求凸包+旋转卡壳算法——求平面点集S内点对的最远距离 - CSDN博客

问题：求平面点集S内点对的最远距离解答：

其实这个问题的解法就是先求这个点集的凸包，然后运用旋转卡壳算法求此凸包的直径，便得到最远距离（即为此凸包直径）。

一． 凸包

点集Q的凸包(convex hull)：是指一个最小凸多边形，满足Q中的点或者在多边形边上或者在其内。右图中由红色线段表示的多边形就是点集Q={p0,p1,...p12}的凸包。如图1所示。

图1

平面凸包求法：**Graham's Scan法求解凸包问题**

问题：给定平面上的二维点集，求解其凸包。

过程：

1. 选取基点： 在所有点中选取y坐标最小的一点H，当作基点。如果存在多个点的y坐标都为最小值，则选取x坐标最小的一点。坐标相同的点应排除。

2. 排序：按照其它各点p和基点构成的向量与x轴的夹角进行排序，依据夹角从小到大进行排序，如果夹角相等，则按照与基点的距离从小到大进行排序。实现中无需求得夹角，只需根据向量的内积公式求出向量的模即可。

3. 去掉凹进去的点：按照排序后各点得顺序进行扫描，即逆时针扫描。基准点在最前面，设这些点为P[0]..P[n-1]；建立一个栈，初始时P[0]、P[1]、P[2]进栈，对于P[3..n-1]的每个点，若栈顶的两个点与它不构成“向左转”的关系（即栈顶点是凹进去的点），则将栈顶的点出栈，直至没有点需要出栈以后将当前点进栈。所有点处理完之后栈中保存的点就是凸包上的点了。如何判断A、B、C构成的关系不是向左转呢？如果b-a与c-a的叉乘小于0就不是。a与b的叉乘就是a.x*b.y-a.y*b.x。

复杂度：

由于在扫描凸包前要进行排序，因此时间复杂度至少为快速排序的O(nlogn)。后面的扫描过程复杂度为O(n)，因此整个算法的复杂度为O(nlogn)。

二． 旋转卡壳算法

当点集的凸包求出之后，便运用旋转卡壳算法求此凸包的直径。

有二种卡壳方法，第一种是卡住两点，如图2所示。 第二种是卡住**一条边和一个点**，如图3所示。由于实现中卡住两点的情况不好处理，所以我们通常关注第二种情况。

图2

图3

在第二种情况中，我们可以看到，**一个对踵点和对应边之间的距离比其他点要大**，也就是一个对踵点和对应边所形成的三**角形**是最大的。如图4所示。下面我们会据此得到对踵点的简化求法。

图4

```
/*********************************************************************/
/* 函数功能:卡壳简化算法                    */
/*********************************************************************/
double RotatingCalipers(vector result, int n)
{
    int j = 1;
    double maxLength = 0.0;//存储最大值
    result[n] = result[0];
    for(int i = 0; i
    {
        while(CrossProduct(result[i+1], result[j+1], result[i]) > CrossProduct(result[i+1], result[j], result[i]))
            j = (j+1)%n;
        maxLength = max(maxLength, max(Distance(result[i], result[j]), Distance(result[i+1], result[j+1])));
    }
    return maxLength;
}
```

上面就是简化的**旋转卡壳寻找对踵点**的过程。

其中叉积函数CrossProduct(A, B, C) 返回BA到CA的二维定义下的叉积。这里主要用到了叉积求三角形面积的功能。

对于判断凹凸的实现，基本都采用向量的叉积而不是去计算角度。根据右手法则，逆时针旋转为正，顺时针旋转为负，因此可用以下函数判别：

```
// 求向量p2-p1,p3-p1的叉积
int CrossProduct(const Point &pre, const Point &cur, const Point &next)//pre是上一个点，cur是当前点，next是将要选择的点
{
    int x1 = cur.x - pre.x;
    int y1 = cur.y - pre.y;
        int x2 = next.x - pre.x;
        int y2 =  next.y - pre.y;
    return (x1*y2 - y1*x2);>0是满足凸包的点
}
/**********************************************************************/
//凹凸测试
/**********************************************************************/
bool LeftTurnTest(const Point &p1,const Point &p2,const Point &p3)
{
       int product = CrossProduct(p1,p2,p3);
       if(product>0)
               return true;
       //p2点必须在p1-p3线段上面
       if(product==0)
               return (p3.x-p2.x)*(p2.x-p1.x)>0||(p3.y-p2.y)*(p2.y-p1.y)>0;
       return false;
}
```

我们对于一条对应边求出距离这条边最远的点CHj，则由上面第二种情况可知 CHi和CH j 为一对对踵点，这样让CHj绕行凸包一周即可得到所有的对踵点。如图5所示。

图5

接下来考虑，如何得到距离每条对应边的的最远点呢?

稍加分析，我们可以发现凸包上的点依次与对应边产生的距离成单峰函数。具体证明可以从凸包定义入手，可以利用反证法解决。如前面图4所示。

这样我们再找到一个点，使下一个点的距离小于当前的点时就可以停止了。而且随着对应边的旋转，最远点也只会顺着这个方向旋转，我们可以从上一次的对踵点开始继续寻找这一次的。

由于内层while循环的执行次数取决于j增加次数，j最多增加O(n)次，所以求出所有对踵点的时间复杂度为O(n)。

意外问题:

1.“倒刺”：

测试发现，当点的数量比较多的时候（>1000）就容易出现“倒刺”现象，如图6所示。

图6

“倒刺”都有一定的规律—很尖。“尖”说明向量方向问题，也就是说，当叉积为0的时候，会有两种情况：同向和对向。如图7所示。

图7

对向的时候是“尖刺”，并不满足凸包条件，同向的时候是平边，应该保留，否则最远点对会出现遗漏。

2. “包外点”：

当发现并解决了“倒刺”问题的时候，又发现了居然有些点在凸包外，如图8所示。

图8

原来在第二步“排序”的时候，如果出现角度一样的点怎么弄？如序列：B -A - C三个点，B - A向量和 C - A和x轴的夹角是一致的，那么可能出现“先长后短”和“先短后长”两种情况，前者就是“包外点”的罪魁祸首。如图9所示。所以在排序时，当出现与X轴夹角相等时，按照与基点H的距离从小到大排序。

图9

三.　算法总复杂度

Graham's Scan法求解凸包为O(nlogn)，旋转卡壳算法O(n)，所以总的复杂度为O(nlogn)。

四． C++实现代码

```
#include "stdafx.h"
#include
#include
#include
#include
#include
#include
using namespace std;
//点类
class Point
{
public:
    Point(){}
    Point(int m_x, int m_y):x(m_x),y(m_y){}
    int x;
    int y;
    friend ostream& operator<< (ostream &out, const Point &point);
};
//用来存储点集
vector vec;
/************************************************************************/
//交换二个点
/************************************************************************/
void swap(Point *p1,Point *p2)
{
        Point temp = *p1;
        *p1 = *p2;
        *p2 = temp;
}
/************************************************************************/
//函数功能：寻找基点
/************************************************************************/
int BasicPoint(vector vec)
{
        int p = 0;
        for(int i = 1;i<(int)vec.size(); ++i)
        {
                if(vec[i].y
                        p = i;
        }
        return p;
}
/************************************************************************/
//函数功能：返回二者较大值
/************************************************************************/
double max( double len1,double len2)
{
        return len1>len2?len1:len2;
}
/************************************************************************/
```

```
/* 函数功能：求两个向量的叉积  */
/****************************************************************************/
int CrossProduct(const Point &pre, const Point &cur, const Point &next)//pre是上一个点，cur是当前点，next是将要选择的点
{
    int x1 = cur.x - pre.x;
    int y1 = cur.y - pre.y;
        int x2 = next.x - pre.x;
        int y2 =  next.y - pre.y;
   return (x1*y2 - y1*x2); //>0是满足凸包的点
}
/****************************************************************************/
/* 函数功能：求两点间的距离的平方                          */
/****************************************************************************/
double Distance(const Point &point1, const Point &point2)
{
   return (point1.x - point2.x)*(point1.x - point2.x) + (point1.y - point2.y)*(point1.y - point2.y);
}
/**********************************************************************/
/* 函数功能：比较两个点，依据夹角从小到大进行排序，如果夹角相等，则按照与基点的距离从小到大进行排序。
/**********************************************************************/
bool Cmp(const Point &left, const Point &right)
{
        int product = CrossProduct(vec[0],left,right);
        if(product != 0)
                return product>0;
        double d1= Distance(left,vec[0]);
        double d2 = Distance(right,vec[0]);
        return d1
}
/**********************************************************************/
//函数功能：凹凸测试
/**********************************************************************/
bool LeftTurnTest(const Point &p1,const Point &p2,const Point &p3)
{
       int product = CrossProduct(p1,p2,p3);
       if(product>0)
               return true;
       //p2点必须在p1-p3线段上面
       if(product==0)
               return (p3.x-p2.x)*(p2.x-p1.x)>0||(p3.y-p2.y)*(p2.y-p1.y)>0;
       return false;
}
/**********************************************************************/
//函数功能：输出流重定义
/**********************************************************************/
ostream& operator<< (ostream &out, const Point &point)
{
   out<<"("<<<","<<<")";
   return out;
}
/****************************************************************************/
/* 函数功能：获取凸包
```

```
  参数vec存放输入的点，result存放凸包上的点*/
/***************************************************************************/
void GetConvexHull(vector vec, vector &result)
{
        //寻找基点,并把基点与vec[0]交换位置
        int p = BasicPoint(vec);
        swap(&vec[0],&vec[p]);
        //除基点外，对于其余的点，按照与基点的连线与x轴的夹角从小到大进行排序，
        //如果夹角相等，则按照与基点的距离从小到大进行排序
    sort(vec.begin()+1, vec.end(), Cmp);
    int size = vec.size();
    if(size < 3)
    {
        copy(vec.begin(), vec.end(), back_inserter(result));
    }
    else
    {
        result.push_back(vec.at(0));
        result.push_back(vec.at(1));
        result.push_back(vec.at(2));
        int top = 2;
        for(int i=3; i
        {
            while((top>0) && (!LeftTurnTest(result.at(top-1), result.at(top), vec.at(i))) )
            {
                result.pop_back();
                top--;
            }
            result.push_back(vec.at(i));
            top++;
        }
    }
}
/***************************************************************************/
/* 函数功能:简化的旋转卡壳算法                          */
/***************************************************************************/
double RotatingCalipers(vector result, int n)
{
    int j = 1;
    double maxLength = 0.0;//存储最大值
    result[n] = result[0];
    for(int i = 0; i
    {
        while(CrossProduct(result[i+1], result[j+1], result[i]) > CrossProduct(result[i+1], result[j], result[i]))
            j = (j+1)%n;
        maxLength = max(maxLength, max(Distance(result[i], result[j]), Distance(result[i+1], result[j+1])));
    }
    return maxLength;
}
int _tmain(int argc, _TCHAR* argv[])
{
       //产生N个随机点
```

```
    const int N = 20;
    for(int i=0; i
    {
        vec.push_back(Point(rand()%20, rand()%20));
    }
    cout<<"平面上的点："<
    copy(vec.begin(), vec.end(), ostream_iterator(cout, "\n"));
    cout<
    vector result;
        //求凸包
    GetConvexHull(vec, result);
        //打印凸包上的点
    cout<<"凸包上的点："<
    copy(result.begin(), result.end(), ostream_iterator(cout, " "));
    cout<
    double distace = RotatingCalipers(result, result.size()-1);
    cout<<"最远距离为："<<
        system("pause");
        return 0;
}
```

五． 测试

平面上的点：

(7,1)

(0,14)

(4,9)

(18,18)

(4,2)

(5,5)

(7,1)

(11,1)

(2,15)

(16,7)

(4,11)

(13,2)

(2,12)

(16,1)

(15,18)

(6,7)

(18,11)

(12,9)

(19,7)

(14,15)

凸包上的点：

(7,1) (7,1) (11,1) (16,1) (19,7) (18,18) (15,18) (2,15) (0,14) (4,2)

最远距离为：20.2485

请按任意键继续. . .