第35卷 第10期 计 算 机 工 程 2009年5月
Vol.35 No.10 Computer Engineering May 2009
·软件技术与数据库· 文章编号：1000—3428(2009)10—0040—04 文献标识码：A 中图分类号：TP311

快速动态优先搜索树的实现及其应用

黄惠萍，陆伟成，肖林甫，赵文庆
(复旦大学专用集成电路与系统国家重点实验室，上海 201203)

摘 要：对形如([$x1$: $x2$], [-∞: y])的二维查询问题，提出一种快速的、易于实现的动态优先搜索树数据结构及其相关算法，采用只在叶节点存储数据的结构，以及在常数时间内实现旋转操作的算法。设 n 为数据点的个数，k 为满足搜索条件的解的个数，则该动态搜索树空间复杂度为 $O(n)$，插入、删除操作的时间复杂度为 $O(\log n)$，搜索复杂度为 $O(\log n+k)$。

关键词：动态优先搜索树；区域树；堆

Realization and Application of Fast Dynamic Priority Search Tree

HUANG Hui-ping, LUK Wai-shing, XIAO Lin-fu, ZHAO Wen-qing
(State Key Lab of ASIC & System, Fudan University, Shanghai 201203)

【Abstract】 A fast Dynamic Priority Search Tree(DPST) is proposed for 2-D range query in form of ([$x1$: $x2$], [-∞: y]). The proposed DPST stores input data only in its leaf nodes and performs tree rotation in constant time. Let n be the total number of leaf nodes in the tree, and k be the number of solutions in query. The tree requires takes $O(n)$ storage space and takes $O(\log n)$ for insertion and deletion, and $O(\log n+k)$ time for query.

【Key words】 Dynamic Priority Search Tree(DPST); range tree; heap

1 概述

优先搜索树(Priority Search Tree, PST)[1]是一种二叉搜索树，存储二维坐标点(x,y)，主要应用于形如([x_1:x_2], [-∞:y])的一端无边界的特殊二维区域搜索。PST 同时具备一维区域树和堆的特性：对于任何节点，它的左子树的 x 坐标总是不大于右子树的 x 坐标值。即从 x 坐标来看，PST 具有一维区域树的特征；而且任何子树根节点的 y 坐标值都是该子树的最小值，这又使得PST具有最小堆的特点。图1给出了一棵PST。

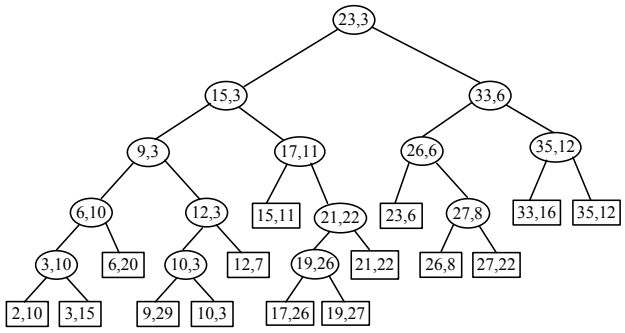

图 1 优先搜索树

动态优先搜索树(Dynamic Priority Search Tree, DPST)[2]具有 PST 的性质，同时支持节点的动态插入与删除等操作，具有更加灵活的应用。

但是文献[2]提出的 DPST 数据结构和算法过于复杂，难以实现：数据有可能同时存储在叶节点和内部节点中，逻辑相当复杂；不包含父节点指针，遍历不方便；平衡操作的时间复杂度为 $O(\log n)$(n 为数据点个数)；采用递归的方法实现查询算法，无法中途终止递归过程。本文提出的 DPST 有以下改进：只在叶节点存储数据；数据结构包含父节点指针；常数时间内实现树的重新平衡；查询操作采用类似 STL 中的迭代器(iterator)的设计模式以实现解集的遍历。

2 动态优先搜索树的应用

DPST 应用广泛，不仅适用于几何方面的查询，还可以用于构造动态路由表、Linux 内核优化等，而且尤其适合于超大规模集成电路物理设计方面的应用。将其应用于交替移相掩模算法以搜索矩形的相交情况从而构造冲突图[3]，以下具体说明该应用原理。

数轴上的线段可通过某种映射与平面上的点一一对应，从而线段之间相交情况的查询可以转换为二维区域搜索[4]。具体地，如果将[l:u]存储成(u,l)的形式，则查询与 q 有交叠的线段就转化为查询([l_q:+∞], [-∞:u_q])区域内的二维坐标点。

结合线扫描算法[1]和 DPST 的区域搜索方法，可以用来探测平面上与 x 轴、y 轴平行的矩形之间的相交关系。2 个矩形相交的充分必要条件为它们在 x 轴和 y 轴上的投影线段都分别相交。因此，可以用与 x 轴垂直的直线对平面上的矩形从左到右进行一次扫描，判断出各个矩形的 x 轴投影线段的相交情况，并保存与当前扫描线所属矩形 *rect* 在 x 轴上有交叠的矩形集合 R。然后利用 DPST 的区间搜索功能判别 R 中矩形与 *rect* 在 y 轴上投影线段的相交情况，即将 y 轴上的线段[y_1:y_2]存储为(y_2, y_1)。设 *rect* 的 y 轴投影线段区间为[y_{lq}:y_{uq}]，则 y 轴线段的相交情况可转换为([y_{lq}:+∞], [-∞:y_{uq}])范围内的二维查询。最终搜索得到的解集即为与 *rect* 相交的矩形集合。

由于版图的数据量十分庞大，因此构造的 DPST 很大，

基金项目：国家自然科学基金资助项目(90307017, 60676018)；教育部高等学校博士学科点专项科研基金资助项目(20050246082)；上海市自然科学基金资助项目(05JC14007)

作者简介：黄惠萍(1982－)，女，硕士研究生，主研方向：集成电路计算机辅助设计；陆伟成，副教授；肖林甫，硕士研究生；赵文庆，教授

收稿日期：2008-10-22 **E-mail：**052021072@fudan.edu.cn

而且需要不断地进行节点的插入与删除以及大数量级的重复搜索。但是版图的局部关联性强，全局关联性弱，每次搜索解集个数并不是很多。本文提出的 DPST 在这样的应用中优势非常突出，速度比文献[2]提出的 DPST 快。

3 数据结构

本文提出的 DPST 只采用叶节点来存放数据。设输入的数据点集合为 K，则 K 中所有数据都存储于叶节点。内部节点只存放搜索信息，所有内部节点的左右子节点都非空。若集合 K 中有 n 个点，则生成的 DPST 包含 $2n+1$ 个节点。这样的数据存储方式使得 DPST 逻辑更加简单，更易于实现。

节点的数据结构定义如下：

```
Node v {
   Point p;   //坐标值
   Node *left;   //左子节点
   Node *right;   //右子节点
   Node *parent;   //父节点
   bool winner;   //竞赛中是否为胜者的标志
   Nodecolor color;   //颜色标志
};
```

在数据成员中，p 用来存储节点坐标；$p.x$ 和 $p.y$ 分别表示节点的 x 坐标值和 y 坐标值。对于任意内部节点 v_{int}，其 x 坐标都是以它为根节点的子树中某个叶节点的 x 坐标值，且保证 $v_{int}.p.x$ 大于其左子节点的 x 坐标值且不小于其右子节点的 x 坐标值，即对 x 坐标维护一维区域树的特性。而 v_{int} 的 y 坐标值始终是以它为根节点的子树中所有节点的 y 坐标值的最小值。为了保证这一点，用类似锦标赛的方法来构造 y 坐标下的最小堆。*winner* 成员标记锦标赛中节点是否为胜者，胜者记为真，反之为假。具体来说，锦标赛时自底向上地比较每个节点的左右子节点的 y 坐标值，将较小的一个记为胜者，并取该值为父节点的 y 坐标值。由于在插入和删除节点操作时需要重新进行锦标赛，因此在节点中包含 *winner* 成员可以利用原先节点间的胜负关系，达到减少比较次数的目的。*left*, *right*, *parent* 分别表示节点的左子节点、右子节点和父节点。选择利用红黑树[5]的原理来保持树的平衡，*color* 成员即红黑树中的平衡因子。

4 算法

DPST 主要应用于二维查询：给定平面点集 $K=\{(x_1,y_1),(x_2,y_2),\cdots,(x_n,y_n)\}$和搜索区域 $Q=([x_{q1}:x_{q2}],[-\infty:y_q])$，要求找出所有在 Q 范围内的数据点集合。实际应用中可能需要不断增加或者减少点集 K 中的数据点，因此，DPST 必须支持节点的动态插入与删除。笔者的数据结构支持动态平衡，在每次插入、删除操作后，都通过旋转操作维持树的平衡。

4.1 插入操作

算法 1 描述了节点的插入过程。设待插入的新节点坐标值为(x_1,y_1)。首先在 DPST 中寻找新节点待插入的位置。由于对 x 坐标而言，DPST 是一维区域树，因此从根节点开始，自顶向下进行遍历，如果 x_1 比当前节点的 x 坐标小，则接下来遍历该节点的左节点，反之遍历该节点的右节点，如此不断重复直至遍历到某个叶节点。设该叶节点为 v，则 v 的节点位置为待插入位置。创建一个新节点 z，用 z 取代 v 在搜索树中的位置；再创建一个新节点 w，设定 w 的坐标值为(x_1,y_1)，并使得 v 和 w 成为 z 的子节点。

算法 1 插入节点

输入 新节点(x_1, y_1)

根据 x_1 的值找到待插入的节点位置，记为 v
创建新节点 z，用 z 代替 v
创建新节点 w，使得 w.p=(x_1,y_1)
令 v 和 w 为 z 的子节点
设置 z 的 x 坐标
进行锦标赛以维持 y 坐标下的堆的特性
进行平衡操作

由于插入过程中已经根据 x_1 的值对二叉树进行遍历，因此只需要设置节点 z 的 x 坐标为 v, w 两者的较大坐标值，就能维持搜索树在 x 坐标下的一维区域树的特性。图 2 所示为将坐标为(18,5)的节点插入到图 1 所示 DPST 的第 1 个步骤。此时 x 坐标已经更新，但 y 坐标还未更新，即 y 坐标的最小堆还未建立。图中粗线条表示需要重赛的路径。

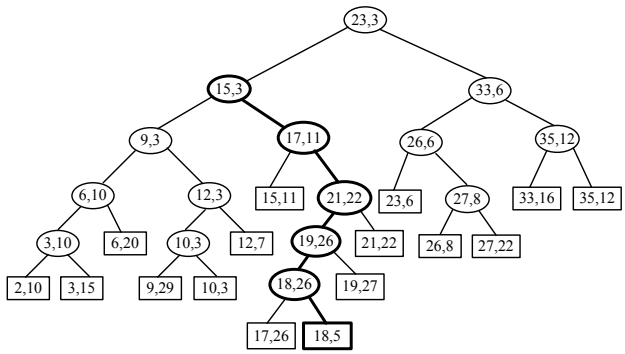

图 2 锦标赛之前的插入操作

接下来，必须更新某些节点的 y 坐标值以维持搜索树在 y 坐标下堆的特性。记从根节点到新节点 w 的路径为 P。因为节点的插入不影响其他路径上的节点，所以只需要在路径 P 上进行锦标赛就能保证搜索树在 y 坐标下仍然是一个堆。具体地，从新节点 w 开始，自底向上进行锦标赛，直至 y_1 不再是胜者的坐标值。由于最多只需要遍历路径 P 上的所有节点，因此该过程的算法复杂度为 $O(\log n)$。

最后进行平衡操作。由于采用红黑树平衡原理，因此平衡算法复杂度为 $O(1)$。图 3 所示为插入操作完成后的 DPST。

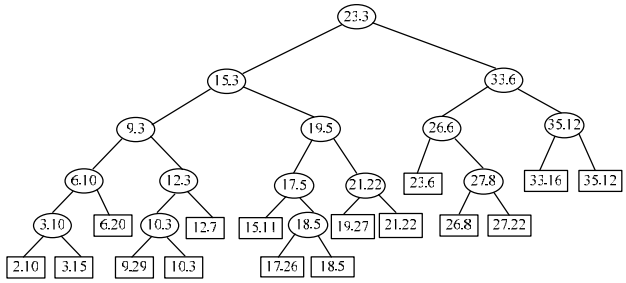

图 3 锦标赛和平衡之后的插入操作

综上所述，插入节点的时间复杂度为 $O(\log n)$。

4.2 删除操作

删除算法与插入算法类似，步骤见算法 2。删除操作的时间复杂度为 $O(\log n)$。

算法 2 删除节点

输入 待删除节点 v

令 z 为 v 的父节点，w 为 v 的兄弟节点
用 w 替换 z，删除 v 和 z
进行锦标赛以维持 y 坐标下的堆的特性
进行平衡操作

4.3 查询操作

DPST 应用于形如$([x_{q1}:x_{q2}],\ [-\infty:y_q])$的一端无边界的二维

区域搜索。利用 DPST 的一维区域树特性可以确定 x 坐标下的子树分支，而 DPST 的最小堆特性则能够对数据结构进行高效的遍历。

文献[2]采用递归方法并最终返回一个数组点集。本文算法采用迭代器的方法，先找出查询范围的上界和下界分别对应的节点，然后通过迭代器遍历解集，从而对查询区域内的数据点进行处理。算法 3 给出遍历解集的方法。每个迭代器都指向满足搜索条件的某个叶节点。LowerBound 函数用来初始化迭代器 it_1，使其指向查询解集中 x 坐标值最小的叶节点。类似地，UpperBound 函数返回指向查询解集中 x 坐标值最大的叶节点的迭代器。然后利用迭代器自加(iterator++)的方法来遍历整个解集。迭代器方法封装了实现的具体细节，使得解集的遍历更为简便直观，提高了用户友好性。下文将具体说明迭代器的初始化和自增操作。

算法 3 遍历解集

输入 $x_{q1}, x_{q2}\ (x_{q1}<x_{q2}), y_q$

$it_1 \leftarrow$ LowerBound(x_{q1}, x_{q2}, y_q)

$it_2 \leftarrow$ UpperBound(x_{q1}, x_{q2}, y_q)

do

  进行一些需要的处理

  ++it_1

while it_1 != it_2

4.3.1 迭代器的初始化

记 v_{lm} 为查询解集中 x 坐标值最小的数据节点，即 v_{lm} 满足搜索条件且在所有满足搜索条件的叶节点中位于 DPST 的最左端。算法 4 返回指向节点 v_{lm} 的迭代器：首先从根节点开始，利用二分法自顶向下地遍历 DPST 以寻找分支节点 v_s，v_s 是遍历过程中满足 x 坐标范围要求的第 1 个节点。根据一维区域树的特性，显然只有 v_s 的子树中包含的叶节点才可能在查询范围内[1]。之后调用算法 5 从 v_s 开始向下遍历，以确定 v_{lm} 在 DPST 中的位置，从而得到指向 v_{lm} 的迭代器。

算法 4 LowerBound

输入 $x_{q1}, x_{q2}\ (x_{q1}<x_{q2}), y_q$

利用二分法找到第 1 个节点 v_s，满足 $x_{q1} < v_s.p.x < x_{q2}$

return iterator(LowerBoundRecur(v_s, x_{q1}, x_{q2}, y_q))

算法 5 LowerBoundRecur

输入 节点 $u, x_{q1}, x_{q2}\ (x_{q1}< x_{q2}), y_q$

w←NULL

while u!=NULL

 if y_q< u.p.y

  return NULL //未找到

 if u 为叶节点

  if x_{q1}<=u.p.x<=x_{q2}

   return u //找到数据节点

  else

   return NULL //未找到

 if x_{q11}<=u.p.x

  w ←LowerBoundRecur(u.left, x_{q1}, x_{q2}, y_q) //遍历左子树

 if w!=NULL

  return w //找到数据节点

 u ← u.right //遍历右子树

return w

算法 5 经由递归方法查找 v_{lm}。设当前节点为 u。由于任意节点具有其子树中所有节点的 y 坐标的最小值，因此当 u 的 y 坐标比 y_q 小时，u 的子树中包含的的所有叶节点都不满足搜索要求，不必再往下遍历。否则，若 u 为叶节点且 u 的 x 坐标在$[x_{q1}:x_{q2}]$内，则 u 即为要找的数据节点。而当 u 为内部节点且 $x_{q1} \le u.p.x$ 时，u 的左子树可能包含满足要求的点，因此必须递归地遍历 u 的左子树。当 u 的左子树中未找到满足要求的叶节点时，则继续在其右子树中递归查找。如此不断迭代，最终将找到 v_{lm}。

类似地，UpperBound 函数返回指向位于 DPST 的最右端的满足搜索条件的叶节点的迭代器。

4.3.2 迭代器自增

完成迭代器的初始化后，就可以通过迭代器自加的方法来遍历所有满足查询要求的数据点。

每个迭代器都指向某个满足搜索条件的叶节点即某个解节点。迭代器的初始化限定了迭代器的范围，迭代器通过增减操作在这个范围内移动。设当前迭代器指向解节点 v。迭代器进行自增操作则迭代器前进一步，指向满足搜索条件且紧挨着 v 的往右的下一个叶节点。这样就可以从 DPST 中最左边的第 1 个解 v_{lm} 出发，不断进行迭代器的自增操作得到往右的下一个解，直到遍历到最右边的解节点，从而得到整个解集。

算法 6 详细说明了迭代器的自增操作。每次外层循环都尝试得到当前节点 n 的按 x 坐标进行中序遍历时的后件节点，但当 n 为内部节点且其 y 坐标超出 y_q 范围时必须跳过对其子树的遍历以节省遍历时间。由于迭代器必然指向解节点，因此若为非叶节点或者不满足 y 坐标的范围要求时，则必须不断循环直到找到符合条件的叶节点，最终返回指向该节点的迭代器。

算法 6 ++iterator

输入 迭代器 it, y_q

n← it.node

do

 if n 为叶节点 or (n.y>y_q and n 不是根节点)

  while n= n.parent.right and n 不是根节点

   n←n.parent

  n←n.parent

 else

  n←n.right

  while n.y<=y_q and n 不是叶节点

   n ←n.left

 if n=NULL

  break

while n 不是叶节点 or n.y>=y_q

it.node←n //设置自增后的迭代器指向节点 n

return it

4.4 平衡操作

应用红黑树的原理来维持 DPST 的动态平衡。红黑树通过节点的旋转操作来达到树的平衡，而且任何不平衡的状态都可以在 3 次旋转之内得到解决[5]。由旋转的定义可以得出，旋转前后 DPST 在 x 坐标下的一维区域树的特性不会发生改变，因此，不需要调整节点的 x 坐标值。但是旋转操作会破坏 DPST 在 y 坐标下最小堆的特性，因此，旋转的同时必须重新进行锦标赛并调整相关节点的 y 坐标值。下文仅讨论旋转时进行锦标赛的方法，以下提及的坐标值均指节点的 y 坐标值。

不失一般性，只讨论左旋转的情况。可以证明：对于每一次旋转操作，最多只需要重新进行一次锦标赛。设 u, w 为待旋转的节点，其中 w 为 u 的右节点。lu, lw, rw 分别表示 u

的左子节点和 w 的左右子节点。问题可划分成以下 2 种情况。

情况 1：旋转前在锦标赛中 w 是输家。

情况 2：旋转前在锦标赛中 w 为胜者。

对于情况 1，如图 4 所示，旋转之前 lu 为胜者，即 lu 小于 w 从而小于 lw，所以旋转后 lu 保持为胜者，故 u 仍取 lu 的值。同时，u 旋转前为 rw 的前辈节点，所以与 rw 进行锦标赛必然为胜者。因此，旋转后只需要改变 w 节点的坐标值为 u 节点的坐标值即可，不用进行任何重赛。旋转操作只需要花费常数时间。

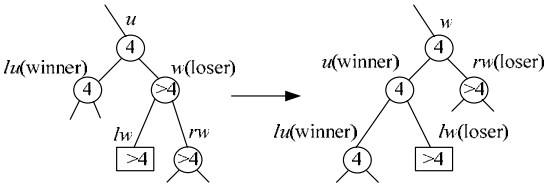

图 4　旋转操作情况 1：w 为输家

而情况 2 又可以划分为 2 类。

情况 2.1：lw 为胜者。

情况 2.2：lw 为输家。

图 5 给出情况 2.1 的示意图，可以看出旋转之后各节点的状态不发生改变，不用进行任何重赛。旋转操作在常数时间完成。

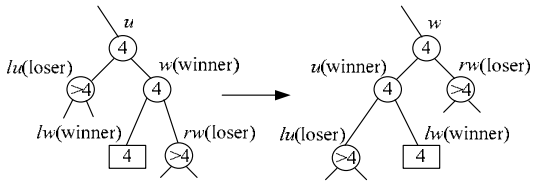

图 5　旋转操作情况 2.1：w 和 lw 皆为胜者

而对于情况 2.2，如图 6 所示，旋转前 lu 与 lw 都为输家，无法确定其大小关系，所以旋转后必须重新比较 lu 与 lw 的值以确定 u 的坐标值。因此，需要一次重赛，旋转操作仍然在常数时间内实现。

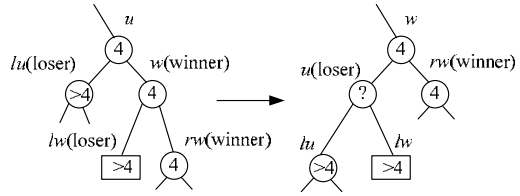

图 6　旋转操作情况 2.2：w 为胜者且 lw 为输家

综上所述，该 DPST 可以在常数时间内实现一次旋转操作，从而可以在常数时间内达到重新平衡。

5　实验结果

用 C++语言实现了以上算法，并在 Linux 服务器(2.80 GHz Intel Xeon CPU, 8 GB 内存)上进行测试。实现了 2 种平衡的 DPST，包括本文版本(记为 sye DPST)和文献[2]的版本(记为 ori DPST)，以及一种非平衡的 DPST(Radix PST, RPST)，并对它们的插入、删除和查询操作进行比较。

5.1　插入操作

笔者同时比较了创建整棵树的时间以及创建完成后再向其中插入 10 个节点所用的时间。表 1 给出以平面分布均匀的点集作为输入的实验结果。表 2 实验数据对应的输入点集中各点在平面中分步很不均匀，而且其中大部分点包含于一个很窄的区间。

表 1　均匀取点情况下的插入操作

节点个数	创建整棵树所用时间/ms			插入 10 个点所用时间/μs		
	sye DPST	ori DPST	RPST	sye DPST	ori DPST	RPST
1 000	0.5	0.66	0.3	5	6	9
5 000	3.0	4.10	1.7	14	16	5
10 000	6.6	8.80	3.7	8	9	5
50 000	62.0	116.00	25.0	14	28	7
100 000	143.0	183.00	66.0	17	22	10
500 000	1 186.0	1 342.00	572.0	26	60	17

表 2　不均匀取点情况下的插入操作

节点个数	创建整棵树所用时间/ms			插入 10 个点所用时间/μs		
	sye DPST	ori DPST	RPST	sye DPST	ori DPST	RPST
1 000	0.49	0.67	0.56	5	6	11
5 000	2.90	4.00	2.20	11	8	5
10 000	6.60	8.80	5.20	7	11	5
50 000	57.00	75.00	60.80	13	23	28
100 000	137.00	180.00	294.00	16	19	61
500 000	1 082.00	1 377.00	9 146.00	28	29	369

可以看出，无论点集分步均匀与否，sye DPST 的插入速度总是比 ori DPST 快。这是由于 sye DPST 完成旋转操作只需要常数时间，平衡速度较快。但是因为 RPST 不是平衡树，所以它在不同情况下表现差异很大：点集分步均匀时插入速度很快，然而点集分步不均时由于搜索树严重不平衡，因此需要花费很长的时间。

5.2　删除操作

同样地，分别取分步均匀与分步不均匀的点集作为输入，比较删除整棵树以及删除 10 个节点的时间。实验结果如表 3、表 4 所示。可以看出，与插入操作类似，sye DPST 的删除操作总是比 ori DPST 快，而 RPST 在不同情况下仍然存在很大的表现差异。

表 3　均匀取点情况下的删除操作

节点个数	删除整棵树所用时间/ms			删除 10 个点所用时间/μs		
	sye DPST	ori DPST	RPST	sye DPST	ori DPST	RPST
1 000	0.29	0.4	0.2	7	9	5
5 000	1.90	2.5	1.3	9	11	8
10 000	6.56	6.7	2.9	13	18	6
50 000	56.20	87.0	24.1	21	40	13
100 000	143.00	176.0	70.0	26	28	18
500 000	1 090.00	1 380.0	614.0	34	58	21

表 4　不均匀取点情况下的删除操作

节点个数	删除整棵树所用时间/ms			删除 10 个点所用时间/μs		
	sye DPST	ori DPST	RPST	sye DPST	ori DPST	RPST
1 000	0.29	0.40	0.30	7	9	5
5 000	1.90	2.70	1.70	10	13	8
10 000	4.87	7.20	3.84	16	19	11
50 000	62.40	76.00	63.40	23	29	36
100 000	145.00	189.00	317.00	27	31	74
500 000	1 081.00	1 224.00	9 017.00	37	44	349

5.3　查询操作

接下来比较 3 种数据结构执行查询操作的运行时间，表 5 给出了实验结果。

表 5　查询操作

节点个数	解的个数	运行时间/μs		
		sye DPST	ori DPST	RPST
10 000	1	3	6	5
	2	4	7	4
	7	6	8	11
	17	8	13	16
100 000	12	11	14	11
	23	25	19	17
	127	58	52	39
	275	117	101	86
1 000 000	27	20	25	20
	125	75	65	46
	256	187	100	78
	520	277	198	169

可以看出，当解集中包含点的个数较少时，sye DPST 可以较快地完成搜索。当解集点数较多时，其他方法会快一些。这是因为 sye DPST 中所有数据存储于叶节点，遍历解集需要花费较多的时间。 (下转第 48 页)