Two Center Disk Covering 问题的实现

问题背景

一家电信公司希望能在一个城市里面设置两个信号基站，要求基站发出来的信号能覆盖该城市里面的所有居民区和商业区。每个基站的覆盖范围是一个以自己为圆心，以某个值（与基站功率大小有关）为半径的圆盘。当然，信号越强，对基站的要求越高，成本就要提高，所以电信公司想给这两个基站选定两个位置，使得它们发出信号能覆盖所有预定区域并且功率尽可能的小。我们就是要帮助这个电信公司选定两个好的位置，并且给出基站应该配置多大的功率（也就是计算出基站要覆盖的区域的半径大小）。

问题描述

我们可以将这个城市的布局用二维平面来表示，商业区和居民区对应着平面上的点，而基站的覆盖区域对应于平面上两个圆盘。于是这个问题可以抽象为下面的数学问题：

给定平面上的 n 个点的集合 S，确定两个封闭圆盘的圆心位置和圆盘的最小半径，使它们的并集能覆盖 S 中的所有点。

这就是著名的几何优化问题——“*Two-Center*”问题。

如果使用 p 个全等圆盘进行覆盖，要求半径最小，就是 *p-center* 问题，如果 p 也是输入的参数的一部分，则在[5]中被证明是 *NP-complete* 问题了。因此人们开始寻找固定了 p 值后的一些特殊情况，比如 p=1，就是 *One-Center* 问题，也称为“*smallest enclosing disk problem*”，可以在 $O(n)$ 时间内解决，见参考文献[6]，下一个问题就是 p=2 的情况了。在 1996 年之前，人们提出的最好算法是 $O(n^2 \log n)$，见参考文献[2]，之后 *Micha Sharir* 首次提出了 $O(n \log^9 n)$ 的算法，将复杂度降低到了平方量级以下，见参考文献[4]。随后一年，*David Eppstein* 在 *Micha Sharir* 的算法基础上，作了改进，将复杂度降低到了 $O(n \log^2 n)$，见参考

文献[1]。

算法原理

基本按照参考文献[4]提出的方法实现 Two-Center 问题的整体框架，考虑到要将并行处理转化为串行运算，我们对模块内部的实现采用了一些不同的处理方法。下面具体说明算法的实现步骤。

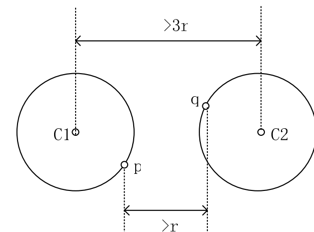
- 整体框架

  参考文献[4]提出的算法是 *Parametric search*，要求我们用并行处理机对不同的情况（包括半径 r 和圆心连线的角度）同时进行测试，选出最优的结果作为输出。在实现的时候，我们采用的是串行处理，分别按照两圆距离很远、两圆一般接近、两圆重合部分很大这三种情况来求出满足条件的最小半径，然后选择一种最优的半径、圆心位置作为最后的结果。

- 位置关系讨论

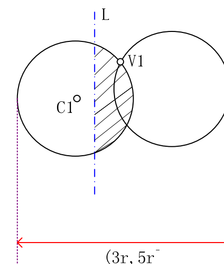
  三种位置如下：

  - $\|c_1c_2\| > 3r$



在两个圆距离很远的情况下，假设圆盘 D_1（对应圆心 c_1）在 D_2（对应圆心 c_2）的左侧。设 D_1 的最右点为 p，设 D_2 的最左点是 q，则 pq 之间的距离一定大于 r。根据这个特点，可以比较容易的找出来 p 和 q，并且根据 r 来确定圆心。

  - $r < \|c_1c_2\| \le 3r$

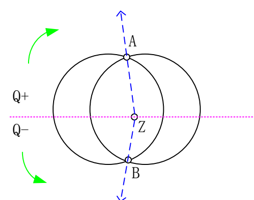


当两圆的圆心距满足(r，3r] 的时候，可以找出一条竖直线 L，把点集 S 分成左右两部分，其中左边的部分完全包含在一个圆内，我们可以找出左侧圆的圆心可能位于的一个区域，然后用右侧的点做实验，显然在阴影部分的点具有性质“以这些点为

圆心，r 为半径的圆一定和左侧的圆的圆心区域有交集”。我们就可以进一步的试验左侧圆心的位置，检验在哪个位置上，可以保证左圆盘外的点能被另一个半径为 r 的圆盘全部覆盖。只要找到了这样的圆心位置，也就能同时确定另一个元新的位置了。

- $\|c_1c_2\| \le r$

当两个圆盘重叠部分很大的时候，可以很容易地找到位于交集的点 Z，经过 Z 点的水平线把两个圆分成上下的两部分，利用 Z 点作为极坐标原点，根据其他点的极角可以对所有点进行排序，上面圆周顺时针方向，下面圆周逆时针方向，就可以排成两个队列 Q+和 Q-。如果两个圆的位置关系确实这样，就一定可以在 Q+和 Q-中找到两个点 A 和 B，使得 Q+中排在 A 前面的点和 Q-中排在 B 前面的点都是完全属于左边圆，其余的点全都属于右边圆，于是我们根据半径 r 的范围就能确定出圆心所在位置。



- 子程序模块

在前面测试的所有情况中都要调用一个基本的功能“给定点集 P 和半径 r，判断是否存在一个半径为 r 的圆盘可以覆盖 P 中的所有点”。这个功能我们使用了两种子程序实现，一个是 K(P)树，一个是 One-Center。

- One-Center

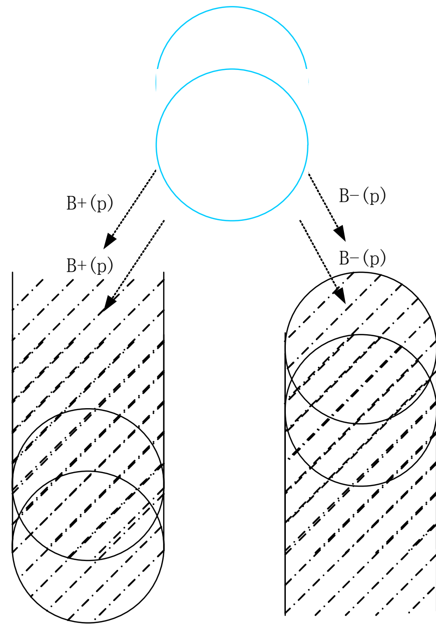
作为经典的算法，已经有论文提到了确定性的线性算法，不过实现起来比较困难，在这里我们用随机增量算法来实现。算法描述见参考文献[7]中的第四章。由于是随机算法，期望的复杂度是 O(n)，但是最坏的情况下是 $O(n^2)$。在我们的执行过程中，点集实际是排好了顺序的，所以随机增量是算法恰好是最差情况，当点数增多的时候，要比 K(P)树慢很多，可以参见后面的程序运行结果。

- K(P) tree

K(P)树的含义是，以 P 中的每一个点为圆心、以 r 为半径作圆，K(P)树的根存储的是这些圆盘的交集，如果 K(P)的根节点非空，则说明我们可以找到一个点，这个点到 P 所有点的距离都小于等于 r，等价于说 P

中的点可以被一个以 r 为半径的圆盘覆盖。

因为圆形在任何方向上都不是单调图形，为了求得若干圆的交集，我们需要利用单调曲线的性质，于是将原来的圆盘分成了两部分，$B_r^+(p)$ 与 $B_r^-(p)$。$B_r^+(p)$ 表示以 p 为圆心，以 r 为半径的圆的内部和上面的部分（试管状）；$B_r^-(p)$ 表示以 p 为圆心，以 r 为半径的圆的内部和下面的部分，见右侧图。



这样就可以将判断圆盘的交集改成判断单调图形的交集了。具体的算法参见数据结构中的说明。

数据结构

实验中用到的关键数据结构是 K(P)树，在程序中以类 CDIT2 来描述。

```
class CDIT2
{
public:
    CDIT2(void);
    CDIT2(const CDIT2& tree);
    virtual ~CDIT2(void);

    CDIT2& operator = (const CDIT2& tree);

    // member functions
    // 根据输入点集和覆盖圆半径建立起K(P)树
    void   Build(CPointArray *pa, int r, const CBoolArray& ba);
    // 修改输入点集或者覆盖圆半径，重建整棵K(P)树
    void   ReBuild(int r, const CBoolArray& ba);

    // 向当前合法点集中添加一个点
    void   AddPoint(CPoint point);
    // 从当前合法点集中删除一个点
    void   RemovePoint(CPoint point);
    // 向当前合法点集中添加一个指定下标的点
```

```
    void   AddPoint(int index);
    // 从当前合法点集中删除一个指定下标的点
    void   RemovePoint(int index);
    // 向当前合法点集中添加一系列点，由这些点的下标序列给出
    void   AddPoints(const CIntArray& ia);
    // 从当前合法点集中删除一系列点，由这些点的下标序列给出
    void   RemovePoints(const CIntArray& ia);

    // 判断整棵K(P)树的公共区域是否为空
    bool   IsEmpty();
    // 判断给定点是否属于K(P)树的公共区域内
    bool   Search(CPoint point);
    // 从K(P)树的公共区域内挑选一个点作为覆盖圆圆心返回
    bool   Query(CPoint& point, bool isLeft);
    // 获得K(P)树公共区域的所有break points
    bool   GetBreakPoints(CPointArray& breakPoints);
    // 将当前K(P)树的公共区域与给定圆求交并报告交点
    bool   Intersection(CPointArray& intPA, const CPoint& center);

    // 打印整棵K(P)树到指定文件
    void   Print(CString filename);
    // 清空K(P)树
    void   Clear();

private:
    // 拷贝一棵K(P)树
    void   Copy(const CDIT2& tree);
    // 从点集数组到K(P)树静态数组的下标转换函数
    int    LineToTree(int index);

private:
    // 指向K(P)树根节点的指针
    CDIT2Node *K_tree;
    // 指向整个点集数组的指针
    CPointArray   *ptrPA;
    // 当前覆盖圆的半径
    int radius;
    // 当前点集元素个数，即叶节点个数
    int numLeaf;
    // K(P)树的高度
    int level;
};
```

与类CDIT2相关联的是CDIT2Node类，它描述了树中节点的数据结构。

```
class CDIT2Node
{
public:
    CDIT2Node(void);
    CDIT2Node(const CDIT2Node& node);
    virtual ~CDIT2Node(void);

    // properties
    ......

    // member functions
    // 设定叶节点状态
    void    SetLeafNode(CPoint point, int r, bool avail);
    // 设定内部节点状态
    void    SetInternalNode(CDIT2Node *left, CDIT2Node *right, int r);
    // 更新叶节点状态
    void    UpdateLeafNode(int r, bool avail);
    // 更新内部节点状态
    void    UpdateInternalNode(int r);
    // 拷贝某一节点状态
    void    CopyNode(const CDIT2Node& node);

    // 判断给定点是否属于当前节点的K(P)区域内
    bool    SearchNode(CPoint point, int r);
    // 从当前节点的K(P)区域内挑选一个点作为覆盖圆圆心返回
    bool    QueryNode(CPoint& point, int r, bool isLeft);
    // 获得K(P)区域的所有break points
    bool    GetBreakPoints(CPointArray& breakPoints, int r);
    // 将当前节点的K(P)区域与给定圆求交并报告交点
    bool    Intersection(CPointArray& intPA, const CPoint& center, int
r);

    // 打印当前节点到文件流
    void    PrintNode(ofstream& fout, int level);

private:
    bool        isAvail;      // 当前节点是否合法
    bool        isEmpty;      // 当前节点所包含的K(P)区域是否为空
    bool        isLeaf;       // 当前节点是否为叶节点
    int         xRange[2];    // 当前节点所含K(P)区域在x轴上的投影区间
    // for leaf node
    CPoint      center;       // 叶节点所对应圆的圆心
    // for internal node
```

```
    CArcArray K_plus;       // K(P)区域的上包络（弧序列）
    CArcArray K_minus;      // K(P)区域的下包络（弧序列）
    // pointers
    CDIT2Node *leftChild, *rightChild;    // 指向左右子节点的指针
    CDIT2Node *parent;                    // 指向父节点的指针
};
```

复杂度分析

这个程序可以使用两种子模块解决 Two-Center 问题，下面我们先看着两个子模块的各自的计算复杂度，然后再看主程序的复杂度。

- 子模块 One-Center 的时间复杂度

实现 One-Center 功能可以有很多种算法，我们选择的是我们能够实现的最快速的算法——随机增量算法。根据参考资料[7]中的介绍，我们知道它的期望复杂度是 O(n)，最坏情况下的复杂度是 $O(n^2)$。虽然这个子模块的形式很简洁，功能也明确，但是它有一个缺点，就是不能重复利用，只要有一个点改变了，就要重新计算。

- 子模块 K(P)树的时间复杂度

K(P)树是用来判断点集 P 中的所有点是否能被某个半径为已知值 r 的圆盘所附盖，因此针对 K(P)树，我们需要三种操作：构造、更新以及查询。注意到 K(P)树实际上是一个 Minimum Height Binary Tree，也就是平衡二叉树，只不过它的叶子节点是按照一个方向从小至大排好了顺序，因此具有 n 个叶子节点的 K(P)树的高度是 log(n)。

  - K(P)树的构造

    构造一棵具有 n 个叶节点的 K(P)树，采用自底向上的顺序。由于每一个叶节点的圆弧都可能出现在最后的 K(P)边界上，所以每个内部节点的存储弧段最多为以它为根子树对应的叶节点数。确定每一层的节点存储的内容，就只须要看它的两个 children 的存储内容，然后在 log(n)时间内可计算出是否空，如果非空还可以计算出弧线的交点以及当前子树对应点集的 K(P)边界弧线。总共只有 O(n)个内部节点，所以 K(P)树的构造

时间复杂度是 O(nlogn)。

- K(P)树的更新

  对于我们的程序来讲，每次更新叶子节点，都要影响 log(n)个内部节点的存储内容，最多的时候需要更新 O(n)个节点，因此复杂度和 K(P)树的构造是一样的，都是 O(nlogn)。

- K(P)树的查询

  我们实际上需要的查询操作和论文中提到的查询不一样。论文中的查询操作是判断一个点是否在 K(P)范围内，而我们的查询需要返回 K(P)树是否为空，如果非空，还要返回一个 K(P)内部的点。我们没有找到用 $O(\log^2 n)$ 时间的算法来判断 K(P)是否为空，现在实现了用 O(n)时间判断 K(P)是否为空，类似于在两个排好了顺序的数列中查找是否有公共元素。因此 K(P)树的查询所用时间是 O(n)。

K(P)树虽然看起来很麻烦，而且在它上面的一些操作的复杂度也达到了 O(nlogn)，但是它是可以重复利用的，如果能够保证每次的改变范围较小，是可以维持原有结构不变，只更改少数节点的存储，这就节省了运算时间。在后面的实际计算中，当计算量增大的时候，能明显的看出来 K(P)树的优势。

- 外层循环

  在算法的实现过程中，我们对于圆的半径以及点集的整体旋转角度分别进行测试，这构成了最外面的两层循环。

  - 点集的旋转角度采用 10 度为间隔，总共测试旋转[0，180]度共 18 个角度，于是这层循环次数是常数。
  - 半径 r 的搜索是要估计出合适的数值，使得 Two-Center 的两圆半径尽可能小。为了能尽快找出 r，我们先找出了一个最小的矩形框（矩形的边平行于屏幕的边框），于是 r 不会大于矩形框对角线的长度 L。我们就用 L/2 作为 r 的初始值，采用 Binary Search 的方法寻找最小的满足 Two-Center 的圆半径 r，终止搜索的条件是。而 L 的大小不会超过屏幕对角线的像素个数的（常数），搜索的次数最多是 O(logL)，由于 L 小于一个常数，因此，这一层循环的次数仍然是常

数。

- 在这两层循环的外层要进行矩形框的确定，所需要的时间是 O(n)，我们只要简单的找出所有给定点的 x、y 坐标的最大、最小值就可以了。这个操作只要进行一次就可以了，不影响最后的总体复杂度。

- 循环内部：实际上就是在确定的点集、确定的旋转角度、确定的半径 r 情况下，找出是否存在两个半径为 r 的圆盘可以将所有的点覆盖。由于我们事先不知道这两个圆的位置关系是怎么样的，所以要分别测试在三种可能的位置关系下，是否有满足条件的两个圆。这三种位置关系见前面算法部分的介绍。实际上这三种位置关系对应的是三种不同的处理方法，我们可以从菜单项中指定采用哪一种处理方法。下面我们分别分析这三种处理方法对应的复杂度。

a) 使用 K(P)树作为循环的内部的调用。

- Case 1

  算法：从左到右逐个扫描，判断每一个划分下左右两部分点集是否能被两个半径为 r 的圆盘覆盖，只要存在，就可以把两个圆心的位置返回给主程序，结束 case1 的调用。

  最差情况下，需要把判断所有的情况，也就是进行 O(n)次判断，每次判断的时候要进行两次操作，一个是更新 K(P)树（有点被插入/删除），另一个是查询左右两个 K(P)树是否为空。根据 K(P)树的复杂度分析，Case 1 的整体复杂度为 $O(n^2 logn)$ 。

- Case 2

  算法：用 0.3r 作为扫描间隔，定出常数种左右分隔，也就是需要最多 O(1)条测试线。在每一种分隔线下面，依次寻找左侧圆心可能的位置（用右边的每个点作半径为 r 的圆，与左侧的 K(P)区域香蕉的交点位置以及原来 K(P)上面的 breakpoint），并一一测试，这些交点最多 O(n)个，每次的测试都是查询是否左右两侧满足 K(P)非空，所用时间是 O(nlogn)。所以 case 2 花费的总的时间是 $O(n^2 logn)$ 。

- Case3

算法：首先找出一个位于两个圆相交部分的点 z（我们采用的是外包矩形的中心点，很容易证明这个点一定是两个圆的公共点），然后经过 z 做一条水平线，以 z 为极坐标的极点，将上下两部分的点的极角算出来，并分别进行排序，要寻找一对点（分别在这条水平线的上、下），使得它们可以将点集分成两部分，左部的点完全落在左侧的圆盘内部，右部的点完全落在右侧的圆盘内部。我们要对上下的取值进行推断，最多要测试 O(n)对点，每次测试要用 O(nlogn)时间，所以 Case 3 的总体复杂度是 $O(n^2 \log n)$。

b) 使用 One-Center 作为循环的内部的调用。

- Case 1

  算法：与 K(P)树基本一样，要从左到右逐个扫描，判断每一个划分下左右两部分点集是否能被两个半径为 r 的圆盘覆盖，最差情况下，需要进行 O(n)次判断，每次判断的时候要调用两次 One-Center 子模块，判断左右划分是否能被一个圆盘覆盖。根据 One-Center 的复杂度 $O(n^2)$，我们知道 Case 1 的整体复杂度为 $O(n^3)$。

- Case 2

  算法不能用用 One-Center 实现，因为是特别针对 K(P)结构提出来的。

- Case3

  算法和 K(P)的一样，只是将左侧和右侧的测试换成 One-Center，测试次数最多 O(n)次，每次测试最多 $O(n^2)$，所以 Case 3 的整体复杂度为 $O(n^3)$。

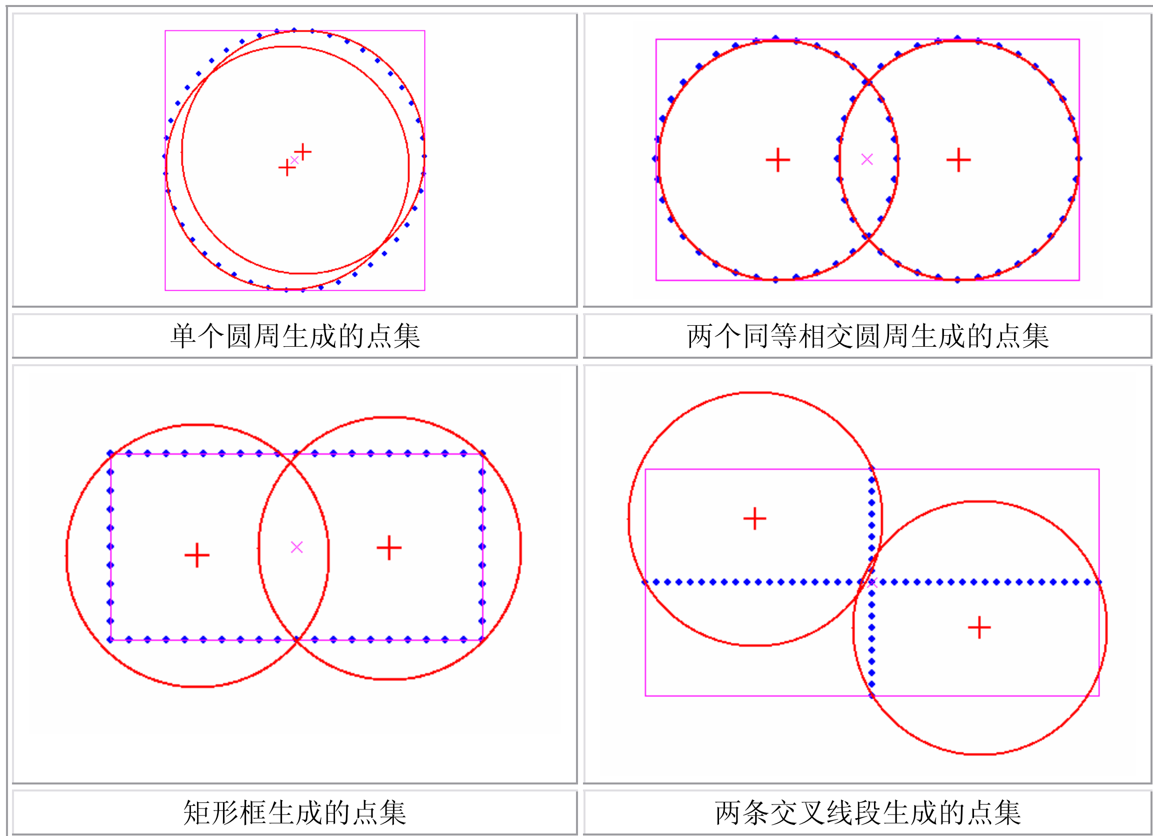
综合上面的分析，我们看出来，采用 One-Center 作为子模块调用，时间复杂度为 $O(n^3)$，而采用 K(P)树作子模块，只要 $O(n^2 \log n)$ 的时间。K(P)树在计算复杂度上比 One-Center 要好一些，但是它的复杂度超过了平方量级，还是有改进的余地的。论文中提到的复杂度是 $O(n \log^9 n)$，我们实现出来的程序的复杂度比

它高出一个 n 的量级，原因在后面的进一步的工作方向中会进行分析。

测试统计与分析

我们主要从正确性和运行效率两方面来对程序进行测试。

一方面我们准备了数个规则图形生成点集的测试样例（./test 目录），使用它们来验证算法的正确性。下面是我们用到的几个测试样例及程序结果。



单个圆周生成的点集　　两个同等相交圆周生成的点集

矩形框生成的点集　　两条交叉线段生成的点集

这些测试样例基本验证了算法的正确性。唯一美中不足的是单个圆周生成点集的测试情况。我们看到，在两个覆盖圆交点附近的部分测试点没有被覆盖住。对于这一点，我们在“误差分析”这部分内容中有更详细的讨论。

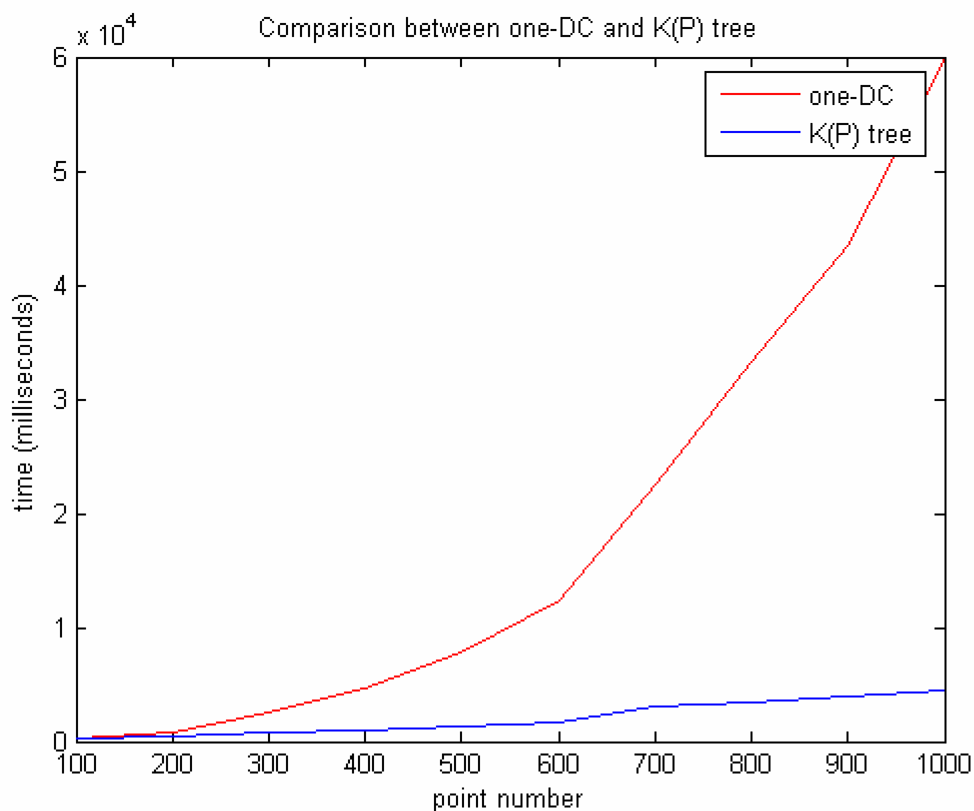
另一方面，我们随机生成指定点数的测试样例，使用大规模测试集来检查算法的效率和鲁棒性，并对两种算法进行了比较。

首先我们从 100 个点开始，每次递增 100 个点，测试了 One-Center 和 K(P) tree 两种算法在运行时间上的差异。

测试环境：Intel® Pentium® 4 CPU 2.00GHz, 512 MB memory, Windows XP

with SP2。

测试结果作图如下。

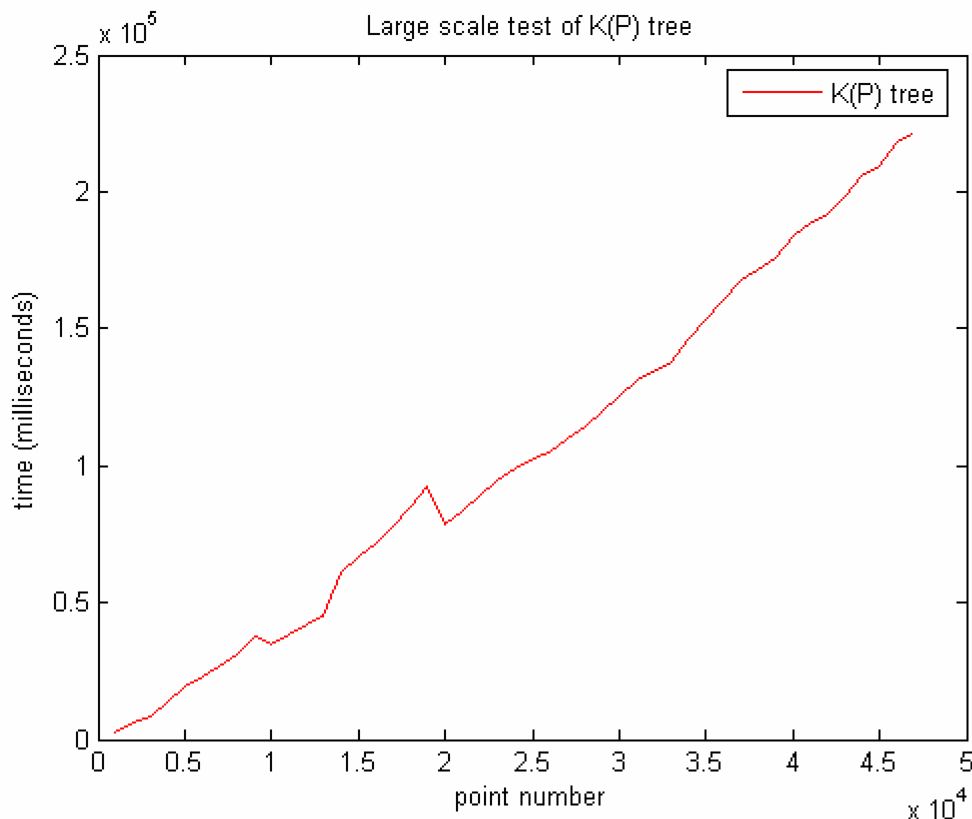


从上图我们可以看到，使用 One-Center 算法的运行时间随测试集规模增大呈指数级上升；相比之下 K(P) tree 算法的运行时间却上升得甚为缓慢，充分体现了该算法在大规模点集上的优越性。

为了更加精确的测量 K(P)树的性能，我们采用更大规模的测试集。我们从 1000 个点开始，每次递增 1000 个点，单独对 K(P) tree 算法进行极限测试。

测试环境：Intel® Xeon™ CPU 2.40GHz×2, 2.00GB memory, Windows 2003 server。

结果作图如下。



从算法运行时间与测试集规模的拟合曲线我们看到，除了在 20000 个点附近有一个稍大的波动外，曲线基本呈现出线性关系。回顾前面对算法复杂度的分析，我们认为 xxxxx 只是针对最坏情况给出的一个上界。在实际运行的例子中，对于平均情况而言，程序性能会有更大的提升。这也是算法的优势所在。

误差分析

1. 来自存储的舍入误差

由于我们把主要精力放在了算法的理解与实现上，对于数据的存储采用的是 Int 类型，这样就会带来误差。如果想减小误差，可以将 Int 换成 Double 类型，然而这样做除了提高精度，在算法的实现上并无别，但这样会使得程序运行的速度下降。一些看起来简单的数值计算步骤（涉及到的是常数，和输入点的个数无关）只会对数量级前面的系数产生影响，对复杂度没有影响，然而这些常数并不是那么小的，比如前面分析过的外层两重循环，就会带来比较大的常数，当 n 比较小的时候，这些常数的影响可能会超过数量级的影响。当采用 Double 的时

候，常数还会比 Int 的时候更大一些。

2．K(P)树的误差

作为经常调用的子程序，K(P)树的构造、查找、更新操作都要涉及到求交点（论文中提到的 breakpoint），计算两段弧线的交点坐标，就算是直接代入公式计算，也会存在计算误差的，当多次修改 K(P)树后，这种误差的积累就可能会使得最后的计算结果不能 100%的达到最优解。然而通过我们对各种情况的测试，发现这种结果误差还是很小的，如果做成实际系统，可以先用我们的程序找出合适的圆心位置关系，然后通过更精确的计算，得到两个圆的圆心的精确坐标以及半径。

进一步的工作方向

根据现有的实现结果，我们会发现，这个 project 还是有很多可以改进的地方，如果想继续完善，可以从下面两个方面入手。

✧ 改进存储精度

在现在的程序中点的存储都是采用 Integer 形式，而不是 Double 的类型。当初采用 Integer 主要是显示图像的需要（屏幕上的点的输出需要给出像素的坐标，point 的坐标必须用整数给出），但是这样也带来了显示精度不足以及后面的计算的误差的累积等问题，从测试数据来看，有些测试图形中，边缘的点没有被包括进圆中。

如果想提高精度，可以考虑在存储点的时候，采用 Double 类型，而在显示的时候定义一个阈值来确定哪一个整数点应该被显示。当然，改变点的存储方式，还牵涉到其他一些问题。

比如在我们的程序开始，要测试各种旋转角度下半径的变化情况，整个平面上的点经过坐标旋转后，新的点坐标虽然可以按照原有的计算公式得到，但是这些数仍然是浮点数，在显示的时候还要重新换算成最合适的整点坐标。

这个精度方面只要掌握合适的阈值，还是比较容易实现的。

✧ 改进算法复杂度

现有实现方法的实际计算复杂度是 $O(n^2 \log n)$，与论文中提到的算法相比较，差距在于 K(P)树的更新和查询步骤。

在查询的环节上，我们没有找到判断 K(P)是否为空的快速有效算法。虽然我们曾经发信给 Sharir 询问相关细节，但还没有收到回信。于是我们自己设计出了一个 O(n)时间的算法来实现查询，然而这需要我们依次存储 K(P)边界的各条弧，然后采用类似 zig-zag 的方法来确定是否有交点。为了保持数据结构的一致，我们在实现 K(P)树的更新时，还是沿用了这个结构，导致更新操作要用 O(nlogn)时间。

如果能找出 O(logn)的算法来实现 K(P)的查询，就可以将整体的复杂度降低。即使不能找出 K(P)树的 O(nlogn)查询算法，也可以在更新的时候采用另外的 Binary search 方法来实现，这样也可以将 2-Center 最后的复杂度降低到 $O(n^2)$。这对于提高效率是很显著的。

✧ 并行化实现与随机算法

原来的论文基本都是按照并行机来给出算法的，如果有条件，也可以真的在并行处理机上实现，这要比串行化处理节约很多时间，而且同时维护很多个 K(P)树避免了重复构造和数据复制。更多的并行化介绍可以参考[4]的最后一部份

如果想做出最新的结果，需要引入随机的思想，大家可以参考[1]，它的实现与[4]类似，对[4]的一些操作作了简化，所以复杂度只有 $O(n\log^2 n)$，这已经是目前为止最快的实现方法了，如果想获得更有效的实现，就要等待理论结果上的突破了。

参考文献

[1] David Eppstein, Faster Construction of planar Two-centers.

[2] J. Jaromczyk and M. Kowaluk, An efficient algorithm for the Euclidean two-center problem, Proc. 10th ACM Symp. On Computational Geometry, 1994, 303-311.

[3] M. Overmars and J. van Leeuwen, Maintenance of configurations in the plane, J. Comp. System Sciences 23(1981), 166-204.

[4] M. Sharir. S near-linear algorithm for the planar 2-center problem. 12th Symp. Computational Geoometry, 1996, pp. 106-112.

[5] N. Megiddo and K. Supowit, On the complexity of some common geometric

location problems, SIAM J. Comput. 13 (1984), 1182-1196.

[6] N. Megiddo, Linear-time algorithms for linear programming in R^3 and related problems, SIAM J.Comput. 12(1983), 759-776.

[7] Computational Geometry: Algorithms and Applications, by Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf, 2nd edition, Springer-Verlag, February 2000 (ISBN: 3-540-65620-0).

作者联系方式

乔　维：计研 4 班 Email：qiaow04@mails.tsinghua.edu.cn

FIT 4-505　Tel: (O) 62797701

孙　帆：计研 4 班 Email：sunfan_00@mails.tsinghua.edu.cn

FIT 4-505　Tel: (O) 62797701

赵英超：计研 4 班 Email：yczhao@mails.tsinghua.edu.cn

FIT 1-208　Tel: (O) 62797304