深 圳 大 学 实 验 报 告

课程名称 算法设计与分析

项目名称 二维最近点问题

学　　院 计算机与软件学院

专　　业 软件工程

指导教师 卢亚辉

报 告 人 郑彦薇 学号 2020151022

实验时间 2022/3/22~2022/4/6

提交时间 2022/4/8

教务处制

一、实验目的与要求

（1）掌握分治法思想。
（2）学会最近点对问题求解方法。

二、实验内容与方法

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=100,1000,10000,100000，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。

实验方法：

根据二维最近点对求解思路，利用c++编程求解正确结果并计时，得到不同方法求解下的实际运行时间和理论运行时间，利用excel作图进行对比。

三、实验步骤与过程

（一）问题分析和数据生成：

二维最近点对问题是在平面中随机生成的 N 个点，求解这 N 个点中距离最小的两个点并得到这个最短距离。

解决问题之前，首先需要随机生成 N 个点：使用 srand()和 rand()函数分别生成 N 个 x 和 N 个 y，根据索引将得到的 x 和 y 一一对应，得到 N 个点。

具体编程如下：

蛮力法生成 N 个随机点：

```
srand((unsigned int)time(0));

for (int i = 0; i < n; i++)
{
    dot[i].x = rand();
    dot[i].y = rand();
}
```

分治法生成 N 个随机点：

分治法生成随机点时设置了另一个点集存储生成的随机点，是因为在生成点之后，需要根据点的横坐标对点进行排序。而对于规模较小的点集，需要重复整个过程实现计时，因此设置另一个点集存储可以保证重复分治过程的点集仍然是随机生成的点集，不会出现第二趟时进入循环的点集已经排过序的情况。

```
struct point* dot_temp = new point[n];
for (int i = 0; i < n; i++)
    dot_temp[i] = dot[i];
```

```
for (int i = 0; i < n; i++)
    dot[i] = dot_temp[i];
```

对于二维最近点对问题，关键过程就是对于随机生成的这 N 个点中，找到最近点对以及这两个点之间的距离，下面分别对蛮力法和分治法的求解过程进行解析。

（二）蛮力法求解：

1. 算法思想

以点集的大小 N 设置双层循环，根据索引依次遍历点集中的每个点，得到当前点（第 i 个点）与在它之后的每一个点之间的距离，将当前得到的距离和最小距离 min_d 进行比较，若比 min_d 小，则更新 min_d 的值。min_d 一开始设置为大数 1e20。

2. 编程实现

```
for (int i = 0; i < n - 1; i++)
    for (int j = i + 1; j < n; j++) {
        d = sqrt((dot[i].x - dot[j].x) * (dot[i].x - dot[j].x) + (dot[i].y - dot[j].y) * (dot[i].y - dot[j].y));
        if (min_d > d) {
            min_d = d;
            k1 = i;
            k2 = j;
        }
    }
```

3. 时间复杂度分析

双层循环中，对于第 i 个点，依次求出这个点与其后面 N-i+1 个点的距离并进行比较，根据比较次数，可以得到时间复杂度为：$T\left(\frac{N(N-1)}{2}\right)\rightarrow O(n^2)$

（三）分治法求解：

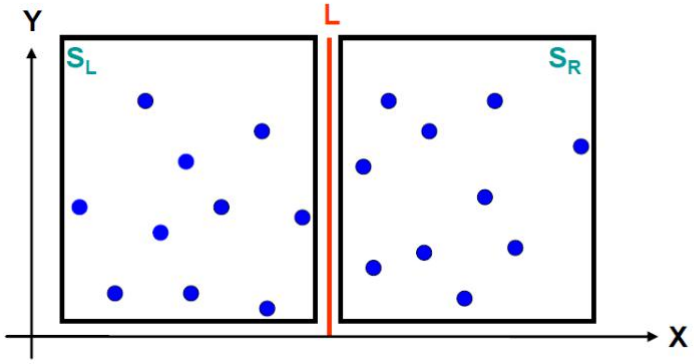
1. 算法思想

①点集的预处理：

对于随机生成的点集，按照点的横坐标进行排序，实验中采用归并排序。

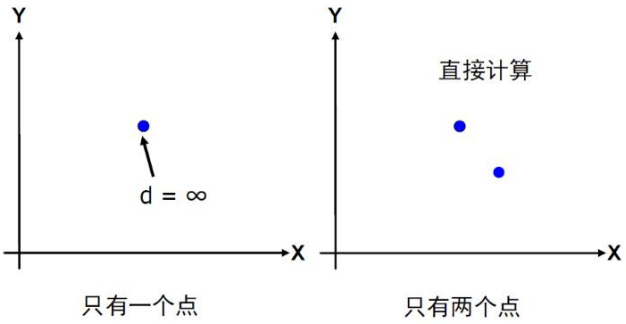
②确定中间线：

N 个随机点生成以后，已经对点按横坐标进行排序，因此左右子集只需根据点的个数进行划分，这样得到的左右子集也保证了左子集的点的横坐标 x 值都小于中间线对应的 x 值，右子集的点横坐标 x 值都大于中间线对应的 x 值。



③递归求解左右子集的最小值：

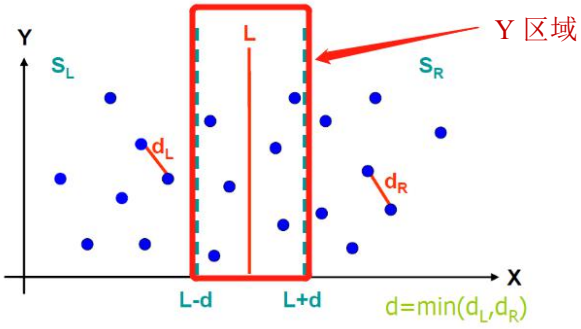
递归重复划分过程，如对于左子集进行再划分，直到”左子集”只剩下 1 或 2 个点，这两种情况都可以在很少的步骤内得到最近点对及其距离：对于点集中只有一个点，返回距离为无穷，这里设置为 99999999；对于点集中只有两个点，直接返回这两个点的距离。右子集同理。



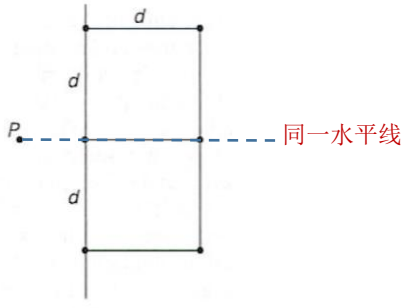
对点集进行划分之后求出当前的最小距离，再对子集进行合并。这里可以借助归并排序的思想：对最小点集（即只有 1 或 2 个点）求解最小距离后，进行点集的合并，再次求解最小距离。直到结束整个递归过程，得到左子集的最近点对距离 min_dl 和右子集的最近点对距离 min_dr。对得到的两个值进行比较，得到当前最近点对距离 min_d。

④根据 min_d 求解跨中间线的点对距离：

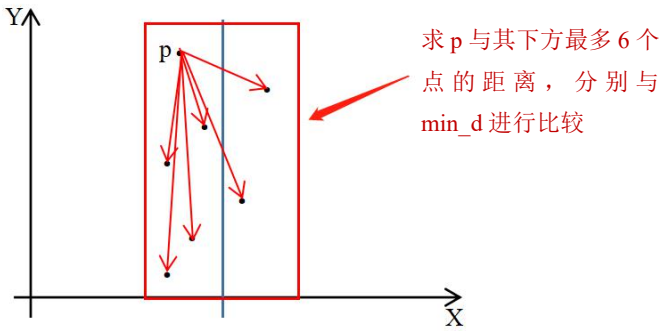
设中间线对应的横坐标为 x，定义 Y 区域为该区域中的点满足点的横坐标在[x-min_d,x+min_d]区间内。将 Y 区域的点存储在点集 temp 中，并用 m 记录 temp 点集的点个数，对 temp 点集按照点的纵坐标 y 进行排序，这里同样采用归并排序实现。



根据下图可以知道，对于 Y 区域中的任意一点 p，在 Y 区域中，与 p 的距离小于等于当前最近点对距离 min_d 的点最多只有 6 个。利用反证法进行简单证明：图中的 d 即为左右子集递归求得的最近点对距离 min_d，已知图中 d*2d 矩阵中，黑点标识的六个点每个点之间的距离都为 d，若矩阵中还有第 7 个点，这个点必定与六个点之一的距离小于 d，不成立。



因此在对 y 进行排序之后，只需遍历 Y 中的 m 个点，分别求出这 m 个点与在 Y 区域中其后 6 个点的距离，再与 min_d 进行比较，得到当前最近点对距离 min_d。



求 p 与其下方最多 6 个点的距离，分别与 min_d 进行比较

2. 编程实现

递归函数编程：

```
double func(point dot[], int left, int right, point dot1[])
{
    double min_d, min_dl, min_dr;
    int mid = (left + right) / 2;
    if (left == right)//一个点，返回无穷
        return 99999999;
    if (left + 1 == right)//两个点，直接返回两点间距离
    {
        dot1[0] = dot[left];
        dot1[1] = dot[right];
        min_d = dis(dot[left], dot[right]);
        return min_d;
    }
```

对 dot1 进行解释：dot1 是额外设置的大小为 2 的点集，用于存储最近点对的点坐标信息

```
    //多于2个点根据点的个数划分为两个点集，递归求解左右两个点集
    point* dotL = new point[2];
    min_dl = func(dot, left, mid, dot1);
    dotL[0] = dot1[0];
    dotL[1] = dot1[1];
    point* dotR = new point[2];
    min_dr = func(dot, mid + 1, right, dot1);
    dotR[0] = dot1[0];
    dotR[1] = dot1[1];
    if (min_dl > min_dr) {
        min_d = min_dr;
        dot1[0] = dotR[0];
        dot1[1] = dotR[1];
    }
    else {
        min_d = min_dl;
        dot1[0] = dotL[0];
        dot1[1] = dotL[1];
    }
```

对 dotL、dotR 进行解释：dotL 用于存储左点集分治递归过程中得到的最近点对的信息；dotR 用于存储右点集分治递归过程中得到的最近点对的信息。通过 dotL 和 dotR 对递归过程中最近点对信息的存储，在最终进行 min_d 的确定时，也可以将对应的最近点对信息复制给 dot1，在程序最后进行输出。

获取 temp 点集以及 temp 点集的排序：

```
//将位于中线两侧的点存储到temp中
struct point* temp = new point[right - left + 1];
int m = 0;
for (int i = left; i <= right; i++)
{
    if (dot[i].x > (dot[mid].x - min_d) && dot[i].x < (dot[mid].x + min_d)) {
        temp[m++] = dot[i];
    }
}
```

```
//对temp按y的值进行排序
MergeSort_Y(temp, 0, m-1);
```

排序编程实现（点的预处理时进行的对 x 的归并排序同理此过程）：

```cpp
void merge_Y(point* temp, int low, int mid, int high)
{
    //struct point* temp1 = new point[n];
    for (int k = low; k < high + 1; k++)
        temp1[k] = temp[k];
    int i = low, j = mid + 1, k = low;
    while (i < mid + 1 && j < high + 1)
    {
        if (temp1[i].y < temp1[j].y)
            temp[k++] = temp1[i++];
        else
            temp[k++] = temp1[j++];
    }
    while (i < mid + 1)
        temp[k++] = temp1[i++];
    while (j < high + 1)
        temp[k++] = temp1[j++];
    //delete[]temp1;
}

void MergeSort_Y(point* temp, int low, int high)
{
    if (low < high)
    {
        int mid = (low + high) / 2;
        MergeSort_Y(temp, low, mid);
        MergeSort_Y(temp, mid + 1, high);
        merge_Y(temp, low, mid, high);
    }
}
```

求解 temp 中点对距离并与 min_d 比较：

```cpp
//求中线两侧点对距离
for (int i = 0; i <= m-1; i++)
{
    for (int j = i + 1; j <= m-1 && j <= i + 6; j++)
    {
        if ((temp[j].y - temp[i].y) > min_d)
            break;
        if (dis(temp[i], temp[j]) <= min_d)
        {
            min_d = dis(temp[i], temp[j]);
            dot1[0] = temp[i];
            dot1[1] = temp[j];
        }
    }
}
```

因为 temp 点集已经按照 y 值进行排序，因此在计算距离前，可以先对两点的 y 值的差进行判断，若已经大于 min_d，则说明距离也大于 min_d，同时也说明在第 j 个点之后的点与第 i 个点之间的距离也大于 min_d。这也可以进一步减少算法的实际运行时间

3. 时间复杂度分析

根据分治法中递归求解：$T(n)=2T(n/2)+n\rightarrow O(n\log n)$

对点集进行归并排序：$O(n\log n)$

跨中间线点对的距离计算：$T(6*n/2)\rightarrow O(n)$

可以最终得到分治法的时间复杂度为：$O(n\log n)$

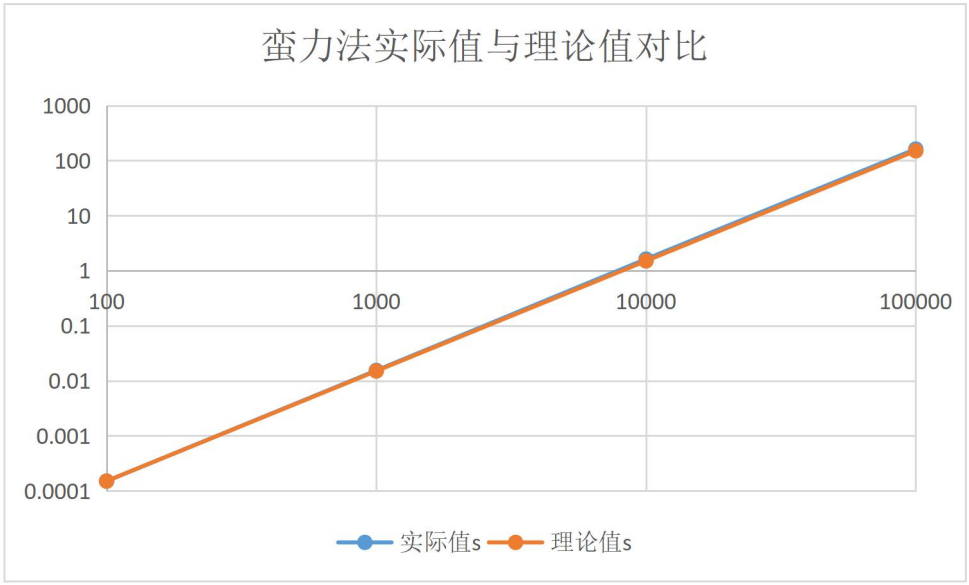
（四）效率分析和对比：

1. 蛮力法运行时间及效率分析

对 N=100，1000，10000，100000 规模下生成的随机点集求解最近点对距离的实际运行时间进行统计。取规模为 100 的实际运行时间为理论运行时间，根据时间复杂度 $O(n^2)$计算出其他规模下的理论运行时间，得到如下表格：

蛮力法	n	实际值s	理论值s
0(n)	100	0.00015	0.00015
	1000	0.0154	0.015
	10000	1.622	1.5
	100000	161.902	150

对实际运行时间和理论运行时间同时做与规模的关系图，并取对数刻度，得到对比图如下：



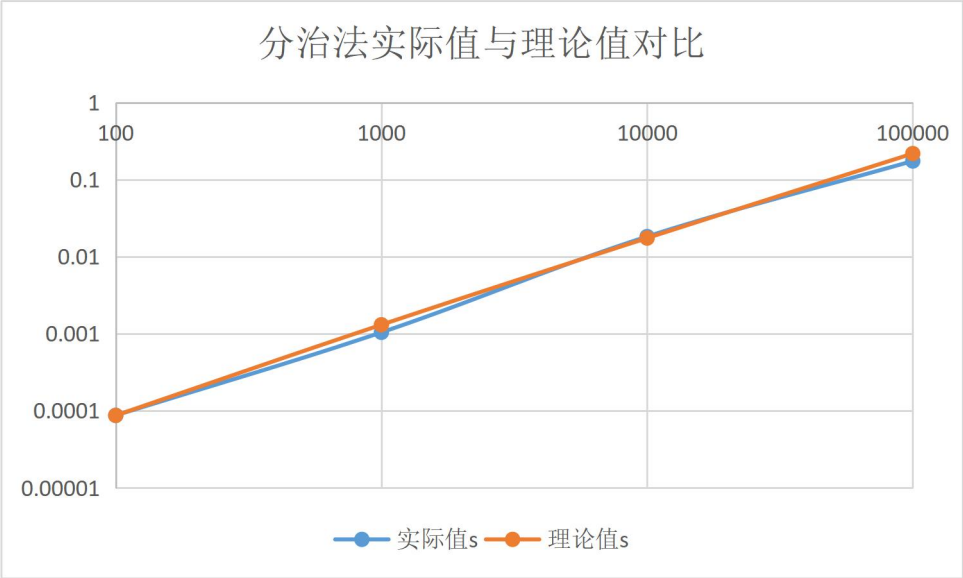
可以看到两条曲线几乎重合。在取对数之前，根据表格可以知道规模增大后，实际值要高于理论值，原因可能是在生成随机点时，进行了坐标范围的限制，而规模小时，生成点集的坐标值也小，计算时所需的时间也比较低，这里又取了最小规模的实际运行时间作为理论运行时间，使得大规模的理论运行时间也是一个较小值，因此会出现规模增大后，实际运行时间略高于理论运行时间。

2. 分治法运行时间及效率分析

对 N=100，1000，10000，100000 规模下生成的随机点集求解最近点对距离的实际运行时间进行统计。取规模为 100 的实际运行时间为理论运行时间，根据时间复杂度 O(nlogn)计算出其他规模下的理论运行时间，得到如下表格：

分治法	n	实际值s	理论值s
0(nlogn)	100	0.000087	0.000087
	1000	0.00104	0.001305
	10000	0.0182	0.0174
	100000	0.174	0.2175

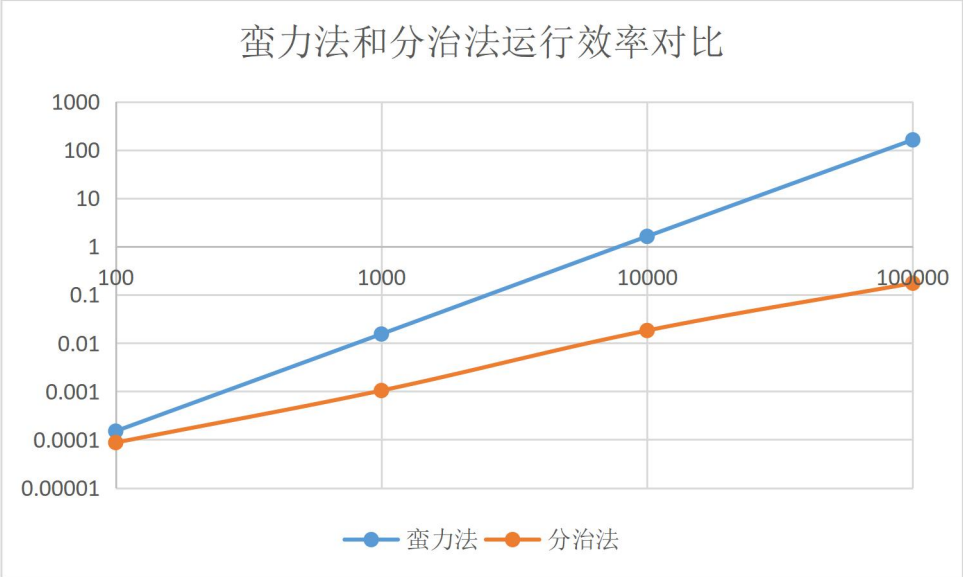
对实际运行时间和理论运行时间同时做与规模的关系图，并取对数刻度，得到对比图如下：



由上图可以看出，实际值与理论值对应的曲线的拟合度并不高，这里尝试对其进行分析，出现这个结果的原因可能是在分治法中，分别进行了一次对横坐标 x 的排序和一次对纵坐标 y 的排序，根据排序算法可以知道，若数据已经相对有序，那么实际运行时间也会相应减少，同样的，如果数据无序性较大，那么实际运行时间也会相应增大。

3. 蛮力法与分治法效率对比

根据实验所得蛮力法和分治法的实际运行时间，得到两者的运行效率对比图如下：



根据上图可以知道，分治法的运行效率高于蛮力法，且随着规模的增大，分治法在处理二维最近点对问题时的效率优势就更明显。但对比规模为 100 时的运行效率，可以看到两点十分接近，因此当规模很小时，也可以采用蛮力法解决问题，这样避免因分治法编程的复杂性造成的错误。

四、实验结论或体会

1．进行编程时，需要逐步检查编程的正确性，通过调试或者观察每一步得到的结果确定算法是否能够根据要求得到正确的结果。要保证编程的正确性，才能对运行时间进行统计。

2．在进行分治求解时，需要将点集分为左右两个点集。在实现这一过程时，首先是根

据得到的点集的 x 的值对点进行排序，再根据下标将点集分为前 N/2 个和后 N/2 个。因为已经对 x 进行点的排序，因此通过下标划分中间线也可以实现中间的点的 x 值恰好为所有 x 的中间值。

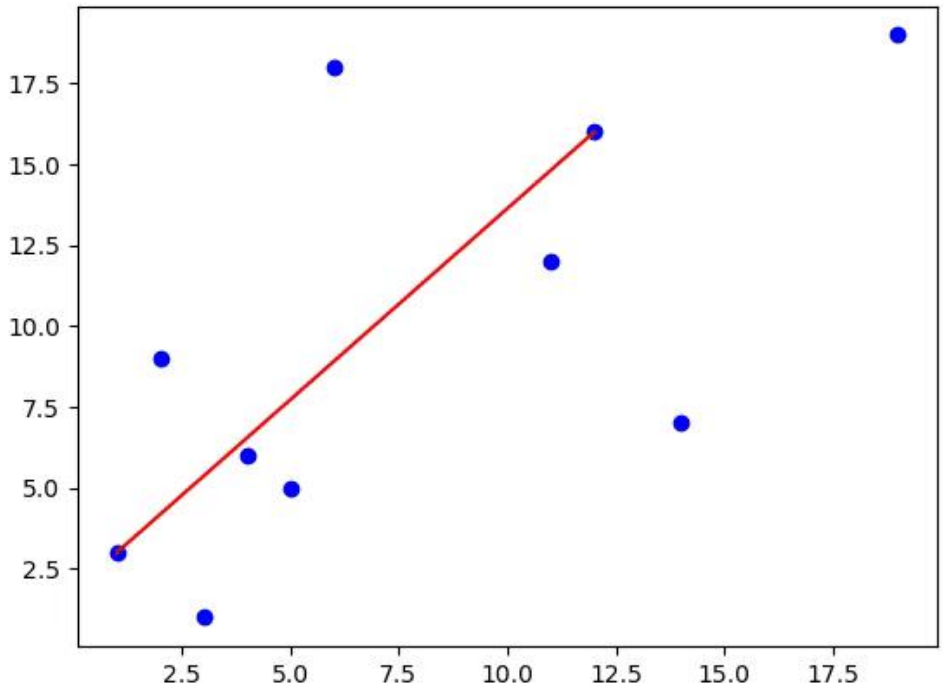
3．分治法求解时，一开始采用记录下标的方式来确定最近点对的点信息，但因为存在递归过程，在最终比较 min_dl 和 min_dr 确定 min_d 时，下标的记录便出现错误。改正方法是定义两个点结构（即分治法中进行解释的 dot1）进行最近点对信息的存储。

4．在实验过程中，我将蛮力法和分治法编程在同一个 cpp 文件中，在运行时只生成一组规模为 N 的随机点集，同时为了防止分治的排序结果对蛮力的运行时间有影响，蛮力法在分治法之前操作。将两种方法在同一个 cpp 文件的好处在于：一是可以通过结果确认分治法编程的正确性，二是待操作的点集相同，可以更好的比较两种方法的效率。

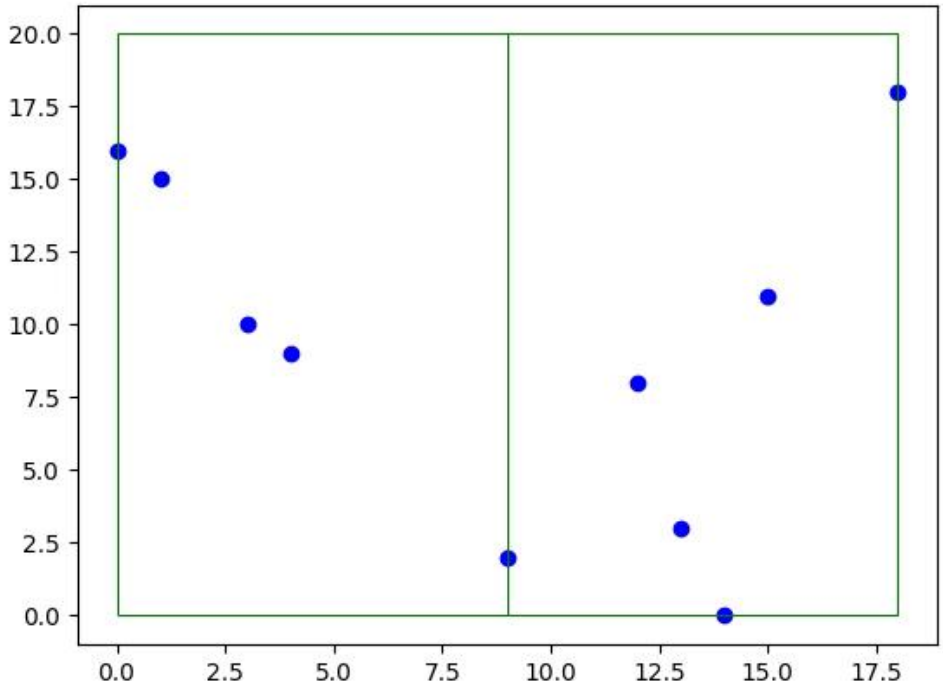
5．对过程进行图形界面输出，可以更加直观的观察实际运行过程。为实现这一目的，对 python 实现图形输出进行了学习。

五、思考

算法的图形界面输出（利用 python 实现：获取每一帧画面，将图片加入视频流）
蛮力法：（查看动图需右击打开超链接）



分治法：（只展示划分至最小点集并得到最短距离的过程）


20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5

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。