**课程设计报告**



题目: 中文分词系统设计

专 业： 信息安全

班 级： 1602班

学 号： U201614840

姓 名： 冯冠云

指导教师： 卢萍

日 期： 2018.3.14

成 绩：

目录

[任务设计书 3](#_Toc509338717)

[1.绪言](#_Toc509338718)

[1.1 研究意义 4](#_Toc509338719)

[1.2 系统的实现思路 4](#_Toc509338720)

[1.3 程序运行环境 5](#_Toc509338721)

[2.重要数据结构](#_Toc509338724)

[2.1 Trie树 6](#_Toc509338725)

[2.1.1 为什么使用Trie树 6](#_Toc509338726)

[2.1.2 Trie树的结构 6](#_Toc509338727)

[2.1.3 如何查询一个词语 8](#_Toc509338728)

[2.2 DAG(有向无环图) 9](#_Toc509338729)

[2.2.1 DAG的直观结构 9](#_Toc509338730)

[2.2.2 DAG的存储结构 9](#_Toc509338731)

[3.关键算法及实现](#_Toc509338732)

[3.1 Trie树的算法 10](#_Toc509338733)

[3.1.1 插入词语 10](#_Toc509338734)

[3.1.2 删除子树 11](#_Toc509338735)

[3.1.3 删除词语 12](#_Toc509338736)

[3.1.4 搜索词语 14](#_Toc509338737)

[3.2 获得DAG的算法 16](#_Toc509338738)

[3.2.1简单的DAG切分 16](#_Toc509338739)

[3.2.2 改进的DAG切分 16](#_Toc509338740)

[3.3 MMSEG分词算法 19](#_Toc509338741)

[3.1.1 MMseg规则一 19](#_Toc509338742)

[3.1.2 MMseg规则二 19](#_Toc509338743)

[3.1.3 MMseg规则三 20](#_Toc509338744)

[3.1.4 MMseg规则四 20](#_Toc509338745)

[3.1.5整个句子的切分 20](#_Toc509338746)

[3.4 MPSEG分词算法 26](#_Toc509338747)

[3.4.1算法思路 26](#_Toc509338748)

[3.4.2算法实现 26](#_Toc509338749)

[3.5 MAXSEG分词算法 29](#_Toc509338750)

[4.其他相关问题](#_Toc509338751)

[4.1 中文编码问题 30](#_Toc509338752)

[4.1.1 几种常见的编码 30](#_Toc509338753)

[4.1.2 编码的存储方式 31](#_Toc509338754)

[4.1.3 编码的检测 31](#_Toc509338755)

[4.1.4 编码的转换 33](#_Toc509338756)

[4.2 图形界面设计 34](#_Toc509338757)

[5.程序测试](#_Toc509338758)

[5.1支持文件切分与文本框内切分 36](#_Toc509338759)

[5.2字典的修改(增加与删除) 38](#_Toc509338760)

[5.2.1字典的添加 38](#_Toc509338761)

[5.2.1 删除词语 40](#_Toc509338762)

[5.3多种切分算法的支持 42](#_Toc509338763)

[5.4中文语句歧义的问题 44](#_Toc509338764)

[5.5程序的兼容性(英文与编码) 46](#_Toc509338765)

[6.编程过程中的问题](#_Toc509338766)

[6.1 一个内存访问越界Bug 48](#_Toc509338767)

[6.2 malloc与new混用造成的错误 49](#_Toc509338768)

[6.3 vector的赋值问题 49](#_Toc509338769)

[参考文献 51](#_Toc509338770)

[附录 52](#_Toc509338771)

[Trie类 52](#_Toc509338772)

[Dict类 55](#_Toc509338773)

[Decode类 55](#_Toc509338774)

[SegCore类 55](#_Toc509338775)

[Segment类 55](#_Toc509338776)

# 任务设计书

|  |
| --- |
| 课题内容  本题要求设计和实现一个中文分词系统，将任意给定的一段中文切分成一个一个单独的词。按所推荐的参考算法和数据结构通过编程实现系统，使学生进一步强化 C 语言程序设计能力，掌握动态存储分配、文件读写等高级编程，掌握  Hash 表、Trie 树等数据结构的使用方法，提高综合运用各门课程知识解决实际问题的能力。 |
| 课题任务要求  中文分词是将一段中文的字序列切分成词序列的过程。比如，“我是一名大学生”切分的结果为“我/是/一名/大学生”。中文分词系统所采用的算法较多， 一种基本的分词算法为基于字符串匹配的分词方法，这种方法需要用到词典。分词过程中需要频繁地对词典进行查找和匹配，为了提高分词速度，需要使用特殊的数据结构对词典进行索引和存储。  本系统需实现以下基本功能：   1. 词典维护功能：可从文件导入词典，对词典进行增加、删除和修改，将维护好的词典保存到文件，数据存放的格式可以自定义。 2. 索引维护功能：为了提高词典的查找和比对速度，需要对词典建立各种索引。因此，当词典发生变化后，索引需相应更新。系统启动后，索引被加载到内存，索引更新后，应采用合适的数据存放格式将索引存到硬盘，以便下次启动系统时，被加载的索引与系统上次运行时的索引完全一致。 3. 待处理中文文本的输入：待处理中文文本可以有多个段落，每个段落可以有多个句子，每个句子字数不限，还可以带各种标点符号。待处理中文文本可以通过键盘输入，也可以从文本文件导入。 4. 对输入的中文文本进行分词处理：分割后的词两两之间用“/”分隔，保持原文次序。提供保存到文本文件的功能。 5. 用户界面功能：系统需要有良好的用户界面，方便操作。用户界面可以采用简单文本菜单界面、下拉式文本菜单界面和图形菜单界面。 6. 系统还能正确切分中文文本中的英文单词、各种数字。 |

# 绪言

## 1.1 研究意义

自然语言处理是一个信息时代最重要的技术之一，简单来讲，就是让计算机能够理解人类语言的一种技术。在其中，分词技术是一种比较基础的模块。对于英文等拉丁语系而言，由于词之间有空格作为词边际表示，词语一般情况下都能简单且准确的提取出来。而中文等文字，除了标点符号以外，字之间紧密相连，没有明显的词边界，因此很难将词提取出来。分词的意义非常大，在中文中，单字作为最基本的语义单位，虽然也有自己的意义，但表意能力较差，意义较分散，而词的表意能力更强，能更加准确的描述一个事物，因此在自然语言处理中，通常情况下词(包括单字成词)是最基本的处理单位。

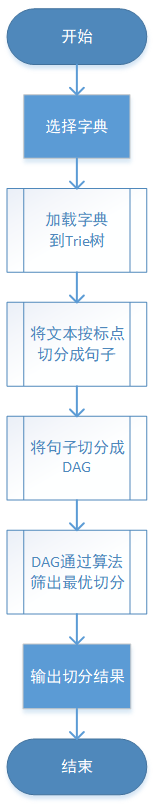
## 系统的实现思路

对于中文分词的大致思路可分为几个关键步骤，其流程图见图1-1:

1. 把字典加载到一定的数据结构(Trie树)中方便查询并加快查询速度。
2. 把一篇文章切分成句子，每个句子通过查询Trie树获得DAG(有向无环图)。即得到句子的所有词语，可以得到切分句子的所有方案。
3. 使用一定的切分方案来从DAG中选取较优的切分路径。这里的切分方案可以是最大正相匹配或者其他的切分方案。

这里用于存储字典的数据结构我选择了Trie树，对于句子的切分方案我提供了三种选择。MMSeg，是一种较最大正向匹配更为复杂也更加精确的算法。MPSeg计算最大概率路径的算法。MaxSeg，最大正相匹配算法。

由于我的界面等实现较为复杂，在画流程图的时候至画出核心的数据结构部分流程。流程图如下：



总体实现思路流程图 图1-1

## **1.3 程序运行环境**

cpu：E5-1620 3.60MHZ

内存：DDR3 24G 1333MHZ

IDE：QT5.9.2 + Visual Studio 2017pro

# 重要数据结构

## 2.1 Trie树

### 2.1.1为什么使用Trie树

Trie树是这次代码实现的关键。Trie树又称字典树。之所以要用到这种数据结构，和切分无关，只是为了提高词语的提取速度。

试想，如果每次查询词语都要遍历一遍字典，切分一个句子需要多次查询；再加上IO操作本来慢，这里面的时间耗费是巨大的。因此必须先把字典文件加载到内存。

加载到内存也是不够的，还需要有好的数据结构做支持。如果直接加载到线性表，可以认为每次查询词语的时间复杂度为O(n)。由于字典文件巨大动辄几十万个词，O(n)显然并不是一种好的查询方式。

而Trie树则是一种较常用的数据结构，其时间与词典中的词数无关，只与查询的词长有关。其时间复杂度为O(strlen)。

### 2.1.2 Trie树的结构是什么样的

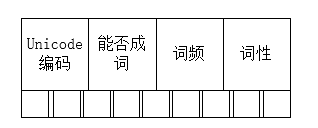
我查阅了一些Trie树的讲解，也读了一些源码，发现其具体结构的实现多种多样，也就是说不必拘泥于其具体的结构多了一点或者哪里不一样了，只要按照Trie树的思想实现即可。

首先Trie树就是一颗树，每个结点有数据域和孩子结点的指针。结构上，每个Trie树结点的数据域存储一个字符，以及一些其他信息。指针域要能够从汉字映射到指针。Trie树的根结点数据域为空，孩子指针指向各个词语的首字结点。

存储时，数据域的汉字可以保存其编码，选用ANSI或者Unicode编码较为合适，因为这两种编码中文都是两个字节，可以用一个unsigned short变量来存储。我认为Unicode编码更加合适，因为其英文部分也把ASCII码扩展成了两个字节，这样可以方便中英文的统一处理。

在孩子指针的问题上，如果使用其编码的值”直接定址”，那么每次指针域均要分配2^16个指针空间，虽然这样能加快查找速度，但会耗费巨大的空间。使用哈希函数映射到指针不失为一个好的解决方案，我在这里使用了Unordered\_map类，其底层实现就是哈希表。

我的Trie树结点数据域还有其他几个数据，能否成词，词频，词性。其中能否成词使用一个bool类型的变量存储，如果这个结点的字符可以作为一个词的结尾，那么变量为true，否则为false。词频和词性在不能成词时是无意义的，能组成词的时候代表了该单词的出现频率和词性。结点具体的结构见图2-1。



Trie树的结点结构 图2-1

假设现在有一个字典，其内容如下:

一 98 m

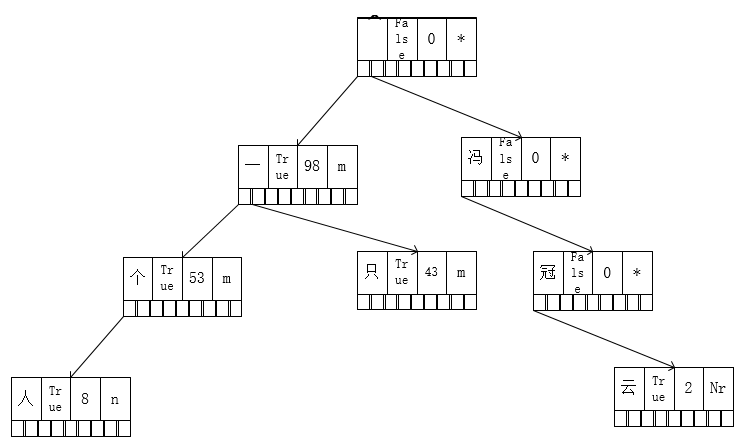
一个 53 m

一只 43 m

一个人 8 n

冯冠云 2 nr

那么加载出来的Trie树结构图可见图2-2：



Trie树的结构 图2-2

### 2.1.3 如何查询一个词语呢

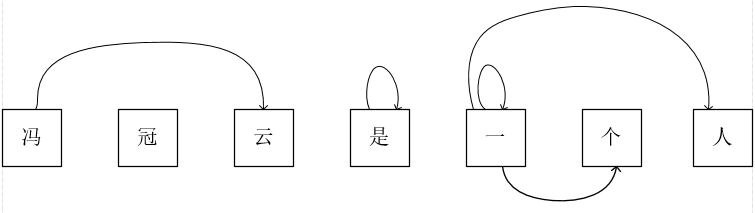
比如要查询“一个”这个词，那么首先会在根结点下找首字，即“一”。如果没有找到就会返回，找到了则指针下移至“一”结点，继续在“一”结点下继续找下一个字。如此重复，直到找完要查询的词语的所有字。如果在这个过程中有一个字未找到，则立即停止，表示词典中没有该词语；找到最后一个字后检查其能否成词的标记，如果能组成一个词就返回这个词的结构体信息（包括词频词性）。

## DAG(有向无环图)

由于后面需要对一个句子的不同方案进行多种方法的筛选，因此先通过查询字典将个句子的所有切分方案提取并存储起来就十分有必要。DAG(有向无环图)就可以存储其切分方案。

### 2.2.1 DAG的直观结构

举个例子来直观的说明DAG在句子切分时到底存储了什么信息。比如现在有一个句子“冯冠云是一个人”，字典和上面Trie树举例用的字典一样， 那么把每一个汉字作为结点，DAG的切分图如图2-3。



DAG的直观结构 图2-3

也就是说，把句子中的每一个字作为一个结点，用一条有向的边连接句子中每个可能词语的第一个字和最后一个字，由此组成一个有向无环图。

### 2.2.2 DAG的存储结构

句子中可以用字所在的位置描述其结点，我们只需要使用一个二维数组就可以存储由句子切分出的有向无环图。假设我们有一个二维数组，dag[1024][1024]。

用上面的句子举例，“冯冠云是一个人”。“冯”字这个结点用0来代表，其余每个字编号依次往后推。那么dag[0]这个数组里就存储了若干个可能以“冯”作为第一个字的单词结尾字的结点位置。这里“冯”只能组成一个词，因此dag[0]只有一个元素。dag[0][0]存储了“云”字的位置，即dag[0][0] = 2。

由于我的词典里面还有词频和词性，我除了结尾字的位置还存储了组成单词的词频和词性。如果一个字与后面字的组合均无法在字典中找到，那么就返回这个字的位置。其词频设置为2，表示出现概率小。词性设为“\*”表示未知。

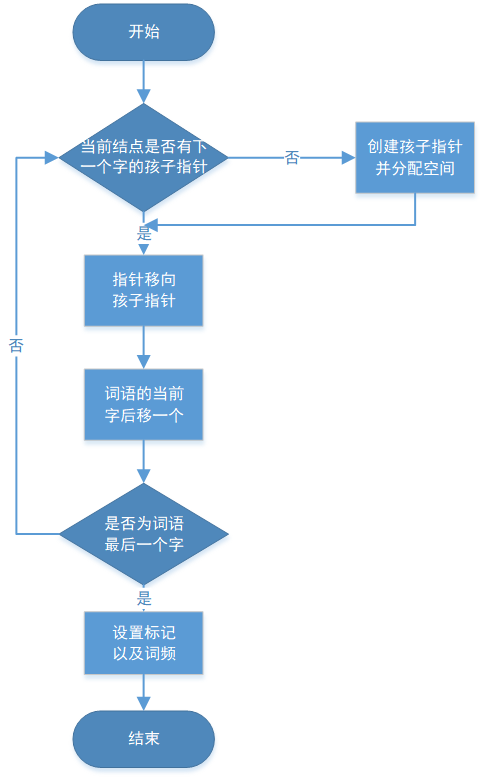
# 关键算法及实现

## 3.1 Trie树的算法

这次分词的实现过程中，Trie树是一个非常关键的数据结构，许多算法也要基于Trie树的基本操作来实现，因此其操作实现就极其重要。

### 3.1.1 插入词语

这个函数用于在Trie树中插入一个词语。Trie树的根结点数据与总为空，各个孩子指针指向各个单词的首字。每次插入词语的时候，首先在根结点孩子指针中搜索单词的首字，如果没有则新分配一块空间并赋值。随后在首字的结点孩子指针中搜索词语的第二个字，同样，如果不存在结点则新分一块空间并赋值。以此类推，依次在第二个字的结点孩子中搜索第三个字，知道词语结束。在最后一个词的结点将其成词的标记置为True，并将其词频词性一起存入结点。其流程图见图3-1，如下：



Trie树插入流程图 图3-1

具体的代码实现如下：

bool Trie::InsertVacab(Info info) {

//把单词添加到Trie树

TrieNode \*pTmp = this->root;

for (unsigned int i = 0; i < info.vacab.size(); i++) {

assert(pTmp);

unordered\_map<word, TrieNode\*>::iterator ite = pTmp->childList.find(info.vacab[i]);

if (ite == pTmp->childList.end()) {

TrieNode \*tmpNode = new TrieNode();

if (!tmpNode) return false;

pTmp->childList[info.vacab[i]] = tmpNode;

pTmp = pTmp->childList[info.vacab[i]] = tmpNode;

tmpNode = NULL;

}

else if (!pTmp->childList[info.vacab[i]]) {

TrieNode \*tmpNode = new TrieNode();

if (!tmpNode) return false;

pTmp->childList[info.vacab[i]] = tmpNode;

pTmp = pTmp->childList[info.vacab[i]] = tmpNode;

tmpNode = NULL;

}

else pTmp = pTmp->childList[info.vacab[i]];

}

//if (pTmp->wordTag) return false;

pTmp->wordTag = true;

pTmp->freq = info.freq;

strcpy(pTmp->prop, info.prop);

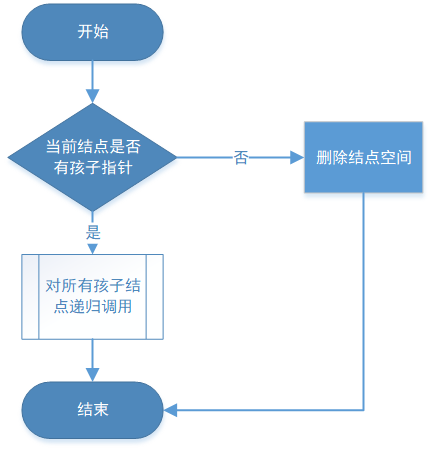
this->numb++;

return true;

}

### 3.1.2 删除子树

这个函数主要用于Trie树的删除，其作用是从一个Trie树的结点开始， 删除其自身和所有子树。这个函数的实现使用了递归，如果当前结点是叶子结点，那么释放这个结点的空间；否则对其所有的孩子结点递归调用这个函数。**其流程图见图3-2，如下：**



Trie树的删除 图3-2

**其代码实现如下：**

//把结点node以及其所以子树删除

void Trie::DeletNode(TrieNode\* &node) {

if (!node) return;

//叶节点

if (node->childList.empty()) {

delete node;

node = NULL;

}

//非叶节点

else {

unordered\_map<word, TrieNode\*>::iterator it = node->childList.begin();

for (; it != node->childList.end(); it++)

this->DeletNode(it->second);

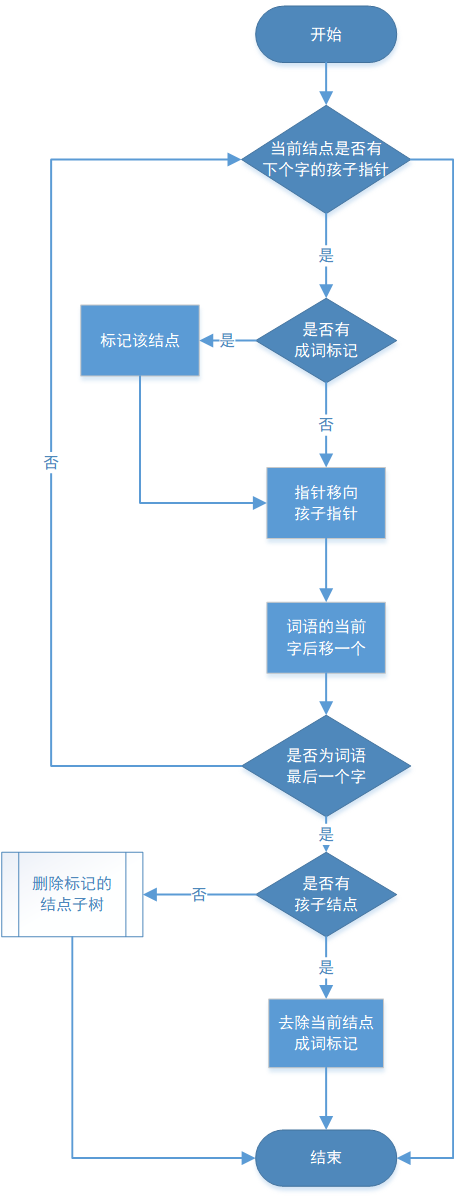
}

}

### 3.1.3 删除词语

这个函数用于从Trie树中删除一个词语。这个算法实现的关键在于不能误删其他的词语。因此大致可以分为两种情况，这个词语的最后一个字是否在叶子结点。如果不在叶子结点只需改变末尾字的成词标记，将其从True改为False。如果末尾字是叶子结点则需要从上一个能够成词的结点开始删除。

具体算法如下，从根结点开始查找这个词语，查找的过程中遇到能够成词的结点则进行标记，如果多次遇到能成词的结点则仅保留最后一次的结点。一直搜索到词语的结尾时，如果结尾字的结点是叶子结点则从标记的结点开始将子树删除；否则只改变结尾字结点能否成词的标记。**其流程图见图3-3如下：**



Trie树词语删除 图3-3

**其具体的代码实现如下：**

//把某一个单词从Tire树中删除，vacab是单词

bool Trie::DeletVacab(vector<word> vacab) {

TrieNode \*pTmp = this->root, \*pTmp3 = pTmp;

word tmpWord = vacab[0];

for (unsigned int i = 0; i < vacab.size(); i++) {

unordered\_map<word, TrieNode\*>::iterator it = pTmp->childList.find(vacab[i]);

if (it == pTmp->childList.end()) return false;

if (pTmp->wordTag) {

pTmp3 = pTmp;

tmpWord = vacab[i];

}

pTmp = pTmp->childList[vacab[i]];

}

//如果还有子树，只是去除成词标记

if (!pTmp->childList.empty()) {

pTmp->wordTag = false;

pTmp->freq = 0;

}

//否则删除上一个可以成词的结点子树

else

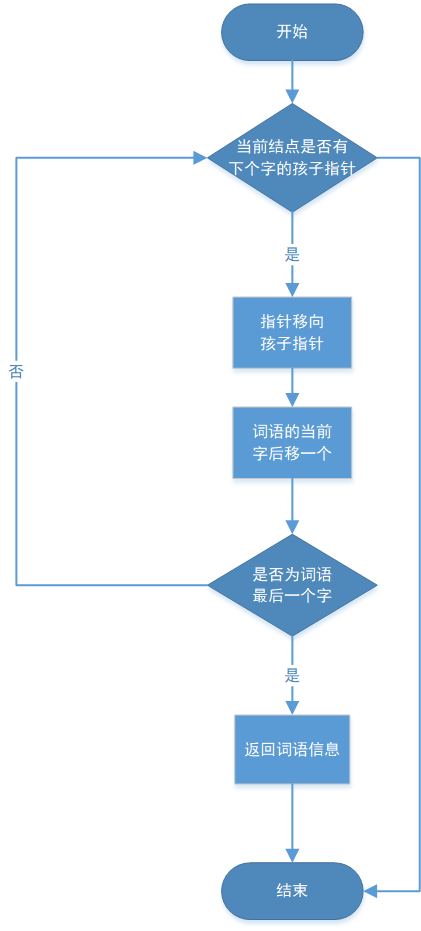
Trie::DeletNode(pTmp3->childList[tmpWord]);

return true;

}

### 3.1.4 搜索词语

搜索词语的算法其实在上面的几个操作中均有体现，只需在根结点孩子中找首字，首字结点孩子结点中找第二个字，一直到词语结尾。中途如果某一次未查找到，则直接返回空的信息。如果一直查询到了词语的结尾，则返回词语的词频词性等信息。**其流程图见图3-4，如下：**



Trie树搜索 图3-4

**其具体的代码实现如下：**

//在加载的Trie树中查找单词str

WordInfo Trie::Search(vector<word> vacab) {

Info info;

TrieNode\* pTmp = this->root;

if (!pTmp) return info; //info.freq == 0可作为是否正确找到单词的判断依据

for (unsigned int i = 0; i < vacab.size(); i++) {

unordered\_map<word, TrieNode\*>::iterator it = pTmp->childList.find(vacab[i]);

//如果Trie树中没有该单词

if (it == pTmp->childList.end()) return info;

pTmp = pTmp->childList[vacab[i]];

if (!pTmp) return info;

}

//如果True树中有该路径，但不能作为一个单词

if (!pTmp->wordTag) return info;

else {

info.vacab = vacab;

strcpy(info.prop, pTmp->prop);

info.freq = pTmp->freq;

return info;

}

}

## 3.2 获得DAG的算法

### 3.2.1简单的DAG切分

按照使用已有Trie树搜索词语函数来切分DAG已经足够了，依旧拿前面的句子举例，“冯冠云是一个人”。那么只需要新建一个二维数组，从“冯”开始查询，查“冯”是不是一个词，“冯冠”是不是一个词，“冯冠云”是不是一个词，“冯冠云是”又能不能作为一个词语，一直到句子的结尾。把所有可能作为结尾的字结点存入dag[0]中。随后再从下一个字“冠”开始查询直到句子的结尾，将这些词存入dag[1]。以此类推。

### 3.2.2 改进的DAG切分

我并不满足于这样的方式得到DAG(有向无环图)。数据结构的学习至少让我知道了，要想改进算法，利用之前搜索过的结果不失为一种好的方法。这里一遍遍在Trie树里面进行搜索，其中存在大量重复的内容而没有加以利用，如果句子的长度为n，那么时间复杂度会达到惊人的O(n^4)。即便句子的长度一般不会达到非常大的规模，这也是难以忍受的。

我为获得DAG图，另写了一种算法来在Trie树中搜索。与之前不同的是，以前的算法是每次传入一个词语，而改进后每次传入一个句子。当我传入一个句子时，每一次搜索会进行到以下几种情况时停止:

1. 句子结束 2.下一个字不在Trie树结点中 3.访问了Trie树的叶子结点。

比如“冯冠云是一个人”这句话，原本从“一”字开始查询要查3次，即“一”、“一个”、“一个人”每个词都要在Trie树中进行一次查询。而现在会一直访问到Trie树的叶子结点也就是“人”这个结点，随后把路径上访问过的词语“一”、“一个”、“一个人”存入“一”开头的数组dag[4]中一起返回。这个算法的时间复杂度在O(n)和O(n^2)之间，具体情况会随句子不同有变化。**其流程图见图3-5，如下：**



获取DAG图 图3-5

**其具体的代码实现如下：**

vector<vector<DAGInfo>> Trie::SearchDAG(vector<word> sentence)

{

vector<vector<DAGInfo>> dag;

vector<DAGInfo> tmp;

DAGInfo tmpInfo;

dag.clear(); tmp.clear();

TrieNode\* pTmp = this->root;

assert(pTmp);

for (unsigned int i = 0; i < sentence.size(); i++) {

pTmp = this->root;

tmp.clear();

for (unsigned j = i; j < sentence.size(); j++) {

unordered\_map<word, TrieNode\*>::iterator it = pTmp->childList.find(sentence[j]);

if (it == pTmp->childList.end()) break;

else if (it->second->wordTag) {

tmpInfo.freq = it->second->freq;

strcpy(tmpInfo.prop, it->second->prop);

tmpInfo.pos = j;

tmp.push\_back(tmpInfo);

}

pTmp = pTmp->childList[sentence[j]];

}

if (tmp.empty()) {

tmpInfo.pos = i;

tmpInfo.freq = 1;

strcpy(tmpInfo.prop, "\*");

tmp.push\_back(tmpInfo);

}

dag.push\_back(tmp);

}

return dag;

}

## 3.3 MMSEG分词算法

这是我的中文分词实现中主要的分词算法。这个算法是Chih-Hao Tsai于2000年提出的。按照理解，这种算法可以看作最大正相匹配算法的一种改进。

正向最大匹配算法每次会找出句子中从前往后最长的词，确定下来这个词后，再从这个词的下一个字开始再次找最长的词。

MMSeg算法也是每次确定一个词后从下一个词继续搜索。只不过确定这个词的方式不再是最大长度，而是使用了四种规则进行过滤。每次从开始查找的地方往后选取三个词或者直到句子结尾，对三个词进行一下考量：

1. 三个词的长度最大（Maximum matching），即三个词的词长之和最大；
2. 三个词的平均词长最大（Largest average word length），即要求词长分布尽可能均匀；
3. 三个词的词长变化最小（Smallest variance of word lengths ）；
4. 三个词中单字的出现词自由度最高（Largest sum of degree of morphemic freedom of one-character words）。

现在举例子来说明这种分词算法，可能有部分引用自网页内容，会在参考文献中写明。

### 3.1.1 MMseg规则一

比如“长春市长春药店”，这个会有如下几种组合

长春市\_长春\_药店\_

长春市\_长\_春药\_

长春\_市长\_春药\_

长春\_市\_长春\_

长\_春\_市长\_

第一种组合长度最长，所以就选择“长春市\_长春\_药店”。

### 3.1.2 MMseg规则二

比如“国际化”，这个会有如下几种组合

国际化\_

国际\_化\_

国\_际\_化\_

显然规则1无法过滤，长度都是3 经过规则2,之后发现第一个组合平均长度为3/1 = 3,第二个是3/2 = 1.5,第三个3/3 = 1。第一个平均长度最大，选择“国际化”。其实并不需要计算平均长度，只要词语的数量最少即可。这个规则则主要使用在句子结尾可能不足三个词的时候。

### 3.1.3 MMseg规则三

比如“北京大学生”，这个会有如下几种集合

北京大学\_生\_

北京\_大学生\_

北京\_大学\_生\_

北京\_大\_学生\_

北\_京\_大学生\_

显然规则1无法过滤，长度都是5，在经过规则2之后剩下

北京大学\_生\_

北京\_大学生\_

在规则3中，前面一句话方差是sqrt(((4-2.5)^2+(1-2.5)^2))/2)=1.5，后面是sqrt(((3-2.5)^2+(2-2.5)^2))/2)=0.5。因此选择“北京\_大学生”。可以看到，在第3条规则时，每种选择中的词语数量均相同(否则会在第2条规则筛掉)，因此除词数开根号这一步骤可以省去，只需计算(c1 - average)^2 +

北京\_大学生\_

### 3.1.4 MMseg规则四

比如“设施和服务”，这个会有如下几种组合

设施\_和服\_务\_

设施\_和\_服务\_

设\_施\_和服\_

经过规则1过滤得到

设施\_和服\_务\_

设施\_和\_服务\_

这时通过前三条规则已经无法选择出最佳的三个词。则看单个词语的词频，取对数相加比大小，选择数值上较大的一种方案。对数相加和直接相乘理论上是一样的，但由于要防止变量溢出，取对数更加稳妥。

### 3.1.5整个句子的切分

另外，每次虽然选取三个词，但最后仅确定下来一个词语。比如有一个句子“武汉市长江大桥已经建成十年了”。从第一个字开始找三个词，可能的结果有：

武汉 市 长

武汉 市 长江

武汉 市 长江大桥

武汉 市长 江大桥

武汉市 长江 大桥

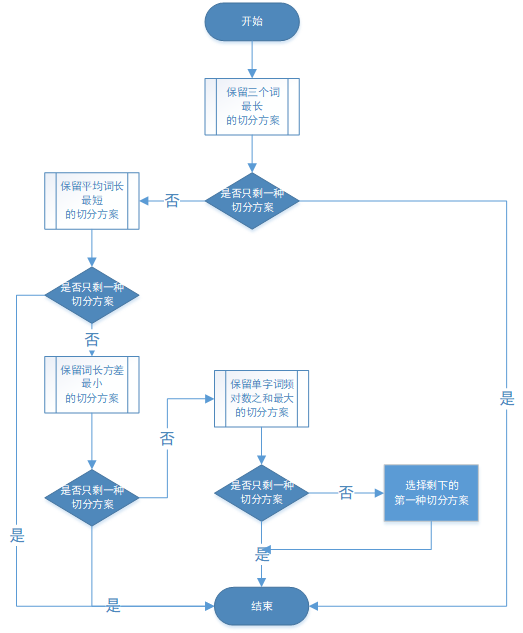
武汉市 长江大桥 已经

那么将会使用第一条规则，选择三个词长总和最长的，即“武汉市/长江大桥/已经”，并将第一个词作为确定的词语。下一次查找会从长字开始。继续查找的结果为：

长江 大桥 已经

长江大桥 已经 建成

于是可以确定“长江大桥”一词，再继续往后查找，以此类推。**其流程图见图如下：**

****

**MMSeg分词算法流程图 图3-6**

**具体的代码实现如下：**

void segment::Cut(vector<word> sentence, FILE\* Outfile) {

unsigned i, j , k, p, q, next, current, n = 0;

unsigned maxlen = 0, numb =3;

char tmpStr[5];

vector<vector<DAGInfo>> dag;

unsigned routs[1024][3];

unsigned rout[3];

dag.clear();

dag = this->dict->tree.SearchDAG(sentence);

//i是第i个字

for (i = 0; i < dag.size(); ) {

n = 0; maxlen = 0; numb = 3;

//j用于遍历i所有能组成的词

for (j = 0; j < dag[i].size(); j++) { //进行第一条规则筛选

memset(rout, 0, 3 \* sizeof(unsigned));

rout[0] = dag[i][j].pos - i + 1;

current = next = dag[i][j].pos + 1;

if(current <dag.size())

for (k = 0; k < dag[current].size(); k++) {

rout[1] = rout[2] = 0;

rout[1] = dag[current][k].pos - current + 1;

next = dag[current][k].pos + 1;

if (next < dag.size())

rout[2] = dag[next][dag[next].size() - 1].pos - next + 1;

if (maxlen <= rout[0] + rout[1] + rout[2]) {

if (maxlen<rout[0] + rout[1] + rout[2])

n = 0;

maxlen = rout[0] + rout[1] + rout[2];

routs[n][0] = rout[0];

routs[n][1] = rout[1];

routs[n][2] = rout[2];

n++;

}

}

else {

if (maxlen <= rout[0] + rout[1] + rout[2]) {

if (maxlen<rout[0] + rout[1] + rout[2])

n = 0;

maxlen = rout[0] + rout[1] + rout[2];

routs[n][0] = rout[0];

routs[n][1] = rout[1];

routs[n][2] = rout[2];

n++;

}

}

}

if (n == 1)

goto CONTINUE;

else {

numb = 3;

for (p = 0, q = 0; p < n; p++) {//进行第二条规则筛选

if (routs[p][2] && routs[p][1] && routs[p][0]) {

if (numb == 3) {

routs[q][0] = routs[p][0];

routs[q][1] = routs[p][1];

routs[q][2] = routs[p][2];

q++;

}

}

else if (!routs[p][2] && routs[p][1] && routs[p][0]) {

if (numb > 2) {

numb = 2; q = 0;

routs[q][0] = routs[p][0];

routs[q][1] = routs[p][1];

routs[q][2] = routs[p][2];

q++;

}

else if (numb = 2) {

routs[q][0] = routs[p][0];

routs[q][1] = routs[p][1];

routs[q][2] = routs[p][2];

q++;

}

}

else if (!routs[p][2] && !routs[p][1] && routs[p][0]) {

if (numb > 1) {

numb = 1; q = 0;

routs[q][0] = routs[p][0];

routs[q][1] = routs[p][1];

routs[q][2] = routs[p][2];

q++;

}

else if (numb = 1) {

routs[q][0] = routs[p][0];

routs[q][1] = routs[p][1];

routs[q][2] = routs[p][2];

q++;

}

}

assert(!(!routs[p][2] && !routs[p][1] && !routs[p][0]));

}

n = q;

}

if (n == 1)

goto CONTINUE;

else { ////进行第三条规则筛选

double average = (double)maxlen / numb;

double variance = 100, tmp = 0;

for (p = 0, q = 0; p < n; p++) {

tmp = (routs[p][0] - average)\*(routs[p][0] - average)

+ (routs[p][1] - average)\*(routs[p][1] - average)

+ (routs[p][2] - average)\*(routs[p][2] - average);

if (tmp < variance) {

variance = tmp;

q = 0;

routs[q][0] = routs[p][0];

routs[q][1] = routs[p][1];

routs[q][2] = routs[p][2];

q++;

}

else if (tmp == variance) {

routs[q][0] = routs[p][0];

routs[q][1] = routs[p][1];

routs[q][2] = routs[p][2];

q++;

}

}

n = q;

}

if (n == 1)

goto CONTINUE;

else { ////进行第四条规则筛选

double mp = 0, tmpMp = 0;

for (p = 0, q = 0; p < n; p++) {

tmpMp = 0;

if (routs[p][0] == 1)

tmpMp += log(dag[i][0].freq);

if (routs[p][1] == 1)

tmpMp += log(dag[i + routs[p][0]][0].freq);

if (routs[p][2] == 1)

tmpMp += log(dag[i + routs[p][0] + routs[p][1]][0].freq);

//

if (tmpMp > mp) {

mp = tmpMp;

q = 0;

routs[q][0] = routs[p][0];

routs[q][1] = routs[p][1];

routs[q][2] = routs[p][2];

q++;

}

else if (tmpMp == mp) {

routs[q][0] = routs[p][0];

routs[q][1] = routs[p][1];

routs[q][2] = routs[p][2];

q++;

}

}

n = q;

}

CONTINUE:

if (this->out\_encode == UCS2\_LE) {

for (p = 0; p < routs[0][0]; p++) {

tmpStr[0] = (char)(sentence[i] & 0x00ff);

tmpStr[1] = (char)((sentence[i] >> 8) & 0x00ff);

fwrite(tmpStr, 1, 2, Outfile);

i++;

}

tmpStr[0] = 0x2f;

tmpStr[1] = 0;

fwrite(tmpStr, 1, 2, Outfile);

}

else {

for (p = 0; p < routs[0][0]; p++) {

tmpStr[1] = (char)(sentence[i] & 0x00ff);

tmpStr[0] = (char)((sentence[i] >> 8) & 0x00ff);

fwrite(tmpStr, 1, 2, Outfile);

i++;

}

tmpStr[1] = 0x2f;

tmpStr[0] = 0;

fwrite(tmpStr, 1, 2, Outfile);

}

}

}

## 3.4 MPSEG分词算法

这次课设的实现过程中，其思路有很大一部分我参考了python的开源库—jieba分词。我在阅读jieba库的python源码时，知道了这种分词算法。Python代码虽然与C++代码有很多不同，但重要的是提供了一种算法的思路。包括前面先把句子切分成dag(有向无环图)再进行操作，这也是阅读其python代码收获的重要思路。不过由于python与c++的区别，我需要重新思考如何在C++下重新实现这些思路。由前面的算法我已经可以得到一个句子的dag了，那么如何通过dag(有向无环图)来切分出句子呢？

### 3.4.1算法思路

首先说明算法的思路。一个句子根据字典里的词语可以有很多种切分方式。由于字典中存储了每个词语的出现次数，那么出现的总概率用词频相乘即可算出。这里需要注意，是词频而不是出现次数。即要除以所有词语出现的总次数。(如果使用出现次数，那么最后的切分结果有很大可能全是单字。)

由于词频是一个小于1的数，当多个出现次数较少的词词频相乘时可能会产生一个很小的数从而导致下溢。解决方法就是取对数log进行运算。由于在n<1时log函数的增长很快，从而起到放大的作用，也就不会产生下溢了。

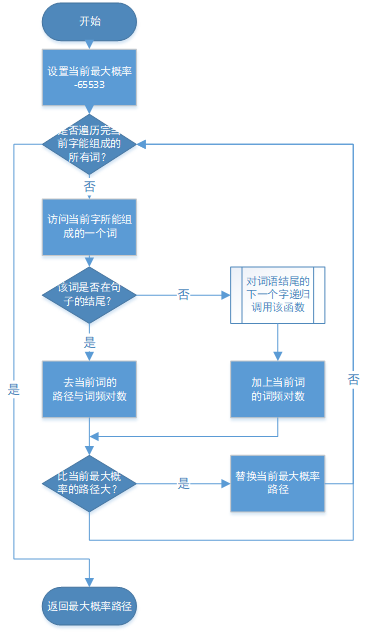
### 3.4.2算法实现

既然是最大概率路径，那么我就需要计算出概率最大的那条路径。怎么获得最大概率的路径呢？最简单的方法就是把所有路径的概率都计算出来，也就是说存储所有的路径，然后进行比较。但是万一句子超长呢？就算用一个二维数组存储起来了，还需要用一次O(n)时间复杂度的遍历。所以说这种思路既耗时，又占用空间，算法还不优美。我放弃了这种思路的算法。

我经过思考使用递归的方法实现了这一算法。递归函数名为CalcPoss(DAG dag,int i,

Unsigned total)。每次传入的参数为，句子的dag图，当前所在句子第几个字，字典中所有词的总词频。函数的返回值是从当前字开始后面有最大概率的路径以及其概率。

进入函数后，如果已经到了句子的最后一个字，直接返回其位置和词频。如果不是最后一个字，则查找当前字能组成的所有词的结尾字。如果结尾字是句子的结尾，则获得其位置和词频。对其他所有该结尾字的下一个字调用CalcPoss()函数并加上当前词的词频对数。每次获得一条后面的路径就与前面的最大概率路径进行对比，如果大于其概率则替换之，否则抛弃。**其流程图见图3-7，如下：**

****

**MPSeg算法流程图 图3-7**

**具体的代码实现如下：**

typedef struct {

double possi;

vector<int> rout;

}MPRout;

void SegCore::MPSeg(vector<unsigned short> sentence, Dict \*dict, wchar\_t \*ws) {

int count = 0, sCount = 0;

vector<vector<DAGInfo>> dag;

dag.clear();

dag = dict->tree.SearchDAG(sentence);

vector<int> rout = SegCore::CalcPoss(dag, 0, dict->GetTotal()).rout;

for (int i = rout.size() - 1; i >= 0; i--) {

while(sCount <= rout[i])

ws[count++] = sentence[sCount++];

ws[count++] = '/';

ws[count] = '\0';

}

}

MPRout SegCore::CalcPoss(vector<vector<DAGInfo>> dag, int i, unsigned long total) {

MPRout r, tmp;

r.possi = -65533; r.rout.clear();

if (i == dag.size() - 1) {

r.possi = log((double)dag[i][0].freq/total);

r.rout.push\_back(i);

}

else {

for (int j = 0; j < dag[i].size(); j++) {

if (dag[i][j].pos == dag.size() - 1) {

tmp.possi = log((double)dag[i][j].freq / total);

if (tmp.possi > r.possi) {

r.possi = tmp.possi;

r.rout.clear();

r.rout.push\_back(dag[i][j].pos);

}

}

else {

tmp = CalcPoss(dag, dag[i][j].pos + 1, total);

tmp.possi += log((double)dag[i][j].freq / total);

if (tmp.possi > r.possi) {

r.possi = tmp.possi;

r.rout.clear();

r.rout = tmp.rout;

r.rout.push\_back(dag[i][j].pos);

}

}}}

return r;}

## 3.5 MAXSEG分词算法

这种分词算法就是最大正相匹配算法。思路非常简单每次从前往后找到最长的词，再从下一个字开始往后找最长的词。由于我已经得到了一个句子的dag(有向无环图)，我只需要首先在dag[0]的数组下面找到最大的数max1，那么句子的0 - max1就是第一个词。如果max1不是句子的最后一个字，就在dag[max1+1]中寻找最大的数max2，max1+1 – max2就是第二个词。如果还没到句子的结尾则继续以上步骤直到句子结尾。由于算法过于简单在这里就不画出其流程图了。其具体的代码实现如下：

void SegCore::MaxSeg(vector<unsigned short> sentence, Dict \*dict, wchar\_t \*ws) {

unsigned i = 0, j = 0, maxpos = 0, count = 0;

vector<vector<DAGInfo>> dag;

unsigned routs[1024][3];

unsigned rout[3];

dag.clear();

dag = dict->tree.SearchDAG(sentence);

for (i = 0; i < dag.size();) {

for (j = 0; j < dag[i].size(); j++) {

maxpos = 0;

if (dag[i][j].pos > maxpos)

maxpos = dag[i][j].pos;

}

while (i <= maxpos)

ws[count++] = sentence[i++];

ws[count++] = '/';

ws[count] = '\0';

}

}

# 4.其他相关问题

## 4.1 中文编码问题

### 4.1.1 几种常见的编码

**ASCII编码**

**所有的空格、标点符号、数字、大小写字母分别用连续的字节状态表示，一直编到了第127号，这样计算机就可以用不同字节来存储英语的文字了。叫做 ANSI 的“ASCII”编码。**

**后来，采用 127号之后的空位来表示新的字母、符号，还加入了很多画表格时需要用下到的横线、竖线、交叉等形状，一直把序号编到了最后一个状态255。从128 到255这一页的字符集被称“扩展字符集”。**

**GB2312编码**

国人自己进行的编码，把127号之后的符号取消并规定：一个小于127的字符的意义与原来相同，但两个大于127的字符连在一起时，就表示一个汉字，前面的一个字节（称之为高字节）从0xA1用到0xF7，后面一个字节（低字节）从0xA1到0xFE。。把这种汉字方案叫做 “GB2312”。GB2312 是对 ASCII 的中文扩展。另外，GB2312扩展之后的编码方案被称为 GBK 标准。

**ANSI编码**

**ANSI编码是常见的中文编码，英文部分占一个字节，直接使用ACSII码；中文占两个字节，使用GBK编码。**

**UNICODE编码**

**Unicode是ISO（国际标谁化组织）重新制定的编码，ISO规定unicode必须用两个字节，也就是16位来统一表示所有的字符，对于ASCII里的那些“半角”字符，unicode包持其原编码不变，只是将其长度由原来的8位扩展为16位。**

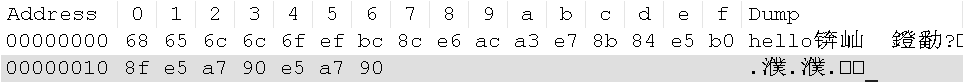
**UTF-8编码**

**UTF-8就是在互联网上使用最广的一种Unicode的实现方式，这是为传输而设计的编码。UTF-8最大的一个特点，就是它是一种变长的编码方式。它可以使用1~4个字节表示一个符号，根据不同的符号而变化字节长度，当字符在ASCII码的范围时，就用一个字节表示；中文字符则将Unicode通过一定的转换变成UTF-8编码，Unicode中一个中文占2个字符，UTF-8中则占2个或3个字节。**

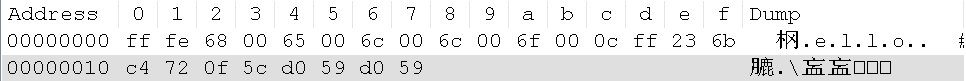
### 4.1.2 编码的存储方式

编码有多个字节，因此存储时有高位在前还是低位在前的区分，成为大端存储和小端存储。其中Unicode分为little endian和big endian，其同样都是Unicode编码，只不过在存储的时候，高字节与低字节的顺序相互颠倒过来。

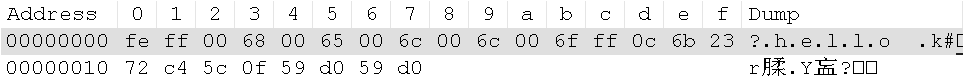
另外，有些编码在文本文档存储时，会有一个BOM头。也就是说，文本文档的前几个字节并非世纪的文本内容，而是用作编码标记的字节。那么大部分不同的编码就可以通过读取文件的前几个字节来判断其编码方式。我通过向一个不同编码的文件写入相同内容来观察。见图4-1，4-2，4-3，4-4。



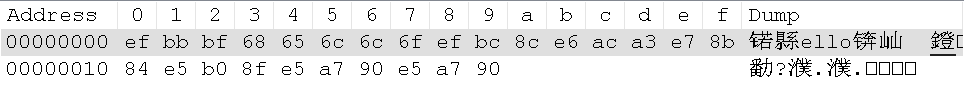
ANSI没有前导符 图4-1



UCS2-LE有前导符FF FE 图4-2



UCS2-BE有前导符FE FF 图4-3



UTF-8(有bom)前导符是EF BB BF 图4-4

### 4.1.3 编码的检测

编码的检测思路大致就是，读取文件的前几个字节看是否能判断出文件的编码方式，如果不能则遍历文件，一般可以认为编码可能是没有BOM头的UTF-8编码或者ANSI编码。因此可以通过检测每个字节的特征是否符合UTF-8编码的特征。**其具体的代码实现如下：**

Encode Decode::CheckEncode(const char \* path) {

FILE\* fp = fopen(path, "rb");

Encode encoding = UNKNOWN;

if (!fp) return encoding;

char\* ch = (char\*)malloc(5 \* sizeof(char));

fgets(ch, 3, fp);

//Unicode

if ((ch[0] == (char)0xFF) && (ch[1] == (char)0xFE))

encoding = UNICODE\_LITTLE\_ENDIAN;

//Unicode\_big\_endian

else if ((ch[0] == (char)0xFE) && (ch[1] == (char)0xFF))

encoding = UNICODE\_BIG\_ENDIAN;

//UTF-8 有bom

else if ((ch[0] == (char)0xEF) && (ch[1] == (char)0xBB)) {

fgets(ch, 2, fp);

if (ch[0] == (char)0xBF) encoding = UTF\_8\_BOM;

}

//UTF-8无bom,或者ANSI

else {

fseek(fp, 0, SEEK\_SET);

encoding = UTF\_8;

while ((ch[0] = fgetc(fp))!= EOF) {

if ((unsigned char)ch[0] < 0x80) continue;

else if ((unsigned char)ch[0] < 0xC0) {

encoding = ANSI;

break;

}

else if ((unsigned char)ch[0] < 0xE0) {

if ((ch[0] = fgetc(fp)) != EOF) {

if (((unsigned char)ch[0] & 0xC0) != 0x80) {

encoding = ANSI;

break;

}

}

else break;

}

else if ((unsigned char)ch[0] < 0xF0) {

if (((ch[0] = fgetc(fp)) != EOF && (ch[1] = fgetc(fp)) != EOF)) {

if (((unsigned char)ch[0] & 0xC0) != 0x80|| ((unsigned char)ch[1] & 0xC0) != 0x80) {

encoding = ANSI;

break;

}

}

else break;

}

else {

encoding = ANSI;

break;

}

}

}

free(ch);

fclose(fp);

return encoding;

}

### 4.1.4 编码的转换

在编码的转换方面，我一般使用了windows的编码转换函数，并进行了封装，将函数封装在了Decode类里面。

另外我自己也编写了Unicode与UTF-8编码的转换函数。主要用于单个字符的操作，转换单个字符时使用自己写的函数会更加灵活。具体的代码实现如下：

vector<unsigned short> Dict::ToUnicode(string str, Encode encoding) {

vector<unsigned short> tmpUnico;

tmpUnico.clear();

//字符串为UTF-8编码

if (encoding == UTF\_8 || encoding == UTF\_8\_BOM) {

for (unsigned i = 0; i < str.size(); ) {

//第一个字节小于127(即字节以0开头),为ASCII字符

if (!(str[i] & 0x80)) tmpUnico.push\_back(str[i++]);

//第一个字节形式为0x110\*\*\*\*\*,那么UTF8编码占2字节

else if ((byte)str[i] < 0xDF && i + 1 < str.size()) {

unsigned short tmpL = (str[i] & 0x1F); tmpL = tmpL << 6;

unsigned short tmpR = str[i + 1] & 0x3F;

tmpUnico.push\_back(tmpL | tmpR);

i += 2;

}

//第一个字节形式为0x1110\*\*\*\*,那么UTF8编码占3个字节

else if ((byte)str[i] < 0xEF && i + 2 < str.size()) {

unsigned short tmpL = (str[i] & 0x0F); tmpL = tmpL << 12;

unsigned short tmpM = (str[i + 1] & 0x3F); tmpM = tmpM << 6;

unsigned short tmpR = str[i + 2] & 0x3F;

tmpUnico.push\_back(tmpL | tmpM | tmpR);

i += 3;

}

//不属于Unicode跳过相应字节(属于UCS-4 编码，重新规范后，这些字节值无法出现在合法 UTF-8序列中)

else if ((byte)str[i] <= 0xF7) i += 4;

else if ((byte)str[i] <= 0xFB) i += 5;

else if ((byte)str[i] <= 0xFD) i += 6;

else return tmpUnico; //exit(-1);

}

}

//字符串为Unicode\_little\_endian编码

if (encoding == UNICODE\_LITTLE\_ENDIAN) {

for (unsigned i = 0; i < str.size(); i += 2) {

unsigned short high = str[i + 1];

high = (high & 0x00FF) << 8;

unsigned short low = str[i];

low = low & 0x00FF;

tmpUnico.push\_back(high | low);

}

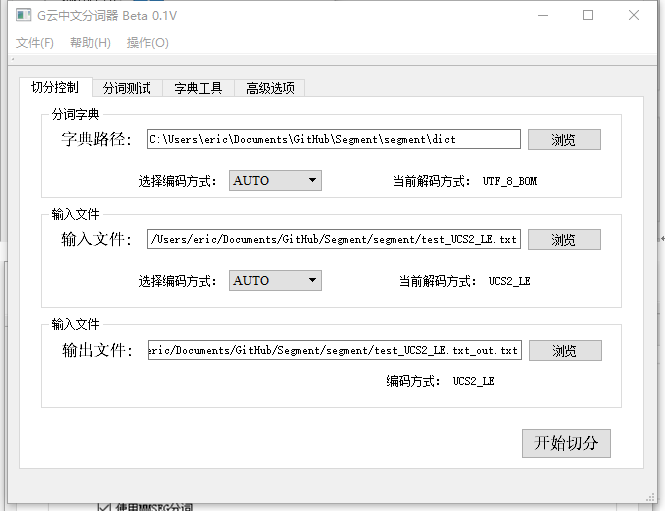
}

## 4.2 图形界面设计

这次的图形界面我使用了QT进行编写。由于我之前有一定QT编程的经历，这次的图形界面做的相对比较美观。

我使用了tab widgte来制作选项卡，使得程序的层次更加清晰。我是用的主窗体是QMainWindow，一个从Qwidgte继承而来的类。

控件我主要使用了QActoin，Qcombobox，QPushButton，QLineEdte，QtextEdite，QCheckBox。并使用了其中的TextChanged（），trieggered（），clicked（）等信号。另外使用了一些layout，groupBox等控件使界面更加有序。由于界面代码较长，具体可在附录中查看。图形界面的接口展示使用会在“程序测试”中有详细地展示。程序的主界面可见图4-5。



程序主界面 图4-5



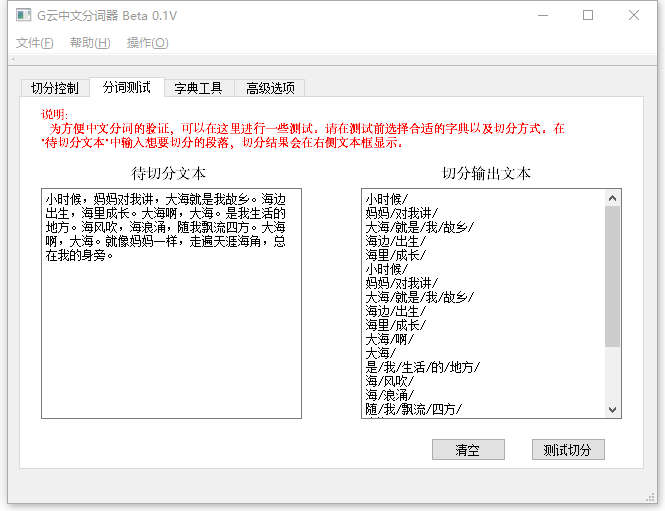
设置界面 图4-6

# 程序测试

## 5.1支持文件切分与文本框内切分

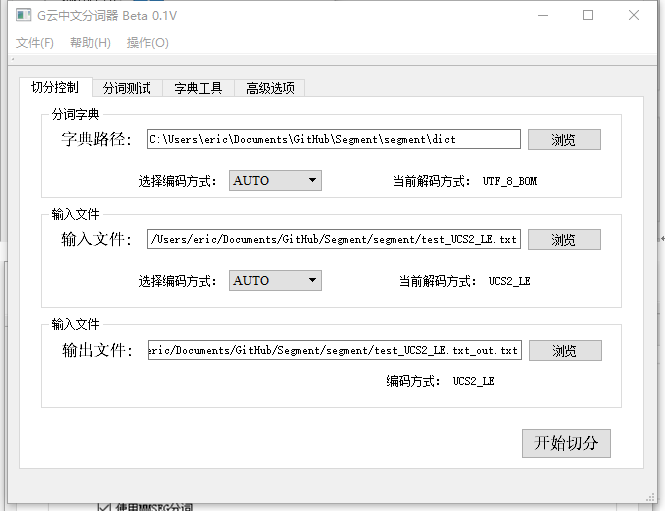
这次我编写的程序支持两种分词的方式，一种是将文本进行切分，将切分后的结果输出到另一个文本中。另一种是直接在程序的文本框中输入，并在程序中直接输出。

接下来进行测试。我支持了三种切分算法，下面切分均在MMASeg切分算法下进行。在程序内文本框测试时，我使用了歌曲《大海，我的故乡》的歌词，在左侧文本框输入文本后点击测试切分，其测试的结果见图5-1。

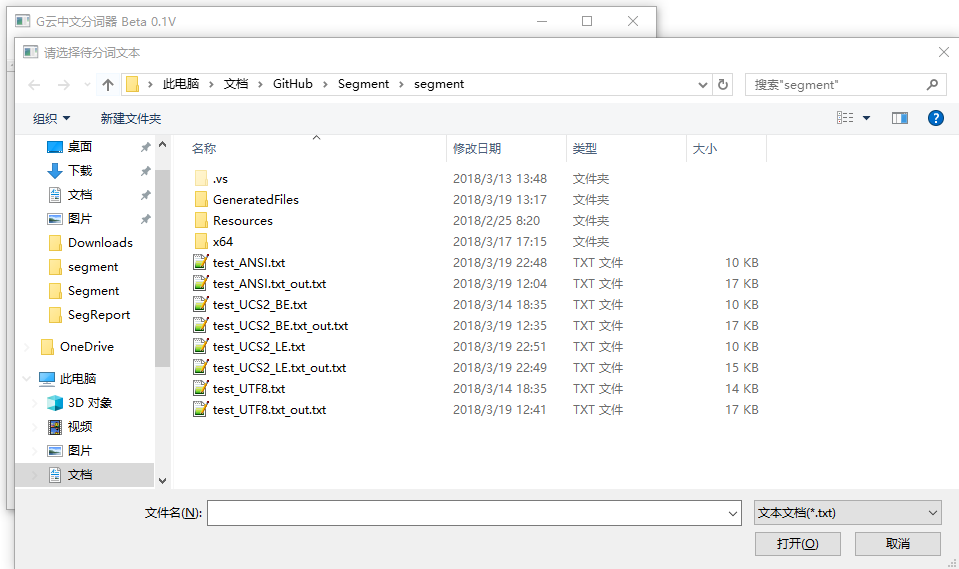


在文本框中对句子进行切分 图5-1

我的程序另一种切分方式是直接对文本文档进行切分，切分后输出到另一个新的文本文档中。在切分前需要选择要切分的文件以及输出的文件路径。选择输出文件的时候默认的文件名为原文件名加上”\_out”。选择浏览的过程界面如图5-2,图5-3。



选择文件界面 图5-2

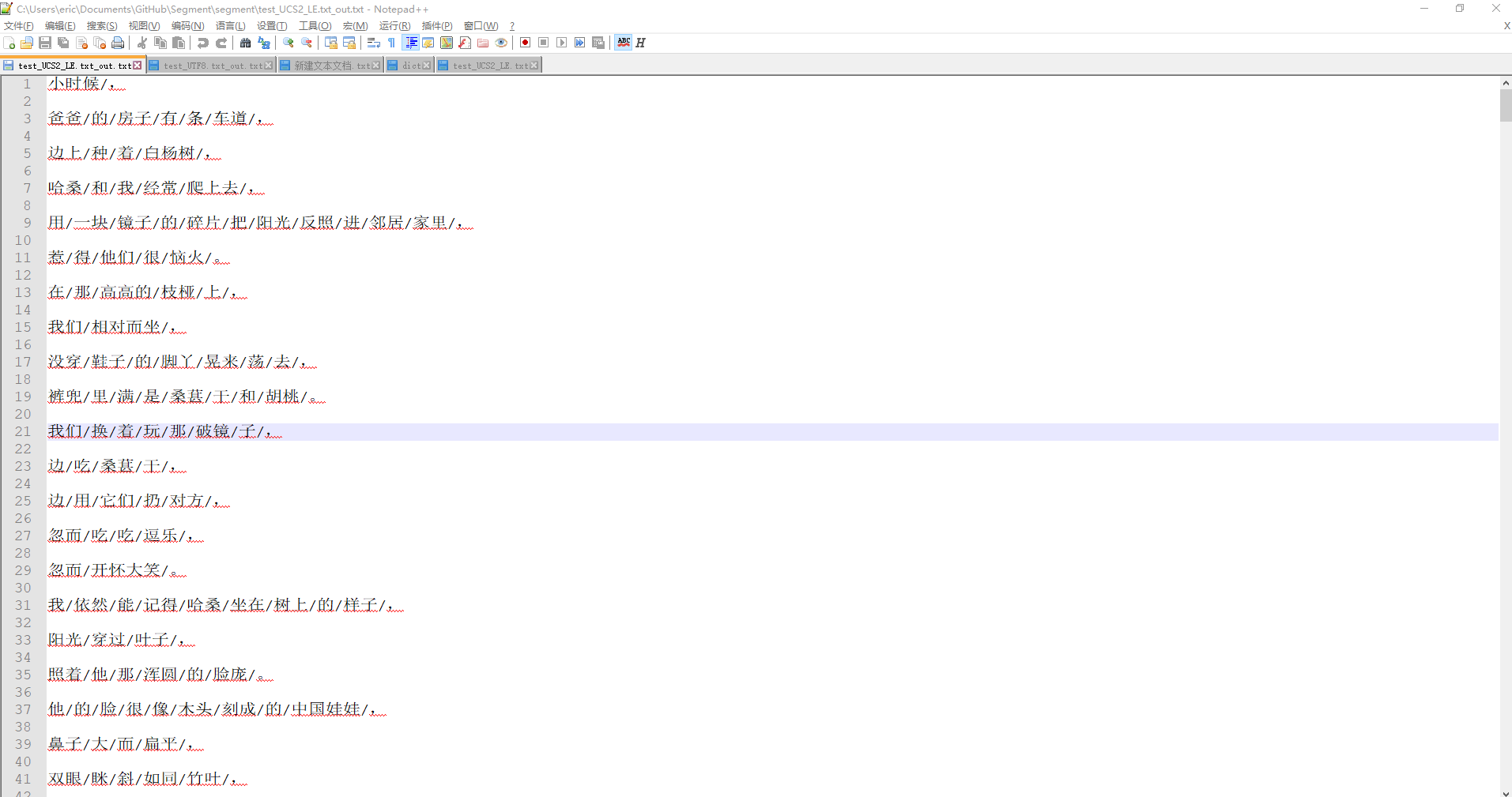


选择界面文本框 图5-3

我在网上找了一篇文章，具体内容是《追风筝的人》的第一章。我把其保存在了一个文本文档中，该文本使用notepad++打开的截图见图5-4。切分后生成了一个新的文件，其用notepad++打开的截图见图5-5。可以看到，切分已经完成了。



待切分文本 图5-4

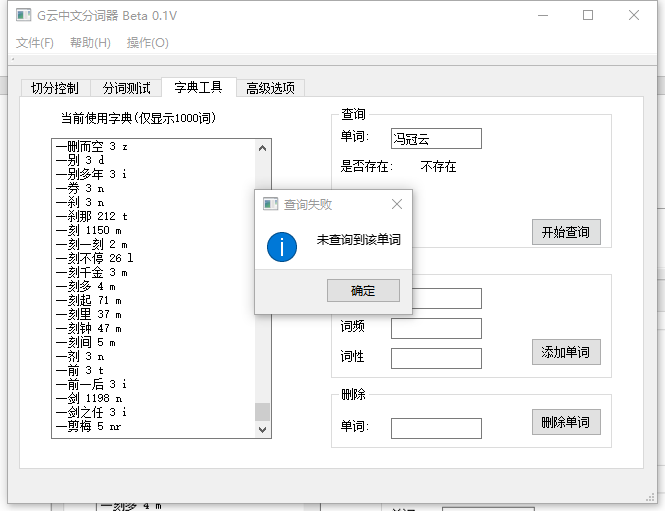


切分输出文本 图5-5

## 5.2字典的修改(增加与删除)

### 5.2.1字典的添加

这里我加载了字典之后，可以查询，添加，删除字典中的词语。由于查询功能在接下来的展示中会多次使用，就不单独展示了。首先在字典中查询我的名字“冯冠云”。并没有查询到。见图5-6。随后在“添加”中添加我的名字(可以省略词频和词性，省略则以词频2，词性\*添加)，可以选择修改写入字典文件。见图5-7。随后再进行查询即可查询到我的名字“冯冠云”，说明添加成功。查看文件也可以看到文件末尾增加了我的名字。



查询词语不存在 图5-6



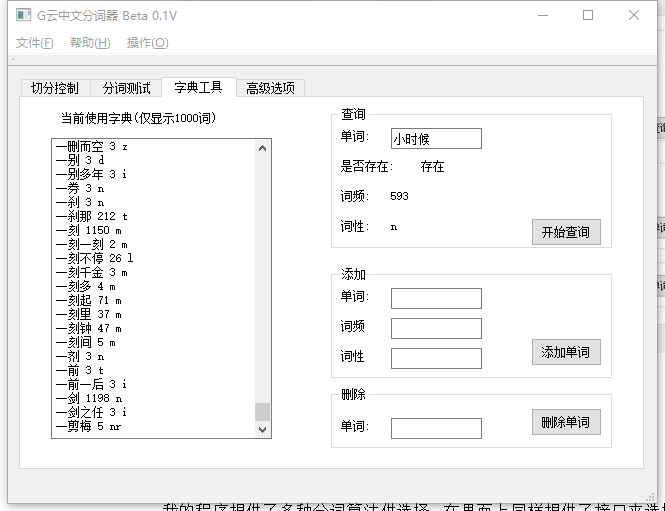
添加词语 图5-7



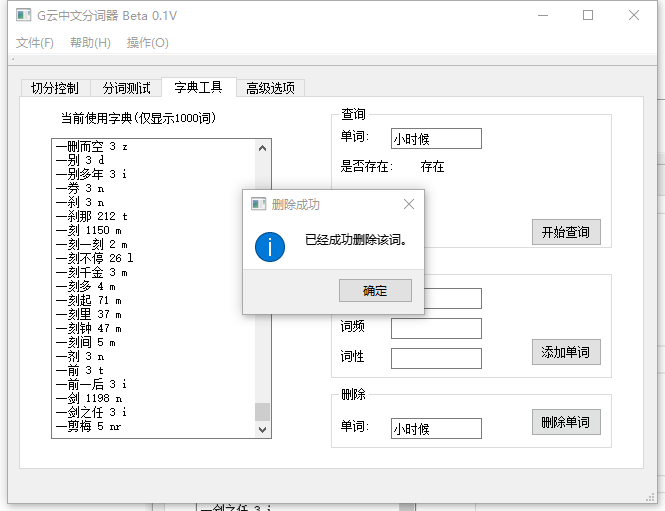
词语添加成功 图5-8

### 5.2.1 删除词语

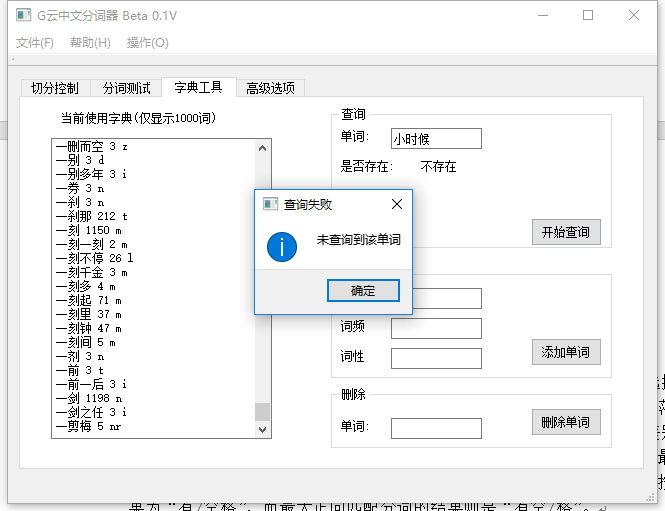
我首先查询一个在字典中存在的词”小时候”，成功的查询到了词语和其相关信息。随后删除，再次查询提示词语不存在。具体见图5-9，5-10，5-11。



查询存在 图5-9



删除词语 图5-10



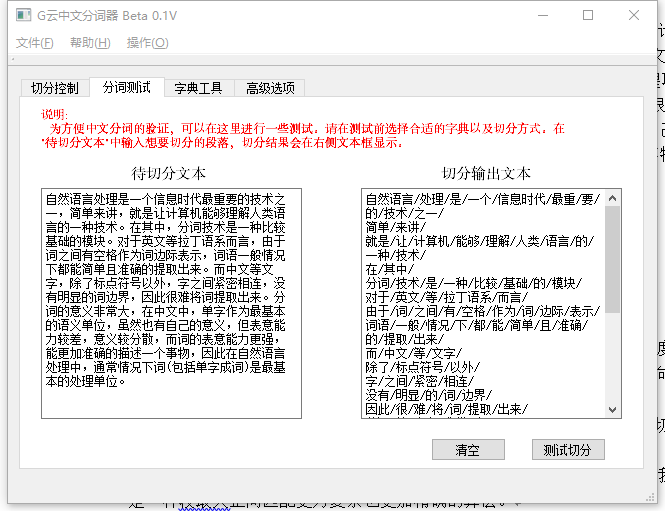
删除成功 图5-11

## 5.3多种切分算法的支持

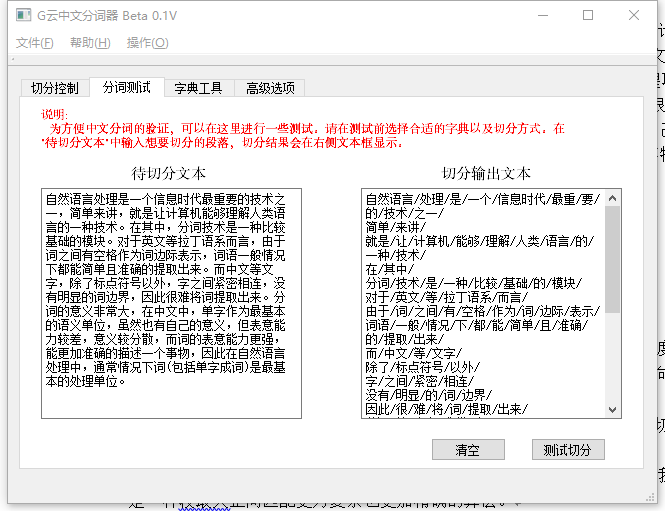
我的程序提供了多种分词算法供选择，在界面上同样提供了接口来选择分词算法。图形界面见图5-9。为测试具体的切分效果，我复制了报告的绪论部分一个段落进行测试。测试的结果见图5-12，5-13，5-14。总的来说，三种分词的结果没有太大的差别。可以看出，最大概率路径的MPSeg算法和用四条规则筛选的MMSeg算法更为精确，而最大正向匹配算法仅少数地方比上面两种分词效果差，比如输出结果第9行的“有空格”，按照前两种分词结果为“有/空格”，而最大正向匹配分词的结果则是“有空/格”。



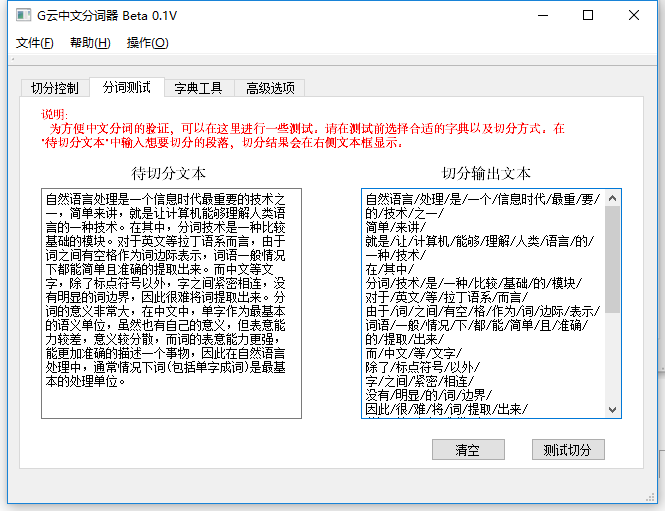
选择分词算法 图5-12



MMSeg分词算法 图5-13



MPSeg分词算法 图5-14

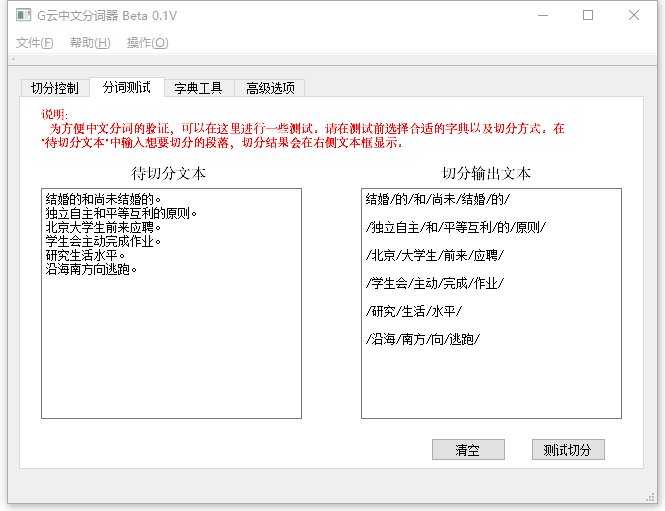


MaxSeg分词算法 图5-15

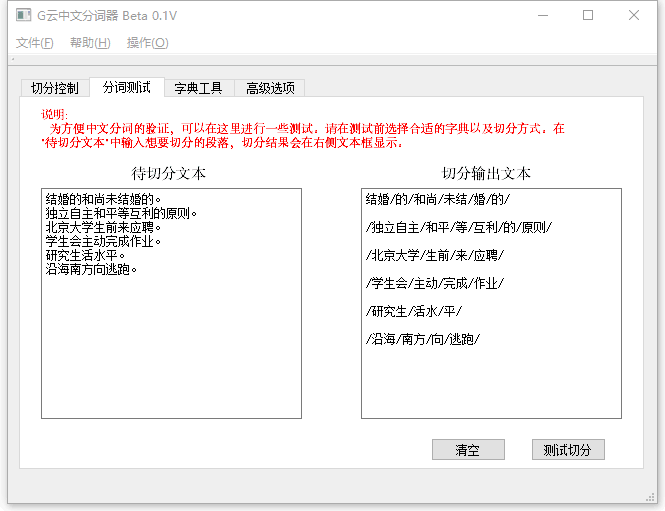
## 5.4中文语句歧义的问题

中文中很多语句会有歧义产生，这里我挑选了一些较难分词的句子，使用不同的分词方式来查看其对于歧义的解决情况。分词的结果见图5-16，5-17。由于MMSeg和MPSeg的分词结果完全一样，因此只截了一张图。

可以看到在这6个较难切分的句子中，使用MaxSeg切分全部出现了问题，而另外两种算法解决了4个句子的切分歧义，有两个句子不能正确切分。比如“沿海南方向逃跑”，无论是MMSeg还是MPSeg均无法正确切分。及主要问题在于字典中“沿”字的词频远小于“向”字的词频。MMSeg分词会筛选到第四条规则，然后比较单字词频，但由于“沿”字的词频小，于是选择了错误的切分方式。而MPSeg中“沿”“海南”“方向”三个词的词频之积小于“沿海”“南方”“向”的词频之积，因而做了错误的选择。可以看出，这两种分词方式还是很依赖字典词频的统计准确性的。



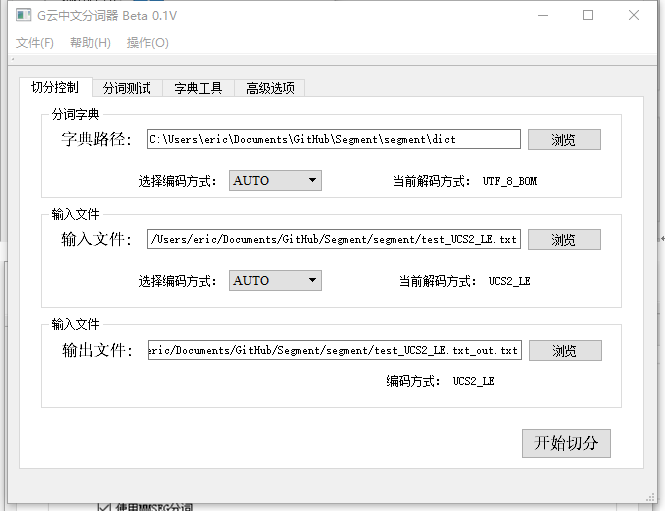
MMSeg和MPSeg的分词结果 图5-16



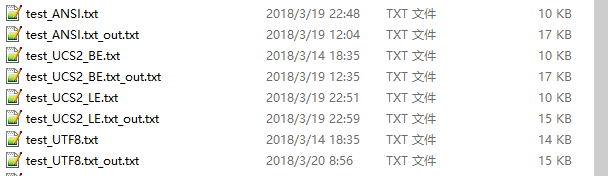
MaxSeg的分词结果 图5-17

## 5.5程序的兼容性(英文与编码)

我的程序支持了多种编码，虽然课设中没有要求，但我认为这是必须的。毕竟ANSI，Unicode，UTF8都是很常见的编码，一个只能支持某一种编码的项目是很难成为一个合格的软件的。加入记事本只能打开ANSI编码，打开其他编码的文本是乱码，那基本就不会有人去使用记事本了，都会去使用notepad++等更好用的编辑器。我测试这切分了不同编码的文本，均可以正确切分。见图5-18，5-19。

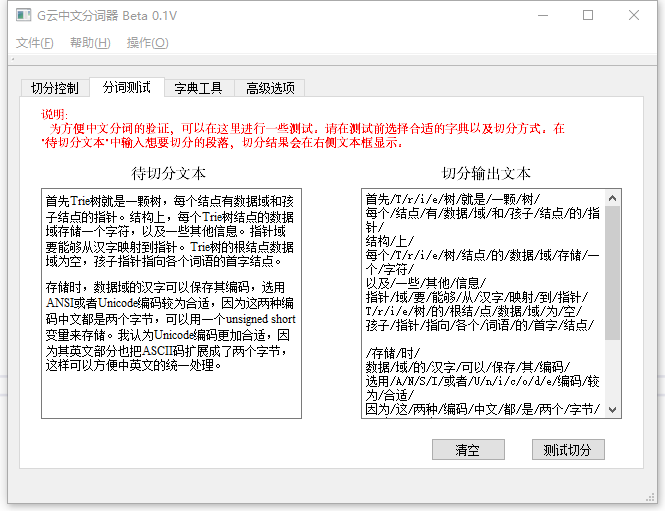


提供不同编码的检测与支持 图5-18

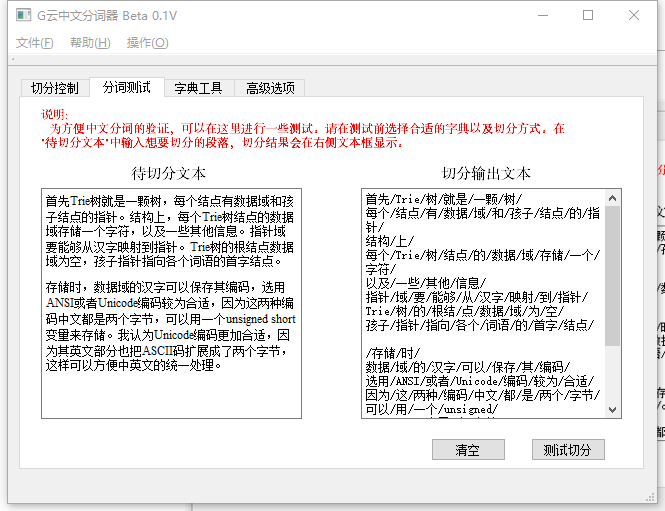


测试不同编码的切分 图5-19

另外我的分词是可以支持英文的，只要往里面添加英文单词即可正确的切分中文中混杂的英文。对于不存在的英文单词也不会报错，但会一个字母进行一次切分。我使用了报告中的一个段落做测试。并在直接测试后向字典中添加了“Trie”“ANSI”“unsigned”“Unicode”等几个在该段落中出现了的英文词语。测试结果见图5-20，5-21。



未添加英文单词时 图5-20



添加英文单词后 图5-21

# 编程过程中体会与总结

总的说我完成这次课设的代码量相当大，为了实现不同的功能也费了不少的心思。以前对C++了解的并不深入，甚至害怕C++的代码风格。在实现的过程中对C++更加熟悉了，也有了更深的体会。这也是我第一次做出一个如此完整，像一个可用的软件一样的东西。为此我非常自豪，很有成就感。

同时，在编写代码的过程中，我也碰到了许多问题，我将一部分记录了下来，以便以后查看。

## 6.1 一个内存访问越界Bug

在调试的时候碰到了一些bug。错误信息：

引发了异常: 读取访问权限冲突。\*\*\_Pnext\*\* 是 0xCDCDCDD1。 见图6-1。



错误信息 图6-1

最后找到了问题的症结所在。我在结构体Info中声明了一个vector的对象，并且用Info的指针手动分配一段内存空间。问题大概在于，我想要往分配的内存中拷贝Info对象的时候，出现了内存访问越界。有问题的代码经过精简之后如下:

#include<stdio.h>

#include<string>

#include<vector>

using namespace std;

struct Info {

vector<int> vacab;

Info() { vacab.clear();}

};

int main() {

Info \*infos = (Info\*)malloc(10 \* sizeof(Info));

//Info infos[10];

Info info;

infos[0] = info;

return 0;

}

也就是说，手动分配的内存空间足够实际上一个Info结构体占用的空间比sizeof(Info)要大。至于为什么目前还没有弄清楚。但应该是malloc时sizeof计算的空间，并没有给成员中类的对象留出足够空间。String对象和vector对象存在同样的问题。

还有一点非常关键，malloc分配空间并不会调用构造函数，这是new和malloc非常重要的一点不同，谨记。

**解决方法：**

1.不使用malloc改用new进行内存的初始化。

2.不要自己分配内存，把Info \*infos = (Info\*)malloc(10 \* sizeof(Info));替换成Info infos[10];

## 6.2 malloc与new混用造成的错误

和上面的错误产生原因是类似的，即malloc和new语法的混用。New会自动的调用其构造函数，而malloc不会。因此在该用new的地方使用了malloc会造成对象的成员未初始化，这样就会导致访问的其成员变量时发生一些错误。

总结起来，给复杂的结构指针分配空间时，尽量使用new而不是malloc可以减少一些错误。

## 6.3 vector的赋值问题

我之前对vector并不很熟悉，那么两个vector变量 之间该如何赋值呢？我在网上查阅了资料，基本说不能直接等号赋值，因为其并没有重载“=”。但是经过测试，发现这可能是以讹传讹，在vs2017编译环境下，可以使用“=”直接赋值。测试使用的代码如下：

#include<stdio.h>

#include<vector>

#include<iostream>

using namespace std;

int main() {

vector<int> v1;

vector<int> v2;

v2.push\_back(10); v2.push\_back(20); v2.push\_back(30);

v1 = v2;

for (unsigned int i = 0; i < v1.size(); i++)

printf("%d ", v2.at(i));

v2.clear();

for (unsigned int i = 0; i < v1.size(); i++)

printf("%d ", v1.at(i));

return 0;

}

可以看到，在赋值之后，再改变其值并不会影响原变量的值，也就是说，“=”是深度拷贝。使用“=”进行赋值是完全可行的。

# 参考文献

[1]Chih-Hao,Tsai. MMSEG: A Word Identification System for Mandarin Chinese Text Based on Two Variants of the Maximum Matching Algorithm. 2000-03-12.

[2] fxsjy. Jieba中文分词Python源码.

URL: <https://github.com/fxsjy/jieba>

[3]Jieba中文分词源码解析(一)

URL: <http://blog.csdn.net/daniel_ustc/article/details/48195287>

[4]Qt Assistant(QT官方帮助文档).

URL: http://doc.qt.io/