平面内二圆覆盖问题的近线性算法的实现

A Implement of a Near——Linear Algorithm for the Planar 2-Center Problem

概要：

M．Sharir（Israelite, from Schoool of Mathematical Sciences, Tel Aviv University, Tel Aviv, Israel）在他的一篇论文（A Near-Linear Algorithm for the Planar 2-Center Problem）中介绍了一种复杂度为$O(n\log^9 n)$的近线性算法，来解决平面上的二圆覆盖问题，即用最小半径的两个圆来覆盖平面上的 n 个点。

问题简介：

二圆问题英文为：The 2-Center Problem，设 S 为平面上 n 个点的点集，对 S 的二圆覆盖问题就是指：用半径尽可能小的的两个相等的闭圆完全覆盖 S 的所有点。二圆问题实际上是普通的 p 圆问题（The p-Center Problem）的特例。

算法简介：

在这篇论文中，Sharir 介绍了一个复杂度为$O(n\log^9 n)$的算法，据他的介绍，这是第一个把复杂度降到$O(n^2)$以内的算法。

这个算法混合使用了多个技术（techniques）：参数搜索技术（parametric searching），在单调区间搜索（searching in monotone matrices），平面结构动态保持（dynamic maintenance of planar configuration），和类似于那些常用预处理“肥”目标。

这个算法可以分成两个部分，主要的部分是对给定的半径 r，判断是否可以覆盖点集 S。在这个基础上，再使用参数搜索技术，就得到了完整的算法。

算法的策略是：假设两圆存在，圆面分别记作D_1， D_2；则我们要进行搜索找到两圆心c_1， c_2；同时圆边界记作：C_1， C_2；不失一般性，如果令两圆心距$|c_1c_2|$尽可能的小，显而易见，在C_1， C_2的凸包边界上必然有 S 中的点。

参数搜索主要体现在两方面：对半径 r 的搜索；对旋转角度 j δ 的搜索。

对半径 r 的搜索是指在一定范围内不断用算法的第一部分来逼近正确地半径 r。

δ 是一个足够小的角度，比如可以取$1°$，取 j = 0,1,..., $\lfloor 2\pi/\delta \rfloor$。因为在对给定的半径 r 进行判断时，我们做了假设：两个圆心c_1 c_2近似在一个水平线上（即 y 轴坐标近似相同），因此对给定的 r，我们要在近似所有方向上都进行判断，所谓近似所有方向就是在原来基础上分别将坐标系旋转 j δ , j = 0,1,..., $\lfloor 2\pi/\delta \rfloor$，在所有的这些方向上，必定有一个方向上$c_1$ c_2连线是近似水平的。

数据结构：

我们用到的主要数据结构是一个平衡二叉树，在介绍数据结构之前，先描述一个会反复用到的过程，同时也是我们采用平衡二叉树的原因。

在下文中我们用到的一些符号如下：

$B_r(p)$ -- 圆心为点 p，半径为 r 的闭圆面。

$B_r^+(p)$ -- 由 $B_r(p)$ 中的点及 $B_r(p)$ 上方的点组成的区域。

$B_r^-(p)$ -- 由 $B_r(p)$ 中的点及 $B_r(p)$ 下方的点组成的区域。

$\gamma(p)$ -- $B_r^+(p)$ 的边界。

同时定义：

$$K(P) = \bigcap_{p\in P} B_r(p)$$

$$K^+(P) = \bigcap_{p\in P} B_r^+(p)$$

$$K^-(P) = \bigcap_{p\in P} B_r^-(p)$$

在算法中需要动态的保持平面上的一个点集 P 中的所有点。每次插入或删除点后，都希望判断交集 $K(P)$ 是否非空。易知：$K(P)$ 是否为非空的状态与点集 P 是否能够被一个半径为 r，圆心在 p 点的圆面覆盖是等价的。

为了判断 $K(P)$ 是否非空，我们将 $K(P)$ 分成了 $K^+(P)$ 和 $K^-(P)$ 两部分分别讨论，由对称性可知，$K^+(P)$ 与 $K^-(P)$ 的研究方法是一样的。且都是 x 轴方向单调的。

于是我们针对 $K^+(P)$ 构造一棵平衡二叉树 T：

1. 将点集 S 中所有的点按照 x 坐标排序，构成所有的叶子节点。
2. 每个叶子节点中存储一个标志位，表示该点 p 是否为当前点集 P 中的点。如果点 p 是点集 P 中的点，则存储 $B_r^+(p)$ 的 x 值的范围，否则，x 值范围存储为整个 x 轴区间。
3. T 中的每个节点 v 记录着交集 $K^+(P_v)$，其中 P_v 是点集 P 的一个子集，包括的点为所有以 v 为根节点的叶子节点所对应的点。
4. 对任意一个内部点 v，设它有一个左孩子 w_l，一个右孩子 w_r，则：

   a. v 中存储了 $K^+(P_v)$ 的 x 值范围。该值就等于 $K^+(P_{w_l})$ 与 $K^+(P_{w_r})$ 的 x 的取值范围的交集。

   b. 如果 $K^+(P_v)$ 的 x 值范围非空，那么根据 $\gamma(p)$ 的性质，$K^+(P_{w_l})$ 与 $K^+(P_{w_r})$ 必有一个并且只有一个焦点 q，存储该点 q，并且存储指向对应的相交的两端弧的指针。

问题的三种情况讨论

当我们对给定半径 r，判断是否能用二圆覆盖所有点时，根据$|c_1c_2|$分三种情况讨论：

1．$|c_1c_2|$ > 3r

2．$|c_1c_2|$ < r

3．r < $|c_1c_2|$ < 3r

◇ $|c_1c_2|$ > 3r

这种情况下，一定有一条竖直线将两个闭圆面D_1和D_2分开，我们将 S 的所有点排序后，从左到右扫描，当前点记做 q，q 左边已扫描过得点包括 q 构成点集S_L,其余的部分记做S_R，动态的维护S_L和S_R，每次将扫描到的点，即S_R得最左边的点从S_R中删除，插入到S_L中。每次变化后，都检测$K(S_L)$和$K(S_R)$是否非空。如果某一个状态下，$K(S_L)$和$K(S_R)$同时非空，则表示我们找到了所求得两个圆。

◇ r < $|c_1c_2|$ < 3r

如果两圆是分离的，那么第一种情况下的算法已经可以将解求出来了。所以我们只需要分析圆相交的情况。

◇ $|c_1c_2|$ < r

这种情况下，$D_1 \bigcap D_2$的面积最少为$r^2\sqrt{3/2} \approx 0.866r^2$，又由于点集 S 所有的点可以被一个边与坐标轴平行，半径为 r 的正方形 R 覆盖，我们可以在正方形 R 中生成有限个点，其中至少有一个是位于$D_1 \bigcap D_2$中的，而且与两圆心c_1 c_2距离相近，设这个点为 z，用通过点 z 的水平线将点集 S 分成两部分Q^+，Q^-（Q^+在Q^-上面）。然后将Q^+，Q^-中的点分别按照与点 z 的连线的角度排序，Q^+中的点按照顺时针方向，Q^-中的点按照逆时针方向。

数据结构的实现

◇ 主结构（二差平衡树）的定义 BSTNode, BSTree:

```
typedef struct BSTNode : public CObject
{
    ElemType        data;               //如果是叶子节点,记录叶子节点的圆心

    float   K_max, K_min;               //记录在该叶子节点处的区间范围
//为了(K+)定义的变量
    //为交点定义的
    float   KU_mpx, KU_mpy;             //记录 K+链表在该节点的交点
    //弧的列表
    ArcList KU_arc;                     //记录该节点处的 K+链表

//为了(K-)定义的变量
    //为交点定义的
    float   KD_mpx, KD_mpy;             //记录 K-链表在该节点的交点
    //弧的列表
    ArcList KD_arc;                     //记录该节点处的 K-链表

//为了排序而定
    float angle;

    BOOL            bInSetP;            //记录叶子是否在点集 P 中
    BOOL            bIsEmpty;           //记录节点是否为空
    BOOL            bIsLeaf;            //记录是否为叶子节点
    int             bf;                 //记录平衡二叉树中的平衡因子
    struct BSTNode  *lchild, *rchild, *parent, *nextleaf, *priorleaf;
    //记录节点的左子树,右子树,父树,如果是叶子节点还记录下一个叶子节点和上一个叶子节点
}BSTNode, *BSTree;
```

◇ ElemType:

```
typedef struct ElemType
{
//为叶子节点定义的圆心
    Circle pCircle;
}ElemType;
```

◇ Circle,MyArc 和 ArcList

```
typedef struct Circle
{
    float cx, cy;
}Circle;
class MyArc : public CObject
{
public:
    Circle * m_pCircle;
    float m_fMax, m_fMin;
```

```
};
class ArcList : public CObList
{
public:
    ArcList();
    virtual ~ArcList();
};
```

算法复杂度分析