**北 京 邮 电 大 学**

**计算机应用编程实践**

**实验报告**



|  |  |
| --- | --- |
| 题目： | 多模式字符串匹配 |
| 姓名： | 田昌昊 |
| 学号： | 2019110680 |
| 班级： | 2019111307 |

计算机学院

2019年11月

目录

[一、实验任务与实验内容 3](#_Toc25168299)

[1. 实验任务： 3](#_Toc25168300)

[2. 挑战： 3](#_Toc25168301)

[3. 程序要求 3](#_Toc25168302)

[二、实验过程 3](#_Toc25168303)

[1. 主要的数据结构和流程 3](#_Toc25168304)

[1.1 版本一：朴素查找（strstr） 3](#_Toc25168305)

[1.2 版本二：多模式串的kmp(Multikmp) 3](#_Toc25168306)

[1.3 版本三：ac自动机匹配（ac\_auto） 4](#_Toc25168307)

[2. 实验过程 5](#_Toc25168308)

[三、遇到的问题 6](#_Toc25168309)

[1. pattern文件和string文件读取出错 6](#_Toc25168310)

[2. 第一个pattern匹配个数问题 8](#_Toc25168311)

[四、结果指标 10](#_Toc25168312)

[五、结论和总结 11](#_Toc25168313)

# 一、实验任务与实验内容

## 1. 实验任务：

1. 给定测试大规模中文文本串，数据量约800M，目标测试中文文本数据约3G
2. 待查找的key大约220万个词作为模式串，从搜狗新闻语料训练
3. GB2312编码格式
4. 需要从大文本串中查找出所有模式串出现的位置和次数

## 2. 挑战：

模式串的规模较大，需要构造一个高效数据结构来处理

## 3. 程序要求

1. 分别实现三个版本，程序分别命名为：
   1. strstr：朴素查找
   2. Multikmp：多模式串的kmp
   3. ac\_auto：ac自动机匹配
2. 输入数据
   1. 词典串pattern.txt(2256691个)
   2. 主串：string.txt
3. 实验结果result.txt
   1. 每行一个，模式串出现次数，但出现次数倒序排列
      1. Keyword1 1124
      2. Keyword2 1098
      3. 最后一行输出两个数字，用空格分割：

– 字符/字节比较次数

– 内存开销（kb）

# 二、实验过程

## 1. 主要的数据结构和流程

### 1.1 版本一：朴素查找（strstr）

数据结构就是普通的char\*。

遍历读取pattern，对于每个pattern遍历读取book进行匹配判断。匹配过程中，将pattern信息和匹配次数写入中间文件。匹配搜索完成后，读取中间文件并进行排序，得到新的排序文件。

### 1.2 版本二：多模式串的kmp(Multikmp)

数据结构也是普通的char\*和int\*。

next数组的获取如pdf教程所示代码：

void get\_next(int\* next, char\* p, int p\_length) {

int i = 0, j = -1;

next[0] = -1;

while (i < p\_length) {

if (j == -1 || p[i] == p[j]) {

i++;

j++;

next[i] = j;

}

else {

j = next[j];

}

}

}

遍历读取pattern，对于每个pattern计算next数组后，遍历读取book进行匹配判断。匹配过程中，将pattern信息和匹配次数写入中间文件。匹配搜索完成后，读取中间文件并进行排序，得到新的排序文件。

### 1.3 版本三：ac自动机匹配（ac\_auto）

ac自动机主要数据结构：

/// 输出函数的链表，存储所有的 pattern 所在的trie树的原node指针

typedef struct PatternList {

struct ACNode \*origin\_node; // 存储pattern所在的trie树的原node指针

struct PatternList\* next;

}\*PatternListPtr;

/// 转向函数的链表，存储结点所有的孩子结点

typedef struct ChildList {

char c; // 存储路径的char值

struct ACNode\* child;

struct ChildList\* next;

}\* ChildListPtr;

/// ac自动机状态结点

typedef struct ACNode {

int state\_id; // 结点编号

int depth; // 结点的深度，记录字符串的长度

int matched\_num; // 用于统计作为trie树，该node上面所带的pattern被匹配的次数

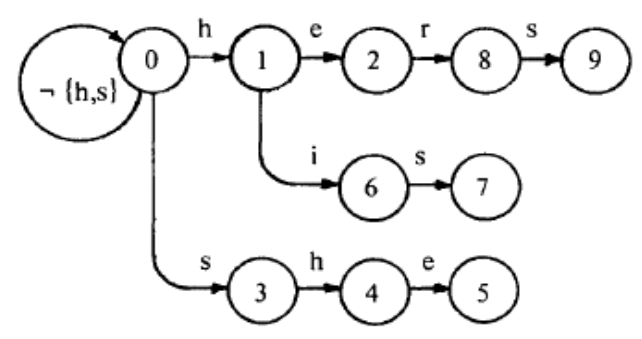
ChildListPtr child\_list[CLASSIFY\_NUM]; // 转向函数

struct ACNode \*fail; // 失效函数，匹配失败转移结点

PatternListPtr pattern\_list; // 输出函数

struct ACNode \*parrent;

}\* ACNodePtr;



以上数据结构中，ACNode是ac自动机的节点结构，包含节点状态id、节点深度和节点所代表的pattern的匹配次数（在output中被更新）；节点的转向函数名为child\_list，代表节点接下来的路径，路径包含一个char变量、所指向的ac自动机节点以及链表的下一条路径，其中CLASSIFY\_NUM是对转向函数进行分类，即对char变量获取hash值来分配到不同类的链表里面，减少查询的次数；失效函数是该节点匹配失效后，转向的下一个ac自动机节点；输出函数是到达该ac自动机后的输出集合链表，包含一个指向ACNode的指针，用于指向输出pattern在trie树结构中的原始节点，next指向下一个输出元素。

队列数据结构：

// 队列数据结构，用于构建失效函数使用，利用链表实现

typedef struct QueueList {

ACNodePtr node;

struct QueueList\* next;

}\*QueueListPtr;

typedef struct Queue {

int size;

QueueListPtr ql;

}\*QueuePtr;

由于需要构建失效函数，过程中要对trie树进行广度优先遍历，因此需要队列结构，该队列结构通过链表实现，Queue包含一个存储元素个数的变量，以及一个链表，链表则包含ac自动机的节点指针和指向下一个链表元素的next变量。

实现过程的逻辑为先读取pattern文件，构建trie树，然后进行广度优先遍历，构建失效函数，在构建trie树和构建失效函数的过程中，构建输出函数。完成以上工作后，遍历BOOK文件进行ac自动机的搜索，并记录匹配的次数。搜索完成后，遍历pattern文件，通过trie树进行搜索并将次数写入中间文件。之后读取中间文件并进行排序，得到新的排序文件。

## 2. 实验过程

在实验过程中，首先打算编写strstr.c程序，在读取数据的时候，发现对大字符串string文件的暴力查找速度很慢，对1个pattern遍历string文件需要30s左右，6个pattern需要3分钟左右，估测2256691个pattern需要783天，无法完成。而且为了估测这个过程，我写了一个中间进度的输出，来隔一段时间输出，从而查看进度。发现每次遍历string文件到50%-60%时，就判断文件结束退出循环了，说明文件的遍历存在问题。

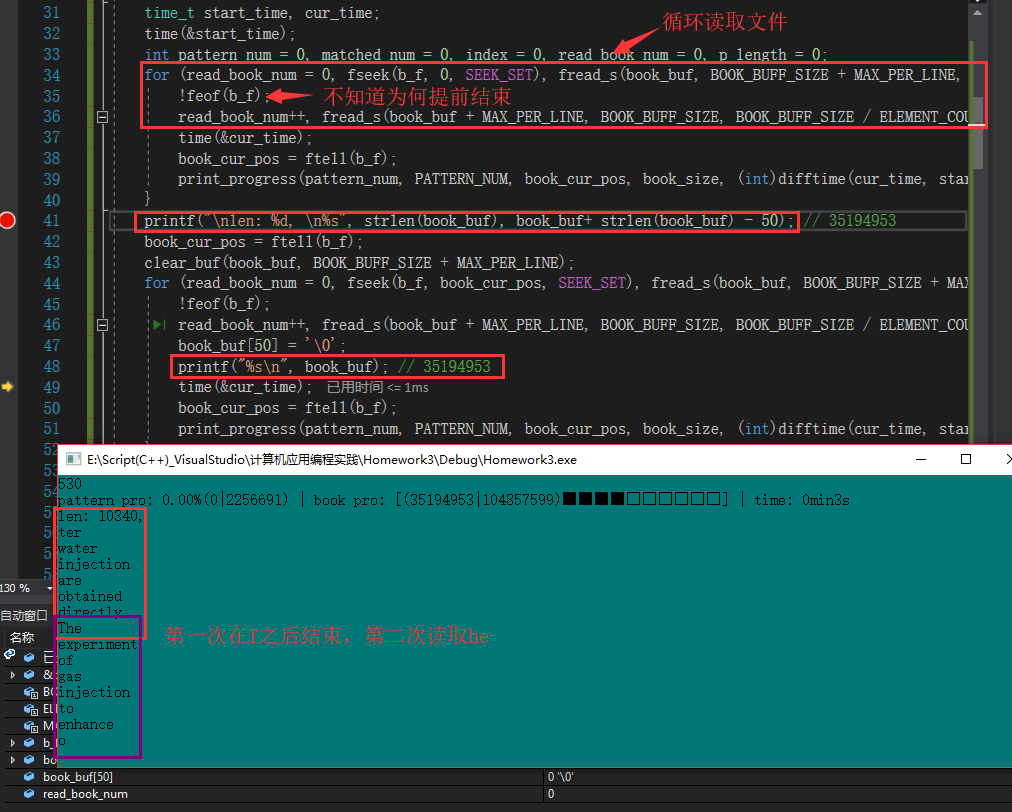
所以就转移了方向：先实现对pattern.txt和string.txt文件遍历读取。但是过程很不顺利，两个文件都存在非GBK编码的不可见字符，导致文件读取错误频出。最后选择了一种较为简单的方式直接对数据进行了过滤和预处理（详见章节三中的第1小节）。

之后开始编写朴素查找。

# 三、遇到的问题

## 1. pattern文件和string文件读取出错

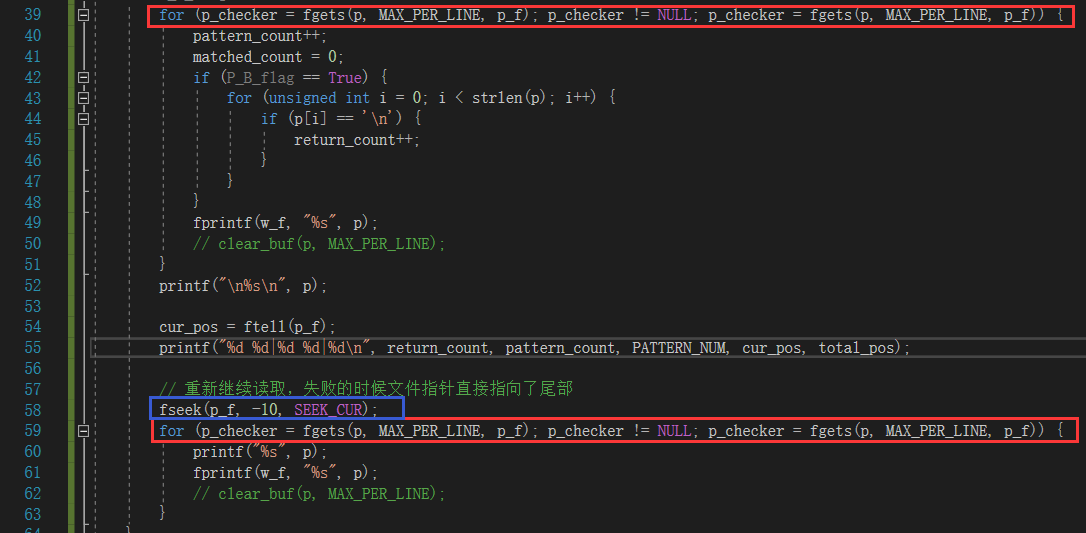
在实验过程中，首先打算编写strstr.c程序，在读取数据的时候，我写了一个进度条输出来展示string文件的查找进度，但是发现在读取到一半的时候经常直接退出循环，即 !feof(FILE\*) 条件判断失败，从而文件的遍历读取结束（如下图所示）。



在上图中即展示了循环读取string文件的代码块，两个printf和两次输入如上图框内所示。发现读取过程中提前跳出了第一个循环，之后必须通过fseek()函数调整文件指针，才能够重新进入文件读取后续内容。但是后续读取也会提前弹出。导致无法完整读取完string文件。

之后去找问题，决定先编写对两个文件的遍历读取函数。发现问题很大。

pattern文件也有相同的情况。不过读取过程采用的不是fread\_s()函数，而是fgets()函数。读取过程中发现读取到倒数第二行“龟龄”之后，文件判断读取完毕，退出循环，后续的“龟龙”和“龟龟”两行pattern未读取到（后续调查发现在“龟龄”之后有两个 ^z 符，导致文件识别结束）。调整文件指针（下图fseek中的参数SEEK\_CUR改为SEEK\_END也可，因为第一次跳出循环的时候，文件指针已经指在了文件末尾，此时SEEK\_CUR==SEEK\_END）倒退10个字节时，才刚好能够读取出来最后两行（如下图所示）。且后续查询资料表示，应该使用fgets()的返回值是否为NULL来判断是否成功读取，而不是feof(FILE\*)。

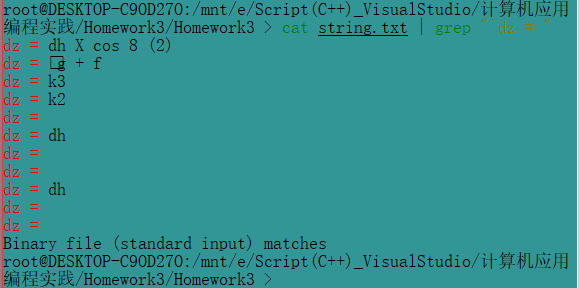


然后读取到最后两行后，重新生成了文件pattern\_fix.txt，和源文件比较如下。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **版本** | **大小（KB）** | **行数** | **字数** | **字符数** | **统计示意图** |
| **pattern.txt** | 17664 | 2256691 | 1579364 | 18087750 |  |
| **pattern\_fix.txt** | 19868 | 2256691 | 1579364 | 20344438 |

对string.txt的处理就很麻烦了，中间尝试了超多方法，而且文件太大，无法查看内容是否正确（如下图，第一次弹出时读取到了 [dz = ] 字符串，利用命令grep匹配上下文，发现读取一部分之后检测到了二进制流之后程序退出了，无法验证）。

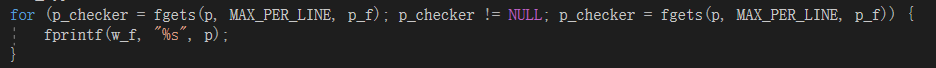
同时发现fread\_s()函数在读取较大文件出错后，得到的文件指针位置可能是错误的，所以舍弃该读取方式，利用fgets()函数读取。



最终决定利用iconv -c -f gbk -t utf-8 string.txt > string\_2\_utf.txt 命令转换为 utf-8编码（-c是用于忽略转化过程中的错误，不忽略会导致命令运行过程中检测到编码错误而退出），再将 string\_2\_utf.txt 转回为 gbk 编码的string\_3\_utf-gbk.txt，之后又重复验证了一遍，得到string\_5\_utf-gbk-utf-gbk.txt，并用string\_5\_utf-gbk-utf-gbk.txt作为初始预处理文件。对string\_5\_utf-gbk-utf-gbk.txt利用fgets()遍历读取，并写入新文件string\_debug.txt，发现一次性读取成功。对string\_debug.txt重新遍历读取，生成的新文件string\_debug\_debug.txt，发现和string\_debug.txt完全相同。故将string\_debug.txt作为稳定版本文件。（这些文件的参数比较如下）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **版本** | **大小（KB）** | **行数** | **字数** | **字符数** |
| **string.txt** | 1140050 | 28978391 | 39079428 | 1167411061 |
| string\_2\_utf.txt | 1507905 | 28978391 | 43294573 | 1544093880 |
| string\_3\_utf-gbk.txt | 1139565 | 28978391 | 39078902 | 1166913991 |
| string\_4\_utf-gbk-utf.txt | 1507905 | 28978391 | 43294573 | 1544093880 |
| string\_5\_utf-gbk-utf-gbk.txt | 1139565 | 28978391 | 39078902 | 1166913991 |
| string\_debug.txt | 990318 | 28971496 | 39069316 | 1014085434 |
| string\_debug\_debug.txt | 990318 | 28971496 | 39069316 | 1014085434 |
| **string\_fix.txt** | 990318 | 28971496 | 39069316 | 1014085434 |
| **string\_fix-utf.txt** | 1358614 | 28971496 | 43284346 | 1391220127 |
| **string\_fix-utf-gbk.txt** | 990318 | 28971496 | 39069316 | 1014085398 |

文件读取并写入新文件的代码块如下：



至此，后续代码利用稳定的string\_fix文件和fgets()函数进行读取操作。

PS：测试过程中，对string\_debug.txt利用fread\_s()函数读取，并写入新文件，每次得到的结果都和源文件不同（在慢慢变大，也不知道在文件内加入了什么东西），而且统计信息的时候，有ftell(p\_f)操作则一次性读取完成，没有ftell(p\_f)操作会导致fread\_s()函数提前读取失败跳出循环。所以就不采用了。（诶。。。能力不足，似乎fread\_s更应该读取二进制流，同时指定element\_size和element\_count参数，在gbk和utf编码中，英文的编码长度和中文不同，所以可能不适用）。

2019.11.12上午5:50，张铮助教发布了清洗之后的string.txt，测试可以读取完成，使用 wc统计发现如下：



上图显示只有1行，1个字符，1101608280个字节。

2019.11.13日，张铮助教发布了新的清洗后的string,txt，该数据去除了乱码和换行，通过fgetc(FILE\*)进行遍历读取。我用自己清洗的文件将实验完成后，更改为张铮助教的数据，进行运行和提交，结果无误。

## 2. 第一个pattern匹配个数问题

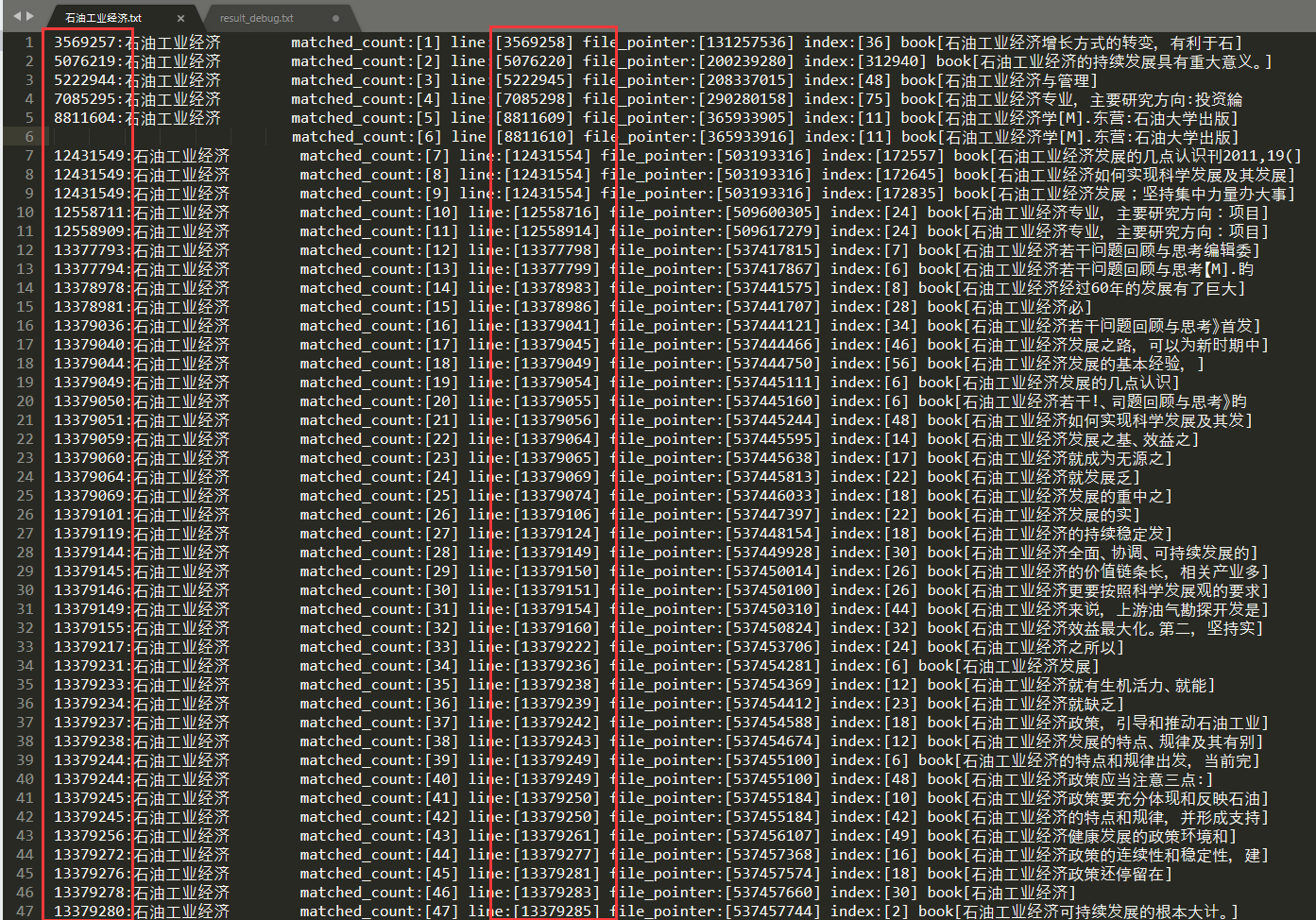
使用暴力搜索strstr.c版本运行之后，得到第一个pattern——“石油工业经济”为61个匹配项，而利用KMP算法搜索Multikmp.c版本得到的匹配个数为49个，说明两者有问题（以下部分着重说明strstr.c的问题，KMP是临界值判断存在bug，后续已改，结果正确，此不赘述）。

在Ubuntu系统中，利用命令：cat string\_4\_utf-gbk-utf.txt | grep -o "石油工业经济" -n > 石油工业经济.txt 对上一节中的中间生成文件进行匹配得到个数为60个。利用命令cat string\_fix-utf.txt | grep -o "石油工业经济" -n > 石油工业经济-utf.txt 对上一节中的中间生成文件进行匹配得到个数为60个。两者得到的信息相同。

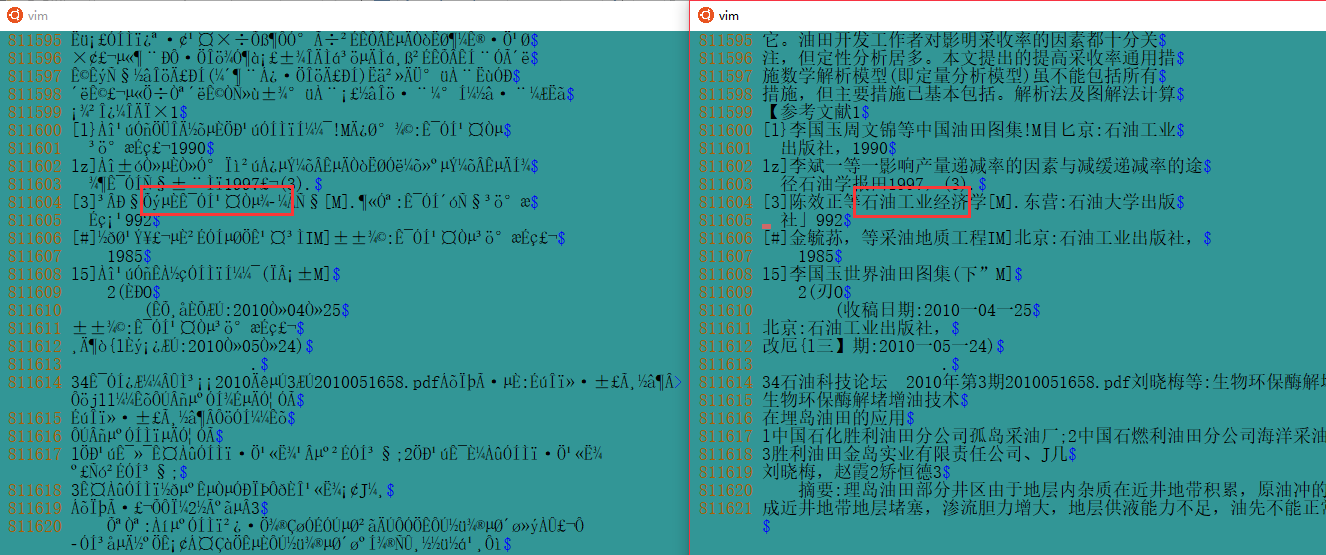
**题外话：**同时对张铮助教发布的文件进行转码 iconv -c -f gbk -t utf-8 string-v2.txt > string-v2-utf.txt 得到文件并利用相同命令进行匹配搜索。Emmm…出不来结果（如下图）。说明文件还存在问题，无法使用。



然后输出匹配信息进行比较，如下：

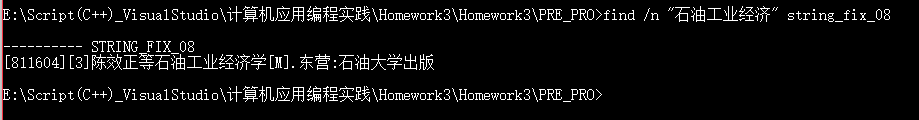


该图左侧为Ubuntu匹配结果，右侧为程序匹配信息结果，红框内为行数，发现60个基本上能够对应，多出来的1个在8811610行处，book内容为“石油工业经济学[M].东营:石油大学出版”。为了查看这一行的状况，利用Ubuntu将大文件string\_fix.txt 按照每100w行split后，vim进入第8个文件查看，如下图：



发现只存在一行，而不是连续的两行。可能是在程序遍历string\_fix.txt时，由于某些原因读取到了相同的一行（在比较匹配信息的图中可知，文件指针后移了，说明前后两次匹配是两次fgets()操作读取的不同行），或者是在转码过程中转码错误，将两行中的一行舍去了。

在windows cmd命令行操作搜索也只得到一行，如下图：



由此分析后，肯定是程序的问题。

添加debug输出book\_len，发现在新的一行中，只有10个字符（解释了前面分析的文件指针后移的问题），但是从第11个字符开始匹配得到了匹配结果。结论：读取新的一行需要将原buff清空，防止前面读取的信息对后续匹配的影响。对应的debug信息如下：

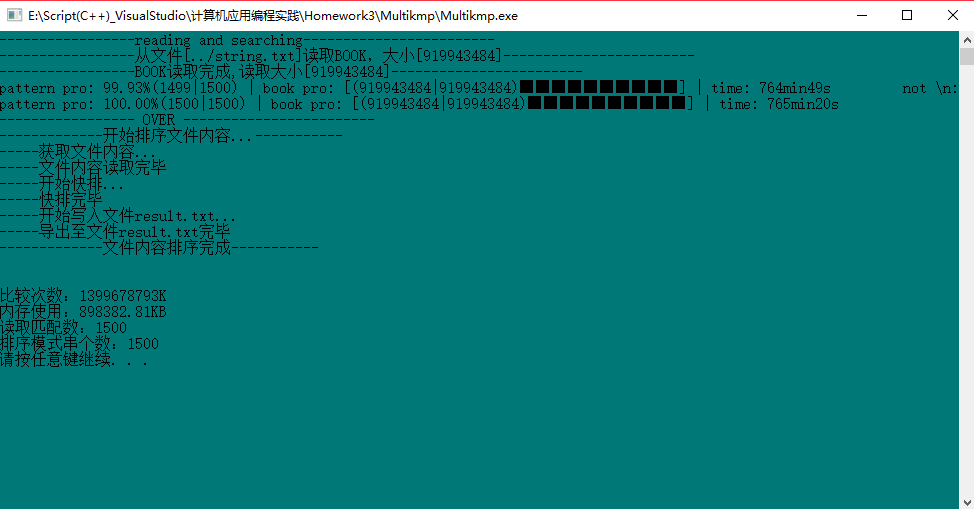
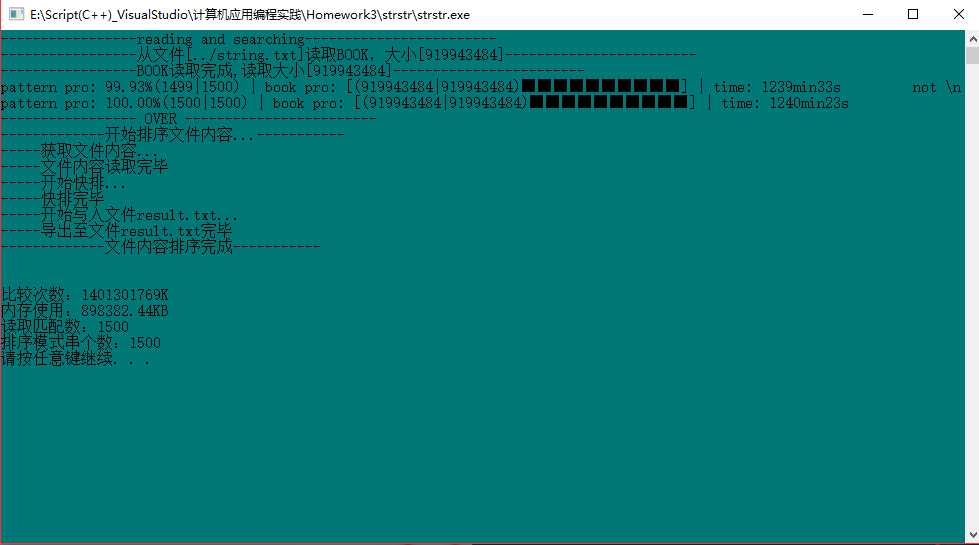
matched\_count:[6] line:[8811605] file\_pointer:[365933916] index:[11] book\_len:[10] book:[石油工业经济学[M].东营:石油大学出版]

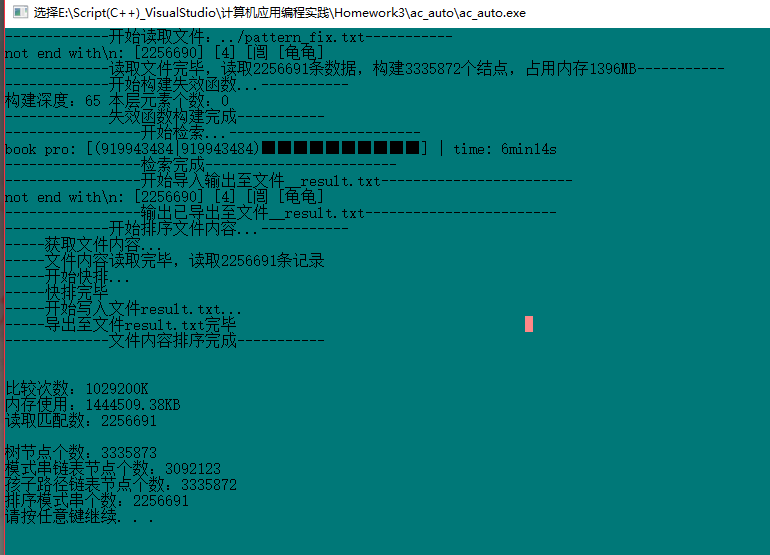
添加clear\_buf操作之后，问题解决！

# 四、结果指标

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **版本** | **字符字节比较次数(K)** | **内存使用（kb）** | **pattern数** | **运行时间（以下仅为检索时间）** |
| **strstr** | 14013011769 | 898382 | 1500 | 1240min23s |
| **Multikmp** | 1399678793 | 898382 | 1500 | 765min20s |
| **Ac\_auto** | 1029200 | 1444509 | 2256691 | 6min14s |

运行结果展示如下：





# 五、结论和总结

本次实验很烦人的地方是文件中的乱码导致读取文件就折腾了3天多，之后对数据进行了清洗和预处理，得到string\_fix数据作为实验数据之后，开始编写三个版本。由于程序运行时间过长，效果无法完全对比，只能对比前面一些pattern的搜索值是否正确。

之后我自行清洗了数据，进行程序编写，过程比较顺利。最后采用张铮助教清洗的数据后，结果对比正确。