词库检索的简单算法比较

词库描述

1. 词库由标准CSV文件保存
2. 数据格式为：

序号	关键词	词性	概念	...

3. 单词+词性构成唯一一个单词
4. 一个单词1-8个词性
5. 单词最长32字节
6. 百万级数据量
7. unicode编码

关于数据处理

1. 实验数据来自搜狗实验室-互联网词库-15万条高频词，采用UTF-8编码
2. 所有数据预先按照关键词直接使用strcmp(char const * _Str1, char const * _Str2)函数排序
3. 原数据中不存在单词多词性的现象，实际数据若出现单词多词性情况，直接将词性编号 $1-8$ 链到关键词尾部即可
4. 数据读到程序中保存关键词，同时使用ftell(_Stream)函数记下该条数据在源文件中的文件偏移量，用long类型保存，查询到相关数据项使用fseek(_Stream, _Offset, _Origin)函数定位在原词库文件中的位置
5. 实验询问所有存在的数据共157202条，运行过程中记录索引的建立时间，每个询问的查询时间，所有查询中最快时间与最慢时间以及整个程序的运行时间

算法

分块

1. 将所有N条数据分成M块，每一块记录块首数据的文件偏移量，即索引
2. 检索前预先依次载入或计算所有块的文件偏移量
3. 由于原先数据有序，所以M个块必然也有序，所以可以二分查询要检索的数据在哪一块中
4. 锁定块后，使用索引定位到文件中，顺序读取每一个数据比较，直到找到数据或者读取到了文件尾或者下一个块的块首

算法分析

建立索引需要预先扫描所有数据，$O(N)$

索引的保存与载入使用二进制文件保存整个数组

分块搜索需要二分查找块，再在块内顺序查找，$O((\log M+(\frac{N}{M}))*Maxlen)$

易知当$M=\sqrt{N}$时，复杂度降到最低，所以建立索引时根据数据量动态分块需要扫描两遍所有数据，第一遍计算数据量，第二遍建立索引

空间需要额外记录M个long类型的索引（文件偏移量）

实验结果

建立索引未包括在时间记录内，时间记录从索引的载入开始

索引载入时间: 0ms

绝大多数数据查询时间分布在0ms以及1ms内，查询过程中需要不断进入文件读取数据，存在几十ms左右的查询时间

查询最快时间:0ms

查询最慢时间:38ms

15W次数据查询总时间:49.923s

代码、实验结果文件

索引建立代码

分块查找代码

运行结果

运行时间记录

二分

1. 记录所有数据的文件偏移量作为索引
2. 载入索引，直接在所有数据中二分查找

算法分析

建立索引需要预先扫描一遍数据，$O(N)$

索引的保存与载入使用二进制文件保存整个数组

二分查找数据，$O(\log N)$

空间需要额外记录N个long类型的索引（文件偏移量）

实验结果

建立索引未包括在时间记录内，时间记录从索引的载入开始

索引载入时间: 1ms

绝大多数数据查询时间分布在0ms，查询过程中需要不断进入文件读取数据，存在几十ms左右的查询时间

查询最快时间:0ms

查询最慢时间:34ms

15W次数据查询总时间:18.000s

代码、实验结果文件

索引建立代码

分块查找代码

运行结果

运行时间记录

Trie

建立标准Trie树，在Trie树上查找

基本结构

```
struct Trie {
    bool is_word;
    char word;
    long file_index;
    Trie *index[200];
    Trie() {
        is_word = false;
        word = '\0';
        file_index = 0;
        for (int i = 0;i < 200;++i)
            index[i] = NULL;
    }
};

class TrieTree {
public:
    TrieTree();
    ~TrieTree();
    bool del(Trie *curr);
    bool insert(char *str, long file_index);
    bool get(char *);
private:
    Trie* root;
};
```

is_word 变量记录当前结点是否走到了一个完整的单词

word 变量记录当前结点的字

file_index 变量记录该单词（若 is_word == true）在原文件中的位置（文件偏移量）

Trie *index[200] 200个结点指针指向每一个字对应的的下一个结点

正常Trie若每个结点存储一个中文字，由于UNICODE中编码了6W+个中文字，若每个结点都开辟6W+个结点指针，极其浪费空间。UTF-8编码中一个中文字基本为三个字节（关于UTF-8编码的说明），且试验中发现每个字节的值都为负数，且取相反数后发现最大不会超过200，因此将一个中文字拆开成多个字节，用char类型保存，这样下来虽然树的深度加深了三倍，每个结点的分叉数却大大减小，且易于保存与查询

实验结果

Trie的建立:2.959s

查询时间全部在0~1ms

建树+15W次数据查询总时间: 3.516s

进程内存: 565MB

代码、实验结果文件

代码

运行结果与时间记录

压缩Trie(Compressed Trie)

标准Trie存在某颗子树退化成单链的情况（将中文字拆分成三个结点的做法更是如此），这样将大大浪费空间。

查询文献发现有Double Array Trie的实现方法，用两个数组实现trie，且压缩了所有单链

尝试实现发现难度较高，思考后自行构造一个压缩Trie(Compressed Trie)，将单链压成一个字符串，地址传给单链子树的第一个父节点

基本结构

```
struct Trie {
    bool is_tail, is_end;
    long file_index;
    char* word;
    Trie *index[200];
    Trie() {
        is_tail = false;
        is_end = false;
        file_index = 0;
        for (int i = 0;i < 200;++ i)
            index[i] = NULL;
        word = "\0";
    }
};

class TrieTree {
public:
    TrieTree();
    ~TrieTree();
    bool del(Trie *curr);
    bool insert(char *str, long file_index);
    long get(char *str);
    bool check(char *a, char *b);
    void adjust(Trie *curr);
    void load();
    void search();
private:
    Trie *root;
};
```

is_tail 变量记录当前结点是否存在单链词

is_tail 变量记录当前结点（不包括单链字符串）是否构成一个单词

file_index 变量记录该单词（若 is_word == true）在原文件中的位置（文件偏移量）

word 变量记录单链字符串

Trie *index[200] 200个结点指针指向每一个字对应的的下一个结点

构造方法

根据插入数据字符串用指针curr从树根往下访问：

若curr指针指向的下一个即将访问的结点为空:

新建结点new_node

若剩下串非空，将new_node的 is_tail 变量标true，将new_node的 is_end 变量标false，同时用插入数据字符串的剩下部分构造一个字符串，地址传给new_node的word 指针

若剩下串为空，则将new_node的 is_tail 变量标false，将new_node的 is_end 变量标true

将new_node结点传给当前结点的对应下一层的结点指针

若curr指针指向的下一个即将访问的结点非空:

若该节点 is_tail值为真，则进行调整，将该节点保存的单链字符串的第一个字符取出。易知该字符指向的下一层结点一定为空，于是新建结点，将剩下的单链字符串拉至新建的结点。同时:

若剩下的单链字符串非空，将 is_tail 变量标true，将is_end 变量标false，同时用剩下的单链字符串构造一个字符串，地址传给word 指针

若剩下串为空，则将 is_tail 变量标false，将is_end 变量标true

curr指向下一个结点

实验结果

Trie的建立:2.584s

查询时间全部在0~1ms

建树+15W次数据查询总时间: 4.665s

进程内存: 231MB

代码、实验结果文件

代码

运行结果

运行时间记录

算法对比

1. 速度：在查询速度上，两种Trie树远远快过二分与分块搜索，因为Trie树的查找速度只与查询数据的长度有关，同时在实现上，Trie树只需要不断访问加载到内存中的的结构即可，而所实现的二分搜索与分块搜索两种

算法需要不断进行文件的读取（但其实也可以将关键词加载到内存中）。二分的查询复杂度在 $O(\log N)$ 量级，而分块的复杂度在最优情况下在 $O(\sqrt{N})$量级。压缩Trie稍稍比标准Trie慢一点点。

2. 空间使用：分块搜素需要M个long类型的变量记录每个块块首的数据在词库文件中的文件偏移量，而二分搜索需要记录所有数据在词库文件中的文件偏移量，试验中使用二进制保存索引，157202条数据使用分块搜索的索引文件大小为2KB， 而二分搜索的索引文件大小为615KB。两种Trie的空间使用远远大于分块搜索与二分搜索算法（这正是典型的空间换时间的例子）。标准Trie存在很多单链子树，15万条数据请求了500+MB的空间；优化后的压缩Trie大大减少了空间的浪费，15万条数据请求了200+MB的空间。百万级的数据可以考虑使用优化之后的压缩Trie

关于UTF-8编码

UTF-8 from Wikipedia

Number of bytes	Bits for code point	First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
1	7	U+0000	U+007F	0xxxxxxx			
2	11	U+0080	U+07FF	110xxxxx	10xxxxxx		
3	16	U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
4	21	U+10000	U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx