**深 圳 大 学 实 验 报 告**

**课程名称： 算法设计与分析**

**实验名称： 实验二 分治法求最近点对问题**

**学院： 计算机与软件学院 专业：计算机科学与技术（创新班）**

**报告人： XXX 学号： XXXXXXXXXX 班级： 高性能班**

**同组人：**

**指导教师： 杨烜**

**实验时间： 2024.4.1**

**实验报告提交时间： 2024.4.**14

**教务处制**

1. **问题描述**

对于平面上给定的N个随机点，给出所有点对中的最短距离，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。

以上问题分别用蛮力法和分治法进行解决，分别取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。

1. **求解问题的算法原理描述**
2. 蛮力法

**算法原理：**

遍历所有的点对，并计算它们之间的距离，然后找出其中距离最小的一对点，即为最近点。

许多电脑萤幕

中度可信度描述已自动生成

图表, 散点图

描述已自动生成

**算法的实现细节的解释：**

1、对于给定的N个点，首先初始化一个最小距离变量min\_dis为一个较大的值，用来记录最小距离。



2、遍历所有可能的点对，对于每一对点(i, j)，计算它们之间的距离new\_dis，即。

图形用户界面, 应用程序, Word

描述已自动生成

3、如果dis小于当前的min\_dis，则更新min\_dis为dis，并记录对应的点对(i, j)。

文本

描述已自动生成

4、完成遍历后，返回记录的最小距离及最小距离对应的点对即可。

5、复杂度分析：当遍历第i个点时，需要第i个点和n-i个点进行距离计算，因此，该算法需要遍历到第n-1个点，遍历到第i个点时需进行n-i次循环。复杂度的计算如下：

1. 分治法

**算法原理：**

将问题划分为更小的子问题，然后递归地解决这些子问题，最后将子问题的解合并起来得到原问题的解。具体来说，递归地将N个点分成左右两部分子点集，求左右子点集的最近点对p1和p2，在合并时在相邻子集间的区域中找到最近点对p3和p4，然后在p1、p2和p3、p4这两对点对中取最近的点对作为该合并子集的解返回。

**算法的实现细节的解释：**

1、将平面上的N个点按照横坐标x进行排序。



2、将问题划分为两个子问题：选择N个点中排序后位于最中间的点作为分割点，其所在的竖直轴作为中位线，根据中位线将点集平均划分为左子点集和右子点集。递归地对左右两个子点集求解，得到左右两个子点集的最短距离ldis和rdis以及两个子点集的最近点对q1、q2和q3、q4。

图表, 散点图

描述已自动生成

图片包含 图表

描述已自动生成

3、递归划分点集的终点：当点集长度为1时，直接返回INF；当点集长度为2时，返回两个点的距离；当点集长度为3时，返回三个点中最近点对及其距离。

图表, 箱线图

描述已自动生成

图片包含 文本

描述已自动生成

文本

描述已自动生成

4、取左右两个子点集的最小距离min\_dis=min(ldis, rdis)和最近点对p1和p2。

文本

描述已自动生成

5、合并左右子集，考虑跨两个子点集可能存在的最近点对：在中位线划分出一个宽为2\*min\_dis、中心在中位线上的区域Y，即所有横坐标距离中位线小于min\_dis的点都被包含在这个区域中。

图表, 散点图

描述已自动生成

这样做的目的是将相距大于min\_dis的跨子点集的点对排除在外（假设一个位于左子点集的点的横坐标距离中位线大于min\_dis，则该点距离右子点集任意一点的距离必然大于min\_dis），减少搜索的数量，即减少计算量。在上图中，夹在两条红色虚线间的区域为Y，位于区域Y外的点为蓝色的，显然蓝色点到相邻子集的任何一个点的距离必然大于min\_dis。

图表, 散点图

描述已自动生成

在上图中，和分别为左右子点集，和分别为左右子点集中最近点对的距离，计算出d=min()，然后划分出的蓝色框即为区域Y。

划分出区域Y后，在该区域中找最近点对，若该最近点对的距离小于min\_dis，则更新min\_dis和最近点对p1和p2。



6、返回min\_dis作为最终的最小距离。

**如何在区域Y中找最近点对**

**策略一：部分蛮力**

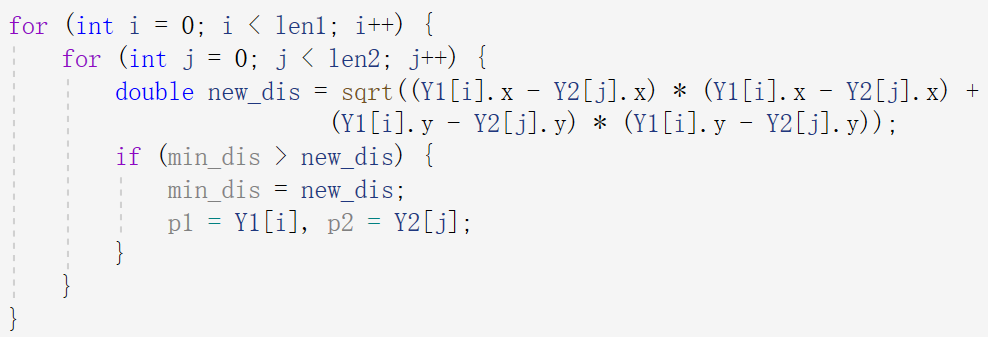
**原理及实现细节解释：**

将Y根据中位线分为左右Y1和Y2两个区域，计算Y1中每个点和Y2中的每个点的距离，计算得到相距距离最短的点对为最近点对。

图表, 散点图

描述已自动生成

在上图中，蓝色点位于区域Y1中，红色点位于区域Y2中，对每个蓝色点计算其与每个红色点的距离，记录下最短距离和最近点对，最终以橙色直线相连的点对即为最近点对。



**正确性验证：**

通过10次随机测试生成5000个随机点，分别用蛮力法和该合并策略下的分治法去求解该点集的最近点对，通过比较两者的结果，以此来验证该合并策略下的分治法的正确性。结果如下图所示。

文本

描述已自动生成

10次测试下两种算法的结果均一致，说明该合并策略下的分治法的正确。

**复杂度分析：**

1、排序：在对点集进行分治之前，需要对点集根据横坐标x进行排序，在本问题中，排序采用的是快速排序，时间复杂度为。

2、分治法：下图摘自《数据结构与算法分析-C语言描述》P280，通过这一段我们可以知道，在均匀分布的随机点集中，位于区域Y中的点的个数平均只有个，故区域Y1和Y2中要遍历的点的个数均为，计算过程中有两层循环，外层和内层循环的次数则均为区域Y1和Y2中点的个数，所以部分蛮力在区域Y中找最近点对的复杂度为，即合并的效率为。

文本

描述已自动生成

结合前面的分析，我们可以得到基于部分蛮力的分治法的递推式为

由于n=2,3时的计算与相差无几，我们可以认为

这样有利于后续对该分治法复杂度的计算。

通过递归树，我们可以计算出该合并策略下分治法的时间复杂度。

图示

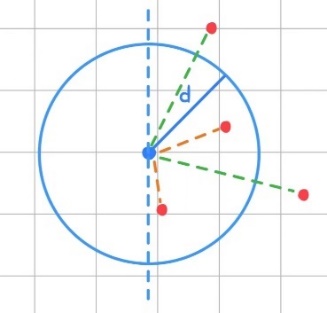
描述已自动生成

综上，根据排序（）和分治法（）最终计算得到的时间复杂度，我们可以认为整个算法的时间复杂度为。

**策略二：维护一个计算区域**

**原理及实现细节解释：**

将Y根据中位线分为左右Y1和Y2两个区域，对Y1和Y2中的点根据纵坐标分别进行排序，计算Y1中每个点和位于特定计算区域内Y2中的每个点的距离，计算得到相距距离最短的点对为最近点对。



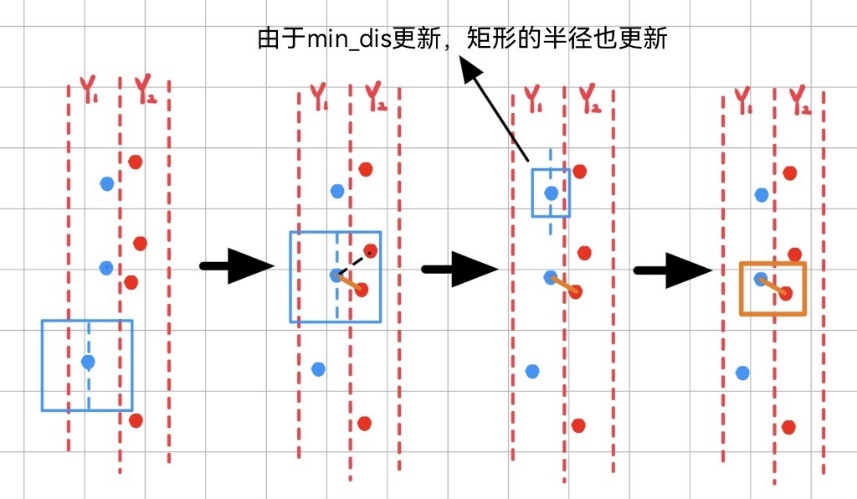
在策略一中，每个Y1中的点要和Y2中的每个点进行一次距离计算，但显然这是不合理的，我们的目的是在区域Y中找寻比min\_dis距离更近的点对，然而在Y1中每个点和Y2中每个点进行计算时，有大部分点对的距离是大于min\_dis的，计算它们的距离不仅不会对答案有任何贡献，反而会降低搜索效率，因此我们需要对部分暴力进行优化避免这种情况。

根据前面的分析，我们可以知道，之所以要在区域Y中找跨子点集的最近点对，是为了找到可能存在的比左右子点集中距离为min\_dis最近点对更近的点对，所以对于区域Y中找跨子点集距离大于min\_dis的点对，我们无需考虑。同理，在计算区域Y1中的点和区域Y2中的点的距离时，我们可以维护一个半径为d=min\_dis的半圆区域，当遍历到Y1中第i个点时，我们以该点为中心，划分出一个半径为d的圆形区域，当前仅当Y2中的点位于这个圆形区域内时，才去计算该点与Y1第i个点的距离并考虑是否更新min\_dis和最近点对，这样便能排除掉相距大于min\_dis的点对，减少计算量，避免了部分暴力中大部分不必要的计算。如上图中，有蓝色点位于区域Y1中，红色点位于区域Y2中，我们可以看到有两个红点位于圆形区域内（以橙色虚线与蓝点相连），故只需要计算该两点与蓝点的距离并更新最短距离min\_dis，而位于圆形区域外的两个红点与蓝点的距离必然大于min\_dis（以绿色虚线与蓝点相连），对min\_dis的更新无贡献，故无需考虑。

图表, 散点图

描述已自动生成

如此，我们便能相对于策略一加快了在区域Y中找跨子点集点对的过程，每次遍历到Y1中的点i时，只需维护一个半径为min\_dis的圆形区域，当有Y2中的点位于该圆形区域内时，才去计算其与Y1中点i的距离并考虑更新min\_dis。在上图中，蓝色点位于区域Y1中，红色点位于区域Y2，相较于策略一，我们能发现计算跨子点集的点对距离的次数大大减少，因此整个过程的速度得以提升。



由于计算机难以对点是否在圆形区域中进行判断，所以将半圆扩展为矩形便于计算机运算，矩形的上下左右边界距离中心均为min\_dis。找最近点对的过程如上图所示。

我们还要考虑一个问题，每次计算矩阵内的点对的距离时，点对的个数最多有多少个？

我们可以画图分析，因为是根据Y1中的点来划分出矩形区域的，故Y2中的点只可能出现在右半边的矩形中，所以下面的分析我们只围绕右半边矩形区域进行讨论。由于之前求得左右子点集最近点对的距离为min\_dis，所以左右子点集中点对的距离必然大于等于min\_dis，因此，位于矩阵内Y2的点之间的距离必然大于等于min\_dis，故在极限条件下且不考虑有重合点，在宽为min\_dis，高为2\*min\_dis的矩形内，最多能存在六个点（如下图，红圈的半径为min\_dis，蓝色点表示Y1中的点，红色点表示Y2中的点），使得彼此间的距离大于等于min\_dis，由于有一个为Y1中的点，故在计算矩阵内的点对的距离时，最多需要计算五对点对的距离。

图示

低可信度描述已自动生成

计算前需要对Y1和Y2中的点根据纵坐标y进行排序，这样便于对点是否在矩形区域的范围内进行判断。

散点图

中度可信度描述已自动生成

在上面的代码中我们能看到，在进入计算点对距离前，先对Y1和Y2区域根据纵坐标进行排序，便于后续范围判断。在每次遍历Y1中的点时，会先对矩形的边界进行更新，使得在后续遍历Y2中的点时，能迅速对点是否在边界中进行判断。然后在遍历Y2中的点时，若该点的纵坐标小于等于矩形下边界或横坐标大于等于矩形右边界时，直接跳过该点；若该点位于矩形内则计算点对的距离并考虑是否更新最短距离；若该点的纵坐标大于等于矩形上边界，直接停止遍历，避免不必要的计算。

**正确性验证：**

通过10次随机测试生成5000个随机点，分别用蛮力法和该合并策略下的分治法去求解该点集的最近点对，通过比较两者的结果，以此来验证该合并策略下的分治法的正确性。结果如下图所示。

文本

描述已自动生成

10次测试下两种算法的结果均一致，说明该合并策略下的分治法的正确。

**复杂度分析：**

1、排序：在对点集进行分治之前，需要对点集根据横坐标x进行排序，在本问题中，排序采用的是快速排序，时间复杂度为。

2、分治法：在找寻最近点对前，需要对区域Y1和Y2根据纵坐标y进行排序，由于区域Y1和Y2中要遍历的点的个数均为，且采用排序算法进行排序，因此这个过程的时间复杂度为。

在找寻最近点对的过程中，由于区域Y1和Y2中要遍历的点的个数均为，计算过程中有两层循环，外层循环的次数为区域Y1中点的个数，每次内层循环由于维护的计算区域缩小遍历的范围和计算时间，相当于每次只需要遍历位于计算区域上界下的点，故每次内层循环次数略小于区域Y2中点的个数，所以这个过程的时间复杂度为。

综合以上步骤，我们可以认为维护一个计算区域在区域Y中找最近点对的复杂度为 ，即合并的效率为。

结合前面的分析，我们可以得到基于维护一个计算区域的分治法的递推式为

由于n=2,3时的计算与相差无几，我们可以认为

这样有利于后续对该分治法复杂度的计算。

通过递归树，我们可以计算出该合并策略下分治法的时间复杂度。

图示

描述已自动生成

综上，根据排序（）和分治法（）最终计算得到的时间复杂度，我们可以认为整个算法的时间复杂度为。

**策略三：维护Y2下界指针**

**原理及实现细节解释：**

将Y根据中位线分为左右Y1和Y2两个区域，对Y1和Y2中的点根据纵坐标分别进行排序，计算Y1中每个点和位于特定计算区域内Y2中的每个点的距离，且每次遍历Y2均从指针下界开始遍历，计算得到相距距离最短的点对为最近点对。

日历

低可信度描述已自动生成

在策略二中，虽然我们通过维护一个特定的计算区域来减少不必要的计算，但在找Y2中在计算区域内的点时，我们仍从Y2底部开始搜索，显然，随着Y1中遍历点纵坐标的升高，Y2中在计算区域内的点的纵坐标也同样要升高，如此，对于Y2中下标靠前的点，它们的纵坐标偏低，显然不会在计算区域内，遍历它们显然会降低搜索的效率。因此，我们需要通过新的优化来避免这些不必要的遍历。

通过观察我们能够发现，在对Y1中的点根据纵坐标y进行排序后，第i个点必然在第i-1个点计算矩阵的上方，即第i个点计算矩阵的下界纵坐标必然大于等于第i-1个点纵坐标，这是因为前面计算出左右子点集的最近点对的距离为min\_dis，故Y1中各点之间的距离必然大于等于min\_dis，且因为矩形上下界距离中心为min\_dis，所以则会有第i个点计算矩阵的下界必然大于等于第i-1个点纵坐标的情况。所以在遍历到Y1中第i个点时，其对应Y2中的点可以从纵坐标位于Y1中第i-1个点纵坐标以上的点开始遍历。

如何可以快速得到遍历Y1中第i个点时开始遍历Y2的下界，我们可以用一个指针来维护。在遍历到Y1中第i个点，对应去计算与Y2中的点的距离时，当遍历到第一个Y2中的点P的纵坐标大于Y1中第i个点的纵坐标时，用一个下界指针指向该点P，在下次遍历到Y1中第i+1个点时，对应遍历Y2中的点从该下界指针指向的点P开始，以避免策略二中遍历指针前的点不必要的计算。

一群人的地图

中度可信度描述已自动生成

如上图，绿色箭头为下界指针，初始化指向Y2中第一个点，当出现第一个Y2中的点P的纵坐标大于当前遍历Y1中的点的纵坐标时，更新指针，令该指针指向该点P，在下次遍历Y1中下一个点时，从指针指向的点开始遍历Y2。

文本

描述已自动生成

在上面的代码中，用了一个flag作为标记，在进行每次外循环遍历前，初始化为1，当找到第一个Y2中纵坐标大于Y1中第i个点的纵坐标的点P时，更新下界指针后，令flag=0,保证了下界指针指向的点为第一个Y2中纵坐标大于Y1中第i个点的纵坐标的点。

**正确性验证：**

通过10次随机测试生成5000个随机点，分别用蛮力法和该合并策略下的分治法去求解该点集的最近点对，通过比较两者的结果，以此来验证该合并策略下的分治法的正确性。结果如下图所示。

文本

描述已自动生成

10次测试下两种算法的结果均一致，说明该合并策略下的分治法的正确。

**复杂度分析：**

1、排序：在对点集进行分治之前，需要对点集根据横坐标x进行排序，在本问题中，排序采用的是快速排序，时间复杂度为。

2、分治法：在找寻最近点对前，需要对区域Y1和Y2根据纵坐标y进行排序，由于区域Y1和Y2中要遍历的点的个数均为，且采用排序算法进行排序，因此这个过程的时间复杂度为。

在找寻最近点对的过程中，由于区域Y1和Y2中要遍历的点的个数均为，计算过程中有两层循环，外层循环的次数为区域Y1中点的个数，每次内层循环由于维护的计算区域缩小遍历的范围和计算时间，且维护的下界指针也同样缩小了遍历的范围，这样基本相当于每次遍历Y2都能从比较接近计算区域的点开始，并且在遍历到位于计算区域上界的点时便结束，又因为计算区域内的点最多有五个，故可以认为每次内层循环次数接近于，所以该过程的复杂度为。

综合以上步骤，我们可以认为维护一个下界指针在区域Y中找最近点对的复杂度为 ，即合并的效率为。

结合前面的分析，我们可以得到基于维护一个下界指针的分治法的递推式为

由于n=2,3时的计算与相差无几，且在本问题中在n取值100000到1000000范围内，因此我们可以认为

这样有利于后续对该分治法复杂度的计算。

通过主定理法，由得，因为当，又因为，故我们可以计算出该合并策略下分治法的时间复杂度。

综上，根据排序（）和分治法（）最终计算得到的时间复杂度，我们可以认为整个算法的时间复杂度为。

1. **算法实现的核心伪代码**
2. 蛮力法

|  |
| --- |
| min\_dis: = n  for i from 0 to n - 1 do  for j from i + 1 to n - 1 do  dis : = Distance(Point[i], Point[j]) // 计算点对间距  if min\_dis > dis then // 更新最短距离  min\_dis : = dis |

1. 分治法

|  |
| --- |
| // 分治法  Divide(points: array of Point, left: integer, right: integer) :  // right - left <= 2直接返回结果  if left = right :  return INF  else if left + 1 = right :  return Distance(points[left],points[right])  else if left + 2 = right :  dis1 := Distance(points[left],points[left+1])  dis2 := Distance(points[left],points[right])  dis3 := Distance(points[left+1],points[right])  dis1 := min(dis1, dis2, dis3)  return dis1  else:  mid := (left + right) >> 1  ldis := Divide(points, left, mid) // 求左子集最近点对  rdis := Divide(points, mid + 1, right) // 求左子集最近点对  min\_dis := min(ldis, rdis)  min\_dis := Merge(points, left, right, mid, min\_dis) // 合并左右子集  return min\_dis |

**合并策略一：部分蛮力**

|  |
| --- |
| // 合并左右子集  Merge(points: array of Point, left: integer, right: integer, mid: integer, min\_dis: floating point numbers) :  Y1 := new Point[right - left + 1] //左子集  Y2 := new Point[right - left + 1] //右子集  len1 := 0  len2 := 0  for i from mid down to left :  if points[i].x > points[mid].x - d:  Y1[len1++] := points[i]  else :  break  for i from mid + 1 to right :  if points[i].x < points[mid].x + d :  Y2[len2++] := points[i]  else :  break  for i from 0 to len1 :  for j from 0 to len2 :  new\_dis := Distance(Y1[i], Y2[j])  if min\_dis > new\_dis: // 更新最短距离  min\_dis := new\_dis  delete[] Y1  delete[] Y2  return min\_dis // 返回最短距离 |

**合并策略二：维护一个计算区域**

|  |
| --- |
| // 合并左右子集  Merge(points: array of Point, left: integer, right: integer, mid: integer, min\_dis: floating point numbers):  Y1 := new Point[right - left + 1] //左子集  Y2 := new Point[right - left + 1] //右子集  len1 := 0  len2 := 0  for i from mid down to left :  if points[i].x > points[mid].x - d:  Y1[len1++] := points[i]  else :  break  for i from mid + 1 to right :  if points[i].x < points[mid].x + d :  Y2[len2++] := points[i]  else :  break  quicksort(Y1, 0, len1 - 1, 'y') // 对Y1根据纵坐标排序  quicksort(Y2, 0, len2 - 1, 'y') // 对Y2根据纵坐标排序  for i from 0 to len1 :  // base\_y1为矩形下边界，base\_y2为矩形上边界，base\_x为矩形右边界  base\_y1 := Y1[i].y - min\_dis  base\_y2 := Y1[i].y + min\_dis  base\_x := Y1[i].x + min\_dis  for j from 0 to len2 :  // Y2中在矩形右侧和下面的点直接跳过  if Y2[j].x >= base\_x or Y2[j].y <= base\_y1 :  continue  // 计算在矩形内的点对距离  else if Y2[j].x < base\_x and Y2[j].y > base\_y1 and Y2[j].y < base\_y2 :  new\_dis = Distance(Y1[i],Y2[j])  min\_dis = min(min\_dis, new\_dis) // 更新最短距离  // 若Y2中的点的纵坐标大于矩形上边界，则停止循环  else if Y2[j].y >= base\_y2:  break  delete[] Y1  delete[] Y2  return min\_dis // 返回最短距离 |

**合并策略三：维护Y2下界指针**

|  |
| --- |
| // 合并左右子集  Merge(points: array of Point, left: integer, right: integer, mid: integer, min\_dis: floating point numbers):  Y1 := new Point[right - left + 1] //左子集  Y2 := new Point[right - left + 1] //右子集  len1 := 0  len2 := 0  for i from mid down to left :  if points[i].x > points[mid].x - d:  Y1[len1++] := points[i]  else :  break  for i from mid + 1 to right :  if points[i].x < points[mid].x + d :  Y2[len2++] := points[i]  else :  break  quicksort(Y1, 0, len1 - 1, 'y') // 对Y1根据纵坐标排序  quicksort(Y2, 0, len2 - 1, 'y') // 对Y2根据纵坐标排序  low\_point := 0 // Y2下界指针  for i from 0 to len1 :  // base\_y1为矩形下边界，base\_y2为矩形上边界，base\_x为矩形右边界  base\_y1 := Y1[i].y - min\_dis  base\_y2 := Y1[i].y + min\_dis  base\_x := Y1[i].x + min\_dis  flag := 1 // 标记  for j from 0 to len2 :  if (flag = 1 and Y2[j].y > Y1[i].y) // 出现第一个纵坐标大于Y1[i]的点Y2[j]  low\_point := j // 更新指针  flag := 0 // 改标记，说明指针已更新，本次遍历Y2无需再更新  // Y2中在矩形右侧和下面的点直接跳过  if Y2[j].x >= base\_x or Y2[j].y <= base\_y1 :  continue  // 计算在矩形内的点对距离  else if Y2[j].x < base\_x and Y2[j].y > base\_y1 and Y2[j].y < base\_y2 :  new\_dis = Distance(Y1[i],Y2[j])  min\_dis = min(min\_dis, new\_dis) // 更新最短距离  // 若Y2中的点的纵坐标大于矩形上边界，则停止循环  else if Y2[j].y >= base\_y2:  break  delete[] Y1  delete[] Y2  return min\_dis // 返回最短距离 |

1. **算法测试结果及效率分析**
2. **蛮力法**

根据10组随机测试样本运行得到的运行时间，得到实际平均运行时间。

理论值的计算公式如下：

根据公式和实际平均运行时间，计算得到理论运行时间。

|  |  |  |
| --- | --- | --- |
| 数据规模n | 蛮力法实际运行时间/秒（s） | 蛮力法理论运行时间/秒（s） |
| 100000 | 41.228600 | 41.228600 |
| 200000 | 164.306200 | 164.914400 |
| 300000 | 369.605900 | 371.057400 |
| 400000 | 656.596100 | 659.657600 |
| 500000 | 1026.045400 | 1030.715000 |
| 600000 | 1513.008000 | 1484.229600 |
| 700000 | 2052.516000 | 2020.201400 |
| 800000 | 2683.069000 | 2638.630400 |
| 900000 | 3390.075000 | 3339.516600 |
| 1000000 | 4170.013000 | 4122.860000 |

理论效率曲线和实际效率曲线

图表, 折线图

描述已自动生成

通过观察理论效率曲线和实测效率曲线，不难发现，理论值与实际值误差较小，且图像符合O()的二次曲线。

1. **分治法**

**合并策略一：部分蛮力**

根据10组随机测试样本运行得到的运行时间，得到实测平均运行时间。

理论值的计算公式如下：

根据公式和实测平均运行时间，计算得到理论运行时间。

|  |  |  |
| --- | --- | --- |
| 数据规模n | 分治法策略一实际运行时间/秒（s） | 分治法策略一理论运行时间/秒（s） |
| 100000 | 0.069600 | 0.069600 |
| 200000 | 0.149800 | 0.147581 |
| 300000 | 0.248600 | 0.228725 |
| 400000 | 0.346200 | 0.311923 |
| 500000 | 0.464600 | 0.396648 |
| 600000 | 0.564800 | 0.482591 |
| 700000 | 0.671400 | 0.569546 |
| 800000 | 0.797800 | 0.657368 |
| 900000 | 0.918800 | 0.745948 |
| 1000000 | 1.059000 | 0.835200 |

理论效率曲线和实测效率曲线

通过观察理论效率曲线和实测效率曲线，可以发现，理论值与实际值之间的误差随着数据规模的增大而增大，图像也基本符合的曲线。误差的变化可能是因为随着数据规模的增大，区域Y中的点的个数也随之增大，规模可能会略大于，导致合并的复杂度也略大于，最终这个分治算法的复杂度也略大于。

**合并策略二：维护一个计算区域**

根据10组随机测试样本运行得到的运行时间，得到实测平均运行时间。

理论值的计算公式如下：

根据公式和实测平均运行时间，计算得到理论运行时间。

|  |  |  |
| --- | --- | --- |
| 数据规模n | 分治法策略二实际运行时间/秒（s） | 分治法策略二理论运行时间/秒（s） |
| 100000 | 0.073400 | 0.073400 |
| 200000 | 0.149600 | 0.155638 |
| 300000 | 0.245400 | 0.241212 |
| 400000 | 0.316000 | 0.328953 |
| 500000 | 0.438800 | 0.418304 |
| 600000 | 0.527400 | 0.508940 |
| 700000 | 0.596200 | 0.600642 |
| 800000 | 0.680600 | 0.693259 |
| 900000 | 0.799000 | 0.786675 |
| 1000000 | 0.924200 | 0.880800 |

理论效率曲线和实测效率曲线

通过观察理论效率曲线和实测效率曲线，可以发现，理论值与实际值之间的误差较小，图像也基本符合的曲线。

**合并策略三：维护Y2下界指针**

根据10组随机测试样本运行得到的运行时间，得到实测平均运行时间。

理论值的计算公式如下：

根据公式和实测平均运行时间，计算得到理论运行时间。

|  |  |  |
| --- | --- | --- |
| 数据规模n | 分治法策略三实际运行时间/秒（s） | 分治法策略三理论运行时间/秒（s） |
| 100000 | 0.070600 | 0.070600 |
| 200000 | 0.143800 | 0.149701 |
| 300000 | 0.230200 | 0.232011 |
| 400000 | 0.296800 | 0.316404 |
| 500000 | 0.406600 | 0.402347 |
| 600000 | 0.481800 | 0.489525 |
| 700000 | 0.558200 | 0.577729 |
| 800000 | 0.619600 | 0.666813 |
| 900000 | 0.735400 | 0.756665 |
| 1000000 | 0.844800 | 0.847200 |

理论效率曲线和实测效率曲线

通过观察理论效率曲线和实测效率曲线，可以发现，理论值与实际值之间的误差较小，图像也基本符合的曲线。

1. **各算法间运行时间的比较**

通过观察，我们得到本次实验的结论：在求最近点对的问题中各算法的效率从高到低为基于维护一个下界指针的分治法>基于维护一个计算区域的分治法>部分蛮力的分治法>>蛮力法。蛮力法的时间复杂度到达，显然其效率无法和时间复杂度到达的分治法相比。基于三种不同合并策略的分治法虽然时间复杂度均为，但显然通过层层优化，它们的运行效率也有了高低之分，也侧面说明每一次的优化确实能加快算法的运行效率，且该优化效果会随着数据规模的增大而逐渐凸显。

1. **经验总结**

在本次实验里，在利用分治法求解最近点问题中，采用了逐步优化的方法展开实验。

第一次基于部分蛮力的合并策略的分治法来求解问题，但在实验过程中不难发现，部分蛮力的合并策略会产生很多不必要的计算，因为这些计算并不能对最终答案做出贡献，并且还会降低搜索的效率。

为了解决部分蛮力合并策略的缺陷，在策略二中对此进行了优化，通过划分出一个合理的计算区域来避免位于计算区域外不必要的计算，通过实验确实能发现该计算区域的划分明显地提高了搜索效率，为此，我还通过证明计算区域内最多只需要计算5个点对，进一步验证了该策略的正确性。虽然如此，该策略仍有优化的空间。每次在Y2中搜索位于计算区域中的点时，都是从Y2区域最底部开始遍历的，但通过观察我们能发现，实际在每次搜索中，计算区域在平面中不断地抬高，故位于计算区域内Y2的点的纵坐标也应该不断地抬高，而位于计算区域下的点，显然是可以不需要遍历的，遍历它们反而会降低搜索效率。因此，合并策略三就通过维护一个下界指针来解决该问题。

在合并策略三中，通过每次搜索Y2的过程中，不断地更新下界指针，令下次搜索可以直接从下界指针开始搜索，无需从底部开始遍历。通过实验，该优化也确实再一次提高了算法的运行速度。

观察最终的结果可知，从最开始的部分蛮力的分治法到两次优化后的分治法，在一百万的数据规模测试下，两次优化后的分治法明显比部分蛮力的分治法运行速度上提高了20%。说明对基于算法原本的框架，一步一步对细节进行层层优化的策略是可行的。

|  |
| --- |
| 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。