深 圳 大 学 实 验 报 告

课程名称：算法设计与分析

实验名称：分治法求最近点对问题

学院：计算机与软件学院 专业：软件工程

报告人：郑杨 学号：2020151002 班级：腾班

同组人：陈敏涵

指导教师：李炎然

实验时间：2022.4.3~2022.4.11

实验报告提交时间：2022.4.11

教务处制

一．实验目的

1. 掌握分治法思想。
2. 学会最近点对问题求解方法。

二．实验内容

1. 对于平面上给定的N个点，给出所有点对的最短距离，即，输入是平面上的N个点，输出是N点中具有最短距离的两点。
2. 要求随机生成N个点的平面坐标，应用蛮力法编程计算出所有点对的最短距离。
3. 要求随机生成N个点的平面坐标，应用分治法编程计算出所有点对的最短距离。
4. 分别对N=100000~1000000，统计算法运行时间，比较理论效率与实测效率的差异，同时对蛮力法和分治法的算法效率进行分析和比较。
5. 如果能将算法执行过程利用图形界面输出，可获加分。

三．实验步骤与结果

1. 蛮力法解决平面最近点对距离问题

算法思想

蛮力法求解这个问题的思想十分的简单，就是直接枚举所有点对，计算距离之后更新当前最小答案即可。

伪代码

$\boldsymbol{BRUTE\_FORCE(Points, n)}$

$ans = INF$

$for\ i = 1\ to\ n$

$for\ j = i + 1\ to\ n$

$ans = \min(ans, dist(Points[i], Points[j]))$

算法流程

蛮力法的算法流程就如思想所述，把每个点附上一个编号，如图 1 所示。首先把最终的答案初始化为正无穷，然后枚举点对，在枚举点 1 的时候，需要与点 2、3、4、5 计算距离并更新答案（距离计算公式为$dist(i,j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$），假设答案更新为$dist(1,3)$；在枚举点 2 的时候，需要与点 3，4，5 计算距离并更新答案，假设答案没有更新。其他的点依次类推，枚举完剩下的点对之后得到最终的答案。**在实际代码实现中，可以把距离计算公式中的根号去掉，这样不会影响最终结果，而且可以优化常数，当然最终算出来的最近点对距离需要开根号。**

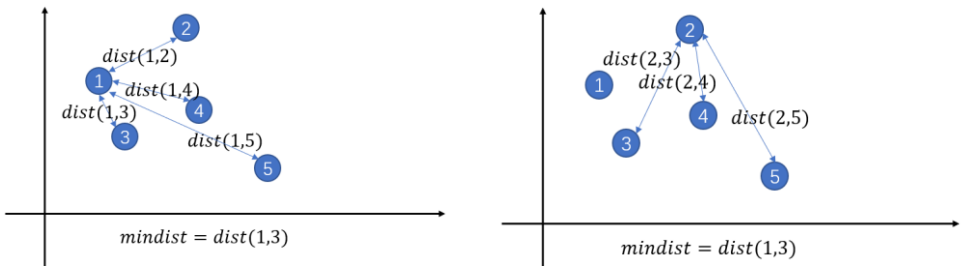


图 1 蛮力法示意图

复杂度分析

i. 时间复杂度

本人还是使用伪代码来进行时间复杂度分析：

$\boldsymbol{BRUTE\_FORCE(Points, n)}$	$\boldsymbol{cost}$	$\boldsymbol{time}$
$ans = INF$	c_1	1
$for\ i = 1\ to\ n$	c_2	$n+1$
$for\ j = i+1\ to\ n$	c_3	$\sum_{i=1}^{n} n-i+1$
$ans = \min(ans, dist(Points[i], Points[j]))$	c_4	$\sum_{i=1}^{n} n-i$

$Then$：

$$T(n) = c_1 + c_2(n+1) + c_3(\sum_{i=1}^{n} n-i+1) + c_4\sum_{i=1}^{n} n-i$$

$$= c_1 + c_2 n + c_2 + c_3\frac{n(n+1)}{2} + c_4\frac{n(n-1)}{2}$$

$$= \frac{1}{2}(c_3+c_4)n^2 + (c_2 + \frac{1}{2}c_3 - \frac{1}{2}c_4)n + c_1 + c_2$$

所以蛮力法解决平面最近点对问题的时间复杂度为：$O(n^2)$。

ii. 空间复杂度

蛮力法没有使用到额外空间，故空间复杂度为：$O(1)$。

2. 分治法解决平面最近点对距离问题

算法思想

从蛮力法中可以看到，每个点对都需要被枚举以求得最终答案。而实际上，一些距离明显比当前答案大的点对，不需要被枚举也不会对最终答案照成影响。分治法在合并过程中，利用点对满足的一些特殊数学性质，把整个枚举的区间缩小，以达到优化的目的。分治法解决这个问题的基本思想就是把整个平面分为两个均等的半平面，然后递归的求解两个半平面的最近点对距离，之后再求解跨越两个半平面的最近点对距离（如图 2），把三个距离取个最小值就可以得到当前平面的最近距离了。**所以算法思想分为两部分，一部分是递归求解左右半平面各自的最近点对距离，一部分是求解跨越两平面的最近点对距离。**具体算法流程见算法流程部分。

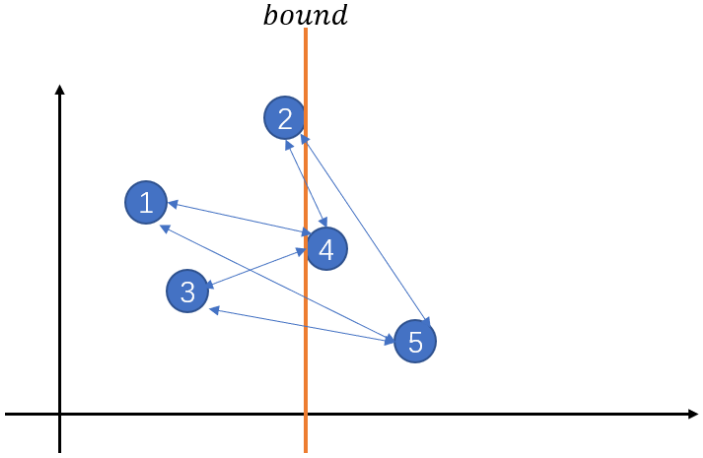


图 2 分治法算法思想示意图

伪代码

$\boldsymbol{PLANE\_CLOSEST\_POINT\_PAIR(Points, l, r)}$

$mindist = INF$

$if\ l == r\ return\ mindist$

$if\ l == r - 1\ return\ dist(l, r)$

$if\ l == r - 2\ return \min(dist(l, r), dist(l + 1, r), dist(l, l + 1))$

$mid = (l + r)/2$

$d1 = PLANE\_CLOSEST\_POINT\_PAIR(Points, l, mid)$

$d2 = PLANE\_CLOSEST\_POINT\_PAIR(Points, mid + 1, r)$

$d3 = Solve(Points, l, r)$

$mindist = \min(d1, d2, d3)$

算法流程

这一部分给出分治法求解平面最近点对的算法流程，对于算法流程中需要证明的内容做出**标注**（如[1]）并在下一部分**‘算法正确性证明’**中加以证明。

首先定义一些变量，把平面中所有点组成的点序列记为$Points[]$，当前序列左端点记为l，序列右端点记为r，当前序列长度为$r - l + 1$，记为n。
那么规模为n的原始问题就是求出点序列$P[]$处于区间$[1, n]$的最近点对距离。

- 把$Points[1, .., n]$按照x坐标从小到大排序[1]，然后进入函数 $PLANE_CLOSEST_POINT_PAIR(Points, 1, n)$中求解。

接下来对函数中的规模n分情况讨论解决策略。

- 如果$n = 1$，函数返回INF，因为此时问题中只有一个点，没有构成点对，因此不对最终答案产生影响，返回正无穷以避免对答案照成影响。
- 如果$n = 3$，暴力枚举 3 对点对距离，算出最小距离之后返回。
- 如果$n \geq 3$，函数把$Points[l, \dots, r]$分为两个点序列$Points[l, \dots, mid]$和$Points[mid + 1, \dots, r]$，其中$mid = (l + r)/2$。
  - 求解子问题$d1 = PLANE_CLOSEST_POINT_PAIR(Points, l, mid)$
  - 求解子问题$d2 = PLANE_CLOSEST_POINT_PAIR(Points, mid + 1, r)$
  - 记$d = \min(d1, d2)$，取出横坐标x在$[Points[mid].x - d, Points[mid].x + d]$[2]范围内的所有点构成点集$P[1, \dots, k]$（其中$k$是点集$P$的大小），并把点集$P$中的点按照纵坐标$y$从小到大排序[3]。
  - 按照纵坐标从小到大枚举点集P，对于当前枚举的点为P_i，枚举处于P_i之后且处于点集P中的点P_j（满足$j > i$并且$j \leq k$并且$P_j.y - P_i.y < d$[4]），使用$dist(P_i, P_j)$更新答案。可以证明P_j的个数不会超过五个[5]。

算法流程到此结束，接下来用一个**实际例子**更加具体地叙述流程。
假设有以下例子，平面上有 10 个点构成点集$Points[1 \dots 10]$，首先先按照x坐标给点排好序，如图 3 所示，然后进入函数$PLANE_CLOSEST_POINT_PAIR(Points, 1, 10)$求解。

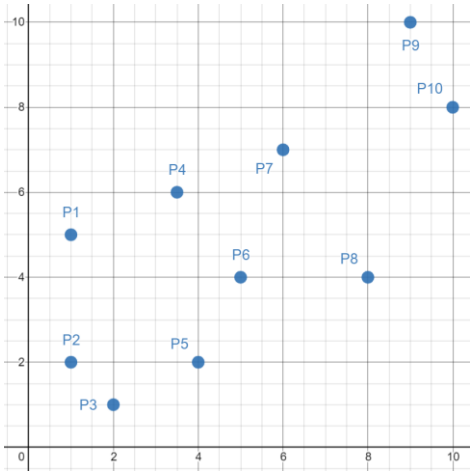


图 3 分治法流程示意图（1）

由于$n=8>3$，所以把点集$Points[1\ldots10]$划分为两部分：$Points[1\ldots5]$和$Points[6\ldots10]$，递归调用函数$PLANE_CLOSEST_POINT_PAIR(Points,1,5)$求出$d1$，并且调用函数$PLANE_CLOSEST_POINT_PAIR(Points,6,10)$求出$d2$，如图 4 所示。

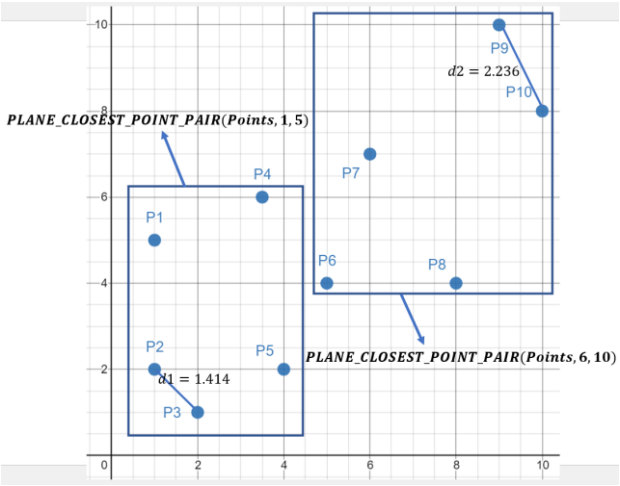


图 4 分治法流程示意图（2）

那么此时$d=\min(d1,d2)=\min(1.414,2.236)=1.414$，此时需要取出横坐标在$[Points[mid].x-d,Points[mid].x+d]$范围内，由于此时的$mid=5$，所以这个范围也就是$[4-1.414,4+1.414]$即$[2.586,5.414]$，画出这个范围，如图 5 所示。

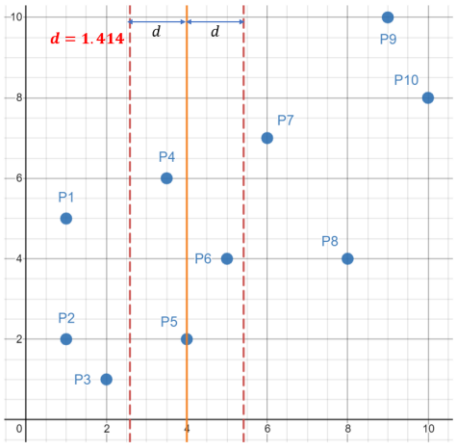


图 5 分治法流程示意图（3）

接下来只关注这个范围内的点，首先把这些点构成的点集$P[1\ldots3]$按照纵坐标从小到大排序，排序完之后的点集$P[1\ldots3]=\{P5,P6,P4\}$。然后再对排序完的点集从头到尾遍历，对于$P5$，只需要往后遍历到$P6$，因为$P6$与$P5$的y坐标距离已经超过d了，如图 6 所示。剩下两个点的枚举也类似。最终$d=1.414$，函数返回，算法结束。

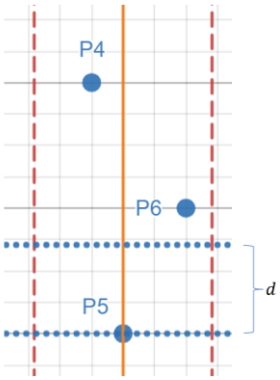


图 6 分治法算法流程示意图（4）

算法正确性证明

接下来对算法流程中的一些做法进行正确性说明。

[1]在算法开头对所有点按照x坐标从小到大进行排序

正确性说明：按照x坐标排序本质上是把所有点的x坐标与点数组下标对齐，以便于快速地把整个点集均分为两个点集，就是直接以数组中点所处位置为划分界限即可。而如果没有排序的话，如图 7 所示，若此时直接以数组中点$P5$所在位置为划分界限的话，就会导致划分不均匀。

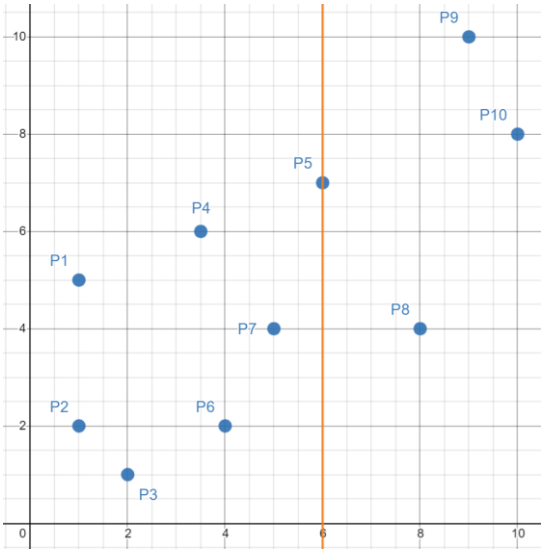


图 7 算法正确性说明示意图（1）

[2] 在递归求解完两个子问题之后，取出横坐标在下面范围中的点作为备选点集合P：

$$[Points[mid].x-d,Points[mid].x+d]$$

正确性说明：求解完两个子问题之后，可以知道当前的最近点对距离为$d=\min(d1,d2)$，因此在考虑**跨越分界线的点对**时，无需再枚举所有点对，分界线两边与分界线距离大于d的那些点我们无需再考虑。只需要考虑在上述范围内的点即可。

[3] 把备选点集P按照纵坐标排序 **[4] 按照纵坐标从小到大枚举点集P，对于当前枚举**

的点为P_i，枚举处于P_i之后且处于点集P中的点P_j（满足$j > i$并且$j \le k$并且$P_j.y - P_i.y < d$[4]）[5]**可以证明P_j的个数不会超过五个**

正确性说明：这三点我觉得放在一起说比较好。这里主要讨论的是如何快速计算备选点集中的最近点对距离并更新答案。类似横坐标缩小枚举范围的思想，考虑纵坐标是否也可以缩小范围，答案是肯定的。对于备选点集中的任意两点，如果要对当前的答案d造成影响的话，**纵坐标方向的距离必须小于d**，因为如果大于等于d的话，根据勾股定理可知，两点的距离必然大于等于d，必然不会更新当前答案。那么对于某个处于点集P中的某个备选点P_i而言，所需要考虑的区域如图 8 阴影部分所示：

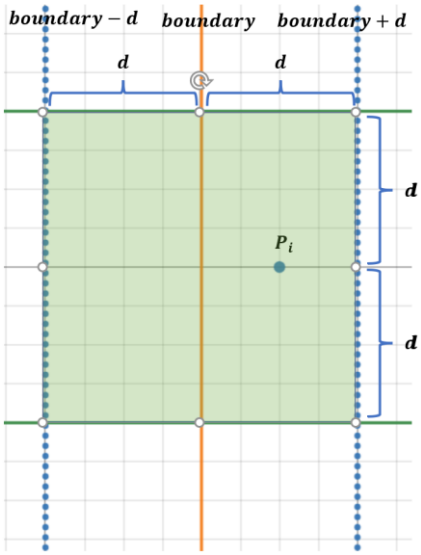


图 8 算法正确性说明示意图（2）

但是如果对于点集P中的每一个点都直接按照上述范围去考虑的话，需要枚举其他所有点判断是否位于这一范围，这样一来时间复杂度就变成$O(n^2)$了，使整个算法失去了意义。可以发现，如果直接枚举的话，会出现如图 9 所示的重复枚举情况，在考虑P_1时会枚举到P_2，在考虑P_2时会枚举到P_1。

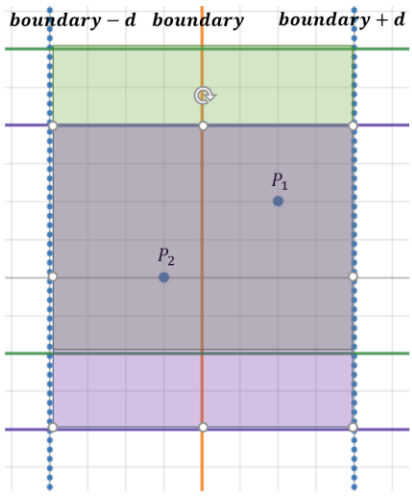


图 9 算法正确性说明示意图（3）

那么其实可以根据纵坐标对点集P进行排序，然后按照纵坐标从小到大枚举备选点，对于某个处于点集P中的某个备选点P_i而言，所需要考虑的区域转化为之前区域的上半部分（如图 10），因为纵坐标有序所以下半部分在之前已经枚举过了。同时，由于纵坐标

有序，区域内的点就是数组中排列于P_i之后的几个点，所以可以按照数组顺序枚举，一旦遇到纵坐标超过区域的点，就可以停止枚举了。

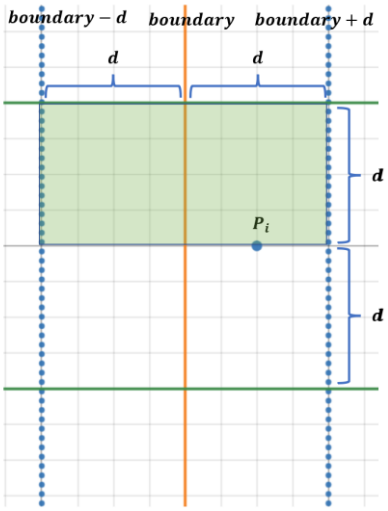


图 10 算法正确性说明示意图（4）

接下来需要证明的是，对于区域内点的枚举时间花费为常数时间花费，只有这个结论存在这个算法才有意义。由于上述区域中边界的左右两部分的点分布是独立的，互不影响，所以可以把证明的问题转化为，在一个$d \times d$区域内最多有多少个点，满足点与点之间的距离大于等于d。如图 11 所示，在极限情况下，最多可以画出 3 个满足条件的点，故在P_i所需要考虑的区域中，最多只存在 6 个点（包括P_i），所以对于某个处于点集$\boldsymbol{P}$中的某个备选点P_i而言，最多需要往后枚举五个点，为常数时间。

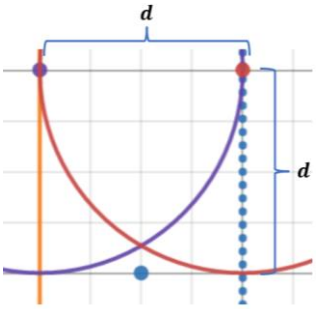


图 11 算法正确性说明示意图（5）

那么还存在一个问题就是，如何快速地根据纵坐标对点集$\boldsymbol{P}$进行排序，由于算法是分治法，不难想到可以利用归并排序在分治之后进行合并，可以做到时间复杂度为$O(n)$的排序。

算法正确性验证

正确性验证方面，本人采用与蛮力法运行结果对比进行验证，如图 12 所示：

```
void test_divide()
{
    float d1, d2;
    printf("BRUTE_FORCE: %.5f\n", (d1 = brute_force()));
    printf("DIVIDE_AND_CONQUER: %.5f\n", (d2 = divide_and_conquer()));
    if (d1 == d2) puts("RIGHT!!\n");
    else puts("WRONG~~~\n");
}
```

图 12 算法正确性验证（1）

使用了数据规模为 10000 的 20 组数据进行验证，均通过验证，如图 13 所示：

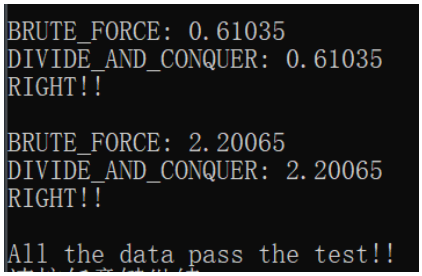


图 13 算法正确性验证（2）

复杂度分析

i.时间复杂度

分治算法的时间复杂度分析，本人采用递推式+递归树来进行分析。

首先由伪代码推导分治算法的递推式：

假设分治算法的实际运行时间为$t(n)$， 伪代码分析如图 14 所示：

```
PLANE_CLOSEST_POINT_PAIR(Points, l, r)
    mindist = INF
    if l == r return mindist
    if l == r − 1 return dist(l, r)
    if l == r − 2 return min (dist(l, r), dist(l + 1, r), dist(l, l + 1)
    mid = (l + r)/2
    d1 = PLANE_CLOSEST_POINT_PAIR(Points, l, mid)
    d2 = PLANE_CLOSEST_POINT_PAIR(Points, mid + 1, r)
    d3 = Solve(Points, l, r)
    mindist = min (d1, d2, d3)
```

O(1)　　2t(n/2)　　O(n)　　O(1)

图 14 时间复杂度示意图（1）

故有：

$$t(n)=\begin{cases}2t(n/2)+O(n), n>3\\ O(1), n<=3\end{cases}$$

把上界函数具体化为：

$$t(n)=\begin{cases}2t(n/2)+cn, n>3\\ c, n<=3\end{cases}$$

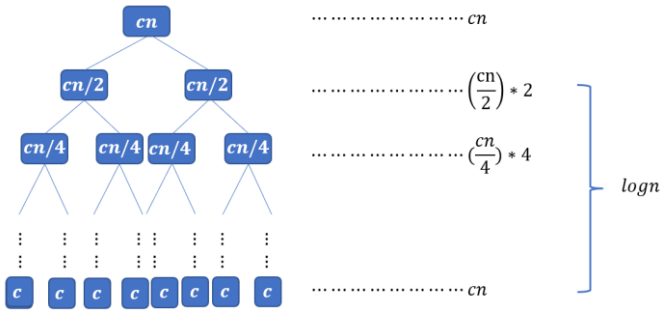
利用递归树推导如图 15 所示，得到：

$$t(n)=c(n+n\log n)=O(n\log n)$$

由于在分治算法执行之前需要将点集根据x坐标进行从小到大排序，时间复杂度为$O(n\log n)$，令整个算法的实际运行时间随数据规模的变化函数为$T(n)$，则有：

$$T(n)=O(n\log n)+t(n)=O(n\log n)$$

故，分治算法的时间复杂度为$O(nlogn)$



$$t(n) = (1 + logn) * cn = c(n + nlogn)$$

图 15 时间复杂度示意图（2）

ii. 空间复杂度

在归并排序部分需要使用额外空间进行存储，空间复杂度为$O(n)$

3. 数据测试与运行效率比较分析

（1）蛮力法性能测试

使用了数据规模为 10000、20000、30000、40000、50000、60000 的随机数据进行测试，每个规模的数据有 20 组，取平均运行时间。对于蛮力法优化前和优化后的运行效率进行测试，对比理论值（**以规模 10000 的实际运行时间为基准值**）与实验值。这里的优化指的是**两点距离计算**的优化，在算法运行中使用用距离的平方进行比较，最后输出答案时再开根号。测试结果如下：

优化前不同数据规模下的效率测试：

表 1 蛮力法优化前运行效率随数据规模变化表

数据规模（个）	10000	20000	30000	40000	50000	60000
理论值（ms）	546.340	2185.359	4917.058	8741.437	13658.495	19668.233
实验值（ms）	546.340	2178.502	4894.144	8703.998	13606.661	19603.077
误差	0.000%	-0.314%	-0.466%	-0.428%	-0.380%	-0.331%

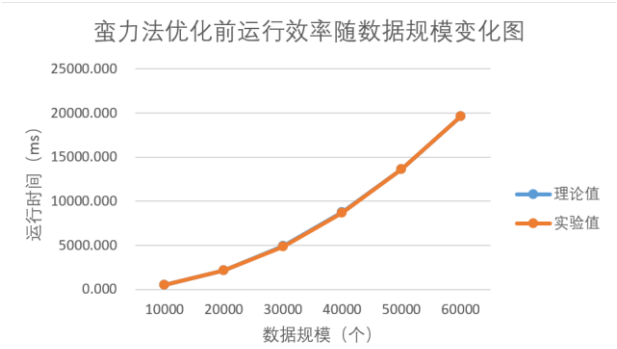


图 16 蛮力法优化前运行效率随数据规模变化图

优化后不同数据规模下的效率测试：

表 2 蛮力法优化后运行效率随数据规模变化表

数据规模（个）	10000	20000	30000	40000	50000	60000
理论值（ms）	265.960	1063.839	2393.637	4255.355	6648.993	9574.549
实验值（ms）	265.960	1063.524	2394.907	4258.898	6667.589	9585.261
误差	0.000%	-0.030%	0.053%	0.083%	0.280%	0.112%

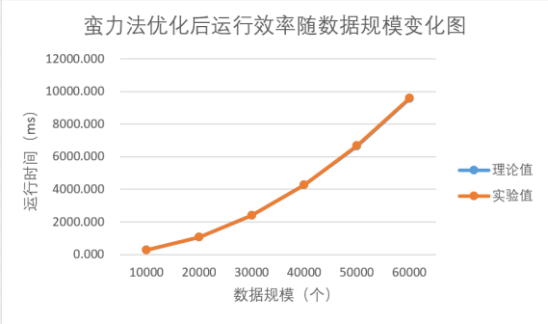


图 17 蛮力法优化后运行效率随数据规模变化图

通过实验可以看出，蛮力法在优化前后实验值和理论值的拟合都较好，误差非常小，因为蛮力法的实现比较简单，常数较小，容易拟合实验值和理论值。

优化前与优化后的对比：

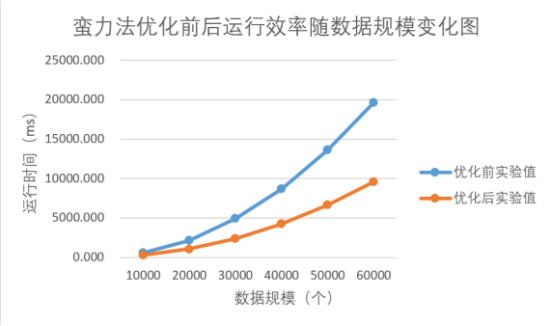


图 18 蛮力法优化前后运行效率对比图

通过小小的优化，可以大幅提升蛮力法的性能，通过实验可以看到，算法性能提升了一倍左右，可见$sqrt$函数的常数较大。

（2）分治法性能测试

使用了数据规模为 100000、200000、300000、400000、500000、600000 的随机数据进行测试，每个规模的数据有 20 组，取平均运行时间。对于分治法优化前和优化后的运行效率进行测试，对比理论值（**以规模 300000 的实际运行时间为基准值**）与实验值。这里的优化指的是**两点距离计算**的优化，在算法运行中使用用距离的平方进行比较，最后输出答案时再开根号。测试结果如下：

优化前不同数据规模下的效率测试：

表 3 分治法优化前运行效率随数据规模变化表

数据规模（个）	100000	200000	300000	400000	500000	600000
理论值（ms）	41.272	87.514	135.631	184.967	235.208	286.172
实验值（ms）	43.120	88.994	135.631	179.285	225.171	270.779
误差	4.476%	1.692%	0.000%	-3.072%	-4.268%	-5.379%

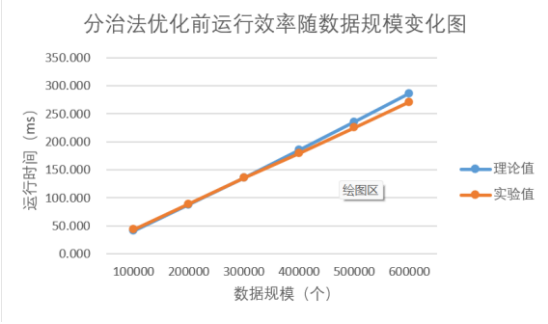


图 19 分治法优化前运行效率随数据规模变化图

优化后不同数据规模下的效率测试：

表 4 分治法优化后运行效率随数据规模变化表

数据规模（个）	100000	200000	300000	400000	500000	600000
理论值（ms）	33.856	71.788	111.259	151.729	192.942	234.748
实验值（ms）	35.286	73.418	111.259	150.563	188.152	237.662
误差	4.224%	2.271%	0.000%	-0.769%	-2.483%	1.241%

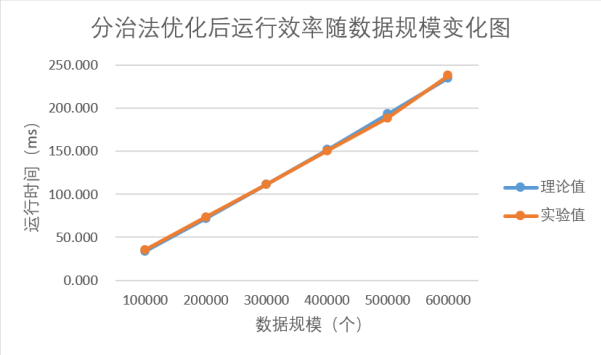


图 20 分治法优化后运行效率随数据规模变化图

测试了以几个规模的实际运行时间为基准之后，发现以 300000 规模时的实际运行时间为时，误差最小。优化前后理论值与实际值的误差都在可控范围内，由于分治法运行效率较快，故容易由系统的一点小波动产生误差。

优化前与优化后的对比：

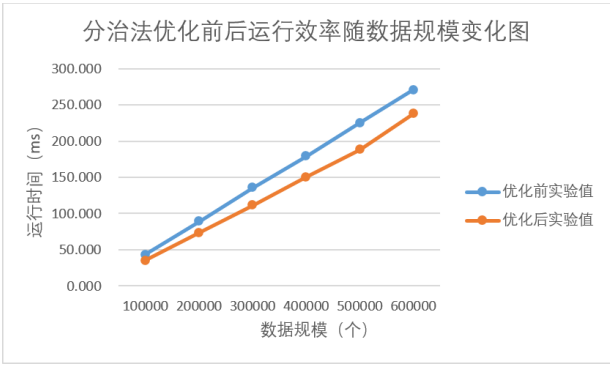


图 21 分治法优化前后运行效率对比图

通过实验发现，分治法距离函数优化之后，性能改变没有蛮力法幅度大。因为优化了距离函数之后，在比较的时候需要增加一些乘法，使得性能提高的程度减小。

（3）两种算法综合比较

使用了不同量级的数据进行测试，数据规模为 10、100、1000、10000、100000，每个规模的数据有 20 组，取平均运行时间，对两个算法的运行时间进行测试，两个算法均使用优化后的版本，测试结果如下：

表 5 不同算法运行效率随数据规模变化表

数据规模（个）	10	100	1000	10000	100000
蛮力法实验值（ms）	0.000	0.030	2.765	267.092	26637.267
分治法实验值（ms）	0.001	0.018	0.231	2.946	35.359

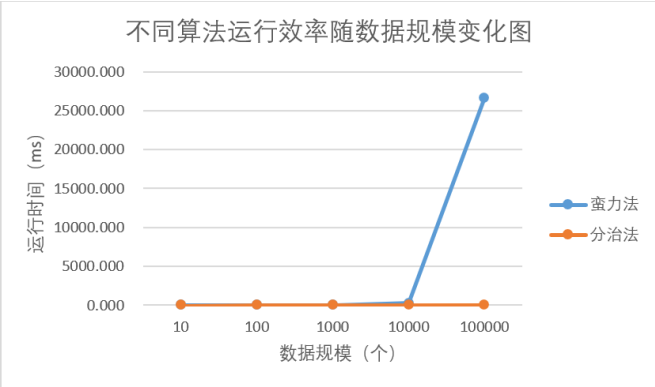


图 22 不同算法运行效率随数据规模变化图

通过对比实验可以得出，蛮力法与分治法在数据规模较小时效率差距较小，但是由于蛮力法的运行时间随数据规模指数增长，在数据规模达到 10000 时，分治法的效率远高于蛮力法。

4. 对分治算法的一点优化

尝试在某个规模时直接暴力枚举计算并返回最小距离，不再往下递归

通过设置一个阈值，如果当前问题规模小于等于这个阈值，则暴力枚举每一对点对，计算距离并返回最小距离，不再往下递归调用。由于合并的时候按照y坐标是使用归并排序，需要保证部分有序，在达到阈值时，需要把点按照y坐标排序，这里使用插入排序进行排序。在数据规模为 10000000 的数据下进行测试，设置不同的阈值（3-17），测试结果如下：

表 6 不同阈值下算法的运行速率表

数据规模（个）	10000000																			
threshold	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	20	25	30	35	40
平均运行时间（ms）	4418	4394	4354	4525	4445	4511	4623	4331	4348	4476	4340	4335	**4328**	4373	**4324**	4571	4575	4604	4578	5287

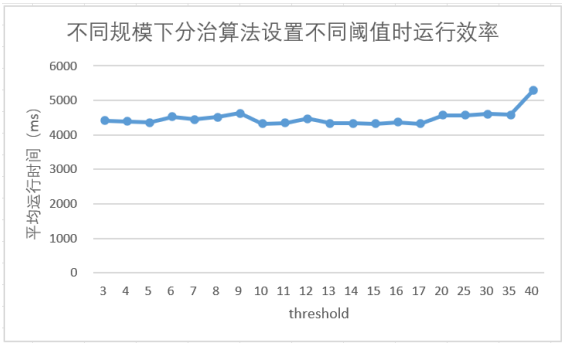


图 23 不同阈值下算法的运行效率图

由实验结果可以看出，在数据规模较大的情况下，选择合适的递归阈值可以略微降低运行时间。初步实验看出，阈值设置在 17 左右比较合适。

5. 图形界面输出算法运行过程

本人使用了*python*的*matplotlib*图形库可视化算法过程。没有花太多时间去学习太多的可视化方法，我采用了最笨的方法，把动态图的一帧一帧画出来之后，按照一定的时间间隔展示，组成了最终的结果。

对于蛮力法过程的可视化：

蛮力法过程比较简单，枚举时把当前枚举的点对标注出来，并实时更新当前的最近点对，如图 24 所示，青色的线表示当前枚举的点对，粉色的线表示当前的最近点对。

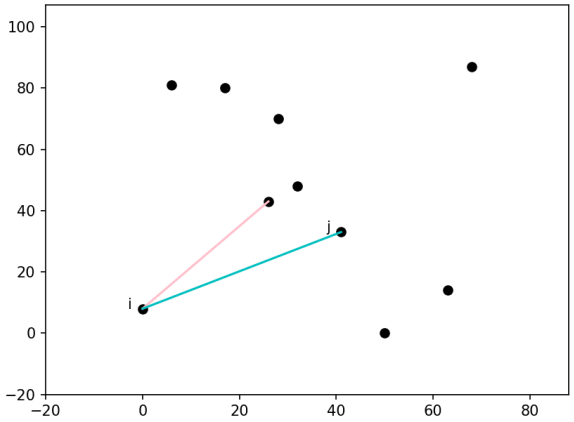


图 24 蛮力法过程可视化

对于分治法过程的可视化：

分治法过程相对蛮力法而言比较复杂一点，我每次递归解决子问题时，用不同颜色的方框表示不同深度的子问题，用相同颜色的方框表示同一深度的子问题，如图 25 所示：

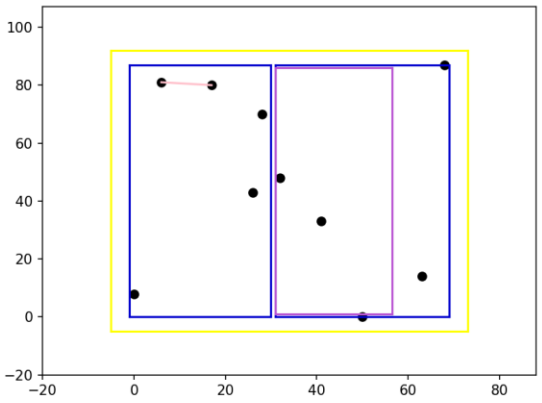


图 25 分治法过程可视化（1）

对于子问题求解之后的合并，我把分界线以及$[Points[mid].x-d, Points[mid].x+d]$这个范围画了出来，分别使用红线和绿线，在枚举点对时，与蛮力法一致，使用青色线表示当前枚举点对，使用粉色线表示当前最近点对。如图 26 所示。

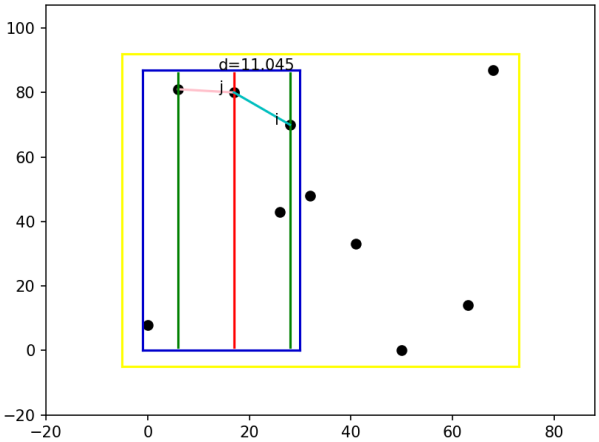


图 26 分治法过程可视化（2）

四．实验心得

算法分析方面，详细地阐述了蛮力法与分治法解决平面最近点对的过程，并辅以例子加以说明。同时详细地证明了分治算法的正确性，对两个算法的复杂度也进行详细地分析。对于复杂度分析更加游刃有余。

数据测试方面，测试了不同情况，不同量级的数据并加以分析，并对不同算法进行对比。

可视化方面，学习了$python$进行简单的可视化，更加细致的对算法过程进行理解。

五．附件说明

- 讲解 PPT
- Code 文件夹
  - better_divide_and_conquer.cpp（距离函数优化之后的分治法）
  - bruteforce.cpp（蛮力法，内含优化前与优化后的距离函数）
  - divide_and_conquer.cpp （分治法）
  - divide_and_conquer_with_threshold.cpp （带阈值的分治法）
  - generatedata.cpp （数据生成器）
  - brute_force.py（蛮力法可视化）
  - divide_and_conquer.py（分治法可视化）
- result.xlsx（测试结果数据表）
- brute_force.gif（蛮力法可视化结果展示）
- divide_and_conquer.gif（分治法可视化结果展示）

指导教师批阅意见： 成绩评定： 指导教师签字： 年　　月　　日
备注：

注：1、报告内的项目或内容设置，可根据实际情况加以调整和补充。

2、教师批改学生实验报告时间应在学生提交实验报告时间后 10 日内。