

数据库期中设计实验报告

实验名称: Flyish 课程设计

院系: 数据科学与计算机学院软件工程

班级: 教务三班 软工六班

组别: 第六组

小组成员: **16340251 谢冰澄**

16340198 孙肖冉

16340196 苏依晴

一. 实验目标

建立一个数据库，用于快速返回一个查询的 k 个近似的最近邻

二. 功能

1. 建立数据库 2. 处理查询 3. 数据库动态维护

三. 实验过程

(1.) 建立数据库

A) 实验内容:

1. 读取数据，将数据按页存储在磁盘；
2. 数据预处理，将处理后的数据按页存储在磁盘；
3. 产生投影矩阵，将投影矩阵按页存储在磁盘；
4. 将处理后的数据进行投影和必要的处理，将哈希后的数据按页存储在磁盘。

B) 算法设计思路:

首先明确一些数据结构:

帧(页)的形式: 存储向量 N 个, 以及它们对应的 id ; 并在页的结尾存储一数组 $check$, 用来统计向量的存在情况(为 0 则对应的槽数中的向量不存在, 为 1 则对应槽数的向量存在)。因为每一页的存储空间有限, 根据向量维度的大小, 每一页中可存储的向量数量 N 不同, 具体数值, 将在最后部分将以表格的形式列出。

读取的方式, 申请一个可以存储 30 页的缓存空间(经过尝试, 申请 40 页的空间, 电脑无法运行程序, 直接崩掉)用二维数组进行读取, 以页码数为横坐标, 页内包含的向量 ID 和内容向量以及 $check$ 数组的内容为纵坐标; 写入的方式, 写入磁盘时, 同样以页的形式写入, 每次写 30 页。

在进行投影矩阵中根据题目的要求设计出具体投影矩阵(高斯投影矩阵, 果蝇投影矩阵), 并利用矩阵乘法得到投影向量, 因为得到的数据维度有变化, 所以需要重新考虑具体页面的存储, 在得到投影向量时, 主要用到了 `rand()` 函数。高斯投影矩阵的存储方式是直接将向量存入磁盘中, 而果蝇投影向量因为只有 0,1, 所以为了方便存储和计算我们只存储了为 1 的下标的位置。但因为果蝇投影矩阵的数据过于大, 所以, 我们利用了分次读取再合并的方式进行处理。

对经过果蝇投影矩阵投影的产生的向量进行处理。首先是进行 `random` 的处理, 随机产生 32 个一定范围内的数, 然后以此作为下标进行降维; 第二步是 `WTA` 处理, 我们仍然是采

用 `qsort` 函数对向量中的数据进行排序，然后选取其中的 32 个最大数保留，其余变为 0；第三部分是在 WTA 的基础上进行 BINARY 操作，原本设想是将 WTA 与 BINARY 两步同时进行，后来发现这样会使得缓存区内容过多，导致运行的十分缓慢。这一部分主要还是运用了文件读写的方面的知识。得到的数据仍然全部存储进磁盘中，而非只存储有效数位。

类型	维度	每页存储的向量数
原始数据	784	20
预处理数据	784	20
高斯投影数据	32	481
果蝇投影数据	7840	2
果蝇_random	32	481
果蝇_WTA	7840	2
果蝇_BINARY	7840	2

C) 核心算法的代码：

1. 以二进制方式写入数据： `fwrite(buffer_pages, sizeof(buffer_pages), 1, fout)`
//buffer_page 即为内存中的设定数组（包含 30 页内容）；
2. 以二进制方式从文本文件中读取数据： `fscanf(fin, "%f", &buffer_pages[i][v*v_size + d]);`//在一个大循环中，只截取了 fscanf 的用法
3. 对数据进行预处理：//先读取 30 页的数据，对其进行分析，然后存入磁盘

`fread(buffer_pages, sizeof(buffer_pages), 1, fin)`

//读取了 30 页信息；

```
for(int v = 0; v < v_per_page; v++){
    float sum = 0.0;
    for(int d = 1; d <= dimension; d++){
        sum += buffer_pages[p][v * v_size + d];
    }
    float xishu = product / sum;
    for(int d = 1; d <= dimension; d++){
        buffer_pages[p][v*v_size + d] *= xishu;
    }
}
```

//对数据进行预处理

```
if(fwrite(buffer_pages, sizeof(buffer_pages), 1, fout) != 1){
    printf("Error fwrite.\n");
    return false;
}
```

//存入文件中

4. 产生高斯投影矩阵
- ```
for(int i = 0 ; i < K; i++){
 for(int j = 0 ; j < dimension ; j++){
 float u1 = rand()%100/(double)101;
 while(u1 == 0.0){
```

```

 u1 = rand()%100/(double)101;
 }
 float u2 = rand()%100/(double)101;
 float z = (sqrt(-2*log(u1))*cos(2*PI*u2));
//根据公式进行计算得出高斯投影矩阵，在一开始时忘记避免 u1 为 0 的情况，出现了数据
溢出的现象。

```

```

 printf("%f ",z);
 Gauss_Matrix[i][j] = z;
 }
 puts("\n");
}
}

```

5. 得到高斯投影向量:

```

for(int write_did = 1; write_did <= K; write_did++){

 for(int d = 1; d < dimension; d++){

 reflect_pages[write_pid][write_vid * Gauss_v_size + write_did]
+= buffer_pages[p][v*v_size + d] * Gauss_Matrix[write_did - 1][d-1];
 }
}

```

//利用了矩阵的乘法

6. 得到果蝇投影向量:

```

srand((unsigned)time(NULL))
for(int row = 0; row < i; row++){
 printf("\n row = %d\n", row);
 for(int col =0; col < j ; col++){
 int num = rand()%(dimension-1)+1;

 if(1!=fwrite(&num,sizeof(int),1,fout)){
 puts("error\n");
 fclose(fout);
 return false;
 }
 }
 // printf("\n");
}
}
}

```

7. WTA 与 binary 处理时用到的 qsort 函数:

```
int cmp (const void* a, const void* b){
 return (*(struct Node *)a).distance > (*(struct Node *)b).distance ? 1: -1;
}
qsort(store, v_range, sizeof(store[0]), cmp);
```

8. WTA 将前 32 位数保留, 其余归零实现方法:

```
qsort(store, Fly_dimension, sizeof(store[0]), cmp); //得到从小至大排列的数组
for(int d = 1; d <= Fly_dimension - K; d++)
{
 outpages[p][IDoffset + (int)store[d-1].id] = 0;
 //置零
}
for(int d = Fly_dimension - K + 1; d <= Fly_dimension; d++)
{
 outpages[p][IDoffset + (int)store[d-1].id] = Flypages[p][IDoffset +
(int)store[d-1].id];
 //保留
}
```

## (2) 处理查询

### A) 实验内容:

目标: 给定一个查询点, 表示为原始向量, 完成以下查询

- 1) 返回原始点集中距离查询点点最近的 K 个点 (KNN) 的 id
- 2) 返回处理后的点集中距离查询点点最近的 K 个点 (KNN) 的 id (注意要对查询点做预处理)
- 3) 使用高斯投影矩阵哈希, 返回查询点的近似 KNN 的 id
- 4) 使用果蝇投影矩阵哈希, 返回查询点的近似 KNN 的 id
- 4) 使用果蝇投影矩阵哈希后, 用 random 做处理, 返回查询点的近似 KNN 的 id
- 6) 使用果蝇投影矩阵哈希后, 用 WTA 做处理, 返回查询点的近似 KNN 的 id
- 7) 使用果蝇投影矩阵哈希后, 用 binary 做处理, 返回查询点的近似 KNN 的 id

距离计算使用欧式距离。

重复实验 1, 改变哈希后向量的维度 k, 以 k 为横坐标, 分别以实验 1 的 3 个指标为横坐标绘制 3 个曲线图。

K 的取值变化为: 2, 4, 8, 16, 32, 64.

对实验结果进行分析。(15 分)

注意，实验 2 改变 k 的时候，需要重新生成哈希后的数据。

## B) 算法的设计思路

在这一阶段主要实现的是 KNN（最近邻）搜寻算法，以及计算 MAP

KNN 最近邻域搜索主要的实现方式：我们随机选择一个向量，并计算每一个向量与其的欧式距离，我们将这些向量的 id 与 distance 用一个结果 Node 存储，在比较完毕之后，利用 qsort（）函数以 distance 为比较对象，将向量的 id 从小到大排列，最终取前 200 个向量 id 即为结果。读取方式依然是每次读取 30 页，读取完毕之后在进行操作。

mAP 的计算方式：我们将得到的 KNN 最近邻域搜索得到的结果存储在数组中，并且利用数组进行计算，根据所给公式计算出实际的 mAP。

## C) 核心算法的伪代码

1. 计算欧式距离：

```
for(int d = 1; d <= dimension; d++){
 distance_add_product += (float)pow((mydatad[d-1] -
 buffer_pages[p][v*v_size + d]), 2);
}
```

2. qsort 函数的应用（与上文中的一致，皆为从小至大排列）

```
int cmp (const void* a, const void* b){
 return (*(struct Node *)a).distance > (*(struct Node *)b).distance ? 1: -1;
}
qsort(store, v_range, sizeof(store[0]), cmp);
```

3. Node 结构

```
struct Node{
 float id;
 float distance;
};
```

4. 从文件中随机选取向量：

```
int random_id = rand()%v_range;//产生随机数

int page_id = random_id/v_per_page;//计算所在页面
int slot_id = random_id%v_per_page;//计算所在槽数

for(int j = 0 ; j < page_id ; j++){
 fread(store[0], sizeof(float)*p_size, 1, fin);
}
for(int j = 0 ; j < slot_id ; j++){
 fread(myid, sizeof(float)*v_size, 1, fin);
```

```

 }
 fread(myid, sizeof(float)*v_size, 1, fin); //读取所需向量

```

#### 5. 遍历所有向量并计算欧式距离

```

int count = 0 ;
for(int j = 0 ; j < p_range/buffer_page_max ; j++){

 fread(store, sizeof(float)*p_size*buffer_page_max, 1, fin);
 for(int k = 0 ; k < buffer_page_max ; k++){
 for(int l = 0 ; l < v_per_page ; l++){
 if(count==v_range) {
 break;
 } //当读取向量达到 v_range 时，停止
 finall[count].id= store[k][l*v_size];

 float dis = 0 ;
 for(int d = 1 ; d < dimension ; d++){
 // printf("%f",myid[d]);

 dis +=
 (store[k][l*v_size+d]-(float)myid[d])*(store[k][l*v_size+d]-(float)myid[d]);
 }
 // printf("%f",dis);
 finall[count].distance = sqrt(dis);
 count++;
 // printf("%d\n ",count);
 }
 }

}

```

#### 6. 计算 map 的代码;

```

int main() {

 float result;
 srand((unsigned)time(NULL));
 for(int i = 0 ; i < random_time ; i++){
 int random_id = rand()%v_range; //产生随机数
 float test[200];
 float test2[200];
 if(knn_search1(random_id, test)&&knn_search2(random_id, test2)) {
 //两个函数返回的是搜寻结果
 result += map(test, test2); //得到 AP
 }
 }
}

```

```

 }
 printf("%d\n", i);
 }

 result = result / (float) random_time; //得到 mAP
 printf("%f", result);

}

float map(float test1[] , float test2[]){
 float sum = 0 ;
 for(int i = 0 ; i < 200 ; i++){
 for(int j = 0 ; j < 200 ; j++){
 if(test1[i-1]==test2[j-1]){
 sum += (float) (i+1) / (float) (j+1);
 break;
 }
 }
 }
 float chushu = 200;
 float num = sum / chushu;
 return num;
}

```

### (3) 数据库动态维护

#### A) 实验内容:

1. 插入一个新的向量（对该向量预处理→作投影→保存到磁盘）
  2. 删除一个向量
- 涉及磁盘页的管理，空闲槽的跟踪，向量 id 的分配等问题。

#### B) 算法的设计思路

插入一个新向量：插入向量时，我们首先检测磁盘页的尾部数据，是否等于已知磁盘页可容纳的最大向量数，如果等于则继续检测，如果不等于则检测槽数为 0 的位置，并且在该位置插入数据（有关向量的 id 问题，我们选择在插入时跳过 id 插入即保留原来的 id），如果



没有空闲槽，则不再进行新页面的创建，插入失败。

删除一个向量：删除向量时只需输入向量的 id，然后进行判断，如果不存在，则删除失败，如果存在，则将向量所在页的 check 数组中下标为删除向量对应的槽数“1”的值变成“0”；并将 check 数组中存储页面总向量的数字减少一。具体实现是将该页的数据读取，然后进行操作，并将结果重新输入文件原位置中。

### C) 核心算法的具体代码：

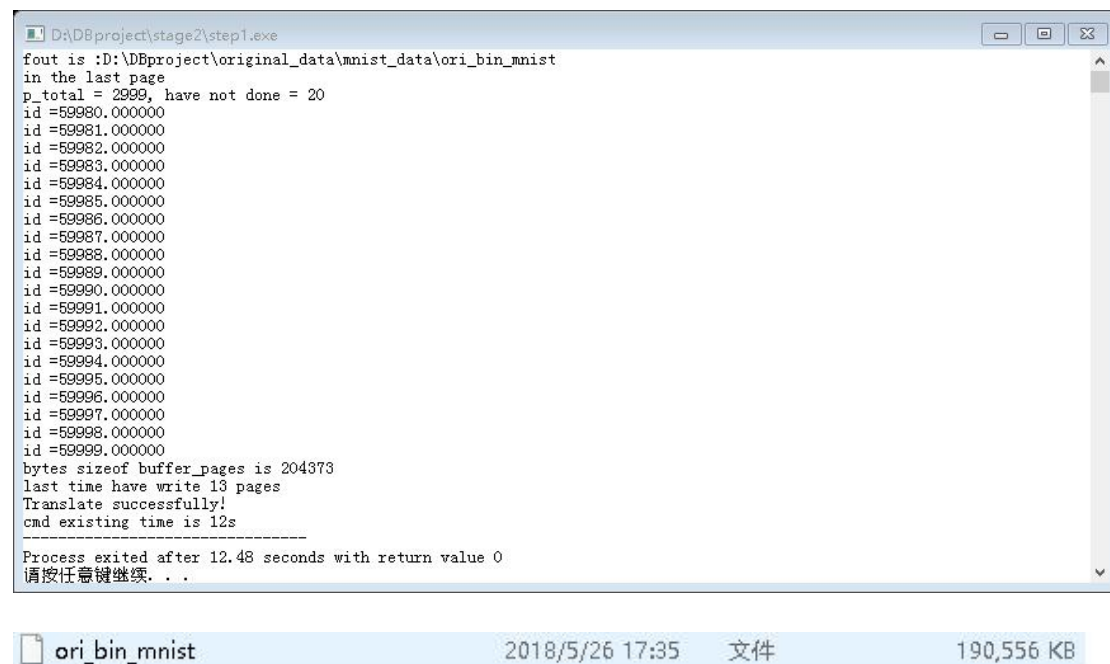
#### 1. 删除向量时 check 数据的处理：

```
buffer_pages[mypage_in_buffer][v_size*v_per_page + myid_in_buffer] = 0.0;
//使之变成 0
buffer_pages[mypage_in_buffer][p_size - 1]--;
//总向量数减一
```

## 四：实验结果及分析

### (1) 建立数据库的实验结果及分析：

#### 1. 读取原始数据并以二进制的方式存进新文件中（ori\_bin\_minst）：



```
D:\DBproject\stage2\step1.exe
fout is :D:\DBproject\original_data\mnist_data\ori_bin_minst
in the last page
p_total = 2999, have not done = 20
id =59980.000000
id =59981.000000
id =59982.000000
id =59983.000000
id =59984.000000
id =59985.000000
id =59986.000000
id =59987.000000
id =59988.000000
id =59989.000000
id =59990.000000
id =59991.000000
id =59992.000000
id =59993.000000
id =59994.000000
id =59995.000000
id =59996.000000
id =59997.000000
id =59998.000000
id =59999.000000
bytes sizeof buffer_pages is 204373
last time have write 13 pages
Translate successfully!
cmd existing time is 12s

Process exited after 12.43 seconds with return value 0
请按任意键继续. . .
```

|               |                 |    |            |
|---------------|-----------------|----|------------|
| ori_bin_minst | 2018/5/26 17:35 | 文件 | 190,556 KB |
|---------------|-----------------|----|------------|

#### 2. 预处理后

```
D:\DBproject\stage2\step2.exe
Are you sure that:
file mnist_data has:
 60000 vectors
 784 dimensions
 20 vectors per page
buffer process has:
 10 pages in buffer
 3000 pages totally
 15721 p_size
 785 v_size
process times is 300
?y/n: y
fin is D:\DBproject\original_data\mnist_data\ori_bin_mnist
fout is D:\DBproject\pre_process_data\mnist_data\pre_process_mnist
Pre process successfully!
time is 1s
请按任意键继续. . .
```

 pre\_process\_mnist 2018/5/29 11:02 文件 19,038 KB

## 2. 生成高斯投影矩阵并存储，进行投影并存储

```
D:\DBproject\stage2\Form_Gauss_Matrix.exe
.083273 0.954318 -1.416614 -0.473395 -1.494416 2.460961 1.267359 -0.582372 0.890043 0.987144 1.040474 -1.345345 -0.02547
9 -0.123689 0.154070 -1.063730 -0.391041 1.758642 -0.857058 0.797554 -0.349100 2.631483 0.478824 0.430511 -0.373038 -2.0
00490 -1.397561 0.010901 -0.381371 -0.752982 -0.344026 -0.190199 0.441947 2.155155 0.618593 0.316948 -1.440802 -1.302891
0.305471 -0.689069 -0.131089 -1.449532 0.117133 -2.365945 -0.929246 0.463054 -0.549827 -0.376072 1.829660 -1.344632 1.2
94844 -0.385780 -0.773913 -0.052178 0.090071 -1.713988 -1.460421 0.586179 0.241958 -0.278047 0.203116 0.866194 -1.440802
0.380450 0.317147 -1.345221 -1.230558 0.288921 0.616295 -0.592255 -0.453627 -0.673371 -0.277935 -1.044156 0.193843 0.59
5864 -0.845004 -0.181566 0.750321 1.430585 -0.349100 2.404481 0.990521 -0.143934 -1.344632 0.624338 0.264712 1.534001 0.
338123 -0.402756 -1.771485 -0.065966 -0.451483 0.355675 -1.827670 0.090071 -0.862406 -1.214463 -3.036662 -1.269669 1.631
705 -0.441531 -1.438294 1.234035 -0.465628 -0.181566 -0.066098 -1.451643 -1.629408 -0.301730 -1.650808 -0.351610 -0.9767
97 1.855025 0.717981 -1.748369 -1.887105 0.569641 0.112283 0.087413 0.809560 1.184747 -0.110831 0.003110 -1.748642 -0.33
8701 -0.592796 -0.176261 -0.049615 1.296742 -0.585087 0.201619 0.938648 0.846055 -0.400280 -0.209355 -0.504510 0.197409
-0.487893 1.143412 -0.182100 -2.640446 -1.113065 0.385618 0.948188 -0.923554 -0.341550 1.732401 2.199043 -0.351610 -0.51
2700 0.833082 1.824404 0.221396 -0.971919 0.754642 -0.753318 -0.443503 0.776280 -0.968317 0.965505 0.096434 -0.693822 -1
.368012 1.450867 1.706950 0.438525 1.860525 -0.524691 -0.626775 -1.897618 -1.279750 0.385872 0.724532 0.396208 1.835949
0.330403 -0.671422 -0.695750 1.250333 0.776280 1.262142 -0.446467 -0.920711 -0.540868 0.957484 0.282089 1.433603 -1.0989
21 -0.645290 0.211502 -0.113099 0.028884 1.692778 -0.858506 -1.872760 -0.829890 0.405468 1.244133 -0.968772 -1.302891 -1
.358425 1.596694 1.313336 0.493075 0.100298 -1.477134 -0.651168 1.411443 -0.573510 -0.178998 0.485484 -0.125198 0.691999
0.782643 -0.782611 -0.184201 -0.968670 0.308846 -1.782172 0.133635 1.046272 -0.666739 -0.409799 0.406907 -1.410910 -0.3
16190 1.350538 -1.051519 0.870210 0.384333 -0.982282 -0.782678 -0.714392 -0.191268 0.039072 -2.289780 0.364340 -0.382760
-2.481192 -0.241266 -0.822666 1.813809 -0.719677 2.089535 0.896542 -0.341718 1.418817 0.578529 -1.874328 0.677942 0.241
958 2.714440 -0.187699 0.675699 -0.553662 1.492843 -0.421845 0.161811 0.189889 -1.469222 0.378281 -1.810812 1.261751 0.8
05805 0.137682 -0.693518 -0.039938 -0.409302 -1.353124 -0.449201 0.170177 0.177080 1.595217 -1.020071 -1.327162 -2.26957
6 1.018813 0.047164 0.809385 -0.463822 1.827534 -1.429185 -0.113099 -1.756780 -2.269576 0.243675 0.624524 0.713347 -1.22
5821 -0.526550

Forming successfully!

Process exited after 2.111 seconds with return value 0
请按任意键继续. . .
```


Gauss reflect seccessful!

 mnist\_Gauss\_Matrix 2018/5/30 11:43 文件 98 KB  
 mnist\_Gauss\_Reflect 2018/5/30 11:46 文件 7,667 KB

3. 生成果蝇投影矩阵并存储，生成投影向量并存储

|                                                                                                       |                 |    |          |
|-------------------------------------------------------------------------------------------------------|-----------------|----|----------|
|  mnist_Flyish_Matrix | 2018/5/30 12:02 | 文件 | 2,389 KB |
|-------------------------------------------------------------------------------------------------------|-----------------|----|----------|

4. 生成 WAT 处理后的果蝇哈希并存储

|                                                                                          |                 |    |
|------------------------------------------------------------------------------------------|-----------------|----|
|  FlyWTA | 2018/5/31 19:19 | 文件 |
|------------------------------------------------------------------------------------------|-----------------|----|

5. 生成 BINARY 处理后的果蝇哈希并存储

|                                                                                             |                 |    |              |
|---------------------------------------------------------------------------------------------|-----------------|----|--------------|
|  FlyBinary | 2018/5/31 19:26 | 文件 | 1,838,086... |
|---------------------------------------------------------------------------------------------|-----------------|----|--------------|

6. 生成 Random 处理后的果蝇哈希并存储

|                                                                                             |                 |    |          |
|---------------------------------------------------------------------------------------------|-----------------|----|----------|
|  FlyRandom | 2018/5/31 19:42 | 文件 | 8,306 KB |
|---------------------------------------------------------------------------------------------|-----------------|----|----------|

## (2) 处理查询

有关 KNN 查询时间结果截图，放在了文件 image 中。

有关 WTA 查询时间结果截图，放在了文件 imag 中。

表格：

|                    | 查询所需时间(s) | mAp(以原始数据的KNN作为真正KNN) | mAP (以使用处理后数据的KNN作为真正的KNN) |
|--------------------|-----------|-----------------------|----------------------------|
| 真正的KNN (使用原始数据)    | 70        |                       |                            |
| 真正的KNN (使用处理数据)    | 69.2      |                       |                            |
| 近似KNN (gaussian)   | 4         | 0.994168              | 1.005784                   |
| 近似KNN (fly)        | 589.6     | 0.09859               | 0.1002                     |
| 近似KNN (fly+random) | 4         | 0.064212              | 0.060849                   |
| 近似KNN (fly+WTA)    | 423.6     | 49.465908 (?)         | 49.059608                  |
| 近似KNN (fly+binary) | 534.5     | 0.136382              | 0.123005                   |

对应的数字：

|               |  |          |           |
|---------------|--|----------|-----------|
| 70            |  |          |           |
| 69.2          |  |          |           |
| 4             |  | 0.994168 | 1.005784  |
| 589.6         |  | 0.09859  | 0.1002    |
| 4             |  | 0.064212 | 0.060849  |
| 423.6         |  |          | 49.059608 |
| 49.465908 (?) |  |          |           |
| 534.5         |  | 0.136382 | 0.123005  |

## (3) 动态维护

删除向量的测试结果;

```
D:\DBproject\stage2\delete.exe
60000 vectors
784 dimensions
20 vectors per page
buffer process has:
10 pages in buffer
3000 pages totally
15721 p_size
785 v_size
and the id to delete is: 0
y/n?:

fin is D:\DBproject\original_data\mnist_data\ori_bin_mnist
page number = 1 needs buffer turn : 1 pagen_umber_in_buffer = 0 id_in_buffer = 0
y/n? :
y
find id = 0.000000

In the file, the check list is :

check[0] = 0.000000 check[1] = 1.000000 check[2] = 1.000000 check[3] = 1.000000 check[4] = 1.000000 check[5] =
1.000000 check[6] = 1.000000 check[7] = 1.000000 check[8] = 1.000000 check[9] = 1.000000 check[10] = 1.000000
check[11] = 1.000000 check[12] = 1.000000 check[13] = 1.000000 check[14] = 1.000000 check[15] = 1.000000 ch
eck[16] = 1.000000 check[17] = 1.000000 check[18] = 1.000000 check[19] = 1.000000 total check = 19.000000
Delete successfully!

cmd exsting time is 10s

Process exited after 10.78 seconds with return value 0
请按任意键继续. . .
```

## 五. 反思与总结

在处理接口方面，我们组实现的很差，使得代码文件过多，有些只需修改目标路径即可，因为没有留出接口，使得我们的实现都是直接重复进行源文件的编写。

### 小组分工：

谢冰澄 算法设计，算法实现，算法测试，查阅资料  
孙肖冉 算法设计，算法测试，编写报告，查阅资料  
苏依晴 查阅资料