华东师范大学计算机科学与软件工程学院上机实践报告

课程名称：数据结构与算法实践	**年级**：21	**实践成绩**：
指导教师：王丽萍	**姓名**：覃翊茹	
上机实践名称：**查询与 K 近邻图**	**学号**：10215101524	**实践日期**：2022 春

一、内容与设计思想

任务一：商户地理位置查询：

随着智能手机的普及，地理信息在诸如高德地图、大众点评、饿了么等 App 中得到广泛的应用，此次数据结构期末大作业将模拟实际生活中的查询需求，完成基于地理信息和文本信息的查找任务。问题的说明如下：系统中已经收集到许多商户的信息，每家商户包括以下三项信息：

- 位置(x,y)，x>0 && y>0；
- 商家名称；12 位 A-Z 字符串，不含小写；
- 菜系， 6 位 A-Z 字符串，不含小写；

查询任务：用户输入自己感兴趣的商家名称如 KFC，程序输出 KFC 的位置信息。在此保证查询成功时结果唯一，当查询的信息不存在时，查询失败，输出 NULL。

【输入】

第 1 行：商户的数量 m 和查询的数量 n，m 和 n 为整数，均不超过 10^9；

第 2-（m+1）行：商户的信息，包括商家名称，位置 x，位置 y 和菜系；

最后的 n 行：查询的信息，即每行包括一家商户的名称；

【输出】

输出应该为 n 行，每一行对应的是每次查询的结果，即商户的位置。

例如：

【输入】

5 3

JEANGEORGES 260036 14362 FRENCH

HAIDILAO 283564 13179 CHAFINGDIS

KFC 84809 46822 FASTFOOD

SAKEMATE 234693 37201 JAPANESE

SUBWAY 78848 96660 FASTFOOD

HAIDILAO

SAKEMATE

SUBWAY

【输出】//查询的商家位置

283564 13179 // HAIDILAO

234693 37201 // SAKEMATE

78848 96660 // SUBWAY

请根据本学期学习的知识，设计算法实现查询功能，并尝试分析算法的空间复杂度和时间复杂度，可结合数据规模、原始数据的特性等分析查询影响因素等。

1）数据规模：200 个商家、4000 个商家、$8*10^5$ 个商家等等；

2）数据特性：每个规模的数据包含 1 组按商家名称升序，1 组按商家名称降序，10 组随机数据共 12 组数据集；

3）查询任务中的 1 组数据的查找时间说明，在本次实践中 1 组数据的查找时间是指：对该组数据中的每个商家查询 1 次后所需的平均时间。例如：对于 200 个商家的数据，需要分别统计每个商家查询时间（200 次查询），再计算 200 次查询的平均值。

4）任务说明：统计每组数据下的平均查询时间；

5）书写要求：若采用教材内的算法实现查询，仅仅需要说明所用算法名称；若实践过程中涉及到自己设计的数据结构或者书本外的知识请在“实验记录和结果”中说明算法的基本思想。

任务二：K 近邻图

K近邻问题在机器学习和数据管理等领域有着重要的应用，例如：机器学习邻域的k近邻分类、数据管理领域的k近邻查询等。本任务要求求解混合-k近邻图连通（无向图）的最小k值。问题的描述为：给定n个欧几里得空间的点，计算使其构造的混合-k近邻图连通（无向图）的最小k值。在此混合k近邻图的边生成的规则是：当空间中的点q是点p的k最近邻，同时点p也是点q的k最近邻时，p和q有一条边。

【参考材料】K近邻图的相关定义　　KNN的相关工作

【输入格式】

第一行以空格分隔的整数n，代表顶点个数, 0 <= n <= 4000

第二行到第n+1行是顶点的坐标x和y，均为浮点数, 0<= x,y <= 10000

【输出格式】

使得混合-k近邻图连通的最小k值

请根据本学期学习的知识，设计算法实现 k 值的计算，并尝试分析算法的空间复杂度和时间复杂度，可结合数据规模、原始数据的特性等分析查询影响因素等。

1）数据规模：100、500、1500 等等；数据在空间中随机分布。为了客观起见，一个数据规模下可以产生多组数据，然后计算其平均值。

2）任务说明：统计每组数据下的平均查询时间；

书写要求：若采用教材内的算法，仅仅需要说明算法名称；若实践过程中涉及到自己设计的数据结构或者书本外的知识请在“实验记录和结果”中说明算法的基本思想。

三、本地实验环境（CPU，内存，操作系统，编程语言，编译器）

CPU：AMD Ryzen 5 5600H with Radeon Graphics

内存：16 GB (3200 MHz)

操作系统：Windows

编程语言：C++

编译器：Code::Blocks

四、实验记录和结果

任务一记录：

任务 1::实验记录 1（对应 *03_OJ 超时 97AVL 时间测试.cpp*）	
数据存储结构：	AVL Tree 存储
查找算法：	AVL 树 查找
数据规模	默认是 3 组规模（200，4000，800000）
数据分组	默认是 36 组
是否有课堂外的算法	否

对应 main 函数内容：

```
318  int main()
319  {
320      LARGE_INTEGER Frequ;
321      LARGE_INTEGER nBegin;
322      LARGE_INTEGER nEnd;
323
324      ifstream fin("rand_800000_10.txt");
325      AVL_tree mytree;
326      int sum,sumq;     fin>>sum>>sumq;
327
328      QueryPerformanceFrequency(&Frequ);
329      QueryPerformanceCounter(&nBegin);//获取开始的时刻
330
331      while(sum--){
332          string s1,s2,s3,s4;
333          fin>>s1>>s2>>s3>>s4;
334          record x(s1,s2,s3);
335          mytree.insert(x);
336      }
337
338      while(sumq--){
339          string tmp;
340          fin>>tmp;
341          mytree.retrieve(tmp);
342      }
343
344      QueryPerformanceCounter(&nEnd);
345      //获得结束的时刻
346      double time = (nEnd.QuadPart - nBegin.QuadPart)/(double)Frequ.QuadPart;
347
348      printf("total time(s):%lf s\n",time);
349
350      return 0;
351  }
352
```

任务 1::实验记录 2（hash+list 对应 *04_0J 超时 93hash_list 顺序搜索_时间测试.cpp*）	
数据存储结构：	采用*哈希表的思想*和*链表*结合的方式 具体实现方法： 将餐厅店名的前两个字母作为哈希表的索引，例：AA—0、AB—1、BA—26、BB—27； 对于冲突的项，采用链表的结构进行按字母顺序的存储：判断①是否比第一项小②是否比最后一项大③都不是，则从头开始一次寻找插入位置
查找算法：	① ：先由餐厅名字的前两个字母得到索引值 ② ：在对应组内进行*顺序查找*
数据规模	默认是 3 组规模（200，4000，800000）
数据分组	默认是 36 组
是否有课堂外的算法	没有参考外界资料

对应 main 函数内容：

```cpp
int main()
{
    LARGE_INTEGER Frequ;
    LARGE_INTEGER nBegin;
    LARGE_INTEGER nEnd;

    ifstream fin("rand_800000_10.txt");
    int sum,qsum;    fin>>sum>>qsum;

    QueryPerformanceFrequency(&Frequ);
    QueryPerformanceCounter(&nBegin);//获取开始的时刻

    while(sum--){
        record data;
        string tmp;
        fin>>data.name>>data.addr1>>data.addr2>>tmp;
        data.index = string2num(data.name);
        insert(data);
    }

    while(qsum--){
        string target;
        fin>>target;
        retrieve(target);
    }
    QueryPerformanceCounter(&nEnd);
    //获得结束的时刻
    double time = (nEnd.QuadPart - nBegin.QuadPart)/(double)Frequ.QuadPart;

    printf("total time(s):%lf s\n",time);
    //printf("total time(ms):%lf ms\n",time*1000);

    return 0;
}
```

任务 1::实验记录 3（最终代码对应 *01 最终 AC 代码_AVL.cpp*）	
数据存储结构：	采用*哈希表的思想*和 *AVL 树结构*结合的方式 具体实现方法： 将餐厅店名的前两个字母作为哈希表的索引，例：AA—0、AB—1、BA—26、BB—27； 对于冲突的项，采用 AVL Tree 的结构进行存储（即课堂练习中的 AVL tree）
查找算法：	③ ：先由餐厅名字的前两个字母得到索引值 ④ ：在对应组内进行 *AVL Tree 的查找*
数据规模	默认是 3 组规模（200，4000，800000）
数据分组	默认是 36 组
是否有课堂外的算法	没有参考外界资料

对应 main 函数内容：

```
int main()
{
    AVL_tree mytree[680];
    int sum,sumq;    scanf("%d %d",&sum,&sumq);
    char target[13];

    while(sum--){
        record x;
        scanf("%s %d %d %s",x.name,&x.addr1,&x.addr2,target);
        int pos=getnum(x.name);
        mytree[pos].insert(x);
    }

    while(sumq--){
        scanf("%s",target);
        int pos=getnum(target);
        mytree[pos].retrieve(target);
    }
    return 0;
}
```

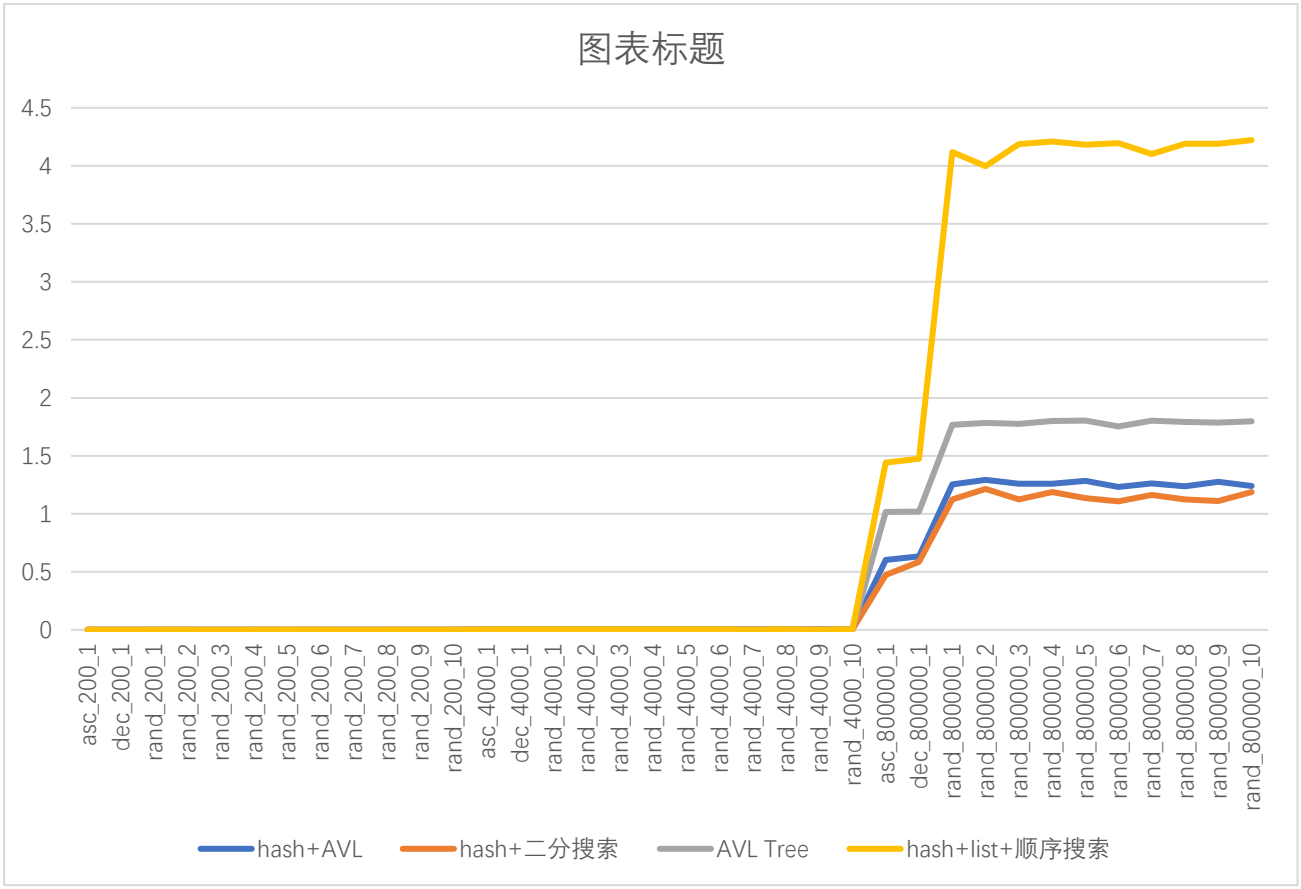
任务 1::实验记录 4（通过的另一版本对应 *02AC_vector_hash 版本*）	
数据存储结构：	采用*哈希表的思想*和利用 vector 工具结合的方式 具体实现方法： 同样将餐厅店名的前两个字母作为哈希表的索引，例：AA—0、AB—1、BA—26、BB—27； 对于冲突的项，采用 vector 的顺序结构进行顺序存储（与记录 2 中插入方法一样，判断①是否比第一项小②是否比最后一项大③都不是，则从头开始一次寻找插入位置）
查找算法：	① ：先由餐厅名字的前两个字母得到索引值 ② ：采用 forgetful 版本的*二分查找*
数据规模	默认是 3 组规模（200，4000，800000）
数据分组	默认是 36 组
是否有课堂外的算法	参考了 vector 迭代器的使用方法

对应 main 函数内容：

```
93  int main()
94  {
95
96      int sum,qsum;    cin>>sum>>qsum;
97
98      vector<vector<record >> table(680);
99      while(sum--){
100         record new_one;
101         scanf("%s %d %d %s",new_one.name,&new_one.addr1,&new_one.addr2,target);
102         int pos=s_getnum(new_one.name);
103         insert(table[pos],new_one);
104     }
105     while(qsum--){
106         scanf("%s",target);
107         int pos = s_getnum(target);
108         recursive_search(table[pos],0,table[pos].size());
109     }
110     return 0;
111 }
112
```

备注：一开始直接采用 clock()求时间，导致 200 数据组计时太短。于是重新采用老师参考代码中的方法，具体在代码文件中有标注。

测试各组数据时间（s）结果折线图：



测试各组数据时间（s）结果详细表格：

测试文件 / 时间(s)	asc_200_1	dec_200_1	rand_200_1	rand_200_2	rand_200_3	rand_200_4
hash+AVL	0.0001812	0.0001542	0.0001701	0.0001521	0.0001531	0.0001492
hash+二分搜索	0.000587	0.000532	0.000563	0.000558	0.000567	0.000523
AVL Tree	0.00018	0.000181	0.000187	0.000185	0.000188	0.000186
hash+list+顺序搜索	0.000144	0.00014	0.000142	0.000141	0.000139	0.000157

测试文件 / 时间(s)	rand_200_5	rand_200_6	rand_200_7	rand_200_8	rand_200_9	rand_200_10
hash+AVL	0.000141	0.0001575	0.0001511	0.0001531	0.0001496	0.0001485
hash+二分搜索	0.00054	0.000548	0.00054	0.000541	0.000485	0.000506
AVL Tree	0.000188	0.000188	0.000176	0.00019	0.000189	0.00018
hash+list+顺序搜索	0.000159	0.000145	0.00014	0.000143	0.000145	0.000142

测试文件 / 时间(s)	asc_4000_1	dec_4000_1	rand_4000_1	rand_4000_2	rand_4000_3	rand_4000_4
hash+AVL	0.0023803	0.0026282	0.0025584	0.0024906	0.0025066	0.0025346
hash+二分搜索	0.002836	0.002915	0.002824	0.003092	0.00306	0.002956
AVL Tree	0.003991	0.004	0.004039	0.0038	0.003775	0.004006
hash+list+顺序搜索	0.002397	0.0025	0.00259	0.00263	0.00286	0.00305

测试文件 / 时间(s)	rand_4000_5	rand_4000_6	rand_4000_7	rand_4000_8	rand_4000_9	rand_4000_10
hash+AVL	0.002533	0.002555	0.0025446	0.0025165	0.0025667	0.0025098
hash+二分搜索	0.002949	0.003005	0.002986	0.003106	0.002996	0.002942
AVL Tree	0.003856	0.004073	0.003879	0.004	0.003969	0.003878
hash+list+顺序搜索	0.002789	0.00311	0.002177	0.0025	0.00225	0.00269

测试文件 / 时间(s)	asc_800000_1	dec_800000_1	rand_800000_1	rand_800000_2	rand_800000_3	rand_800000_4
hash+AVL	0.6030048	0.63412	1.255226	1.292729	1.258505	1.258229
hash+二分搜索	0.473703	0.58706	1.125447	1.214557	1.124429	1.1888
AVL Tree	1.016967	1.019425	1.767594	1.783934	1.774538	1.80124
hash+list+顺序搜索	1.441379	1.475216	4.11897	3.99756	4.187189	4.20861

测试文件 / 时间(s)	rand_800000_5	rand_800000_6	rand_800000_7	rand_800000_8	rand_800000_9	rand_800000_10
hash+AVL	1.283591	1.23298	1.262711	1.23791	1.276071	1.23877
hash+二分搜索	1.135525	1.10882	1.164112	1.124363	1.109363	1.186533
AVL Tree	1.80375	1.753787	1.803012	1.792589	1.786421	1.797125
hash+list+顺序搜索	4.182001	4.196357	4.10057	4.19108	4.18964	4.221748

任务二记录：

备注：在所有记录尝试中，都进行了计算距离的初步处理，在后面记录表格中不再一一说明了。初步处理计算距离思路如下：因为 A 与 B 之间的距离就是 B 与 A 之间的距离，所以采用倒三角这一模式进行计算。

例：　计算 0 号→1 号的距离之后，将距离直接赋值给 1 号→0 号的存储空间中。

以此类推;

采用 for 循环时

即：

```
for(int i=0; i<n; ++i){
        for(int j=i+1; j<n; ++j){
                //计算 distance[i][j];
```

```
                distance[j][i] = distance[i][j];
            }
        }
```

任务 2::实验版本 1(对应 02KNN_AVL 超时.cpp)	
数据存储结构：	对这个有 N 个点的图，采用 AVL Tree 存储点与点之间的距离——AVL[n]。AVL[i]记录第 i 个点与剩下的其他（i-1）个点的距离。AVL 树的每一个叶子 node 都有存储相互之间的距离和（从而使整个结构有序）
算法：	整体上使用 BFS 算法，广度优先搜索。 K 从 1 开始遍历； 0 号点首先入队； 每一轮：已知 i 号点的第 1~k 个最近的点 逐一判断这 1~k 个点的第 1~k 个相近的点中是否有 i 点;若有则将 i 点入队，无——continue 进行下一轮循环，判断第 i+1 号点。 最后队列 empty，判断遍历过的点的和是否为 n； 若==n——图连通；否——k++继续新一轮更大 k 的判断。
数据规模	N=3/4/50/66/70/80/89/100/500/655/1500/5000/9999
是否有课堂外的算法	否
附加说明	因 AVL 树的搜索树性质，编写了 KthSmallest（k）函数，找到第 k 小的点，具体实现如下图：

```
441  void AVL_tree::KthSmallest(int k,record &get)
442  {
443      node *cur = root;
444
445      while( cur ){
446          int lsum = getSize(cur->left);
447
448          if(lsum < k-1){
449              cur = cur->right;
450              k = k-lsum-1;
451          }
452          else if(lsum == k-1)
453              break;
454          else
455              cur = cur->left;
456      }
457      get = cur->data;
458  }
```

对应 main 函数内容较长，在.cpp 文件中展示

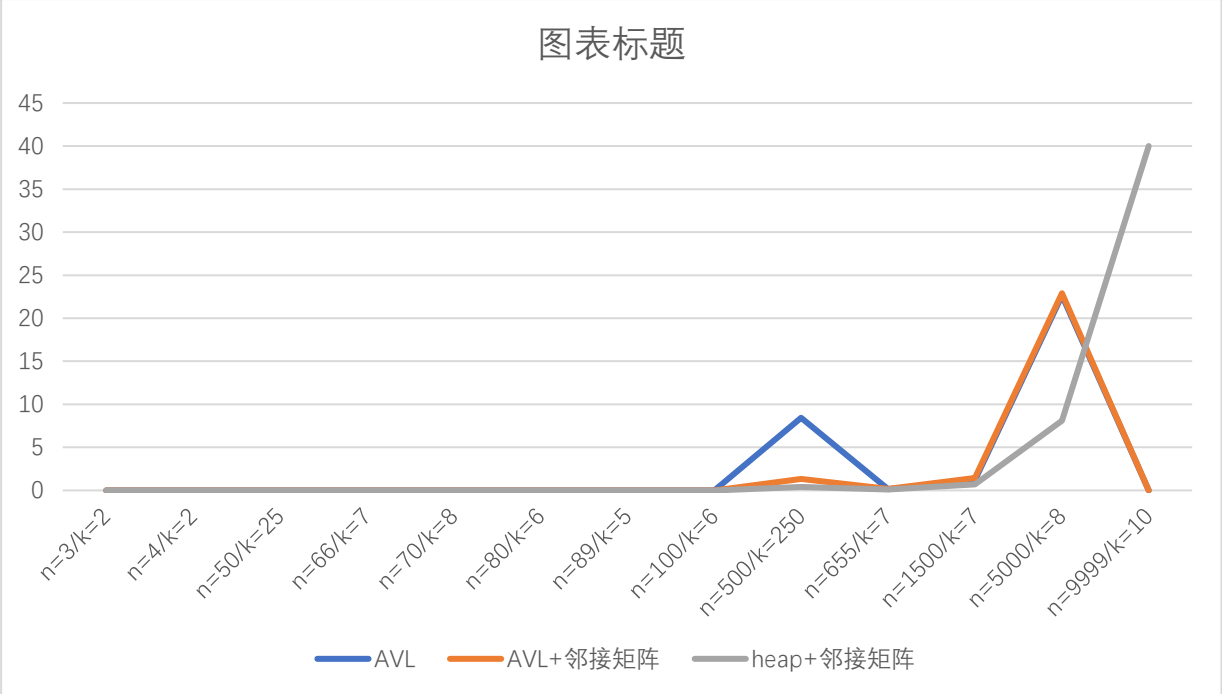
根据测试时间结果显示，本方法受到 k 的大小的影响很大，所以应该寻找能够尽量减小 k 的影响的方法。时间结果由于当时对图的内容还不是很了解，没有想到邻接矩阵这一存储结构;于是在版本二进行改进。

任务 2::实验版本 2(对应 *03KNN_AVL_邻接矩阵超时.cpp*)	
数据存储结构：	同版本 1
算法：	整体上使用 *BFS 算法*，广度优先搜索。 K 从 1 开始遍历； 0 号点首先入队； 每一轮：更新邻接矩阵，这样每次只用检查新的第 k 近的点是否互相是 k 近邻，从而得出有没有这条边。 这样在进行 BFS 中间就无需进行判断 k 近邻点步骤，只需调用邻接矩阵中对应 ij 位置是否为 true；应该能够节约很大一部分时间。
数据规模	N=3/4/50/66/70/80/89/100/500/655/1500/5000/9999
是否有课堂外的算法	否

表现依然不好...于是我开始从存储结构找问题，我想到了大根堆这一结构，同时可以 heapsort 得到距离的从小到大排列，**也便于访问**。决定尝试一下。

任务 2::实验版本 3 • AC(对应 *01KNN_heap_AC.cpp*)	
数据存储结构：	采用 *heap 大根堆*，建立 heap[n]存储点与点之间的距离。 heap [i]记录第 i 个点与剩下的其他（i-1）个点的距离。Heap 大根堆的每一个叶子 node 都有存储相互之间的距离和（从而使整个结构有序）
算法：	整体上仍然使用 *BFS 算法*，广度优先搜索。 K 从 1 开始遍历； 0 号点首先入队； 每一轮：更新邻接矩阵，每次只用检查新的第 k 近的点是否互相是 k 近邻，判断有没有这条边。这样在进行 BFS 中间就无需进行判断 k 近邻点步骤，只需调用邻接矩阵中对应 ij 位置是否为 true。 同时访问直接转变为下标访问，不用进行移动！
数据规模	N=3/4/50/66/70/80/89/100/500/655/1500/5000/9999
是否有课堂外的算法	否

测试各组数据时间（s）结果折线图：



测试各组数据时间（s）结果详细表格：

	n=3/k=2	n=4/k=2	n=50/k=25	n=66/k=7	n=70/k=8	n=80/k=6	n=89/k=5
AVL	0.000053	0.000081	0.001646	0.001279	0.001583	0.001672	0.00179
AVL+邻接矩阵	0.000058	0.000071	0.001442	0.001434	0.001668	0.001875	0.002012
heap+邻接矩阵	0.0001	0.0001	0.001322	0.001624	0.00211	0.002112	0.00249
	n=100/k=6	n=500/k=250	n=655/k=7	n=1500/k=7	n=5000/k=8	n=9999/k=10	
AVL	0.002378	8.448198	0.156352	1.185992	22.6461	no	
AVL+邻接矩阵	0.002488	1.349224	0.187453	1.434491	22.8988	no	
heap+邻接矩阵	0.003021	0.407233	0.12316	0.701198	8.09817	40	

五、结论

Task1：

200、4000 组合都不是大的问题，主要在于八十万这一组数据，数据量庞大，还有升序、倒序、乱序，各种组合。只用一个 AVL 树的时候，树会很深，需要的处理也多，所以是有很大优化空间的。单纯只用哈希表，冲突处理也很有讲究，要有序排列下来便于查询；除了冲突处理，查询的时候也要尽可能节省时间，于是我选择了二分查找，相对于顺序查找的 O(n)，二分查找能够控制在 log2 的复杂度范围内即使在八十万这一组数据的考验下，也不会耗费太多时间

Task2：

一开始没有建图，每一轮更新 k 之后都要重复之前的 k 已经做过的判断这样累加下来，重复而无效的次数数不胜数，所以需要通过中间的建图这一步骤来节约判断耗费掉的时间。

同时如果使用 heapsort，对第 k 近的点就可以之间通过下标索引进行访问了，而不需要像 AVL 树这样还需要向下查找。

总之，本次大作业就是经过重重超时一步一步向上爬，对查找算法更加熟悉，对哈希/AVL/heap 这些存储结构更加了解，在思想上也收获了很多。

心得体会：

首先，在做任务一的过程中，始终在和超时奋战…过程很艰辛，但是我也收获了很多。遇到问题的时候确实应该先分析分析，我有什么地方是好的，我应该保留下来；我觉得我有什么地方好像不太好——那是为什么不太好，进入深度的分析（还有什么地方可以改进，为什么可以这么改进）。

在第一个任务中得到的这一体会，让我在第二任务中，相对于第一个任务节约了比较多的紧盯屏幕的时间和精力，相反，是更多的在大脑中的思考，我觉得这对于平时的写代码也是很重要的，如果毫无头绪就开始写效率是非常低下的，需要在头脑中有一个大概的逻辑再开始。

还有一点，不能把自己的思维固化在一个方面了，就比如说任务 1 中 AVL 有着良好的表现，我就在任务二中首先采用了 AVL，通过 task2 的时间测试可以很明显的看出这是没有必要的，因为观察 AVL 的结构、将其与 heap 结构对比可以看出任务二中用它并不是很合适。

同时，经过本次大作业的训练，我对排序、查找的认识更加深刻，体会到在庞大的数据量下，不同的算法的差距可以如此大，而一个好的算法，可以为我们的生产生活、甚至是科学上的进步做出如此大的突破。