数据结构与算法实验报告

课 程: 数据结构与算法　　班 级：* 班
姓 名:***　　学 号：**********
报告完成日期：2023 年 01 月 31 日

基于 N-最短路径的中文分词器

一、实验目的

1. 掌握图、树、线性表和哈希表等结构的基本知识和使用技术。
2. 运用所学习的数学知识。
3. 培养对问题建模和抽象的能力。
4. 培养设计和使用新工具的能力。
5. 培养自学能力。
6. 能够撰写实验 (技术) 报告，培养沟通能力。

二、需求分析

1. 程序的主要功能：将一段不带标点符号的中文文本，以词典数据为基础，做符合日常语言习惯的词语切分。例如，用户输入“今天天气真好”时，输出分词后的结果“今天/天气/真/好”。
2. 程序的交互模式：以用户和计算机的对话方式执行。
3. 程序执行的命令：
   (1) 要求用户输入待切分字串
   (2) 读取字串后经过处理，输出分词结果
   (3) 询问用户是否需要继续使用程序，若用户同意则重复 (1)(2)(3) 步骤，否则退出程序
4. 程序的用途：作为自然语言处理的早期步骤，为后期的更细致的文本分析打下基础

三、概要设计

1. 特殊数据类型

- 有向无环网。定义：ADT WordString {
  数据对象 V：V 是第一个字前、最后一个字后以及每两个字之间的节点的集合，称为顶点集。
  数据关系 R：R = {VR}，VR = {<v, w>|v, w 属于 V 且 P(v, w), <v, w> 表示从 v 到 w 的弧，P(v, w) 定义了弧 <v, w> 的信息，包括频率、费用、分词标志}
  基本操作 P：InitWordString(&S)，操作结果：初始化有向网 S。
  CreateWordString(&S, s[ ])，操作结果：读取用户输入的字串 s，计算总字数以及可在词典中查找到的词数，为每个可能的词增添一条弧。
  SegmentWordString(&S)，操作结果：使用迪杰斯特拉算法计算从第一个节点到最后一个节点的最短路径，并在路径途径节点上设置一个分词标志。
  PrintWordString(S)，操作结果：输出带有分词斜杠的字串，即分词结果。
  DestroyWordString(&S)，操作结果：销毁有向网 S。
  } ADT WordString

- Trie 树。定义：ADT Trie{
  数据对象 T：T 是词典中每一个字所组成的集合，在树中表示为一个个节点。
  数据关系 R：R={<e1,e2>|e1,e2 均属于 T，<e1,e2> 表示一条从 e1 到 e2 的边}
  基本操作 P：Trie.InitTrie()，操作结果：初始化 Trie 树；Trie.InsertTrie(&s)，操作结果：将一个词典中的词语添加进 Trie 树中；Trie.SearchTrie(s)，操作结果：搜索 Trie 树是否存储字符串 s 的信息，即词典中是否包含串 s。
  }

2. 模块化

本程序含有 4 个模块：

1. 主程序模块：void main() {
   初始化（构建词典）;
   while (true) {
   接受命令;
   处理命令;
   询问用户是否继续使用;
   if (用户决定退出) break;
   } }
2. 有向无环网单元模块：实现有向网的抽象数据结构

3. PreNode 结构单元模块：根据有向无环网确定输入字符串的 PreNode 数组

4. n 最短路径结构单元模块：根据输入字符串的有向无环网找到 n 个最短路径

各模块之间的调用关系如下：

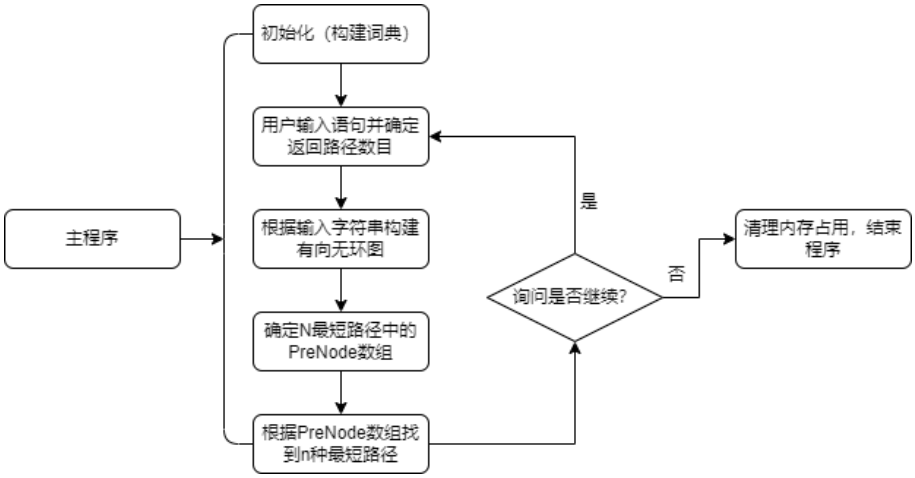

图 1: 分词器结构

四、详细设计

1. 所用宏

名称	数值	描述
OK/ERROR	1/0	状态码
LIST_INIT_SIZE	10	顺序表初始大小
LIST_INCREMENT	5	顺序表每次增量
STACK_INIT_SIZE	100	顺序栈初始大小
STACKINCREMENT	10	顺序栈每次增量
MAX_INPUT_LEN	100	字串的最大字数
MAX_LINE_LEN	100	词典一行的最大长度
MAX_PATH_NUM	30	路径的最大数量
MAX_WORD_LENGTH	25	单个词语的最大字数
ENCODE_NUM	2	使用的编码格式规定的字节数

表 1: 宏设定

2 所用全局变量

名称	类型	描述
filename	string	词典名称
total_word_num	int	词表总词数
total_word_freq	long long	词表总词频
path_id = 0	int	路径序号
Nodelist	SqList<PreNode>*[ ]	每个节点路径相关信息表
trajectory	int[ ][ ]	最终路径

表 2: 全局变量设定

3. 所定义数据类型

1. 每个字的存储结构：

```
struct character{ // 自定义数据结构，表示每个字，利用字符数组进行存储
    char ch[ENCODE_NUM+1];
    character(){
        memset(ch, 0, ENCODE_NUM+1);
    }
};
```

2. Trie 树每个节点的结构：

```
struct TrieNode{ // 自定义Trie树的基本单位，该Trie树采用二叉结构
    character* word; // 每个节点所代表的字
    int freq; // 当该节点是一个词语的尾部时，freq表示该词的词频
    TrieNode* LNode; // 左子节点，表示下一个字
    TrieNode* RNode; // 右子节点，表示与该词同级的兄弟节点
    TrieNode(char* wd, int len):freq(0){
        LNode = nullptr;
        RNode = nullptr;
        word = new character;
        strncpy(word->ch, wd, len);
    }
};
```

3. 输入字符串每个节点 PreNode 的结构：

```
typedef struct{
    int pre; // 指向该节点的上一个节点
    int pre_index; // 该路径的序号
    int dis_index; // 这条路径的总距离
    double all_dis; // 该路径在到达该点的所有路径中的排名
```

```
}PreNode; // PreNode节点记录路径相关信息
```

4. 关键函数

1. Trie 树添加词语的函数

```
Status Trie::Insert(char* s, int frequency){
    for(/*对于词语中的每一个字*/){
        if(/*当前节点左子节点是否为空*/){
        // 若无左子节点就用当前字创建一个
        }else{
            // 若有左子节点令指针指向左子节点
            // 重复在右子节点中寻找是否有匹配的节点
            // 找不到匹配的节点就在最后一个右子节点创建一个新节点储存当前词语
            }
        }
    }
    // 添加完成后在最后一个字所在节点处添加频率
}
```

2. Trie 树查找词语的函数

```
Status Trie::Insert(char* s, int frequency){
    for(/*对于词语中的每一个字*/){
        if(/*当前节点左子节点是否为空*/){
        // 若空即当前字不在Trie树上，返回ERROR
        }else{
            // 若有左子节点令指针指向左子节点
            // 重复在右子节点中寻找是否有匹配的节点
            // 找不到匹配的节点就返回ERROR
            }
        }
    }
    // 查找到返回频率
}
```

3. 构建有向无环图

```
Status dag_construction(/*Trie树，输入字符串，DAG存储邻接矩阵*/){
    for(/*对字符串每个字i*/){
        for(/*对当前字向后的每一个字j*/){
            // 在Trie树中寻找从i到j的字符串
            if(/*是否找到*/){
            // 如果找到，将频数保存在邻接矩阵中
```

```
            }else if(/*未找到*/){
            // 如果未找到就跳出循环
            }else{
            // 如果是前缀就继续找
            }
        }
        // 如果某个汉字在词典中不存在，就假设存在并设置为-1
        // 在最后一个字后面添加一个假想字便于最后输出

        // 对每个词的频数进行处理，方法为该词在词典中出现的
        // 概率的负自然对数，其中计算概率时采用拉普拉斯Add-one平滑
    }
}
```

4. 构建 PreNode 数组

```
void get_prenode(/*输入字符串DAG图邻接矩阵*/)// 将每个字看做节点
{
    for(/*每个字i*/){
        // 对到达当前节点的所有路径按距离进行排序标号
        for (/*对当前节点能够到达的所有节点j*/){
                //如果到达i有k条不同的路径，那么就在j的PreNode数组中添加k次从i到j
                    的路径，路径序号从1到k，路径距离为i的k种路径长度加上从i到j的
                    路径长度
            }
        }
    }
}
```

5. 返回 n 最短路径

```
void n_shortest_path(/*希望得到路径种类n，输入字符串*/){
    for(/*对每种路径*/){
        // 利用递归从后向前进行查找
        if(/*递归是否成功*/){
            // 如果失败，报错
        }else{
            // 如果成功，储存该路径
        }
    }
    // 按顺序输出不同分词结果
    }
}
```

6. 利用递归返回 n 最短路径

```
Status retro_back(/*当前节点序号，希望得到路径的序号*/){
    if(/*当前是否位于头节点*/){
        // 储存栈中序列
        // 再将数组中的数放回栈中，便于递归
    }else{
        if(/*路径序号大于该节点最大的节点序号*/){
            // 报错
        }
        for(/*对于当前节点所拥有的所有路径*/){
            // 将符合路径序号的前序节点入栈
            // 重复调用递归函数寻找上一节点的路径
        }
        return OK;
    }
}
```

五、调试分析

1. 读取字典时有时会将词典每一行最后的词性读取进来，经过反复调试，发现是词典文件的格式设置错误，应该设置成 GBK 格式。

2. 读取字典时有时会查不到词语，后来发现是在每一次查询结束后没有把文件指针恢复到指向文件开头，于是在查词函数最后加入了一行 rewind 函数解决这一问题。

3. 由于词典规模较大，因此我采用了 **Trie 树结构进行存储**，这样能够保证对于词语的搜索最快。

4. 在输入需要进行分词的字符串时，我忽略了 win 终端和工程文件之间编码格式的差异，导致字符串始终无法正常输入，后来我把文件格式全部改为 GBK，并在文件开头加上了 system(”chcp 936”) 的代码，使得无论用户所用电脑终端是何种格式，都可以先改为 GBK 编码，这样能够正常输入汉字。

5. 我在定义邻接矩阵和 PreNode 数组时没有处理好最后一个字，导致在输出时出现较大的偏差，后来在矩阵和 PreNode 数组添加一个元素，即最后一个字符后的“结束字”，这样使得有向无环图能够正常建立，输出的结果也是正确的。

6. 我在调试过程中发现对于较长的词典无法得知词表是否构建完成，于是我在构建词典的函数中添加了**进度条**，这样使得添加词典的过程更加可视化，对于错误也能及时处理。

7. 我发现有些时候词典中可能会出现重复的词语，于是添加了处理重复词的语句，保留词频最大的词。

8. 有时我忘记对一些变量进行初始化，导致输出异样甚至程序异常退出。后来补充了完善的初始化步骤。

9. 算法设计与分析：我采用了 **Unigram 模型**，使用词语在词典中出现的概率（该词的词频/词典总词频）的负自然对数，作为该词的权值（路径长度）。并且，在计算概率时，做了**拉普拉斯 Add-One 平滑**，这样可以使求得的最短路径就是在初始语料库中出现概率最高的分词方法。

10. 对于某些词典中没有的词，我会给它赋予词频 0，并给词典的总词数加 1。这样一来，经过拉普拉斯平滑，它将拥有一个很小的出现概率和很大但并非无穷的权值。这是为了，当用户输入的字串中有词典里没有的生僻字时，我构造的有向图不会在这里断开，通过这种方法，可以处理那些**包含标点符号**的中文语句。

六、实验总结

1. 对于数据结构中的树结构和图结构更加熟悉，对于它们的存储结构也有了更深的理解。

2. 极大地锻炼了我的编程能力和处理程序错误的能力。

3. 锻炼了我对于解决复杂问题的思维和能力。

4. 提高了我对于计算机汉字编码格式的理解。

5. C++ 编程能力还需进一步提高，有时无法快速形成有效解决方法，还需要上网查询相关资料才能解决。