动态凸包最优算法实现及程序演示

计算几何大作业

2015-6-14

计研 142 班 范 典 2014210896

计算机系高性能所 杨 珂 2014310561

计算机系高性能所 曾开胜 2014210957

1.程序演示说明

程序设计思路

希望实现一个带界面交互的 REALTIME CONVEX HULL 算法演示程序，由用户输入待求凸包的点集，分别采用 OPTIMAL 及 DYNAMIC 算法计算凸包，并将运算的实时结果与运算过程中维护的平衡树在窗口中显示。

演示程序由前端和后端两个部分组成，前端采集用户的输入信息，通过接口调用后端程序得到反馈，并将运算过程中的每个步骤按照一定的顺序呈现在界面中。后端算法接受前端界面的调用，通过两种算法实时计算当前点集对应凸包并将重点步骤以 SNAPSHOT 的形式储存。

程序界面概览

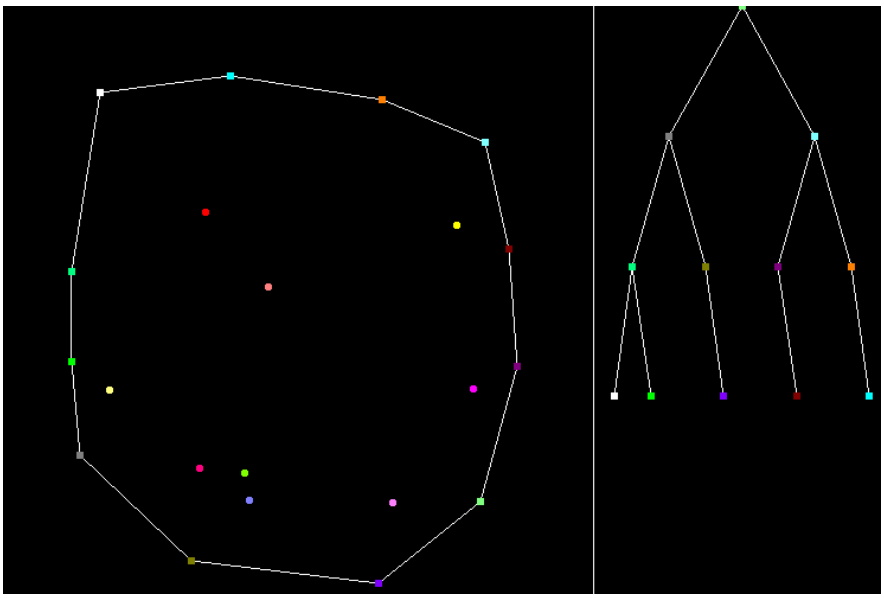


图 1 一组点集及其对应凸包与当前平衡树

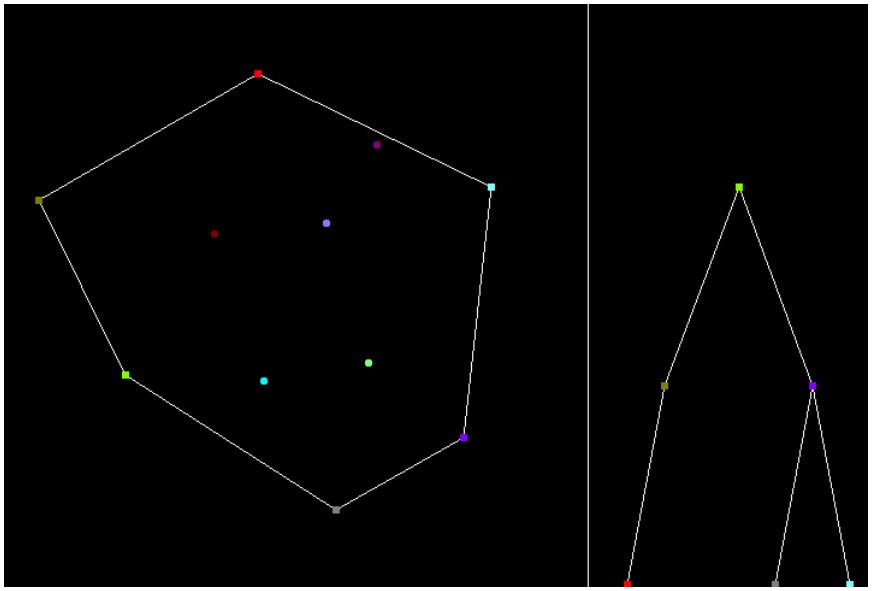


图 2 另一组点集及其对应凸包与当前平衡树

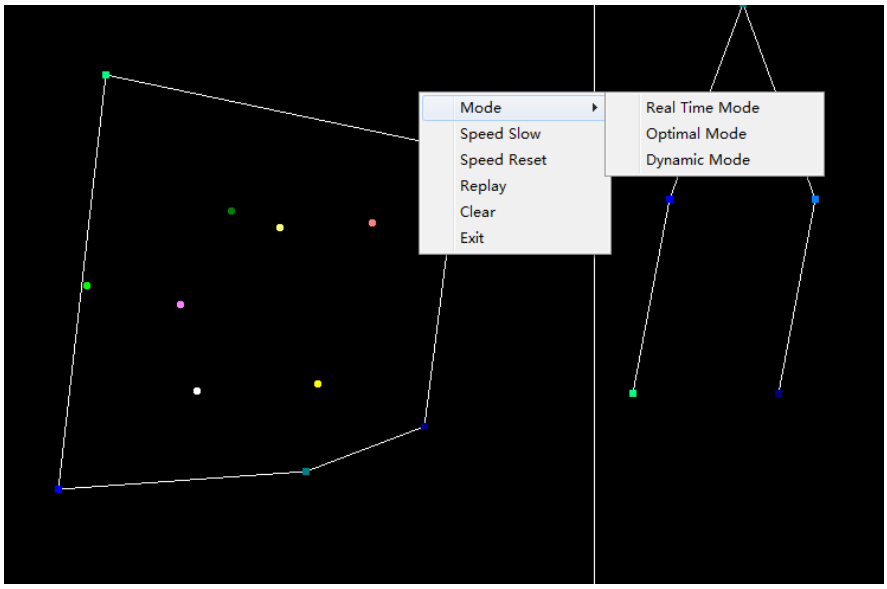


图 3 右键菜单支持功能概览

程序功能说明

前端分为左右两个部分，左半部分为用户输入区域，可以通过鼠标点击指定下一个顶点的位置。右半部分为平衡树显示区域，显示当前凸包上各顶点在平衡树上的位置。

程序提供了三种显示模式，REALTIME 模式直接显示新生成的凸包及平衡树的形状，DYNAMIC 模式和 OPTIMAL 模式逐步显示搜索过程中每个时刻的变化，三种模式可以在过程中随时相互切换。

程序还提供了一些附加功能，比如速度调节，可以更清晰的观察到搜索的具体步骤，还支持单步的重放以及整体的重绘，功能比较完整且鲁棒，可以用于凸包算法的教学演示。

特别地，演示程序实现了输入点集与平衡树显示的互不干扰。每个步骤中被检查的关键顶点予以高亮显示。每个点的颜色不同，边界点与内部点的形状不同，以此进行区分。平衡树高度可以自适应调整等。

程序后端实现了两种不同复杂度的动态凸包算法，具体内容将在第二节中进行详细的描述。

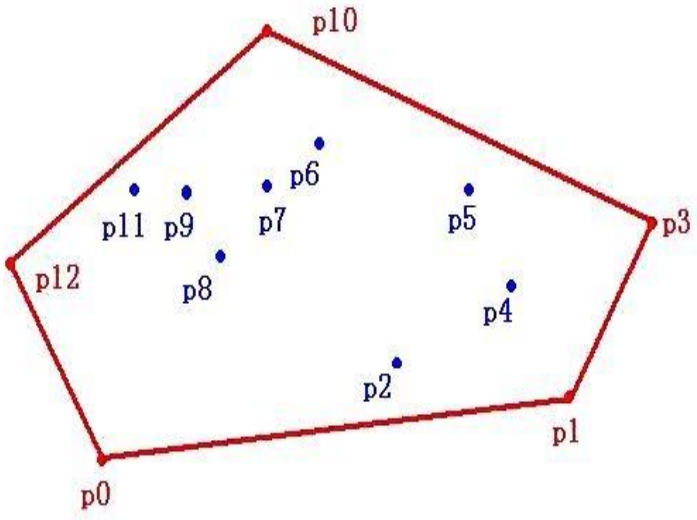
编程环境说明

演示程序使用 C++编程。大部分后端程序在 Linux 环境调试，前端显示部分在 WINDOWS 下使用 VS2012 平台配合 OPENGL3.7 完成。

2.算法原理

静态二维凸包算法

凸包（Convex Hull）是一个计算几何（图形学）中的概念。一组平面上的点，求一个包含所有点的最小的凸多边形，这就是凸包问题了。这可以形象地想成这样：在地上放置一些不可移动的木桩，用一根绳子把他们尽量紧地圈起来，并且为凸边形，这就是凸包了。在二维欧几里得空间中，凸包可想象为一条刚好包著所有点的橡皮圈。用不严谨的话来讲，给定二维平面上的点集，凸包就是将最外层的点连接起来构成的凸多边型，它能包含点集中所有点的（具体的例子如图一所示）。



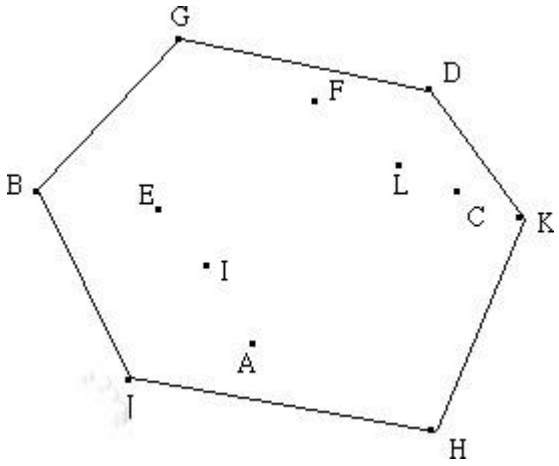
图一 凸包示例

事先给定一个点集合 P，在求出这个集合的凸包，这样的问题由于不涉及动态的点插入，所以称为静态二维凸包问题。针对这个问题现在以有成熟的算法在 O(nlgn)的时间复杂度下解决，下面将对这类算法做简单介绍。

Graham's Scan 算法

这个算法是由数学大师 Graham 提出的，算法的具体实现步骤分为以下三个部分：

（1） 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。然后按照其它各点 p 和基点构成的向量<H,p>；与 x 轴的夹角进行排序，夹角由大至小进行顺时针扫描，反之则进行逆时针扫描。实现中无需求得夹角，只需根据余弦定理求出向量夹角的余弦值即可。以图二为例，基点为 H，根据夹角由小至大排序后依次为 H，K，C，D，L，F，G，E，I，B，A，J。下面进行逆时针扫描。



图二

（2） 线段<H,K>；一定在凸包上，接着加入 C。假设线段<K,C>；也在凸包上，因为就 H，K，C 三点而言，它们的凸包就是由此三点所组成。但是接下来加入 D 时会发现，线段<K,D>；才会在凸包上，所以将线段<K,C>；排除，C 点不可能是凸包。

（3） 即当加入一点时，必须考虑到前面的线段是否会出现在凸包上。从基点开始，凸包上每条相临的线段的旋转方向应该一致，并与扫描的方向相反。如果发现新加的点使得新线段与上线段的旋转方向发生变化，则可判定上一点必然不在凸包上。实现时可用向量叉积进行判断，设新加入的点为 pn + 1，上一点为 pn，再上一点为 pn - 1。顺时针扫描时，如果向量<pn - 1,pn>；与<pn,pn + 1>；的叉积为正（逆时针扫描判断是否为负），则将上一点删除。删除过程需要回溯，将之前所有叉积符号相反的点都删除，然后将新点加入凸包。

在图二中，加入 K 点时，由于线段<H,C>要旋转到<H,K>的角度，为顺时针旋转，所以 C 点不在凸包上，应该删除，保留 K 点。接着加入 D 点，由于线段<K,D>要旋转到<H,K>的角度，为逆时针旋转，故 D 点保留。按照上述步骤进行扫描，直到点集中所有的点都遍历完成，即得到凸包。

这个算法空间复杂度为$O(1)$。但如果将凸包的结果存储到另一数组中，则可能在代码级别进行优化。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$O(nlogn)$。

Jarvis 步进法

1984 年，Jarvis 提出一个求凸包的新算法，算法的执行过程就是像在包礼物，在所有凸包算法中，这个算法似乎最容易理解。算法的具体实现分为以下步骤：

（1）找出点集中横坐标最大和最小的点，分别记做 P 和 Q，过 P、Q 的直线将平面分成左右两个部分。

（2）处理左边的点，重复以下过程：以 P 为起点，向其他“左边部分”的点做射线找一个到 X 轴负方向夹角最小的射线，将这条射线的另一端点加入凸包，将这个点记做 P。直到 Q 被加入凸包。

（3）处理右边的点，重复以下过程：以 P 为起点，向其他“左边部分”的点做射线找一个与 X 轴正方向夹角最小的射线，将这条射线的另一端点加入凸包，将这个点记做 P。直到 Q 被加入凸包。

该算法的复杂度是 O(NH)O(nk)，n 是全部的点数,k 是最终在凸包上的点数。

动态二维凸包算法

Shamos 动态凸包算法

算法的具体执行流程如下:

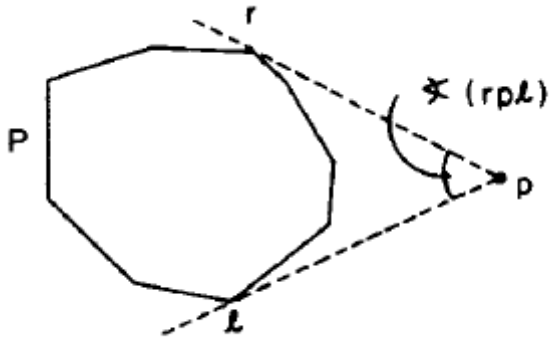
当一个新的点P_i加入点集合,设在已有点集合$\{P_0, P_1, 、、、, P_{i-1}\}$已经求出凸包$H_{i-1}$，选取基准点 O 是凸包$H_{i-1}$中的一个点，然后凸包$H_{i-1}$中的所有点都以基准点 O 来求极角序，并且按照极角序的排序放在一棵平衡二叉树上。通过在这棵平衡二叉树上可以进行查找，找到P_i的相邻的两个顶点，如果P_i点在凸包H_{i-1}的内部，那么P_i点的加入不会产生新的凸包，原有的凸包不变；如果不是这样的情况，那么我通过二分查找定位的相邻的 l,r 两个顶点将成为新凸包的关键顶点，通过P_i连接 l,r 两个顶点的切线段将构成新的凸包H_i，P_i也将成为凸包H_i的一个新的顶点。

所以对于一次更新而言，在一棵平衡二叉树上做查找的最坏情况的时间复杂度是O($(logi)^2$) = $O((logn)^2)$。

改进的动态凸包算法

这个改进的动态凸包算法和 2.2.1 所提到的算法相比，真正意义在于将一次更新的时间复杂度从$O((logn)^2)$降到了$O(logn)$。算法实现主要两个过程来实现：TANGENTS 和 RESTRUCTURE。

$TANGENTS(P,m,p)$过程的输入是准备插入的点 p，已有的凸包 P 和顶点 m=min T(P)（T(P)是存储这 P 中所有顶点的具有特殊性质的平衡二叉树）。这个过程将测试点 p 是否在凸包 P 的内部，然后确定两个顶点 l 和 r,如图一所显示的具有特殊性质的两个顶点。如要特别指出的是我们规定$\varphi(rpl)<\pi$。如果 p 在凸包 P 的内部过程将终止并且对 T(P)不做任何改变；否则在 r 和 l 之间的所有顶点将会被删除，然后在将 p 点插入 l 和 r 中间。这个功能将是有$RESTRUCTURE(P,p,l,r)$实现。



图一 关于 l 和 r 顶点的定义

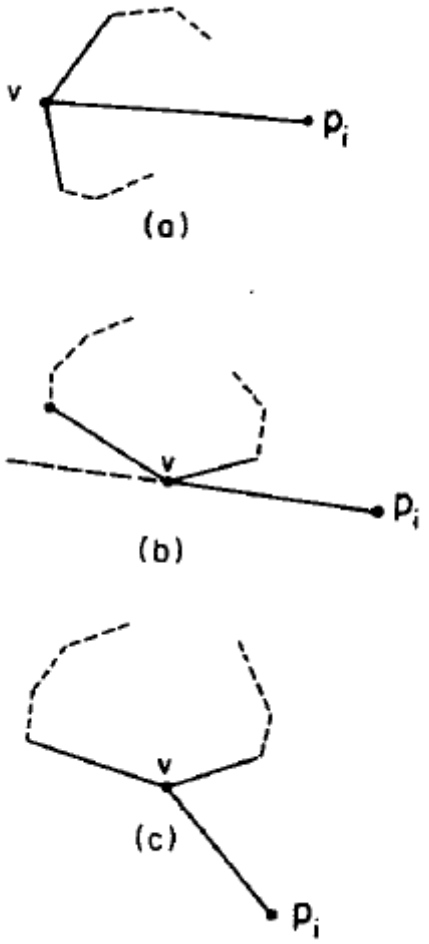
关于插入更新的伪代码如下：

CONVEX-HULL UPDATA
Input: $T(H_{i-1}),p_i$
Output: $T(H_i)$
1. Begin m:=min $T(H_{i-1})$
2. $(l,r):=TANGENTS(H_{i-1},m,p_i)$
3. if $(l,r)\neq(\emptyset,\emptyset)$ then $H_i:=RESTRUCTURE(H_{i-1},p_i,l,r)$ else $H_i:=H_{i-1}$ end

很显然，第一步是在一个具有 i 个元素的特殊平衡二叉树上作搜索，故时间复杂为$O(logi)$，下面我们将证明 TANGENTS 和 RESTRUCTURE 两个过程的时间复杂度最多也为$O(logi)$。

在 TANGENTS 过程中，我们令 $T=T(H_{i-1}),m=min(T),M=$

$ROOT(T)$;L(M),R(M)分别代表 T 的左子树和右子树。考虑一个新点p_i和凸包H_{i-1}中的一个顶点 v，我们称 v 为凹点（concave）如果线段p_iv与H_{i-1}的内部相交；称 v 为支撑点（supporting）如果 v 相邻的连个顶点都在p_iv的同侧；剩下的情况就称 v 为反射点（reflex）（如图二所示）。



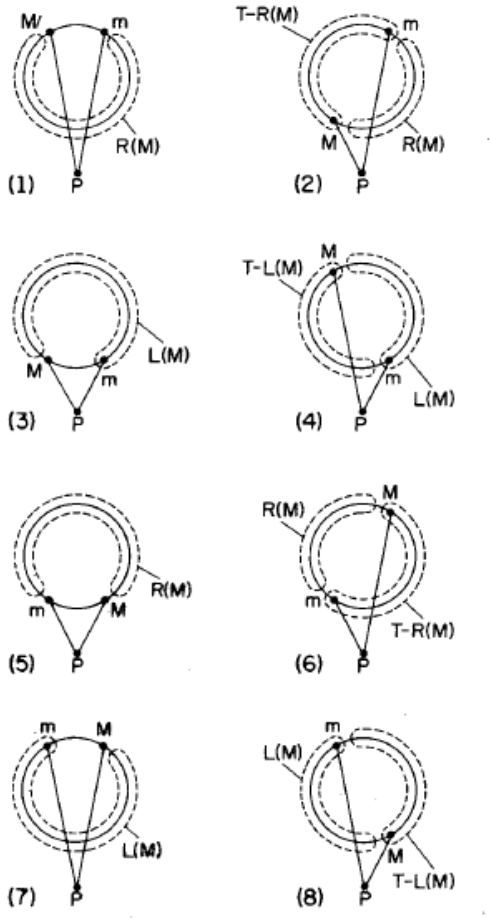
图二 凹点、支撑点、反射点的定义

我们用∝定义mp_iM的夹角：显然有前面的定义我们可以知道对于凹点而言∝≤ π，对于反射点而言∝> π，由顶点 m,M 和∝我们可以得到以下 8 种情况（如表一所示）：

Case	α	m	M
1	convex	concave	concave
2	convex	concave	nonconcave
3	convex	nonconcave	reflex
4	convex	nonconcave	nonreflex
5	reflex	reflex	reflex
6	reflex	reflex	nonreflex
7	reflex	nonreflex	concave
8	reflex	nonreflex	nonconcave

表一 m,M 和∝ 对应分类

图三将说明过程中可能出现的 8 种情况：



图三

在整个过程中首要的任务就是根据表一给出的情况分类来对给定情况进行区分，然后判断的结果来决定是递归调用自身还是通过两个简单的过程（LEFTSEARCH 和 RIGHTSEARCH）来进行处理，这两个过程将分别确定顶点 l 和 r。下面以 LEFTSEARCH 为例子简述过程执行流程。

```
Procedure LEFTSEARCH
Input: a tree T, describing a sequence of vertices.
Output: a vertex l
1.  Begin c:=ROOT(T)
2.          If pc is supporting then l:=c
3.            else begin if c is reflex then T:=L(c) else T:=R(c)
4.                        l:= LEFTSEARCH(T)
                    end
5.            Return l
    End
```

很显然 LEFTSEARCH 在树 T 上探索了一条路径，并且 T 是棵拥有 i-1 个顶点的平衡二叉树，这个过程的时间复杂度为$O(logi)$。就 TANGENTS 过程的时间复杂度而言，关于 8 种情况的判断将会花费一定常数级别的时间，所以执行时间的最终将决定于过程自己的递归调用和 LEFTSEARCH,RIGHTSEARCH 两个过程的调用。过程 TANGENTS 将从根节点到$T(H_{i-1})$的某一个节点 c 确定一条路径，通过递归调用其自身。如果p_i是在H_{i-1}的内部那么 c 点将是 $T(H_{i-1})$的叶子节点，否者将从 c 点出发继续调用 LEFTSEARCH,RIGHTSEARCH 直到 l,r 点确立。TANGENTS 过程的执行时间决定于树 $T(H_{i-1})$的深度，故时间复杂度为$O(logi)$。

最后我们考虑过程 RESTRUCTURE 的时间复杂度，设n_{i-1}是H_{i-1}的顶点数，由于 l,r 之间的顶点必需删除，然后插入新的点p_i。考虑到删除操作，这个过程因为 l 与 r 点在 $T(H_{i-1})$中的顺序而有所不同，在第一种情况下，我们需要在平衡二叉树上进行两次分裂和一次接合操作，在第二种情况下只需要两次分裂操作。我们都知道在平衡二叉树上进行分裂和接合都是标准的操作，需要的时间复杂度为$O(logi)$，所以整个过程的时间复杂度为$O(logi)$。

综上所述，整个 CONVEX-HULL UPDATE 的时间复杂度为$O(logi)$。

3.算法实现

这个算法本身不算复杂，但是实际实现起来编程的工作量还是很大的。虽然如此，即使代码量大，边界情况复杂，然而代码只是这个算法的另一种表现形式，在对算法已经做了详尽分析过后，再来描述代码意义不大。代码的具体实现放在 src/test/geometry.cpp 里了，这里就只是简要描述一下在算法的实际实现中遇到

的问题：

平衡树的选取：论文中选用了 AVL 作为平衡树，然而这篇论文是 1979 年的，在那之后更多种类的平衡树被提了出来，从而给予我们更多的选择。经过比较，我们最终选用了 size balanced tree（SBT）作为我们的实现。SBT 的效率不弱于 AVL，但是其实现更加简单，而这一特点在此极为重要，因为我们不能只是简单的套用平衡树的标准实现，而是要在上面做很多修改。

平衡树的修改-前驱和后继：由于我们在判断一个点是 concave，reflect 还是 supporting 时，需要同时用到这个点、它的前驱以及它的后继共 3 个点；同时在算法中求切点时对于 m 和 M 点的维护还需要 move next 的操作。所以这个数据结构需要能求其前驱和后继，平衡树的标准实现中这两个操作是 logn 的复杂度，然而这个算法中共有 logn 次这样的操作，这就要求我们把这两个操作的复杂度维护到 O(1)。这里我们是通过给节点增加了两个指针域分别指向其前驱和后继来实现的。

平衡树的修改-范围删除：在动态求凸包时，在一次插入后，最多可能删除 n-2 个点。标准的平衡树中，删除一个点的复杂度是 logn，虽然直接用标准实现，均摊复杂度也是 logn 的（因为每个点最多只会被删除一次），但是就做不到实时的 logn 了。考虑到实际在删除时，虽然一次删除的点的数目可能较多，但是这些点最多分布在两个连续的区间中，为此，我们实现了 logn 的范围删除操作。在删除一个区间[l, r]中的点时，如果当前子树被整个包含在此区间中，我们 O(1)的将其整个的删除，可以证明（类似于线段树的复杂度的证明），不被整个包含的子树的个数是 logn 的，所以复杂度是 logn 的。

论文中没有考虑到的情况：论文中求切线的算法中，在 LSearch 和 RSearch 之前，是同时维护了 m 和 M 两个指针，指向平衡树中的两个点，其中 m 表示当前子树中的第一个点，M 表示当前子树的根。论文中，当前子树的节点数小于等于 1 时，表示搜索失败，新加入的点在凸包内，不需要更新凸包。否则，需要对 m 和 M 指向的点作判断，此时算法假设了 m 和 M 是不同的两个点。但是，考虑这样一种情况，假设 x 点是子树的根，x 的左儿子为空，右儿子不为。此时子树节点个数大于 1，然而 m 和 M 依然表示的是同一个节点，这样会导致算法出错，而简单的将 M 指向其下一个节点，虽然实现容易，而且含义正确，但是当 M 不

再是当前子树的根时，接下来的算法都没法再正确执行了（回顾一下，在论文提出的算法中，M 每次只能移向其左子树，或者右子树）。我们对此的处理是，当出现这种情况时，对子树做一次旋转，使得旋转后的子树的根拥有左儿子。旋转的复杂度是 O(1)的，因此不影响整体负责度。

边界条件：对于三点的共线的情况，需要非常复杂的特殊判断才能处理，为了避免这些特判，我们对输入的点加入了微小的扰动，使其在用 double 存储时，几乎不可能出现 3 点共线的情况。

4.性能测试和结果分析

在进行测试数据的生成时，我们发现随机生成的数据，即使点的数目很多，但是其凸包上的点的个数依然很少，这样的数据对于测试意义不大。于是我们对此进行了改进，将生成的点集控制在半径为 3 万的一个圆周附近，这样随着点数的增多，凸包上的点的个数增加相对较快。我们生成了点集大小为 100、1000、10000、100000、1000000 的 5 种数据对程序进行测试，针对每个种类我们生成了多组测试数据作为程序的输入。

由于程序运行的时间依赖于具体的机器的性能，因此此处我们只用关心相对的效率。作为对比，我们对如下 3 个程序进行了测试：

1. Logn 的在线实时算法，也就是论文里的那个算法
2. Loglogn 的在线算法，采用了二分查找和平衡树的实现
3. Gram 的经典的离线算法，总的复杂度为 nlogn

测试结果如下表所示：

表 3-1 程序效率测试结果对比

输入点数	凸包上的点数	Gram 静态算法(ms)	Logn*Logn 的算法(ms)	Logn 的算法(ms)
100	92	472	1893	1150
1000	340	5803	27371	10552
10000	756	54483	167253	44723
100000	1226	284693	1002537	359412
1000000	1253	2293686	10658961	4106911

从上表我们可以看出，Gram 的算法是最快的，Logn 的算法则相对慢了一倍，

而 Logn*Logn 的算法则相对于 Logn 的算法又慢了 1 到 4 倍。Gram 算法的快速是合理的，因为将数据进行实时的处理肯定是需要付出一定代价的。但是当点数增多时，相对于 Logn*Logn 的算法，我们优化掉的那个 logn 并没有完全体现出来，否则当点数为 1000 的时候，前者应该慢上 10 倍才对。之所以会有这样的结果，是因为优化掉的 logn 主要体现在平衡二叉树的访问上，而几何部分的运算，并没有因此减少，同时现对于平衡二叉树的访问主要由指针的运算而言，几何部分的大量的 double 类型的乘法所消耗的常数是相对较大的，因此，在这样的分析下，我们可以得出结论，这样的测试结果是符合预期的。

测试数据的生成文件 gen.cpp 和 gen2.cpp 放在 src/test 目录下，其中 gen.cpp 是生成随机点集，而 gen2.cpp 是生成圆周分布的点集。测试所用的以上 3 个算法的实现在 src/test/geometry.cpp 中

参考文献

1. Graham, R.L. An efficient algorithm for determining the convex hull of a finite planar set. Inform. Processing Letters 1 (1972), 132-133.

2. Jarvis, R.A. On the identification of the convex hull of a finite set of points in the plane. Inform. Processing Letters 2 (1973), 18-21.

3. Knuth, D.E. The Art of Computer Programming. Vol. 3: Sorting and Searching. Addison-Wesley, Reading, Mass., 1973.

4. Preparata, F.P., and Hong, SJ. Convex hulls of finite sets in two and three dimensions. Comm. ACM 20, 2 (Feb. 1977), 87-93.

5. Shamos, M.I. Problems in computational geometry. Dept. of Comptr. Sci., Yale U., New Haven, Conn., May 1975.

6. Shamos, M.I. Geometric complexity. Proc. Seventh Annual ACM Symp. on Theory of Computing, May 1975, pp. 224-233.

7. Shamos, M.1. Computational geometry. Dept. Comptr. Sci., Yale

杨珂 曾开胜

Semi-dynamic Polygon

计算机系
曾开胜 范典 杨珂
指导老师：邓俊辉

Background

- Semi-dynamic
- Online
- Real time
- Achieve optimal time-bounds

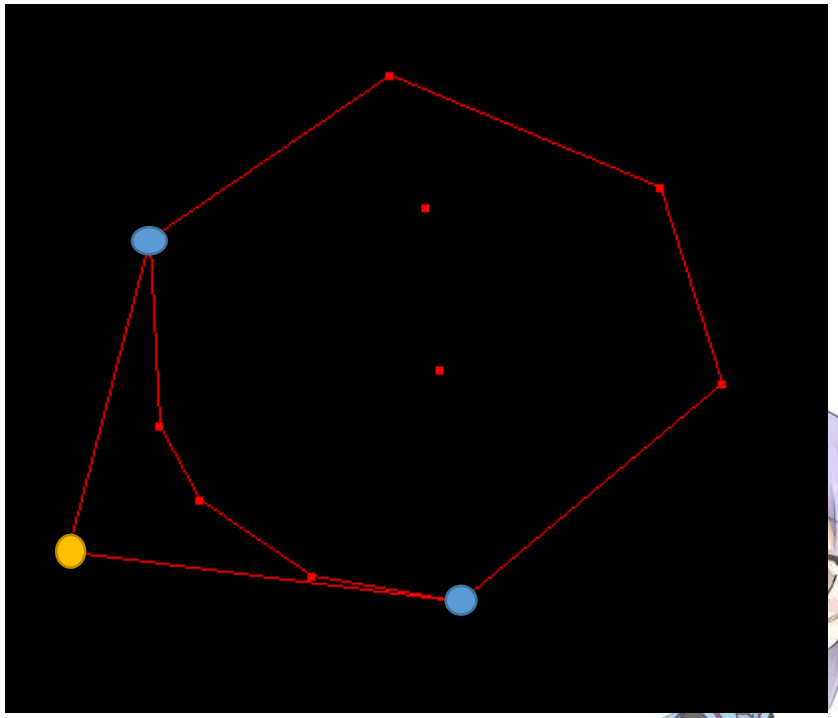
History

- Static： n^2->nlogn->nlogh
- Dynamic : semi-dynamic ->full-dynamic
- Finite precision : logn /(loglogn)

杨珂 曾开胜

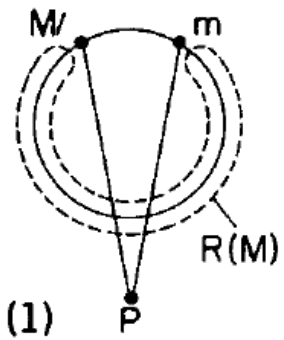
A log(n)*log(n) algorithm

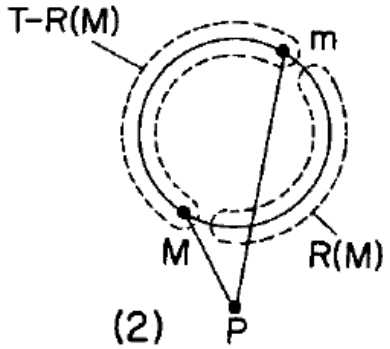
- Balanced tree
- Binary search

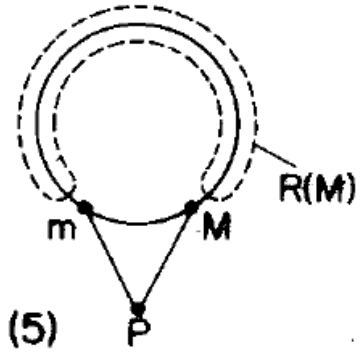


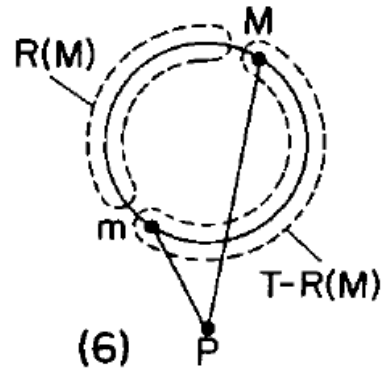
杨珂 曾开胜

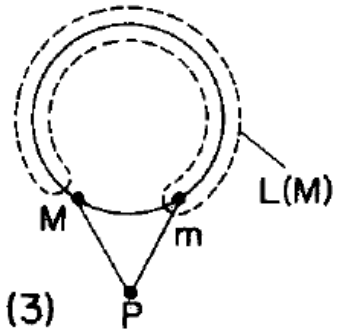
Achieve log(n)

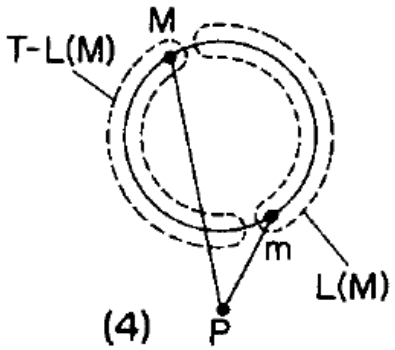


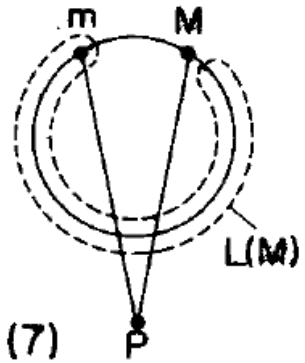


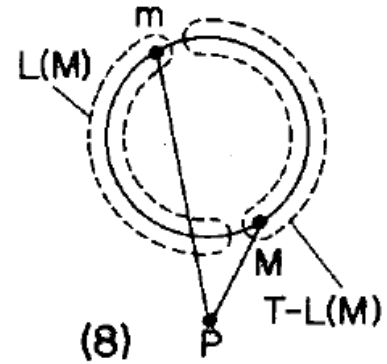












杨珂 曾开胜

Achieve log(n)

```
Procedure LEFTSEARCH
    c<- ROOT(T)
    If pc is supporting
        return c
    else
        If c is reflex
            T <- L(c)
        else
            T <- R(c)
        return LEFTSEARCH(T)
```

Test

- C++
- Demo
- Compare with Gram Algorithm

Reference：

F. P. Preparata. An Optimal real time algorithm for planar convex hulls. Communications of the ACM, 22: 402-405, 1979


Thanks !
计算机系
曾开胜 范典 杨珂
指导老师：邓俊辉