计算机应用编程实验报告

计算机应用编程实验3：

2020110656 高垚 2020111562 杨恩

# 实验目标与意义

# 随着网络技术的普及，互联网的信息逐渐庞大，如何在海量且冗余的信息中找到用户所需的结果是互联网每天都要应对的难题。信息的基数与增长速度，用户查找内容的多样性，查询结果的实时动态变化都使精确搜索成为互联网面临的挑战。

# 本实验的目的是设计一套能够在庞大的词库中寻找特定目标的算法。根据pattern文本建立成树形有限自动机，再利用string文本中匹配相似的字符串。本实验的难点在于，树形自动机的建立，以及搜索算法。树形自动机包括状态转向函数和失效函数的算法，这两种包含了多种情况，需要综合考虑。

# 系统设计与实现

根据实验目标要求，需要平衡消耗时间的长度和空间的占用率，在实验中，利用KMP算法，完成对pattern文件中200多万个字符串的树形自动机的建立，并通过树形自动机算法，搜索在string.txt文件中匹配的字符，并写入result.txt文件中。与之前实验很大的一点不同，在于本次实验逻辑上把string.txt文件中的字符串当成一行，但是文本中却依旧包含换行符和空格符，使字符串和字符串间有间隔。

## KMP算法

KMP算法可以高效地使用已知的信息，达到时间效率地最大利用率。具体操作是每当匹配过程中出现字符串比较不等时，不需回溯主串，而是充分利用已经得到的“部分匹配”结果，过滤掉那些多余的比较，将模式串向右“滑动”尽可能远的一段距离后，继续进行比较，从而提高模式匹配的效率。

### 实验思路

本节概括了实验的流程。

- 为了在海量词库中，精确查找我们所需要的词。我们需要进行以下几个操作。

1. 根据实验二中的程序，我们可以选择不同容量的节点作为查找树的结构。因此我们首先进行了叉树的选择，以保证实验所占用内存最小。

2. 在选择了256叉树作为搜索树后，我们需要设计有效的256叉树的节点数据结构。我们为每一个节点都设置了父亲指针，兄弟指针，孩子指针和失效指针，不同类别的指针有不同的功能，且保证每个节点所包含的数据不为空。

3. 根据pattern文件建立256叉树，此时没有利用失效指针。利用广度优先遍历方法，一个把每一层的开头节点的指针都保存的数组和一个把每一层的最后一个节点的指针都保存的数组，构建失效指针。

4. 根据string文件搜索，当目标字符和正在查找的字符不匹配时，利用失效指针，快速跳到下一个可能和目标字符相匹配的节点。

5. 当匹配成功的时候，为查找树的节点中的输出结构体的参量重新赋值。

6. 把输出的字符及其对应的信息保存，然后用快速排序把字符按照出现的次数大小进行排序。

7. 按照格式要求，在目的文本输出。

其中涉及到的函数有：

|  |
| --- |
| /\*本实验中用到的函数\*/  void init\_tree()//初始化一个搜索树。  void insert\_recoder(char \*str)// 插入一个节点。  FileHandler\* open\_file(char\* file\_name, char\* open\_type) //创建指向文本的指针和内存缓冲区。  void query\_recoder(char \*str) //查询节点。  void make\_turn() //跳转到下一个可能与目标值相匹配的节点。  void out\_to\_result()//输出排序后的结果。 |

### 2.1.2 搜索树的建立

搜索树的节点中，包含五个指针。其中每个节点最多仅有一个子节点指针，指向下一个节点；每个节点最多一个兄弟指针，兄弟指针指向的是同一层的节点，每个节点最多有256个孩子，每个孩子之间用兄弟指针连在一起，而每个节点只有一个指向第一个孩子的指针。

|  |
| --- |
| /\* 构造树的结构 \*/  typedef struct node  {  char str;  unsigned int is\_end;  struct node \*child;  struct node \*father;  struct node \*brother;  struct node \*turn;  struct node \*out;  }Node;  /\* 输出内容 \*/  typedef struct out  {  char \*str;  unsigned int num;  int line\_1;  int line\_2;  int line\_3;  }Out;  /\* 初始化搜索树 \*/  void init\_tree()  {  Root = create\_new\_node();  Return;  }  /\*初始化一个节点 \*/  Node \*create\_new\_node()  {  Node \*\_node = (Node \* )malloc(sizeof(Node));  \_node->str = 0;  \_node->is\_end = 0;  \_node->father = NULL;  \_node->child = NULL;  \_node->brother = NULL;  \_node->turn = NULL;  \_node->out = NULL;  return p;  };  node \*child  node \*father  node \*brother  node \*turn |

该数组中存放着指向每一层最后一个节点的指针

该数组中存放着指向每一层第一个节点的指针

NULL

NULL

NULL

NULL

NULL

NULL

NULL

NULL

root

### 搜索树节点的插入

由于搜索树的结构体中包含的指针过多，所以在建立时有许多初始化操作，导致搜索树的插入操作复杂。因此我们设计了多个不同功能的函数，一起完成节点的插入步骤。其中make\_turn函数是KMP算法的快速匹配核心，能够快速跳到下一个可能匹配的节点。其数据结构如下。

|  |
| --- |
| /\* 插入节点操作 \*/  void insert\_recoder(char \*str){}  /\*给node对应的字符串的出现次数加1\*/  void add\_num(Node \*node, int key, int line\_num){}  /\*遍历每层的节点，构建它们的失效指针\*/  void make\_turn(int key){} |

### 搜索树节点的查询

搜索树的查询需要通过读取string文本中的词，并且忽略文本中的换行符。把目标值与当前指针指向的节点的值进行比较，然后根据比较的结果进入下一个子节点。其数据结构如下。

|  |
| --- |
| /\*查询函数，按字节遍历所给字符串，在不同状态间跳转，并输出 \*/  void query\_recoder(char \*str,int line\_num){} |

### 2.1.5 搜索树节点的输出

本实验要求输出按照匹配次数的大小进行排序，因此在搜索树中就包含了输出结构体，把输出所需要的信息包含在该输出中。此外，利用快速排序算法为输出的字符串排序，并在目的文本中输出。

|  |
| --- |
| /\*一个递归算法，把所有输出的节点用它们的turn指针连起成一个队列\*/  Node \*out\_start;//队头  Node \*out\_end;//队尾  int \*out\_num=0;//记录有多少个输出节点  /\*把node到root的路径上的字符拼成字符串并返回\*/  void get\_out(Node \*node){}  /\*把node到root的路径上的字符拼成字符串并返回\*/  char out[512];//暂存从node到root拼出来的字符串  char \*ch=NULL;//指向拼出来的字符串的开始位置  void calc\_str(Node \*node){};  /\*快速排序算法,把输出的数据按照num的大小排序\*/  void quick\_sort(Node \*\*arr,int begin,int end){};  /\*此函数被main调用，负责遍历树，找到输出节点，整理并输出\*/  void out\_to\_result(){}; |

### 2.1.6 文件的操作

本实验涉及到了三个文件，其中两个为输入文件dict.txt和string.txt，和一个输出文件result.txt。dict.txt文件是包含了127万个词的文件，而string.txt包含了1.7万个词。result.tx文件中所包含的词包含了输入两个文件的交集。

本实验中，利用的文件操作涉及到了指针。其数据结构设计如下：

|  |
| --- |
| /\* 文件指针的建立 \*/  typedef struct FileHandler {  FILE\* file;//  char file\_name[99];  int open\_status;  int point;  unsigned int buffer\_size;  char\* buffer;  }FileHandler;  /\*创建文件 \*/  FileHandler\* open\_file(char\* file\_name, char\* open\_type) {  return file\_handler;  }; |

**(1) 文件的读取**

本实验利用缓存区读取文件。首先通过全局变量为缓存区设定大小。当缓存区内部填充满时，再进行读取。这样读取的次数就大大减少了。文件读取的数据结构如下：

|  |
| --- |
| /\* 文件的读取 \*/  int read\_line(FileHandler\* file\_handler, char\*\* str) {  return 1；//读取成功  } |

**(2) 文件的写入**

本实验中利用的写入操作有一个特点，即直接通过对缓存区进行写入，当缓存区内部填充满时，再进行写入，直到完全把结果输出到文本中。文件写入的数据结构如下：

|  |
| --- |
| /\* 文件的写入 \*/  int write\_line(FileHandler\* file\_handler, char\* str) {  char target[20];//we have 6306 numbers  static int num = 1;//numbers will change  char\* t = target;  sprintf(target,"%d",num);  return 0；//写入成功  }; |

# 实验分工

按照上述的系统设计，实验分工如下：

高垚实现了搜索树的建立和查找。

杨恩实现了搜索树的输出。

# 实验结果与分析

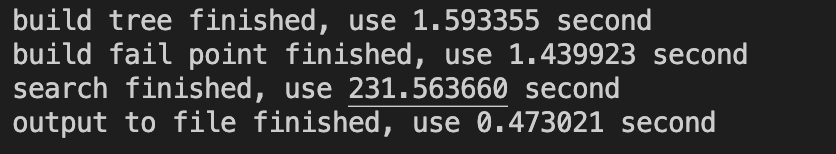
经过实验，程序共爬取了1270574行字符，其中，建立搜索数耗时1.376849 s，建立失效指针耗时1.400058s，搜索共花费210.722672 s，输出文件过程花费0.480729 s。总时间为213.980308s，运行内存峰值216.2M，最终得到45174行字符，程序运行结果如下图1,2。

图 1 实验的输出

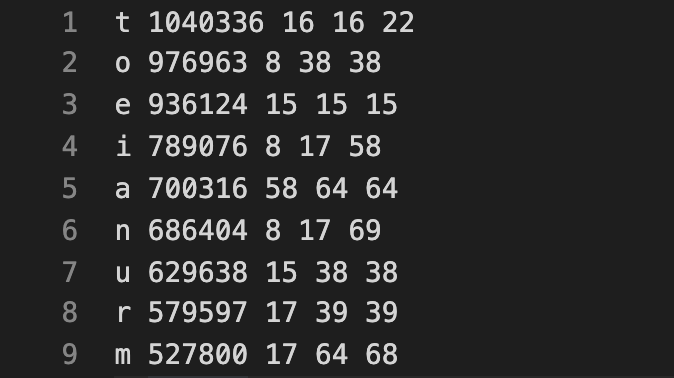


图 2 实验的文本输出

# 实验问题

**（1）把结果输出于目的文本中的输出过于缓慢**

解决方法：为每个节点设置输出结构体，把需要输出的信息保存在结构体中，然后利用快速排序为输出的字符串排序，再统一输出到内存中。

**（2）报错提醒conflicting types for “”**

解决方法：头文件（.h）中与.c程序中的函数的声明不一致，需要改用成一致的声明。

# 实验结论

综上所述，实验程序能够对海量数据进行筛选和搜索，并能在较快时间和满足需求的存储空间内完成计算。实验可以通过对数据文本的遍历，建立多种树形数据结构，达到查询的目的，顺利完成实验目标，符合实验预期结果。