Chapter 5

译者 朱翎

概要

这一节将探索一个问题：给定一个元素数量为N的数组A，如何在快速的查找出在任一范围内的最小值与最大值？例如，“在第8个元素到第23个元素之间最小值是多少？”

5.1目的

这一节跟其他章节不同，这个问题与光线追踪并没有直接关系，它不涉及一个光线的生成，追踪，计算相交，或着色等操作。但是，在进行光线追踪时，尤其是当渲染一个体积数据时，会经常遇到这类问题。数据的体积渲染，无论是结构化或非结构化，通常定义一个标量字段，z=f(x)，这通常是用光线步进的形式绘制。与表面渲染一样，快速确定哪个区域的体积是空的或不重要是快速渲染的关键，通过跳过这些区域，减少采样数量，或使用其他近似方法来加快计算速度。这通常涉及构建一个空间数据结构，每个叶子节点储存底层标量的最小值和最大值。

事实上，由于标量很少直接呈现，所以出现了本章的问题。用户交互的修改某些指定颜色跟不透明度映射到不同标量值的转换函数t(z)，(例如，使肌肉跟皮肤透明，使韧带跟肌肉不透明 )，在这种情况下，一个区域标量的极值对于渲染是不重要的。相反，我们需要将转换函数输出的极值应用到标量中。换句话说，假设我们将转换函数表示为一个数组A[i]，标量场映射到数组索引i(lo)与i(hi)之间的最大值跟最小值，分别是我们想要的数组中A中A[i]的最大值跟最小值 (i ∈ [ilo,ihi])。

乍一看，我们的问题类似于计算一个子数组的和，我们可以用 summed-area tables (SATs) [3, 9] 来完成，但是，min() 跟 max()不是可逆转的，所以SATs在这种情况下不可用。本节剩余部分讨论此问题的四种不同的解决方案，每种都在预计算所需内存与查询时间方面都有不同的权衡。

5.2 整表查询

这是最简单的方法，先预计算一个N × N大小的表，Mj,k = min {Ai,i ∈ [j,k]}，简单的查找我们期望的值 。

这个方法既简单又快速，OSPRay [7]中使用了的getMinMaxOpacityInRange()，提供了一个不错的快速解决方案。但是，这有一个明显的缺点，储存消耗数组大小N的平方(O(N^2))，目前对于非平凡数组(e.g., 1k or 4k entries)，这个表会很大，除了尺寸以外，每次转换函数被修改时，该表都需要重新计算，消耗至少是O(N^2)

基于这种复杂性，完整表查询适用于小尺寸，但是较大的数组可能需要不同的解决方案

5.3 稀疏表方法

GeeksForGeeks论坛中概括了一个鲜为人知但值得一试的改进完整表的方法——稀疏表方法。在查找文献之前，我们并不知道这个方法（我们没有在本书的其他地方讨论过），因此，在这里进行一下简单的描述。

稀疏表方法的核心思想是，任何一个n个元素的范围[i,j]可以被看做是两个（有可能重叠）尺寸为2的整数次方的范围的并集（第一个以i开始，另一个以j结束）。在这种情况下，实际上我们不必预计算所有可能查找范围的完整表，而只需要计算尺寸为2的整数次方的查询范围，然后我们在两个2的整数次方范围的预计算结果中查找，最后组合它们的结果。

在详细一些，假设我们预计算一个所有可能性的查询表，表中有两个元素，我们记为L[0,1] = min(A0, A1)，L[1,1] = min(A1, A2)等等，同样，我可以计算表L(2)

一旦我们有LogN 表 L(i)，对于任何查询范围 [lo, hi] 我们可以简单的执行以下步骤： 计算需要查询的序列的宽度 n = (hi - lo + 1)，然后找到最大整数p，保证2^p小于n，此时范围[lo, hi]范围可以看做是两个范围的并集[lo,lo + 2^p - 1] 与 [hi - 2^p + 1,hi]，因为这些序列已经通过预计算保存在表L(p)中了，我们可以简单的查询L[lo, p] 跟 L[hi - 2^p +1, hi]，计算他们的最小值，返回结果。这个方法详细的图解在图5-1中表示

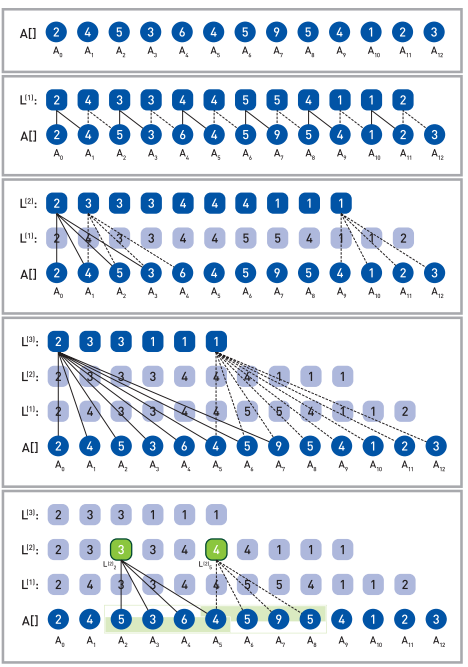


图5-1，稀疏表方法的范例，我们输入13个元素到数组A中，我们预计算L(1), L(2), 和L(3) ，包含了宽度为2，4，8的查询数组，假设我们要查询[A2, A8]范围内7个元素的最小值，我们可以把这个列表分解为两个重叠的长度为4的列表，分别为[A2,A5]跟[A5,A8]，这个分解之后的列表经过预计算，储存在表L(2)中，因此，最终的结果为min(L2(2), L5(2)) = min(3,4) = 3

对于长度为非2的整数次幂来说的区间来说，两个子区间会重叠，意味着某些数组元素会被计算两次，所以这使得该方法不适用于其他类型的优化，例如加法以及乘法。但对于计算最小值与最大值， 双倍的计数不会改变最终结果。在计算消耗方面，该方法的复杂度仍然为o(1),因为所有查询列表可以通过两次查询完成。 在内存开销方面，there are N - 1 entries in L(1), N - 3 in L(2), etc. (这个不会翻译)，总的储存消耗为O(N logN)，明显优于全表查找方法的O(N^2)复杂度。

5.4 递归范围树方法

原文：For ray tracing—where binary trees are, after all, a common occurrence—an

obvious solution to our problem is using some type of range tree, as introduced by

Bentley and Friedman [1, 2, 8]. An excellent discussion of applying range trees to

our problem can be found online [4, 5].2

对于光线追踪，二叉树是很常见的数据结构，毕竟一个常见的情况（这个不知道怎么翻译），对于我们的问题，一个 很明显的解决方案是用某种类型的范围树，Bentley 跟 Friedman介绍的，

范围树是一个二叉树，对于每个节点递归分割输入的范围，储存在对应的子树中，每个叶子节点对应一个数组元素，内部节点有两个子节点（其中一个子节点储存输入范围的上半部分，另一个子节点储存输入范围的下半部分）并且储存着两个子节点的 最大值，最小值，和，以及乘积，等等。图5.2中展示了这样一个树结构如何查询最大值跟最小值的范例。

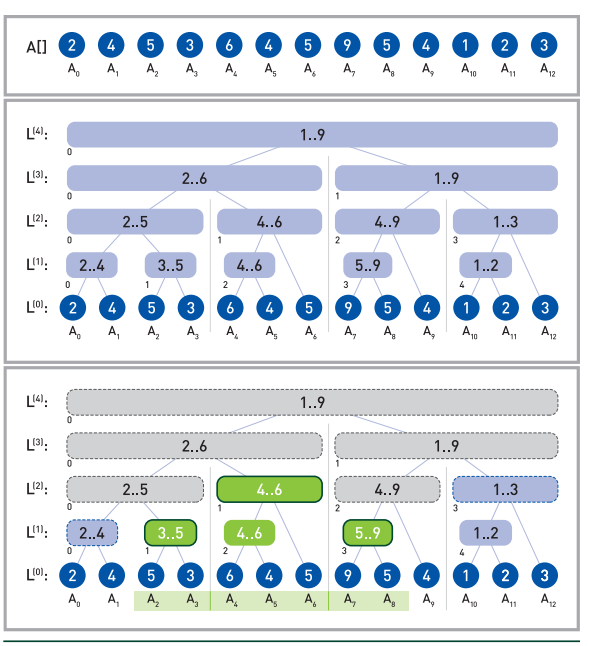


图5-2，展示了递归范围树方法。

上图，指定一个输入数组A，

中图，生成一个二叉树，它每个节点储存了对应叶子节点的最小值跟最大值。

下图，在查询范围内递归遍历二叉树。

伪代码中实现了所有的三种情况

第三种情况，灰色节点对应着两个子节点

第二种情况，黑色轮廓的绿色节点被记录并终止

第一种情况，虚线轮廓蓝色节点完全在需要查询的范围以外

给定这样一个范围树，查询任意范围[lo,hi] 需要找到一系列的节点正好横跨输入范围， 接下来用一个简单的递归算法去执行这个查询

RangeTree::query(node,[lo,hi]) {

    if (node.indexRange does not overlap [lo,hi])

        /\* Case 1: node completely outside query range -> ignore. \*/

        return { empty range }

    if (node.indexRange is inside [lo,hi])

        /\* Case 2: node completely inside query range -> use it. \*/

        return node, valueRange

    /\* Case 3: partial overlap -> recurse into children, & merge. \*/

    return merge(query(node.leftChild,[lo,hi]),query(node.rightChild,[lo,hi])

}

范围树仅仅只需要线性的储存空间以及处理时间，这些整数（todo）因子小于稀疏表方法，它的缺点是，查询的消耗不再是常量时间，取而代之的是O(logN)的复杂度，更糟糕的是，递归查询会实现相对较高的“实现常量”(todo,暂时我把它理解为耗时)，

（尤其在SIMD或SPMD架构上），即使再小心的进行数据布局并且避免指针追踪也会如此。

5.5 迭代范围树查询

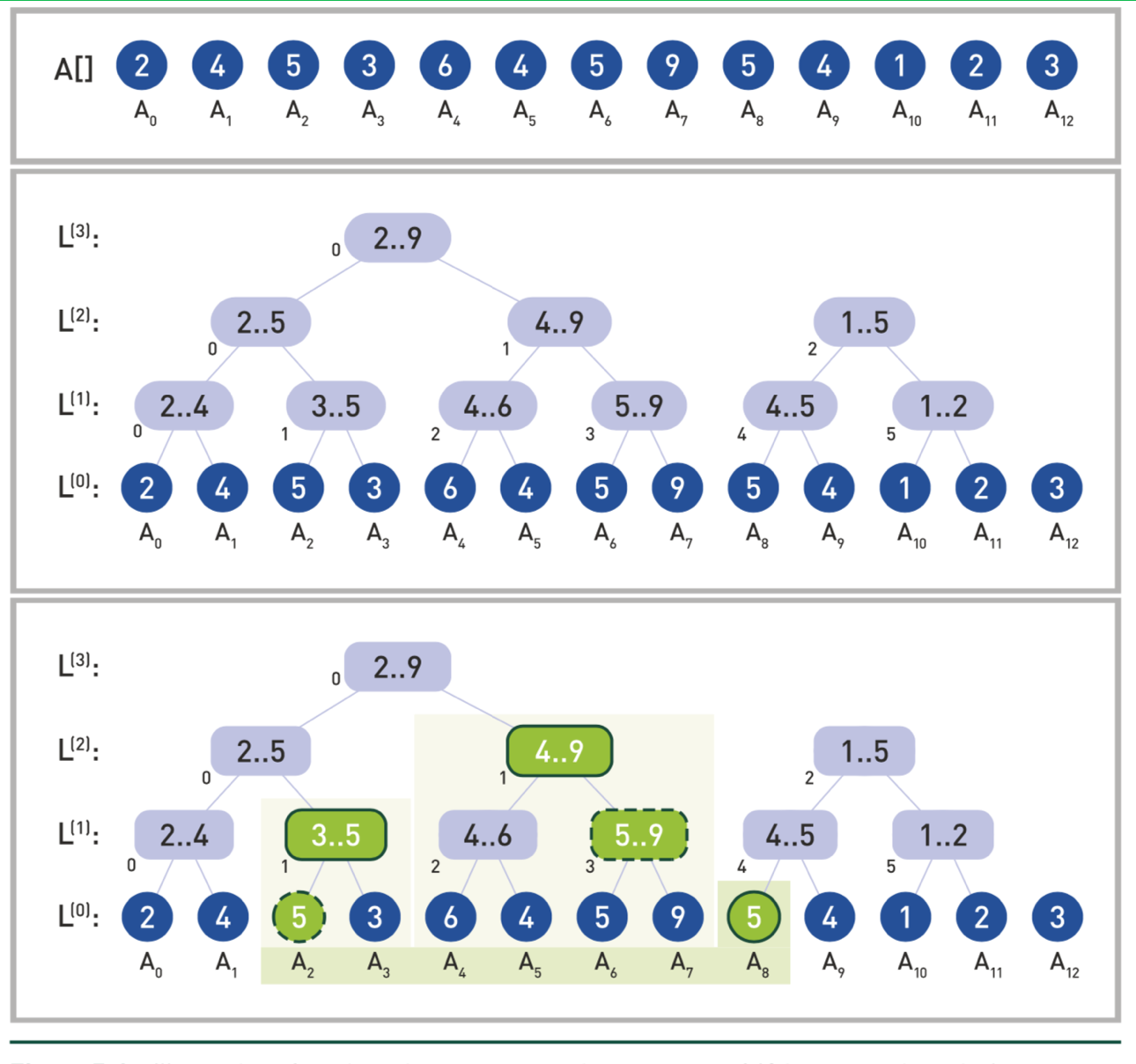
在实际情况下，范围树查询的主要消耗不在于其O(logN)的复杂度，而在于其递归带来的实现常量。所以迭代的方式相比之下要更好一些。

为了推导出这种方法，我们在逻辑上从下往上看一个范围树，那么他就是一个连续对各自更精细级别进行合并的过程。在最精细的一层L[0],我们有N0 = N 个原始数值。Li（0）= Ai 在下一层，我们从上一层的每一（完整的）对值计算最小/最大值，也就是说N1 = floor（N0/2）个值在L（1）i层，且L（1）i=f(L（0）2i,L（0）2i+1),f表示最小值或最大值函数，第2层有N2 = floor（N1/2）个由L1层的数合并的对，依此类推。对非2次幂大小的数组，有些Ni可能是奇数，也就是说这个节点没有父节点，不过我们的算法可以适应这种情况。

从图5-3得到上述结论的数据结构，数据形成了一系列二叉树（当且仅当数组大小是2的倍数时，结果才是一棵二叉树，其他情况下大于一棵）任意层的节点n代表包括这棵子树在内所有数组值的二叉树的根

**图5-3 我们的迭代范围树**

**给定长度为13一个数组，我们迭代地将子节点成对合并成连续地更小的层级。一共形成三个二叉树（本例中）。现在查询[low = 2, high = 8],我们必须找到三个黑色实线框的节点（第0层第8个，第1层第1个，第2层第1个）。算法从第0层开始，此时lo=2，hi=8；这确定了hi是偶数，且应该被计算。**给定一个查询范围[lo, hi]， 我们来看那些子树完全在查询范围内且不是查询范围内其他更大的树的一部分的节点（在图5-3中被粗线框框出来的节点）。显然，这些才是我们需要考虑的节点——所以我们需要找一种有效的方法来遍历这些节点。

 为了做到这点，针对我们的查询范围在每一层L所覆盖到的节点，我们把这些节点列为[loL...hiL]。现在我们来看 loL.，通过构建，我们知道 loL 只有当它的下标是奇数时，才能是一棵子树的根（否则只能是另外一棵树的左孩子）。无论奇数还是偶数，在下一个粗糙层次最左侧的下标被记为loL+1 = (loL+1)/2 。右侧下标hiL也有类似的参数

Iterate(lo,hi) {

Range result = { empty range }

L = finest level

while (lo <= hi) {

if (lo is odd) result = merge(result,L[lo])

if (hi is even) result = merge(result,L[hi])

L = next finer Level;

lo = (lo+1)>>1

hi = (hi-1)>>1 /\* Needs signed arithmetic, else (hi+1)/2-1 \*/

return result

}

}

伪代码中的注释提到，我们必须注意正确处理 hi = 0 时 高位下标的计算。在经典的范围树中，这种迭代方式对输入范围中的每一个值只计算一便因此可以用于除最大值最小值的查询。

关于内存布局，我们已经用一些列数组（每层一组）逻辑地介绍了我们的算法。在实际情况下，我们可以很容易地将所有层级存储在一个包含了每一个L1对应地N1，L2对应的所有值等等的数组中。由于我们总是从最精细的一层遍历到比较粗糙的层级，我们可以隐式计算层级的偏差，可以在（<https://gitlab.com/ingowald/rtgem-minmax>）中看到我们的实现方式。

5.6 比较结果

理论上，迭代方法与经典的范围树方法具有相同的存储复杂度O(N)和计算复杂度O(logN)。 但是，迭代方法的内存布局要简单得多，查询的时间常数也明显低于任何递归实现。 事实上，除了一些包含至少数十万个元素的表之外，在我们的示例代码中， 这个迭代版本的速度几乎与O(1)稀疏表方法一样快， 同时使用的内存要少得多。

例如，使用一个包含4k个元素的数组，并随机选择查询端点lo和hi，迭代方法仅比稀疏表方法慢5%左右，但内存使用量为稀疏表方法的1/10。对于较大的100k个元素的表，速度差增加到约30%，但内存的使用率为稀疏表方法的1/15。 虽然这是一个有趣的权衡方式，但值得注意的是，随机选择的查询端点接近迭代方法的最坏情况:由于迭代计数在∣hi-lo∣中是对数的复杂度，因此小范围查询实际上运行耗时比均匀随机选择的lo和hi值执行的宽范围的查询耗时更短。例如，如果我们将查询值限制为∣hi-lo∣≤N，那么100k元素数组上的迭代方法将比稀疏表方法(在1/15甚至更少内存的情况下)慢30%到快15%

参考引用

* **[1]**Bentley, J. L., and Friedman, J. H. A Survey of Algorithms and Data Structures for Range Searching. http://www.slac.stanford.edu/cgi-wrap/getdoc/slac-pub-2189.pdf, 1978.
* **[2]**Bentley, J. L., and Friedman, J. H. Algorithms and Data Structures for Range Searching. *ACM Computing Surveys 11*, 4 (1979), 397–409.
* **[3]**Crow, F. Summed-Area Tables for Texture Mapping. *Computer Graphics (SIGGRAPH) 18*, 3 (1984), 207—212.
* **[4]**GeeksForGeeks. Min-Max Range Queries in Array. https://www.geeksforgeeks.org/ min-max-range-queries-array/. Last accessed December 7, 2018.
* **[5]**GeeksForGeeks. Segment Tree: Set 2 (Range Minimum Query). https://www. geeksforgeeks.org/segment-tree-set-1-range-minimum-query/. Last accessed December 7, 2018.
* **[6]**GeeksForGeeks. Sparse Table. https://www.geeksforgeeks.org/sparse-table/. Last accessed December 7, 2018.
* **[7]**Wald, I., Johnson, G. P., Amstutz, J., Brownlee, C., Knoll, A., Jeffers, J. L., Guenther, J., and Navratil, P. OSPRay—A CPU Ray Tracing Framework for Scientific Visualization. *IEEE Transactions on Visualization 23*, 1 (2017), 931–940.
* **[8]**Wikipedia. Range Tree. https://en.wikipedia.org/wiki/Range\_tree. Last accessed December 7, 2018.
* **[9]**Wikipedia. Summed-Area Table. https://en.wikipedia.org/wiki/Summed-area\_table. Last accessed December 7, 2018.