

```
keywordstyle=, morekeywords=string, emph=self, emphstyle=,  
commentstyle=, stringstyle=,  
frame=single, framesep=1em  
columns=flexible, numbers=left, numbersep=2em, number-  
style=,
```

# 数据结构实验报告

姚苏航 PB22061220

## 1 问题描述

### 1.1 实验题目

利用哈希表统计两源程序的相似性。

### 1.2 基本要求

对于两个 C 语言的源程序清单，用哈希表的方法分别统计两程序中使用 C 语言关键字的情况，并最终按定量的计算结果，得出两份源程序的相似性。

### 1.3 测试数据

事先给出的 file 文件夹，包含关键词表和三份源程序文件，程序之间有相近的和差别大的。文件内容详见附录 B。

## 2 需求分析

1. 扫描给定的源程序，累计在每个源程序中 C 语言关键字出现的频度 (为保证查找效率，建议自建哈希表的平均查找长度不大于 2)，通过这种方式扫描两个源程序，提取其特征向量。
2. 通过计算向量  $X_i$  和  $X_j$  的相似值来判断对应两个程序的相似性，相似值的判别函数计算公式为：

$$S(X_i, X_j) = \frac{X_i^T \cdot X_j}{|X_i| \cdot |X_j|} \quad (1)$$

通过这种方式，可以初步判断两个源程序的相似性，如图 1 所示。其中  $S$  反映了两向量的夹角的余弦，当  $S$  趋近于 1 时，两向量夹角趋于 0，即两向量趋于相似，反之亦然。

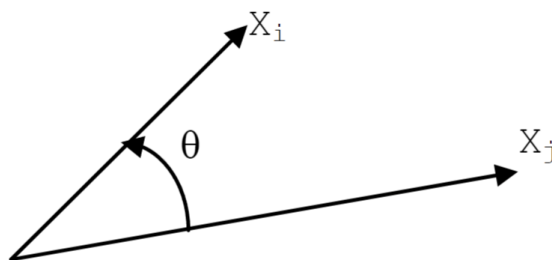


图 1: Similarity

3. 在有些情况下,  $S$  不能很好地反映两向量的相似性, 还需要进一步的考虑。例如, 在  $S$  接近于 1 时, 两向量的模的差距不能很好地被反映, 如图 2 所示。因此引入几何距离  $D$ , 用于反应两向量终点间的距离。

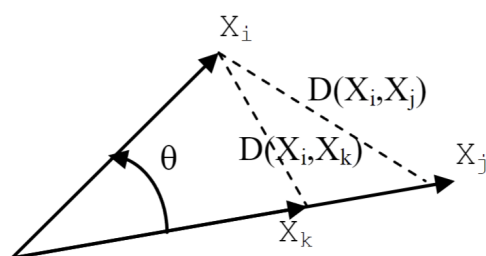


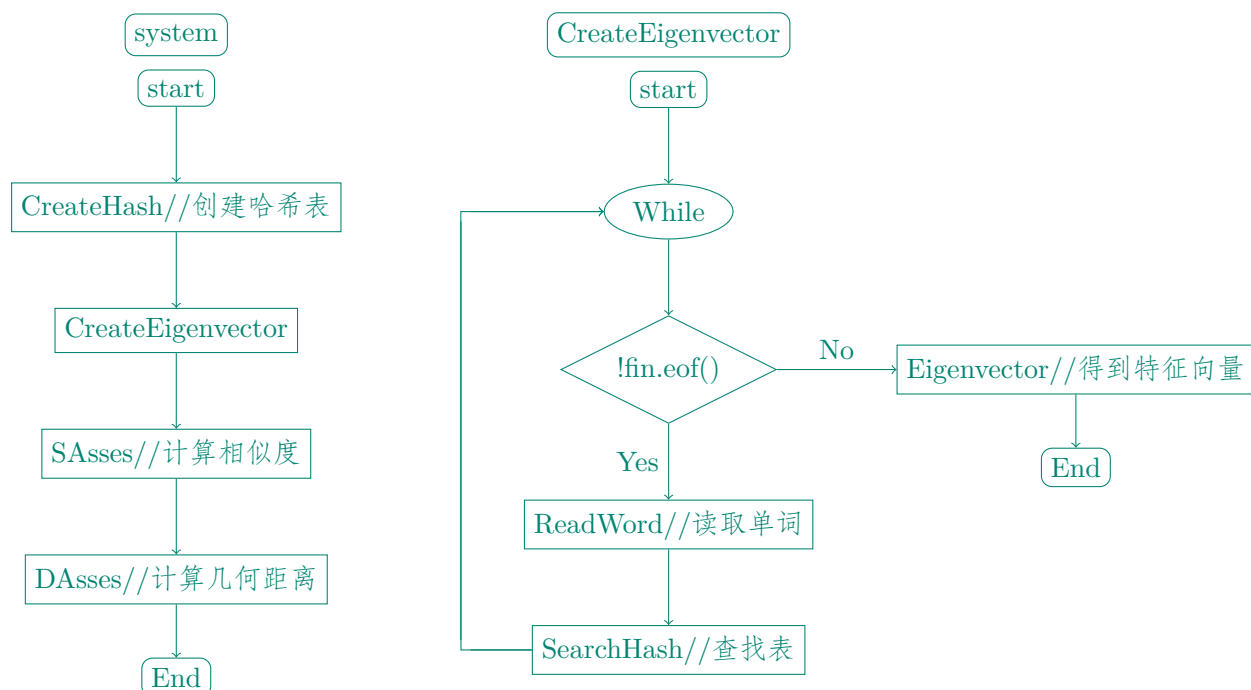
图 2: Distance

4. 通过分别比较很相近和差别很大的三个源代码, 实践这种方法的有效性。

### 3 概要设计

#### 3.1 所用到的数据结构及其 ADT

#### 3.2 主程序流程及其模块调用关系



### 4 详细设计

#### 4.1 实现概要设计中的数据结构 ADT

```
typedef struct {
```

```

    KeyType key;
    Datatype Data;    // 记录关键字出现的次数
} ElemType;          // 包含关键字和数据

typedef struct LNode {    // 哈希表结点
    ElemType data;        // 查找表单元
    LNode *next;          // 后继
} *LHptr;

typedef struct {
    LHptr *elem;
    int count;    // 记录数
int size;        // 容量
} LHashTable;    // 链地址存储法

```

## 4.2 实现每个操作的伪码，重点语句加注释

### 4.2.1 哈希表的创建与查找

---

**Algorithm 1** 创建哈希表

---

```
1: function CREATEHASH(LHashTable &H)
2:   ifstream fin("../file/keywords.txt", ios :: in);
3:   keyword[0]  $\leftarrow$  new char[30];
4:   for each i in [1, 22] do
5:     keyword[i]  $\leftarrow$  new char[10]; ▷ 为每个节点分配空间
6:   end for
7:   fin.get(keyword[0], 26); ▷ 将无用内容读入 keyword[0], 关键字从 1 开始存储
8:   i  $\leftarrow$  1, j  $\leftarrow$  0;
9:   while !fin.eof() do
10:    fin.get(ch);
11:    if ch  $\geq$  97 & ch  $\leq$  122 then ▷ 判断是否为需要读取的关键词
12:      keyword[i][j]  $\leftarrow$  ch; ▷ 读取关键词
13:      j ++;
14:    else
15:      keyword[i][j]  $\leftarrow$  '\0';
16:      j  $\leftarrow$  0, i ++; ▷ 为读取下一个关键词做准备
17:    end if
18:  end while
19:  H.size  $\leftarrow$  43; H.count  $\leftarrow$  0
20:  H.elem  $\leftarrow$  new LHptr[H.size] ▷ 初始化为所有结点指针的头指针
21:  for each i in [0, H.size - 1] do
22:    H.elem[i]  $\leftarrow$  new LHNode; ▷ 为单个结点分配空间
23:    H.elem[i]->next  $\leftarrow$  nullptr;
24:  end for
25:  for each i in [1, 17] do
26:    n  $\leftarrow$  Hash(keyword[i]);
27:    p  $\leftarrow$  new LHNode; p->data.key  $\leftarrow$  new char[10]; ▷ 分配空间
28:    p->next  $\leftarrow$  nullptr; p->data.Data  $\leftarrow$  0; ▷ 计数
29:    p->data.key  $\leftarrow$  keyword[i] ▷ 记录 key 值
30:    if !H.elem[n]->next then
31:      H.elem[n]->next  $\leftarrow$  p;
32:    else
33:      p->next  $\leftarrow$  H.elem[n]->next;
34:      H.elem[n]->next  $\leftarrow$  p;
35:    end if
36:  end for
37:  fin.close();
38: end function
```

---

---

**Algorithm 2** 查找哈希表

---

```
1: function SEARCHHASH(LHashTable H, int n, KeyType key)
2:    $p \leftarrow H.elem[n] \rightarrow next;$ 
3:   while  $p$  do
4:     if !strcmp(key,  $p \rightarrow data.key$ ) then                                ▷ 等于则返回 0
5:        $p \rightarrow data.Data++;$ 
6:       break;
7:     end if
8:      $p \leftarrow p \rightarrow next;$ 
9:   end while
10: end function
```

---

#### 4.2.2 向量的生成与运算

---

**Algorithm 3** 读取文件生成向量

---

```
1: function CREATEEIGENVECTOR(LHashTable H, int X[], const string& FileAddress)
2:   InitHash();                                                                ▷ 初始化哈希表
3:   while !fin.eof() do
4:     ReadWord();                                                            ▷ 读取单词;
5:      $n \leftarrow Hash();$                                                     ▷ 计算哈希值;
6:     SearchHash();                                                          ▷ 搜索哈希表;
7:   end while
8:   for each  $i$  in  $[0, 43]$  do
9:      $p \leftarrow H.elem[i] \rightarrow next;$ 
10:    while  $p$  do
11:       $X[j++] \leftarrow p \rightarrow data.Data;$                             ▷ 用哈希表的记录生成特征向量
12:       $p \leftarrow p \rightarrow next;$ 
13:    end while
14:  end for
15:  fin.close();
16: end function
```

---

---

**Algorithm 4** 计算几何距离

---

```
1: function DASSES(const int X1[], const int X2[])
2:    $k, i, m \leftarrow 0;$ 
3:   for each  $i$  in  $[0, 16)$  do
4:      $m += (X1[i] - X2[i]) * (X1[i] - X2[i]);$                                 ▷ 计算 X1 和 X2 的差
5:   end for
6:    $k = sqrt(m);$                                                             ▷ 计算几何距离 D return  $k;$ 
7: end function
```

---

---

**Algorithm 5** 计算相似度

---

```
1: function SASSES(const int X1[], const int X2[])
2:    $n1, n2, i, m \leftarrow 0$ ;
3:   for each  $i$  in  $[0, 16)$  do
4:      $m+ = X1[i] * X2[i]$ ; ▷ 计算 X1 和 X2 的点积
5:      $n1+ = X1[i] * X1[i]$ ;
6:      $n2+ = X2[i] * X2[i]$ ;
7:   end for
8:    $n1 = \text{sqrt}(n1)$ ; ▷ 计算 X1 的模
9:    $n2 = \text{sqrt}(n2)$ ; ▷ 计算 X2 的模
10: return  $m / (n1 * n2)$ ; ▷ 返回相似度 S
11: end function
```

---

### 4.3 主程序和其他模块的伪码

---

**Algorithm 6** 判断相似性

---

```
1: function SYSTEM(void)
2:   CreateHash( $H$ ); ▷ 根据给定关键词创建哈希表
3:   CreateEigenvector( $H, \text{SimVec}, \text{"../file/similar.c"}$ ); ▷ 读取程序生成特征向量
4:   CreateEigenvector( $H, \text{DifVec}, \text{"../file/different.c"}$ );
5:   CreateEigenvector( $H, \text{MainVec}, \text{"../file/main.c"}$ );
6:    $\text{Similarity} \leftarrow \text{SAsses}(\text{SimVec}, \text{MainVec})$ ; ▷ 计算相似度 S 和几何距离 D
7:    $\text{Distance} \leftarrow \text{DAsses}(\text{SimVec}, \text{MainVec})$ ;
8:    $\text{Similarity} \leftarrow \text{SAsses}(\text{DifVec}, \text{MainVec})$ ;
9:    $\text{Distance} \leftarrow \text{DAsses}(\text{DifVec}, \text{MainVec})$ ;
10: end function
```

---

## 5 调试分析

### 5.1 问题分析与体会

本项目工程主要分为两个部分。第一部分是哈希表的创建和查找，第二部分是根据查找结果得到向量以及对向量的数学计算等处理。

在这两部分中，文件的读取和写入是必不可少的，因此，对文件的读写操作是调试的重点。在本实验中，根据给定的文件，选用适当的函数读取文件，并且注意想要获取的内容间分隔符的处理，是第一个难点，也是调试的重点。此外，文件读取状态的判断也是一个重点。

在正确地读取文件内容后，如何处理向量也是实验重点。在一开始，只是简单地将每个读取的向量的处理步骤转换成代码，造成了代码臃肿，复用性差，并且难以调试的问题。通过对代码的重构和优化，将向量的处理步骤封装成通用的函数，使得代码的复用性大大提高，并且易于调试，在实验过程中数学运算的错误也更易发现。

在本次实验中，注释发挥了重要的作用。通过对一些细节操作的注释，大幅加快了调试的速度。在遇到问题时能够很快找到解决方案。注释也大幅提高了代码可读性，在优化代码时，注释作为参考，指明了数据初始化的状态，重要操作的目的，方便了后期的维护与修改。

通过这次实验，我锻炼了自己处理多个文件的能力，认识了注释等好的编程习惯的重要性，提升了自己对非单一文件的项目工程的函数编写封装思路的理解。它不仅加深了我对哈希表的认识，了解了哈希表在查重方面的应用，还提高了自己代码的编写能力，能够更好更快地写出容易理解且便于调试的代码。

## 5.2 时空复杂度分析

# 6 使用说明

用户将事先准备好的关键词表 (keyword.txt) 和需要统计相似性的程序放入 file 文件夹中，运行时程序将通过关键词表建立哈希表，并通过查找哈希表构建两程序的向量，最后通过判别函数计算两程序相似性和向量的几何距离。

在本项目中，使用事先给出的测试数据，准备三个编译和运行都无误的 C 程序，程序之间有相近的和差别大的，通过 similar.c 和 different.c 两个程序与 main.c 进行比较，可以直观展现出比较的效果。

# 7 测试结果

## 7.1 输入数据

输入数据从给出的测试文件中读取，读取 keyword.txt 文件生成哈希表，再分别读取 main.c,similar.c,different.c 并进行比较。

## 7.2 输出数据

输出结果显示在终端，内容如下：

X\_(../file/similar.c):

0 1 1 4 0 1 0 3 3 1 2 1 2 0 1 1 4

X\_(../file/different.c):

0 1 0 1 0 0 1 2 1 0 3 0 0 0 1 3 2

X\_(../file/main.c):

0 1 1 3 0 1 0 3 2 1 2 1 2 0 1 1 4

S\_(Sim&Main):0.988174

D\_(Sim&Main):1.41421

S\_(Dif&Main):0.740121

D\_(Dif&Main):4.89898

# A 实验源代码文件

为方便查看，附录中的链接文件均为 txt 格式

[define.h](#)

[main.cpp](#)



[OpenHashing.h](#)  
[OpenHashing.cpp](#)  
[system.h](#)  
[system.cpp](#)  
[SimAsses.h](#)  
[SimAsses.cpp](#)

## B 实验用测试文件

[main.c](#)  
[different.c](#)  
[similar.c](#)  
[keyword.txt](#)