数据结构的选择与算法效率
——从 IOI98 试题 PICTURE 谈起

福建师大附中　陈宏

【关键字】
数据结构的选择　线性结构　树形结构

【摘要】
算法 + 数据结构=程序。设计算法与选择合适的数据结构是程序设计中相辅相成的两方面，缺一不可。数据结构的选择一直是程序设计中的重点、难点，正确地应用数据结构，往往能带来意想不到的效果。反之，如果忽视了数据结构的重要性，对某些问题有时就得不到满意的解答。通过对 IOI98 试题 Picture 的深入讨论，我们可以看到两种不同的数据结构在解题中的应用，以及由此得到的不同的算法效率。本文以 Picture 问题为例，探讨数据结构的选择对算法效率的影响。

【正文】

引言

算法通常是决定程序效率的关键，但一切算法最终都要在相应的数据结构上实现，许多算法的精髓就是在于选择了合适的数据结构作为基础。在程序设计中，不但要注重算法设计，也要正确地选择数据结构，这样往往能够事半功倍。

在算法时间与空间效率的两方面，着重分析时间效率，即算法的时间复杂度，因为我们总是希望程序在较短的时间内给出我们所希望的输出。如果在空间上过于“吝啬”而使得时间上无法承受，对解题并无益处。

本文对 IOI98 的试题 Picture 作一些分析，通过两种不同数据结构的选择，将了解到数据结构对算法本身及算法效率的影响。

Picture 问题及算法设计

1、 Picture 问题

Picture 问题是 IOI98 的一道试题，描述如下：

墙上贴着一些海报、照片等矩形，所有的边都为垂直或水平。每个矩形可以被其它矩形部分或完全遮盖，所有矩形合并成区域的边界周长称为轮廓周长。

例如图 1 的三个矩形轮廓周长为 30：

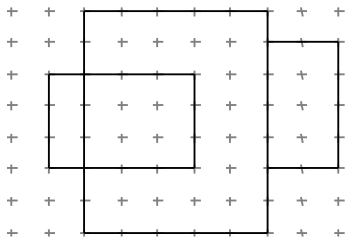

图 1

要求编写程序计算轮廓周长。

数据量限制：

0≤矩形数目<5000；

坐标数值为整数，范围是[-10000，10000]。

2、 算法描述

在算法的大体描述中，将不涉及到具体的数据结构，便于数据结构的进一步选择和比较分析。

（1） 、轮廓的定义

在描述算法前，我们先明确一下“轮廓”的定义：

1、轮廓由有限条线段组成，线段是矩形边或者矩形边的一部分。

2、组成矩形边的线段不应被任何矩形遮盖。图 2 与图 3 分别是遮盖的两种情况。

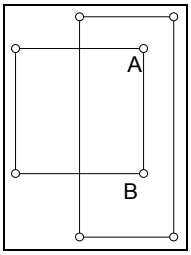

图 2
（AB 被遮盖）

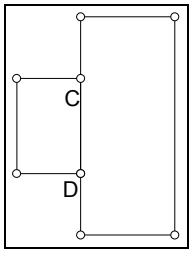

图 3
（CD 被遮盖）

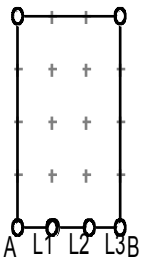

图 4

（2） 、元线段

本题的一大特征是分析矩形的边，而边的端点（即矩形的顶点）坐标为整数，且坐标取值范围已经限定在[-10000，10000]之间。这样，就可以把这个平面理解成为一个网格。由于给出的坐标是整数，所以矩形边一定在网格线上。在网格中，对于一条线段我们最关心其绝对坐标。如图 4，我们认为矩形边 AB 由线段 L_1、L_2、L_3组成。像 L_1、L_2、L_3这样连接相邻网格顶点的基本线段，称之为“元线段”，这样就把矩形边离散化了。显然，有限的元线段覆盖了所有的网格线，且元线段是组成矩形边乃至组成轮廓的基本单位。一条元线段要么完全属于轮廓，要么完全不属于轮廓。这种定义使我们对问题的研究具体到每一条元线段，这样的离散化处理有利于问题的进一步讨论。

（3） 、超元线段

元线段的引入，使问题更加具体。但也应当看到，平面中共有20001*20000*2 条元线段，研究的对象过多，而且计算量受到网格大小的影响，如果顶点坐标范围是[-1,000,000，1,000,000]，元线段数目将达到 8*10^12，这是天文数字。因此有必要对“元线段”进行优化。受到元线段的启发，我们定义一种改进后的元线段——“超元线段”，它将由对平面的“切割”得到。具体做法是，根据每个矩形纵向边的横坐标纵向地对平面进行 2*N 次切割、根据矩形横向边的纵坐标横向地对矩形进行 2*N 次切割（N 为矩形个数）。显然，经过切割后的平面被分成了(2*N+1)^2 个区域，如图 5 所示：

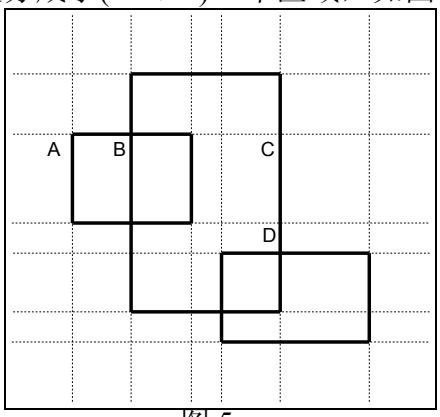

图 5

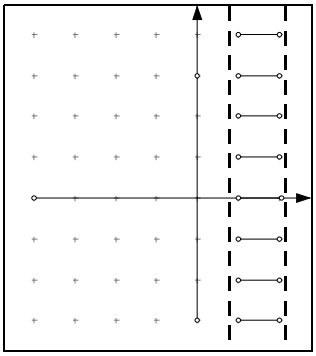
图 6

其中像横向边 AB、纵向边 CD 这样的线段就是“超元线段”。超元线段与元线段有着相似的性质，也是组成轮廓的基本单位。所不同的是，超元线段的数目较少，一般为 4*N 条左右，且超元线段数目不受网格大小的影响。

基于超元线段的优点，算法最终将研究超元线段。

(1)、离散化及算法框架

算法的研究对象是超元线段，但这并不等于逐一枚举，那样耗时过大，而整体考虑又使得问题无从下手。有一种考虑方法是折中的，即既不研究每一条超元线段，也不同时研究所有的超元线段，而是再进一步优化问题的离散化，即将超元线段分组研究。如图 6 所示，夹在两条纵向分割边的超元线段自然地分为一组，它们的共同点是长度相同，并且端点的横坐标相同。纵向线段也可以进行类似的离散化。

这样的离散化处理后，使得问题规模降低，以此为基础，算法的框架可以基本确定为：

1、对平面进行分割。

2、累加器 ans　0。

3、研究每组超元线段，检测其中属于轮廓的部分的长度，并把这一长度累加入 ans。

4、输出 ans 的值。

以上只是算法的基本框架，还很粗糙，求精部分有赖于数据结构的具体选择。

3、 Picture 问题的数据结构选择之一：线性结构

(1)、映射结构的建立

算法的基础是问题的离散化，要进行平面“分割”，一般需要记录分割点，通常采用映射来记录分割点。直观的做法是采用一维数组形式，下标表示分割点的编号，数组元素表示分割点的坐标。利用下标与数组元素的自然对应，实现映射。应该说，这样表示是比较自然的，实现也比较方便。数组的优点主要是存取方便，且可以在 O(NlogN)时间内排序。映射结构定义如下：

```
Type
  Mapped_TYPE = Object
      Len    : 0..Max;                {记下分割点的个数}
      Coord  : array[1..Max] of integer;      {记下分割点坐标}
      Procedure  Creat;                {映射初始化}
      Procedure  Insert(X : integer);        {插入分割坐标 X}
      Procedure  Sort;                 {对坐标排序}
  End
```

以下是三个过程的描述与解释：

```
Procedure  Mapped_TYPE.Creat
1   Len     0
{Creat 用于初始化该映射}

Procedure  Mapped_TYPE.Insert(X : Integer)
1   Len       Len + 1
2   Coord[Len]       X
{Insert 用于插入一个分割坐标，此时坐标之间是无序的}

Procedure  Mapped_TYPE.Sort
略
{Sort 用于将 Len 个坐标排序。由于 Coord 是一维数组，Sort 容易实现，例如
快速排序。设 N = Len，Sort 效率可达 O(NlogN)。针对整数，也可以采用筒排
序得到更好的效率，但这不是问题的关键部分。}

Var
  X_map, Y_map : Mapped_TYPE      {分别记录横纵坐标的映射}
```

以横坐标为例，在程序处理时，首先执行 X_map.Creat 初始化映射。而后通过 X_map.Insert 将每个矩形纵向边的横坐标作为分割坐标插入 X_map.Coord，最后执行 X_map.Sort 进行排序。

至此，映射建立完毕。

应该说，这一部分完全可以满足算法要求，且执行效率较高。三个过程中的 Creat 与 Insert 耗时均为 O(1)，Sort 耗时为 O(NlogN)，但它只需执行一次。

(2)、线性结构的建立

映射建立后，相当于完成了对平面的切割。现在的主要问题是如何描述一组超元线段的状态。由于最终要计算轮廓周长，我们最关心的是一组超元线段中究

竟有多少条属于轮廓。由分组的方法可知，每组超元线段长度相同。以下均以横向超元线段为例进行说明。设：

超元线段组编号 1——N*2-1(N 是矩形数目)

编号为 S 的超元线段组中的线段长度为 Length(S)

编号为 S 的超元线段组中属于图形轮廓的超元线段数目为 Belong(S)

则：

轮廓横向边周长 ANS = $\sum_{S=1}^{N*2-1} Length(s) * Belong(s)$ 算式①

其中 Lenth(s)容易求得。如果超元线段组编号以网格中从左到右为原则，那么 Length(s)就可以表示为： X_Map.coord[s]－X_map.Coord[s-1]，算式①只需求得 Belong（s）即可。

如图 6，可以看到在问题的离散化之后，一组横向超元线段从上到下，数目是有限的，约有 N*2 条。这使得我们很容易想到线性结构。例如用一维数组来描述这么一组线段，用数组下标表示该线段从上到下的编号。数组雏形定义如下

```
Var
  A : array[1..MaxSize] of integer
```

基于对一维数组的使用，可以得到一个称为“累计扫描”的过程，来求解 Belong(s)。累计扫描的思想是，将一维数组的元素看作计数器，计数器 A[I]的内容是覆盖超元线段 I 的矩形上边的数目 — 覆盖超元线段 I 的矩形下边的数目。形象表示如图 7：

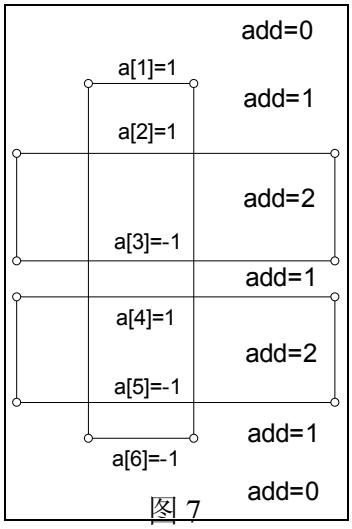

图 7

同时，设立累加器 add，从上至下扫描超元线段，累加 a[I]的值。由图 7 中可以看出，一条超元线段 I 属于轮廓的情况有两种：

1、A[I]≠0 且扫描到该超元线段未累加时 add = 0（超元线段 I 是矩形上边的情况）

2、A[I]≠0 且扫描到该超元线段累加之后 add = 0（超元线段 I 是矩形下边的情况）

这样，对于一组超元线段求解 Belong(s)可以分为两部分：

1、对 A[I]赋值，即累计过程。

2、从上至下扫描一组超元线段并累加 add，即扫描过程。

Belong(s)的值在扫描过程中得到。

至此，描述一组超元线段状态的数据结构基本确立，存储结构是线性一维数组，定义的操作包括累计与扫描两个部分。定义如下：

```
Type
  Group_TYPE = Object
    A : array[1..MaxSize] of Integer;  {线性地记录一组超元线段的信息，如图7}
    Procedure Count;        {累计的过程}
    Function Adding;        {扫描的过程，即求解 Belong(s)的过程}
  End

  Procedure Group_TYPE.Count {累计的过程}
  1  数组 A 清零
  2  for I     1 to N
  3    do if 矩形 I 跨越了超元线段组 S
                                  {即矩形的左右边分别在线段两侧}
  4          then A[矩形 I 的上边]     A[矩形 I 的上边] + 1
  5               A[矩形 I 的下边]     A[矩形 I 的下边] - 1
  {所谓“矩形 I 的上边”指矩形 I 上边纵坐标的映射编号，“矩形 I 的下边”同}

  Function Group_TYPE.Adding {扫描的过程，函数值即为 Belong(S)的值}
  1  调用 Count
  2  add     0
  3  sum     0
  4  for I     1 to 纵坐标的最大映射编号
  5    do if a[I] ≠ 0
  6          then if add = 0
  7                 then sum     sum + 1
                                  {该线段是矩形的上边}
  8               add     add + a[I]
  9               if add = 0
  10                then sum     sum + 1
                                  {该线段是矩形的下边}
  11 return sum
{Count 与 Adding 用于一组超元线段的累计扫描}
  Var
    Scan : Group_TYPE
```

数据结构确立后，Belong(s)通过调用 Scan.Adding 来计算，算式①得以实现。

以上的操作针对一维数组而设计，用于进行一组超元线段的累计扫描过程。执行 Scan.Adding 的时间复杂度为 O(N)。横向超元线段分为 2*N-1 组，固求解横向轮廓周长的算法时间复杂度为 O(N^2)。同理，求解纵向轮廓周长的复杂度也为 O(N^2)，则 Picture 问题的算法时间复杂度为 O(N^2)。虽然这是一个多项式阶，但在最坏情况下（N 接近 5000 时）还有一定的计算量。

对数据结构选择的进一步分析

累计扫描过程体现了一种认识和思维方式，以一维数组作为数据结构基础，这里是否有更好的做法，我们将作进一步分析。

通过求解问题对数据结构选择作的分析中，我们注意到在选择数据结构需要考虑的几个方面：

1、数据结构要适应问题的状态描述。解决问题时需要对状态进行描述，在程序中，要涉及到状态的存储、转换等。选择的数据结构必需先适用于描述状态，并使对状态的各种操作能够明确地定义在数据结构上。在 Picture 问题中，涉及到算法的状态是关于一组“超元线段”的描述，目的是要确定该组超元线段的数目，我们选择了线性结构，采用计数扫描的方法，统计超元线段属于轮廓的数目。这种表示法直观、易于实现，可以说基本适用于描述状态。但采用一维数组，效率并不高，一次扫描耗时较大。其中主要的原因是各组超元线段的扫描分别独立，后面的扫描并不能利用前面的结论。

2、数据结构应与所选择的算法相适应。数据结构是为算法服务的，其选择要充分考虑算法的各种操作，同时数据结构的选择也影响着算法的设计。我们有这样的认识和经历，如果算法是对一个队列进行堆排序，就应当选择能够迅速定位的数据结构，如一维数组等，而不应选择像链表这样定位耗时的数据结构，反之，如果要对一个链表进行排序，则基于链表结构的基数排序应当是首选对象。Picture 问题的算法思想基于问题的离散化，需要对平面进行分割，记录分割点的坐标。通常，使用映射来记录分割点。采用数组形式，利用其下标与数组元素的自然对应，实现映射，直截了当。这样选择基本可以满足算法要求。

同时，在选择数据结构时，也要考虑其对算法的影响。数据结构对算法的影响主要在两方面：

数据结构的存储能力。如果数据结构存储能力强、存储信息多，算法将会较好设计。反之对于过于简单的数据结构，可能就要设计一套比较复杂的算法了。在这一点上，经常体现时间与空间的矛盾，往往存储能力是与所使用的空间大小成正比的。

定义在数据结构上的操作。“数据结构”一词之所以不同于“变量”，主要在于数据结构上定义了基本操作，这些操作都有较强的实际意义。这些操作就好比工具，有了好的工具，算法设计也会比较轻松。Picture 问题中选择了线性结构，它定义的操作比较简单，因此无法很好地将不同组的超元线段统计联系起来。

3、数据结构的选择同时要兼顾编程的方便。许多复杂的数据结构能够得到较好的效率，但编程复杂，不易实现且容易出错。在这种情况下，如果能够选择一种我们较为熟悉的又不会过多地降低程序效率的数据结构，倒不失为一种折中的办法。如 Picture 问题中的 Group_TYPE.Count 过程的 4、5 两步，要求出某个矩形边对应的映射编号。我们定义的映射仅仅是编号　坐标值，并不是坐标值　编号。如果再实现这一映射，势必增加编程难度。所以编程求精时，可以认为以整数而不是以顶点坐标对平面进行横向切割。这样映射关系很好建立，坐标值本身就是编号，减少了编程难度。如果进一步以顶点坐标作横向切割，当然会提高程序效率，但效果并不明显——扫描计数仍需要 O(N)的时间，这是很昂贵的，所以进一步切割并不影响算法主要部分的效率，另一方面，编程难度却会大大提高，得不偿失。由此看出，在算法效率“大局已定”的情况下，有时也需要适当地牺牲程序效率来减少编程不必要的麻烦。

4、灵活应用已有知识。我们对编程都积累了一定的经验，对以后的解题有很大帮助。一个“新问题”有时与“旧问题”有许多内在的联系，往往能够将新问题转化为所学过的知识，或者由所学过的知识得到启发，从而解决问题。所谓“新”数据结构的构造，有时可以是几种基本数据结构的有机结合，或者由基本数据结构得到启发而得到。做到“温故而知新”，是对算法设计者创新意识的要求。当然，对一个问题，要首先考虑现成的、经典的数据结构。如队列、栈、链表等等，其标准结构与标准运算已经有了“公论”，程序实现也经过了“千锤百炼”，效率已经很完美。如果找到一种可行的经典数据结构，那么算法实现一般来说就比较轻松。要做到这一点，要求我们有扎实的基础知识，对各种算法及数据结构了然于胸。在计数扫描过程中采用了经典的线性一维数组，是一个很自然的考虑方向，并且可以很容易上机实现，不足之处在于其效率较低。

总地来说，Picture 问题算法思想的方向还是基本正确的。Picture 问题最大数据应包含近 5000 个矩形，这样大的数据量决定了要降低规模是“大势所趋”，所以对问题的离散化处理是合理的选择。至于效率不高，应当是线性数据结构的选择造成的。由此，我们可以看到使用线性结构来实现 Picture 问题还有一些缺陷，其中最主要的是各组超元线段的统计相互独立，联系不紧，这是算法效率不高的“瓶颈”。为了解决这个问题，我们尝试用其它的数据结构来实现算法，像前面一样，这个数据结构应该符合以下的条件：

1、同线性结构一样，新数据结构要适用于描述一组超元线段的状态。至少，新结构要合理地表示一组超元线段属于轮廓的部分，或者说它要能准确地且较快地计算出算式①中 Belong(s)的值。

2、新结构也要与基本算法相适应。新结构仍然以问题的离散化为基础，映射结构应当保留——事实上映射结构在时间效率上并没有缺点。新结构在描述超元线段组时则要设法将不同组超元线段的统计有机结合起来。

3、新结构还要兼顾编程的方便。如果选择的数据结构编程难度太大，以至于无法上机实现，或者只是理论上的“高效率”，对解题没有实际意义。这么说也许太夸张，但实际上也常常存在“鱼与熊掌不可兼得”的情况。大部分高级数据结构的实现都需要一定的编程技巧。就像问题在时间效率与空间效率上的矛盾一样，算法效率与编程难度也有矛盾，一般来说算法效率越高，编程难度也会越大。考虑新结构时希望能找到实现较为容易的数据结构。

综合以上分析，由于线性结构并不能给我们带来令人满意的效率，所以我们尝试用树形结构来描述一组超元线段的状态，实现 Picture 问题的基本算法。为了提高效率，采用的树结构必需是平衡树，我们姑且称之为“超元线段树”。

Picture 问题的深入讨论

基于对数据结构选择的进一步分析，我们来重新考虑一下 Picture 问题的数据结构的选择，即采用树形结构来描述一组超元线段的状态。

1、 线段树

受到累计扫描过程的启发，一组超元线段属于轮廓的数目，它与跨越该组超元线段的矩形的纵向边位置关系密切。不妨把矩形的纵向边投影到 Y 轴上，这样就把矩形的纵向边看作闭区间，并称之为闭区间 Q。我们以“线段树”的树形数据结构来描述闭区间 Q。作为工具，先简单研究线段树的特点。

线段树是描述单个或若干区间并的树形结构，属于平衡树的一种。使用线段树要求知道所描述的区间端点可能取到的值。换句话说，设 A[1..N]是从小到大排列的区间端点集合，对于任意一个待描述的闭区间 P=[x,y]，存在 1≤i≤j≤N 使得 x=a[i]且 y=a[j]，这里 i, j 称为 x,y 的编号。可以看到，即使是实数坐标，在线段树中也只有整数含义。以下所说的区间[x, y]如无特殊说明，x、y 均是整数，即原始区间顶点坐标的编号。

线段树是一棵二叉树，将数轴划分成一系列的初等区间[I, I+1] (I=1—N-1)。每个初等区间对应于线段树的一个叶结点。线段树的内部结点对应于形如[I, J] (J – I > 1)的一般区间。一般情况下，线段树的结点类型定义如下：

```
Type
  Lines_Tree = Object
    i, j   : integer;      {结点表示的区间的顶点标号 I, J}
    count  : integer;      {覆盖这一结点的区间数}
    leftchild, rightchild : ↑Lines_Tree;   {二叉树的两个子结点}
  end
```

关于 Lines_Tree 的其它数据域与定义的运算将陆续添加。图 8 是一棵线段树，描述的区间端点可以有 10 种取值。其中记录着一个区间[3，6]，它用红色的[3，5]及[5，6]的并采表示。图中红色结点的 count 域值为 1，黑色结点的 count 域值为 0。

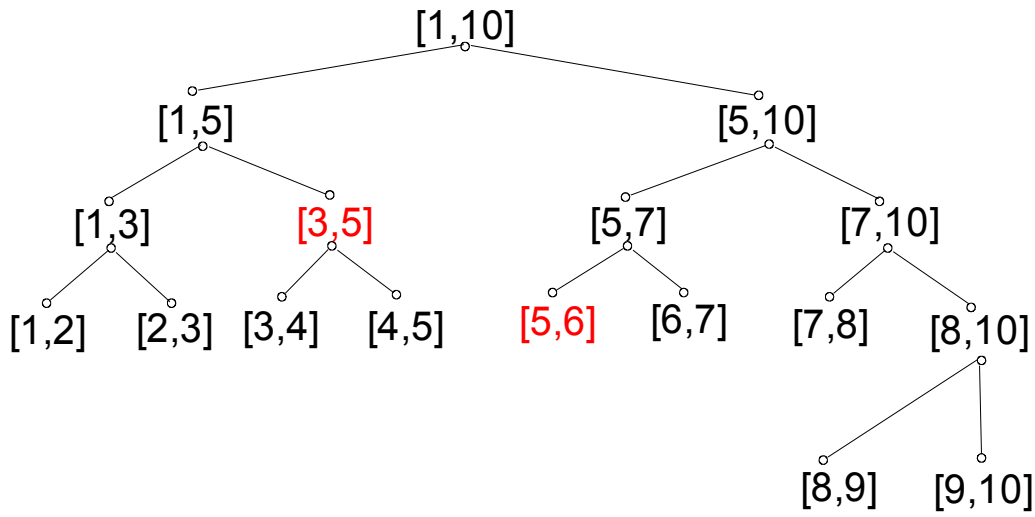

图 8

直观地看，子结点就是父结点区间平均分成两部分。设 L, R 是父结点的区间端点，我们可以增加 Lines_Tree.Build(l, r : integer)递归地定义线段树如下:

```
Procedure  Lines_tree.Build(l, r : integer)
1   I      l          {左端点}
2   J      r          {右端点}
3   Count      0      {初始化}
4   If  r - l  >  1     {是否需要生成子结点，若 r-l=1 则是初等区间}
5       then   k       (l  +  r)      {平均分为两部分}
6                  new(leftchild)
```

```
7                leftchild↑.Build(l, k)   {建立左子树}
8                new(rightchild)
9                rightchild↑.Build(k, r)  {建立右子树}
10     else  leftchild      nil
11               rightchild      nil
```

设根结点是 Root，建树需要执行 Root.Build。

由递归定义看出，线段树是一棵平衡树，高度为 ⌈logN⌉。建立整棵树需要的时间为 O(N)。

以上着重说明了线段树的存储原理，我们还应建立线段树的基本运算。

线段树可以存储多个区间，所以支持区间插入运算 Lines_Tree.Insert(l, r : integer)，定义如下：

```
Procedure Lines_Tree.Insert(l, r : integer)
{[l, r]是待插入区间，l、r 都是原始顶点坐标}
1   if (l <= a[i]) and (a[j] <= r)
2      then count      count + 1          {盖满整个结点区间}
3      else if l < a[(i + j) div 2] {是否能覆盖到左孩子结点区间}
4                then leftchild↑.Insert(l, r)     {向左孩子插入}
5             if r > a[(i + j) div 2 ]  {是否能覆盖到右孩子结点区间}
6                then rightchild↑.Insert(l, r)    {向右孩子插入}
```

类似地，线段树支持区间的删除 Lines_Tree.Delete(l, r : integer)，定义如下：

```
Procedure Lines_Tree.Delete(l, r : integer)
{[l, r]是待删除区间，l、r 都是原始顶点坐标}
1   if (l <= a[i]) and (a[j] <= r)
2      then count      count - 1          {盖满整个结点区间}
3      else if l < a[(i + j) div 2 ] {是否能覆盖到左孩子结点区间}
4                then leftchild↑.Delete(l, r)     {向左孩子删除}
5             if r > a[(i + j) div 2 ] {是否能覆盖到右孩子结点区间}
6                then rightchild↑.Delete(l, r)    {向右孩子删除}
```

执行 Lines_Tree.Delete(l, r : integer) 的先决条件是区间[l, r]曾被插入且还未删除。如果建树后插入区间[2，5]而删除区间[3，4]是非法的。

通过分析插入与删除的路径，可知 Lines_Tree.Insert 与 Lines_Tree.Delete 的时间复杂度均为 O(logN)。(详见[附录 1])

由于线段树给每一个区间都分配了结点，利用线段树可以求区间并后的测度与区间并后的连续段数。

(1)、测度

由于线段树结构递归定义，其测度也可以递归定义。增加数据域 Lines_Tree.M 表示以该结点为根的子树的测度。M 取值如下：

$$M = \begin{cases} a[j] - a[i] & \text{该结点 Count>0} \\ 0 & \text{该结点为叶结点且 Count=0} \\ Leftchild\uparrow.M + Rightchild\uparrow.M & \text{该结点为内部结点且 Count=0} \end{cases}$$

据此，可以用 Lines_Tree.UpData 来动态地维护 Lines_Tree.M。UpData 在每一次执行 Insert 或 Delete 之后执行。定义如下：

```
Procedure Lines_Tree.UpData
1  if count > 0
2     then M      a[j] – [i]    {盖满区间，测度为 a[j] – a[i]}
3     else if j - i = 1       {是否叶结点}
4               then M      0     {该结点是叶结点}
5                        else   M          Leftchild↑.M   +   Rightchild↑.M
                                 {内部结点}
```

UpData 的复杂度为 O(1)，则用 UpData 来动态维护测度后执行根结点的 Insert 与 Delete 的复杂度仍为 O(logN)。

(2)、连续段数

这里的连续段数指的是区间的并可以分解为多少个独立的区间。如[1，2]∪[2，3]∪[5，6]可以分解为两个区间[1，3]与[5，6]，则连续段数为 2。增加一个数据域 Lines_Tree.line 表示该结点的连续段数。Line 的讨论比较复杂，内部结点不能简单地将左右孩子的 Line 相加。所以再增加 Lines_Tree.lbd 与 Lines_Tree.rbd 域。定义如下：

$$lbd = \begin{cases} 1 & \text{左端点 I 被描述区间盖到} \\ 0 & \text{左端点 I 不被描述区间盖到} \end{cases}$$

$$rbd = \begin{cases} 1 & \text{右端点 J 被描述区间盖到} \\ 0 & \text{右端点 J 不被描述区间盖到} \end{cases}$$

lbd 与 rbd 的实现：

$$lbd = \begin{cases} 1 & \text{该结点 count > 0} \\ 0 & \text{该结点是叶结点且 count = 0} \\ \text{leftchild↑.lbd} & \text{该结点是内部结点且 Count=0} \end{cases}$$

$$rbd = \begin{cases} 1 & \text{该结点 count > 0} \\ 0 & \text{该结点是叶结点且 count = 0} \\ \text{rightchild↑.rbd} & \text{该结点是内部结点且 Count=0} \end{cases}$$

有了 lbd 与 rbd，Line 域就可以定义了：

$$Line = \begin{cases} 1 & \text{该结点 count > 0} \\ 0 & \text{该结点是叶结点且 count = 0} \\ \text{Leftchild↑.Line + Rightchild↑.Line - 1} & \text{当该结点是内部结点且 Count=0，Leftchild↑.rbd=1 且 Rightchild↑.lbd=1} \\ \text{Leftchild↑.Line + Rightchild↑.Line} & \text{当该结点是内部结点且 Count=0，Leftchild↑.rbd 与 Rightchild↑.lbd 不都为 1} \end{cases}$$

据此，可以定义 UpData’动态地维护 Line 域。与 UpData 相似，UpData’也在每一次执行 Insert 或 Delete 后执行。定义如下：

```
Procedure Lines_Tree.UpData’
```

```
1   if count > 0          {是否盖满结点表示的区间}
2     then lbd      1
3          rbd      1
4          Line     1
5     else if j - i = 1   {是否为叶结点}
6               then lbd      0   {进行到这一步，如果为叶结点，
                                    count = 0}
7                    rbd      0
8                    line     0
9           else line      Leftchild↑.line + Rightchild↑.line -
                                  Leftchild↑.rbd * Rightchild↑.lbd
                {用乘法确定 Leftchild↑.rbd 与 Rightchild↑.lbd 是否同时为 1}
```

同时，由于增加了 Line、M 等几个数据域，在建树 Lines_Tree.Build 时要将新增的域初始化。

至此，线段树构造完毕，完整的线段树定义如下：

```
Lines_Tree = object
  i, j    : integer;
  count   : integer;
  line    : integer;
  lbd, rbd : byte;
  m       : integer;
  leftchild,
  rightchild : ↑Lines_tree;
  procedure Build(l, r : integer);
  procedure Insert(l, r : integer);
  procedure Delete(l, r : integer);
  procedure UpData;
  procedure UpData';
end
```

有了线段树这个工具，可以考虑利用树形结构来描述一组超元线段的状态。

2、 Picture 问题的数据结构选择之二：树形结构

采用线性结构描述一组超元线段的状态并不能带来太高的效率，其中主要原因是各组超元线段联系不紧。如图 9 所示，超元线段 CD 与 EF 被矩形 AGHB 遮盖，不属于轮廓；而与之相邻 DD'与 FF'则“摆脱”了矩形的遮盖，属于轮廓的一部分。

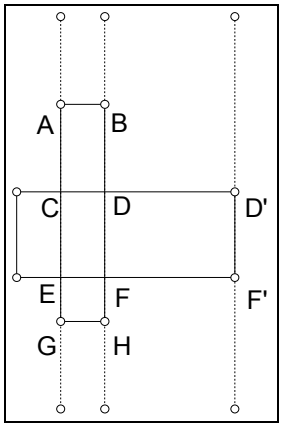

图 9

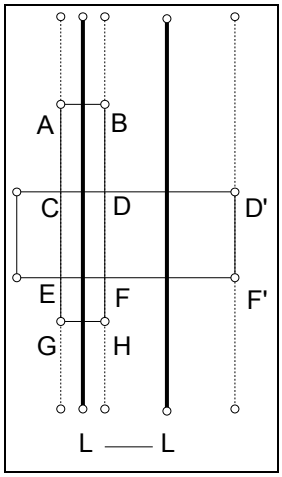

图 10

由此类推，可以看出相邻的超元线段组都有类似的问题。如图 9，DD’与 FF’不被遮盖，可以这样分析：从左往右，CD、EF 首先被遮盖，但随着 BF 的出现，对 DD’、FF’的遮盖自然消失。这一点，正是相邻超元线段组的内在联系。用线性结构无法表示出这一联系，因为各组的累计扫描过程是独立的。现在我们用树形结构来表示将较好地解决这一问题，因为线段树支持插入与删除及动态维护，可以有机联系各组超元线段的状态。我们把“从左往右”当作一个扫描的过程，若将其严格地描述，可以得到一个称为线段扫描的过程：

1. 设立扫描线段 L。
2. L 从左往右扫描，停留在每一超元线段组上。如图 10 所示。
3. L 的状态用线段树来表示，每一条纵向的矩形边看作一个待合并区间。线段树的连续段数*2 表示该组超元线段属于轮廓的线段数目。如图 10，L 的状态首先是[G,A]∪[E,C]，连续线段数是 1，所以 1*2=2 是该组超元线段属于轮廓的数目。接着 L 进一步扫描，状态改变为[E,C], 连续线段数是 1，所以该组超元线段属于轮廓的数目也是 1*2=2。这样，上文所说的“超元线段树”就用线段树来实现。为了统一起见，以后仍称线段树。
4. 扫描过程中动态地维护 L 的状态。参看图 10，L 状态的转换是在线段树中删去区间[H,B]即[G,A]造成的。归纳一下，有以下结论：

   L 初始化为空，即线段树刚建好时的情形。

   扫描时，遇到矩形左边，将其插入（Insert）线段树。

   扫描时，遇到矩形右边，将其从线段树中删除（Delete）。由于从左往右扫描，事先插入了该矩形的左边，所以删除合法。

参看算式①，以上的线段扫描过程可以得到每一组超元线段的 Belong(s)，进一步得到整个图形轮廓的横向边长。同时，线段扫描过程还可以在一次从左到右的扫描中求得图形轮廓的纵向边长。还以图 10 为例。在扫描线状态改变之前，L 是[G,A]∪[E,C]；改变状态之后，[H,F]、[D,B]就“露”了出来，成为轮廓一部分。[G,A]∪[E,C]正是 L 改变前后测度的差。如果描述相邻的扫描线状态的线段树分别为 Tree1、Tree2，则扫描过程中“露出”的纵向边长度为|Tree1.M – Tree2.M|。

在扫描过程中，遇到的插入或删除称为“事件”，待插入或删除的线段称为“事件线段”。在扫描之前，应将事件按横坐标从小到大排序。（详见[附录

2]）

通过以上分析，得到较之线性结构的累计扫描过程改进的线段扫描过程的算法：

1. 以矩形顶点坐标切割平面，实现横纵坐标的离散化并建立映射 X_Map 、Y_Map。
2. 事件排序
3. Root.Build(1,N*2)
4. Nowm　　0
5. NowLine　　0
6. Ans　　0
7. for I　　1 to 事件的最大编号
8. 　do if I 是插入事件
9. 　　then Root.Insert(Y_Map.Coord[事件线段顶点 1]，Y_Map.Coord[事件线段顶点 2])
10. 　　else Root.Delete(Y_Map.Coord[事件线段顶点 1]，Y_Map.Coord[事件线段顶点 2])
11. 　nowM　　Root.M
12. 　nowLine　　Root.Line
13. 　ans　　ans + lastLine * 2 * (X_Map[I] – Y_Map[I-1])
14. 　ans　　ans + |nowM – lastM|
15. 　lasM　　nowM
16. 　lastLine　　nowLine

排序的时间复杂度为 O(NlogN)。事件的最大编号为 N*2，插入或删除的复杂度为 O(logN)，所以整个过程效率为 O(NlogN)。至此，以树形数据结构为基础的算法模式确立，时间效率是令人较为满意的。

3、 两种实现方法的比较

两种数据结构的不同之处不仅在于它们本身的存储差异、定义的运算差异，还在于它们对算法时间复杂度产生的影响。线性结构产生的复杂度为 O(N^2)，而树形结构产生的复杂度为 O(NlogN)。以下是一组数据，将两种数据结构实现程序的运行时间作一个对比，可以感性地认识它们之间的效率差异：

<table>
<tr><th>数据</th><th>实现一：线性结构</th><th>实现二：树形结构</th></tr>
<tr><td>Data_4_1.Txt</td><td rowspan="2">1 秒以内</td><td rowspan="5">1 秒以内</td></tr>
<tr><td>Data_4_2.Txt</td></tr>
<tr><td>Data_4_3.Txt</td><td rowspan="3">30 秒左右</td></tr>
<tr><td>Data_4_4.Txt</td></tr>
<tr><td>Data_4_5.Txt</td></tr>
</table>

注：以上程序在赛扬 300/Borland Pascal 下运行。（程序清单见[程序 1]—线性结构及[程序 2]—树形结构）

4、 Picture 问题的推广

基于对 Picture 问题特征的分析及树形结构的应用，可以使问题得到推广。

离散化思想可以使顶点坐标由整数 实数。在算法及数据结构的选择中，我们已不再使用数据类型为整数的特性。所以 Picture 问题的数据类型可以推广到实型。为了适应这一变化，要将 Mapped.Coord 的基类型改为实型，同时将 Lines_Tree.M 改为实型，线段扫描算法中的累加器 ans 等涉及到顶点坐标的类型也要改为实型。

更重要的是，Picture 问题本身也可以推广，即由 Picture 周长问题 Picture 面积问题。周长问题涉及到扫描线 L 的连续段数与测度，其中横向轮廓涉及到连续段数，纵向轮廓涉及到测度；相应地，面积问题涉及到 L 的测度。在扫描过程中，设事件点为 I，L“扫过”的一组超元线段的面积就是

测度*（X_Map.coord[I]－X_Map.coord[I-1]）。

（程序清单见[程序 3]——Picture 面积问题）

结论

Picture 问题在基于离散化思想的算法下，通过改进数据结构的选择，即由线性结构 树形结构，使得时间复杂度大大降低。改进源于数据结构的选择，更本质地，源于对问题本身与数据结构特点的较为深刻的认识。只有进一步理解问题、进一步认识问题，才能更好地选择适用于问题的数据结构；也只有对数据结构的特性充分理解，才能在各种结构中作出合适的选择。

通过上述讨论，我们进一步认识了数据结构选择的重要性，对选择数据结构的技巧也有了一定认识。其中最重要的一点，就是数据结构要适用于问题、适用于算法，只有这样，才能较为本质地描述状态、解决问题。

从某种意义上说，数据结构与算法是空间与时间的具体体现。在这个意义上，它们有统一的时候，也有矛盾的时候。以 Picture 为例，线性结构可以不占用堆空间，但时间复杂度是 O(N^2)，树形结构要大量占用堆空间，却将复杂度降低到 O(NlogN)。如何协调它们的矛盾，引导其走向统一是算法设计中的一个重要环节。

【参考书目】

《算法与数据结构》 电子工业出版社 编著： 傅清祥 王晓东
《算法 ＋数据结构 ＝程序》 科学出版社 瑞士 N·沃思 著
曹德和 刘椿年 译