实验报告

题目：编制一个使用最短路径匹配算法实现中文文本分词的程序

班级：统计学院 2018 级 4 班 姓名：曹竞轩 学号：2018202075 完成日期：2019.12.22

一、 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。

二、 概要设计

为实现上述程序功能，需要的抽象数据类型为有向网（WordString）。

1. 有向网（WordString）的定义：

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

2. 本程序含有 3 个模块：

1） 主程序模块：

```
void main() {
```

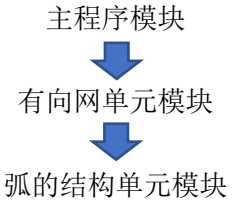
```
        初始化;
        while (true) {
            接受命令;
            处理命令;

            询问用户是否继续使用;
            if (用户决定退出) break;
        }
    }
```

2） 有向网单元模块——实现有向网的抽象数据结构

3） 弧的结构单元模块——定义有向网邻接矩阵的结构

各模块之间的调用关系如下：



三、 详细设计

1. 使用的宏

```
#define MAX_WORD_STRING_LENGTH 100 // 字串的最大字数
#define MAX_WORD_LENGTH 10 // 单个词语的最大字数
#define MAX_LINE_LENGTH 50 // 词典一行的最大长度
```

2. 使用的全局变量

```
FILE *dict = NULL; // 用来读取词典的文件指针
int total_word_freq = 0; // 词典中所有词语的总频数
int total_word_num = 0; // 词典中词语的总数
```

3. 辅助功能的函数实现

```
// 计算词典中词语的总数及所有词语的总频数
void CountTotalWordFreqAndNum() {
    while (!feof(dict)) {
        // 每次读取词典的一行，储存在 line 中
        char line[MAX_LINE_LENGTH];
        fgets(line, MAX_LINE_LENGTH, dict);

        // 将这一行的词语的频数加到词典总频数中
        int freq;
        sscanf(line, "%*s%d", &freq);
```

```
            total_word_freq += freq;

            // 词语总数加 1
            ++total_word_num;
        }

        // 将文件指针恢复到指向词典开头
        rewind(dict);
    }

    // 查询词语在词典中的频数
    int LookUpWordFreq(char *w) {
        while (!feof(dict)) {
            // 每次读取词典的一行，储存在 line 中
            char line[MAX_LINE_LENGTH];
            fgets(line, MAX_LINE_LENGTH, dict);

            // 读取这一行的词语，储存在 word 中
            char word[3 * MAX_WORD_LENGTH + 1];
            sscanf(line, "%s", word);

            // 若要查的词语与该行词语匹配上，则读取并返回该行词语的频数
            if (!strcmp(w, word)) {
                int freq;
                sscanf(line, "%*s%d", &freq);
                rewind(dict);
                return freq;
            }
        }

        // 将文件指针恢复到指向词典开头
        rewind(dict);

        // 若词典中查不到该词，则返回 0
        return 0;
    }
```

4. 有向网的邻接矩阵结构的类型

```
    typedef struct {
```

```
    int freq; // 词语在词典中出现的频数
    double fee; // 词语的"费用"
    char content[3 * MAX_WORD_LENGTH + 1]; // 词语的内容
} WordMatrix[MAX_WORD_STRING_LENGTH + 1][MAX_WORD_STRING_LENGTH + 1];
```

5. 有向网的类型与实现

```
typedef struct {
    WordMatrix words; // 邻接矩阵
    bool seg[MAX_WORD_STRING_LENGTH + 1]; // 分词标记数组
    int character_num; // 字数
    int word_num; // 词语数
} WordString;


Status InitWordString(WordString &S) {
    // 初始化邻接矩阵
    memset(S.words, 0, sizeof(S.words));

    // 初始化分词标记数组
    for (int i = 0; i < MAX_WORD_STRING_LENGTH + 1; ++i) {
        S.seg[i] = false;
    }

    // 初始化字数和词数
    S.character_num = 0;
    S.word_num = 0;

    return OK;
}

Status CreateWordString(WordString &S, char s[]) {
    // 打开词典，失败则报错并退出程序
    dict = fopen("dict.txt", "r");
    if (dict == NULL) {
        printf("词典读取失败！\n");
        exit(ERROR);
    }

    // 计算词典中词语的总数及所有词语的总频数
```

```
    CountTotalWordFreqAndNum();

    // 计算字串的字数
    S.character_num = strlen(s) / 3;

    // 对每一个字，在词典中查询其频数。若查不到，则视作查到了，并赋予其频数 0
    for (int i = 0; i < S.character_num; ++i) {
        strncpy(S.words[i][i + 1].content, s + 3 * i, 3);
        S.words[i][i + 1].freq = LookUpWordFreq(S.words[i][i + 1].content);

        // 若查不到，则视作查到了，因此将词典中词语总数加 1
        if (!S.words[i][i + 1].freq) {
            ++total_word_num;
        }
    }

    // 对每一个词，在词典中查询其频数并依此计算其"费用"
    // 计算费用的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
    for (int i = 0; i < S.character_num; ++i) {
        for (int j = 1; j < S.character_num + 1; ++j) {
            // 若词语并非单个字，那么词典中查得到则根据频数计算费用，查不到则将费用设为无穷
            if (j - i > 1) {
                strncpy(S.words[i][j].content, s + 3 * i, 3 * (j - i));
                S.words[i][j].freq = LookUpWordFreq(S.words[i][j].content);
                if (S.words[i][j].freq) {
                    ++S.word_num;
                    S.words[i][j].fee = log(total_word_freq + total_word_num) - log1p(S.words[i][j].freq);
                } else {
                    S.words[i][j].fee = INFINITY;
                }
            } else if (j - i == 1) {
                // 若词语为单个字，那么不必再重复查询频数，直接计算费用（词典中查不到的词也计算费用）
```

```
                ++S.word_num;
                S.words[i][j].fee = log(total_word_freq +
total_word_num) - log1p(S.words[i][j].freq);
            }
        }
    }

    // 关闭词典
    fclose(dict);

    return OK;
}

Status SegmentWordString(WordString &S) {
    // 记录每个节点在最短路径上的前驱节点
    int prev[S.character_num + 1];
    for (int i = 1; i < S.character_num + 1; ++i) {
        prev[i] = 0;
    }

    // 记录到每个节点的当前距离
    double path_fee[S.character_num + 1];
    path_fee[0] = 0;
    for (int i = 1; i < S.character_num + 1; ++i) {
        path_fee[i] =  S.words[0][i].fee;
    }

    // 记录到每个节点的最短路径是否已被求出
    bool final[S.character_num + 1];
    final[0] = true;
    memset(final + 1, false, S.character_num);

    while (!final[S.character_num]) {
        int v;
        double min = INFINITY;

        // 遍历所有最小路径尚未求出的节点，找到费用最低的
        for (int i = 0; i < S.character_num + 1; ++i) {
            if (!final[i] && path_fee[i] < min) {
```

```
                v = i;
                min = path_fee[i];
            }
        }

        // 将其最短路径的状态改为"已找到"
        final[v] = true;

        // 更新到剩余节点的当前距离
        for (int i = 0; i < S.character_num + 1; ++i) {
            if (!final[i] && min + S.words[v][i].fee < path_fee[i]) {
                path_fee[i] = min + S.words[v][i].fee;
                prev[i] = v;
            }
        }
    }

    // 将位于从第一个节点到最后一个节点的最短路径上的节点的分词标记改为"是"
    int index = S.character_num;
    while (prev[index] != 0) {
        S.seg[prev[index]] = true;
        index = prev[index];
    }

    return OK;
}

Status PrintWordString(WordString S) {
    // 用来记录有分词标志的节点下标的数组
    int seg_indexes[S.character_num - 1];
    // 有分词标志的节点个数
    int seg_num = 0;

    // 遍历所有节点，找到有分词标志的节点，将其下标存入 seg_indexes，每找到一个就使 seg_num 加 1
    for (int i = 1; i < S.character_num; ++i) {
        if (S.seg[i]) {
            seg_indexes[seg_num++] = i;
```

```
        }
    }

    // 用来储存要输出的分词结果的字符串
    char w[4 * MAX_WORD_STRING_LENGTH] = {0};

    // 若有分词标志的节点数不为0，则依此将字串片段与"/"号连接到w上；若为0，则直接将整个字串复制到w中
    if (seg_num) {
        strcat(w, S.words[0][seg_indexes[0]].content);
        strcat(w, "/");
        for (int i = 0; i < seg_num - 1; ++i) {
            strcat(w, S.words[seg_indexes[i]][seg_indexes[i + 1]].content);
            strcat(w, "/");
        }
        strcat(w, S.words[seg_indexes[seg_num - 1]][S.character_num].content);
    } else {
        strcpy(w, S.words[0][S.character_num].content);
    }

    printf("%s\n", w);
}

Status DestroyWordString(WordString &S) {
    // 清空字串S使用的储存空间
    memset(&S, 0, sizeof(S));

    return OK;
}
```

四、 调试分析

1. 读取字典时会失败，之后发现应该把词典文件放在cmake-build-debug文件夹中，而不是项目文件夹中。
2. 读取字典时有时会查不到词语，后来发现是在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代码。
3. 曾经因为分词标志所在的位置只可能是中间的节点而非两端的节点，于是把分词标志数组的长度设置为字串的字数 - 1。但是这样虽然节省了一点点空间，却导致我对这

个数组进行操作时的下标和节点的下标不相同，引起了一些混乱。后来决定还是将分词标志数组长度改为字串的字数 + 1，这样虽然第一个节点和最后一个节点处的数据始终为 false，是空间的小小浪费，但代码的逻辑清晰了许多。

4. 之前我省略了每个节点的前驱数组 prev 以及记录到每个节点的最短路径是否被求出的 final，采用了另一种方法：在每一次遍历未求得最短路径的节点时，将得到的最小距离节点处分词标志 seg 改为“是”，并在下一轮循环中，只考虑这个节点之后的节点（因为词语不会从后往前）。后来我发现前驱数组 prev 是不应该被省略的，这会导致我在几乎每个节点处都打下了分词标志，但并不是每一个被标志的节点都是通往结尾节点的最短路径上的节点。不过 final 数组倒确实可以省略，但后来为了代码的易懂性还是加上了。
5. 在输出分词结果时，我漏考虑了无需分词的情况，即用户输入的字串只有一个词。此时我的输出会出错。后来加入了对需要分词的节点数为 0 的情况的识别和处理。
6. 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。
7. 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（费用）。并且，在计算概率时，做了拉普拉斯 Add-One 平滑。因为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。

   对于两个或两个字以上的词语，我采取的是：在词典里有则按上述方法计算权值，词典里没有则将权值设置为正无穷；而对于单个字的词，当词典中没有它时，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。

8. 附录

源程序文件名清单：

CWS.cpp // 主程序