实验报告–基于 Trie 树和并行式优化的英文文本词频统计

HNU github:Orange_cyanic

摘要:本文基于数据结构与算法课程所学 Trie 树数据结构，结合多线程编程和互斥锁的应用，实现了对给定英文文本文件的读入、数据优化、字典树构建、词频统计，最后输出结果文件的词频统计过程。

一、数据结构设计

1.1 问题分析

任务为编写程序统计一个英文文本文件中每个单词的出现次数（词频统计），并将统计结果按单词出现频率由高至低输出到指定文件中。同时要求采用的数据结构为 trie 树，也称为字典树。题目额外说明了文本为仅由字母组成的字符序列。

1.2 数据结构实现

由要求可知，只需要在树节点留出 26 个子节点指针即可。同时题目只对单词频率有获取要求，因此可将对节点是否为单词的判断(isword)和频率(frequency)合并成一个成员。数据结构设计如下。

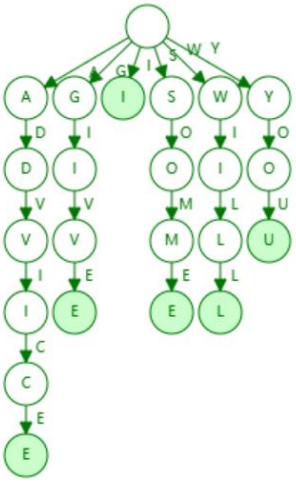
```
class TrieNode {
 public:
    TrieNode* childNode[26];
    int frequency;};
TrieNode() {
        frequency = 0;
        for (int i = 0; i < 26; i++) {
            childNode[i] = NULL;
        }
    }//构造函数
```

同时可实现插入函数。细节如下:

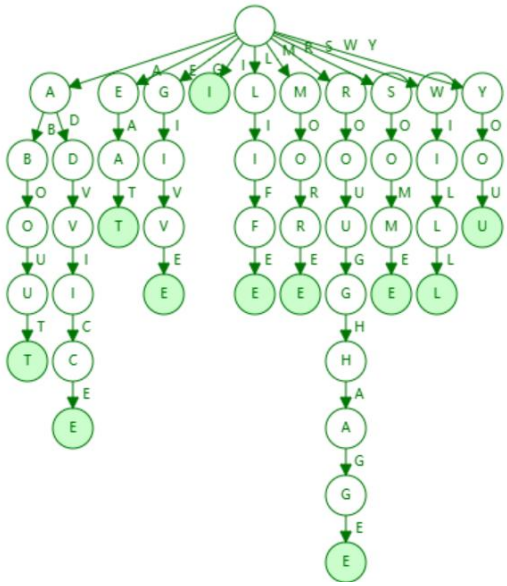
```
void insert(TrieNode* root, string word) {
        TrieNode* node = root;
        for (char c : word) {
            if (node->childNode[c - 'a'] == NULL) {
                node->childNode[c - 'a'] = new TrieNode();
            }
            node = node->childNode[c - 'a'];
        }
        node->frequency++;
    }//这里 string word 的传参是为了适配之后文件读入时的插入操作。
```

1.3 样例演示

以下是插入样例“I will give you some advice”后的字典树



接着，我们插入“eat more roughage”得到如下：



接下来演示在已有 will 单词时插入 willing 的结果：

其他文本样例类似，不一一展示。

二、文件读入与文本预处理

2.1 问题与算法分析

要求为打开当前目录下文件 in.txt，从中读取英文单词进行词频统计。在此单词为仅由字母组成的字符序列。包含大写字母的单词应将大写字母转换为小写字母后统计。此外，由于输入文件是英文小说，因此不会出现无意义的单词。

在打开文件后，为了率先处理数据而不影响后续插入，此时可以利用 getline 读取一行文本，对所有大写字母转小写，非英文字母转空格的处理，最后再将 stringstream 分割成 string 形式插入到字典树中。其中使用 stringstream 的好处是，它可以自动处理字符串中的分隔符，并且提供了一种简单的方式来逐个提取字符串中的单词。代码实现如下。

2.2 函数实现

```
void readWords(TrieNode* root, string filename) {
    ifstream file(filename);
    string line;
    while (getline(file, line)) {
        for (char& c : line) {
           c = isalpha(c) ? tolower(c) : ' ';//将文本字符标准化
        }
```

```
            stringstream ss(line);
            string word;
            while (ss >> word) {
                root->insert(root, word);
            }
        }
        file.close();
    }
```

三、基于多线程的词频统计

3.1 基本遍历选择

要遍历整棵字典树的节点，且保持父节点-子节点的顺序性，首选深度优先搜索（DFS）。

3.2 多线程优化

考虑到文本数量很大，这里使用多线程加快遍历和数据搜集速度。

首先创立线程集和键值对向量。线程集用来存储线程，键值对向量用来获得单词和频率数参加下一步的排序，所有线程共享键值对向量。

在每一次递归使用 DFS 函数时都创立一个新的线程，并将已有的字母加至随递归一同产生的字符串中。一旦线程检测到节点对应的频率不为 0，即使用互斥锁将向量锁住，同时添加单词和频率，然后解锁继续线程。

3.3 代码实现

```
    mutex mtx;
    vector<pair<string, int>> wordsFreq;//定义在全局变量中
    void dfs(TrieNode* node, string word) {
        if (node == NULL) {
            return;
        }
        if (node->frequency > 0) {
            lock_guard<mutex> lock(mtx);
            wordsFreq.push_back({word, node->frequency});
        }
        for (char c = 'a'; c <= 'z'; c++) {
            dfs(node->childNode[c - 'a'], word + c);
        }
    }
    void dfsMultiThread(TrieNode* root) {
        vector<thread> threads;
        for (char c = 'a'; c <= 'z'; c++) {
            threads.push_back(thread(dfs, root->childNode[c - 'a'], string(1, c)));//对于每个字母，函数创建一个新的线程，并将其与对应的子节点和一个长度为 1 的字符串绑定。
        }
        for (auto& t : threads) {
            t.join();
     }
    }
```

3.4 互斥锁应用分析

互斥锁的作用是确保在同一时间只有一个线程可以访问被保护的代码块或共享资源。如果没有互斥锁的保护，多个线程可能会同时修改键值对向量，导致竞态条件（race condition）[1]的发生。竞态条件可能导致数据不一致或不可预测的结果。

通过使用互斥锁，只有一个线程可以获得 wordsFreq 的访问权限，其他线程必须等待该线程释放锁才能继续执行。这样可以确保在任何时候只有一个线程修改 wordsFreq，从而避免了竞态条件的发生。

四、排序函数构建与文件输出

4.1 排序原理

对于上一步获得的键值对向量集需要对其进行排序操作。而由于成员是键值对，sort 函数没有现成的 compare 参数，需要另行实现。

这里使用了 lambda 表达式。Lambda 表达式是一种在被调用的位置或作为参数传递给函数的位置定义匿名函数对象（闭包）的简便方法。Lambda 表达式的基本语法如下[2]:

```
[capture list] (parameter list) -> return type { function body }
```

借助此，我们也可以实现 pair 键值对的比较。首先比较频率，接着比较字母的字典序。

4.2 代码实现

```
void topKFrequentWords(TrieNode* root, int k) {
    dfsMultiThread(root);
    sort(wordsFreq.begin(), wordsFreq.end(), [](pair<string, int>& a, pair<string, int>& b) {
        if (a.second == b.second) {
            return a.first < b.first;
        }
        return a.second > b.second;
    });//排序算法体现
    ofstream file2("out.txt");
    for (int i = 0; i < k; i++) {
        cout << wordsFreq[i].first << " " << wordsFreq[i].second << endl;
        file2 << wordsFreq[i].first << " " << wordsFreq[i].second << endl;
    }
    file2.close();
}
```

此外是头文件引用和主函数:

```
#include <iostream>
#include<thread>
#include <vector>
#include <fstream>
#include <thread>
#include <mutex>
#include <utility>
#include <sstream>
#include <algorithm>
int main( int argc, char* argv[]) {
    string filename = "in.txt";
    readWords(root, filename);
    topKFrequentWords(root, 100);
    return 0;}
```

五、算法性能分析

5.1 时间复杂度

insert 函数的时间复杂度为 O(m)，其中 m 是单词的长度。在最坏情况下，需要遍历整个单词并插入到 Trie 树中。

readWords 函数的时间复杂度为 $O(nm)$，其中 n 是文件中的行数，m 是每行的字符数。该函数需要遍历文件的每一行，并将每个单词插入到 Trie 树中。

dfs 函数的时间复杂度为 $O(26^m)$，其中 m 是单词的长度。该函数通过深度优先搜索遍历 Trie 树的所有可能路径，并将频率大于 0 的单词及其频率存储在 wordsFreq 向量中。

dfsMultiThread 函数的时间复杂度为 $O(\frac{26^m}{p})$，其中 m 是单词的长度，p 是处理器的数量。该函数创建了 26 个线程，每个线程都调用 dfs 函数进行深度优先搜索。由于这些线程是并行运行的，所以总的时间复杂度会除以处理器的数量。然而，由于线程的创建和销毁也需要时间，且存在线程间的竞争，所以实际的时间复杂度可能会高于这个值。

topKFrequentWords 函数的时间复杂度为 $O(nm + klog(k))$，其中 n 是文件中的行数，m 是每行的字符数，k 是要输出的前 k 个频率最高的单词数量。该函数首先调用 dfsMultiThread 函数进行深度优先搜索，然后对 wordsFreq 向量进行排序，并输出前 k 个频率最高的单词。

综上所述，整个程序的时间复杂度为 $O(nm + klog(k) + \frac{26^m}{p})$。

其中 n 是文件中的行数，m 是每行的字符数，k 是要输出的前 k 个频率最高的单词数量，p 是处理器的数量。由于评测机器为四核，p 取 4。

最终经过反复修改，代码在 CG 上的平均运行时间为 5.86s，较单线程优化了接近 30s。

```
共有测试数据:2
平均占用内存:1.980K    平均CPU时间:5.86064S    平均墙钟时间:5.84203S
```

5.2 空间复杂度

TrieNode 类：TrieNode 类占用的空间主要取决于其成员变量。在这里，TrieNode 类有一个大小为 26 的指针数组 childNode，用于存储子节点。由于数组的大小固定为 26，不会随着数据量的增加而改变，因此它的空间复杂度是 O(1)。然而，每个 TrieNode 对象都可能有 26 个子节点，因此 Trie 树的总空间复杂度是 O(n)，其中 n 是插入到 Trie 树中的单词的总数量。

wordsFreq 向量：wordsFreq 向量用于存储单词及其频率。它的大小等于 Trie 树中频率大于 0 的节点的数量，因此它的空间复杂度是 O(n)，其中 n 是插入到 Trie 树中的单词的总数量。

readWords 函数：readWords 函数中定义了两个字符串 line 和 word，它们的大小取决于输入文件中的行长度和单词长度。因此，readWords 函数的空间复杂度是 O(m)，其中 m 是输入文件中最长行或最长单词的长度。

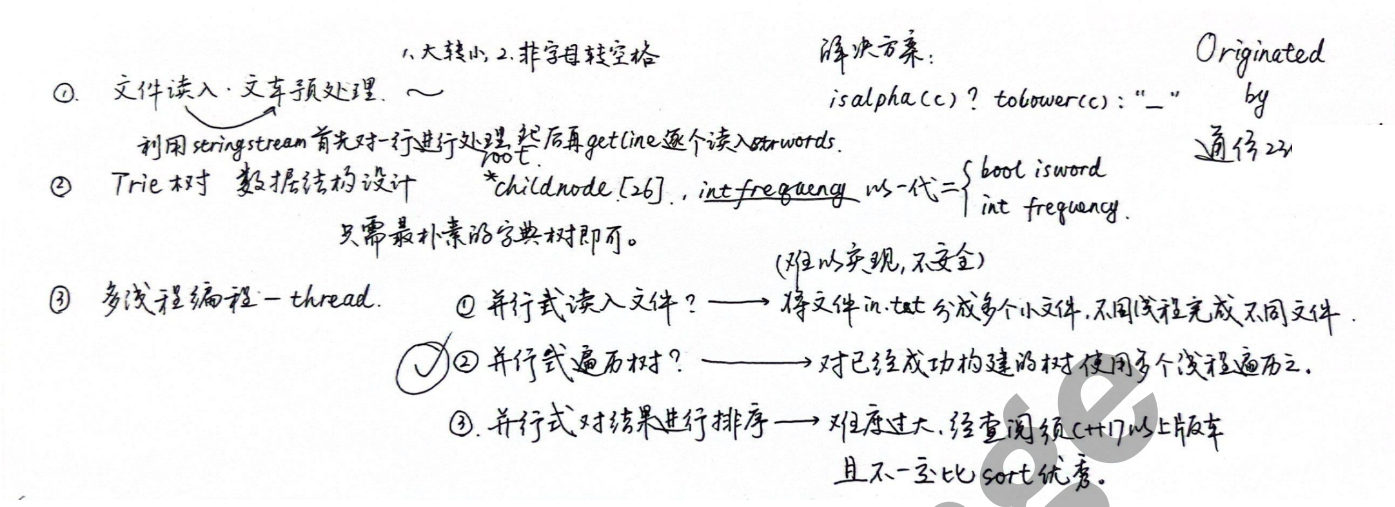
dfs 和 dfsMultiThread 函数：这两个函数的空间复杂度主要取决于递归深度，即 Trie 树的最大深度，因此它们的空间复杂度是 O(m)，其中 m 是插入到 Trie 树中的最长单词的长度。

topKFrequentWords 函数：topKFrequentWords 函数的空间复杂度取决于 wordsFreq 向量的大小，因此它的空间复杂度是 O(n)，其中 n 是插入到 Trie 树中的单词的总数量。

综上所述，**代码的总空间复杂度是 $O(n+m)$**，其中 n 是插入到 Trie 树中的单词的总数量，m 是插入

到 Trie 树中的最长单词的长度。

六、设计手稿与参考文献引用



[1] ISO/IEC 9899:2011 – Information technology — Programming languages — C[引用时间 2024-04-29]

[2] 深入浅出 C++ Lambda 表达式：语法、特点和应用_c++lambda 表达式作为函数参数的用法-CSDN 博客[引用时间 2024-04-29]