

软件工程-作业一-最长单词链-实验报告（41 组）

一、实现原理：

0、数据结构：

单词类的内部成员：每个单词包含一些相关信息，主要有 head（开头字母序号），tail（结尾字母序号），next（指向下一个开头相同的单词的指针）等。

采用邻接表存储单词：将单词按照首字母进行划分，设置一个 26 行的邻接矩阵，0~25 行分别对应开头字母为 a~z 的单词串，同行单词以链表形式存储。

（注：本文中所述单词串，指的是输入文件中以同一字母为首的全部单词序列。）

使用链表存储单词链：找到目标词链后，将其存入链表 nowList，同时设置 maxList 记录查询时最长词链。还存储了一些辅助词链。

1、计算最多单词数量的单词链：

算法思路：采用深度优先搜索算法，以每个单词作为词链开头，对全局进行深度优先搜索，当搜索至尽头（以词链最后一个词的尾字母为开头的行中没有剩余单词可用），则回溯、继续搜索。该算法可保证获取全局最优解，但时间复杂度为指数级，在数据规模较大（例如 100~1000 词）时，性能较差。

若采用贪心算法，则无法保证求得最优解，但能快速得到理想的次优解。采用的策略是：每次选取剩余词数最多的单词串进行搜索，搜索至尽头（条件同上述的 DFS）时直接返回。

关键代码截图：

①单词已存入邻接矩阵 wordList。下图所示循环：遍历每个单词，以其为首，搜索词链。若已达到设定的搜索时间，则直接打印、返回。其它的操作包括对“暂存当前词链”的 nowList 进行初始化、对每个节点设定搜索标记 uFlag、回溯等。

```
for (int i = 0; i < 26; i++) {
    Tmp = wordList[i];
    if (timenow - startTime > timelim) { writeResult(1); return; }
    while (Tmp->word != "") {
        if (timenow - startTime > timelim) { writeResult(1); return; }
        nowLen++;
        nowNum += Tmp->len;
        nowList = new WordNode(Tmp->word);
        nowNode = nowList;
        Tmp->uFlag = true;
        fSearch(Tmp->tail);
        Tmp->uFlag = false;
        Tmp = Tmp->next;
        nowLen = 0;
        nowNum = 0;
    }
}
```

②确定某次开头后，搜索其后继链的 fSearch 函数如下所示。该函数包含的参数 rank 代表上一词尾对应的字母序号，Tmp 首先指向以该序号开头的单词串的头。在未达到搜索尽头（即 Tmp->word 不为初始设置的“”）时，逐个将合规的单词放入 nowList，增加 nowLen 的长度，并进入下一层搜索；回溯时，还原 nowLen 的值和该节点的标记状态。

```

void fSearch(int rank) {
    timenow = time(NULL);
    if (timenow - startTime > timelim) return; //time limit, avoid ...
    WordNode* Tmp = wordList[rank];
    while (Tmp->word != "") {
        if (timenow - startTime > timelim) return;
        if (Tmp->uFlag == true) { //has searched, find next
            Tmp = Tmp->next;
            continue;
        }
        //insert into now list:
        Tmp->uFlag = true;
        WordNode *newNode = new WordNode(Tmp->word);
        nowNode->next = newNode;
        nowNode = newNode;
        nowLen++;
        nowNum += newNode->len;
        fSearch(nowNode->tail);
        Tmp->uFlag = false;
        Tmp = Tmp->next;
    }
}

```

当搜索达到尽头，则将当前长度 nowLen 与最长长度 maxLen 进行比较。若当前链更长，则更新最长链 maxList，回溯。若当前链不够长，则直接回溯。

```

if (Tmp->word == "") {
    if (!tSet.empty()) { ... }
    if ((nowLen > maxLen) && wcFlag) { //update maxList
        maxLen = nowLen;
        Tmp = nowList;
        WordNode *maxNode = new WordNode(Tmp->word);
        maxList = maxNode;
        Tmp = Tmp->next;
        while (Tmp != NULL) {
            WordNode *newNode = new WordNode(Tmp->word);
            maxNode->next = newNode;
            maxNode = newNode;
            Tmp = Tmp->next;
        }
        //back search:
        nowLen--;
        Tmp = nowList;
        if (Tmp == NULL) return;
        if (Tmp->next == NULL) nowList = NULL;
        else {
            while (Tmp->next->next != NULL)
            {
                Tmp = Tmp->next;
            }
            nowNum -= Tmp->next->len;
            Tmp->next = NULL;
            nowNode = Tmp;
        }
    }
    return;
}

```

2、计算字母数最多的单词链：

算法思路：同“词数最多”的算法设计。事实上，我们认为这两种功能的实质都是“寻找有向有环图中的加权最长路径”。如果图中每条边的权值都为 1，则对应以“词数最多”为标准的搜索问题；如果图中每条边的权值对应其指向节点的单词字母数量，则对应“字母数最多”为标准的搜索问题。

实际编程中，给每个单词节点增设成员 len，代表它包含的字母数；在进行搜索、比较的过程中，设置 nowNum 值记录当前的总字母数，更新 nowNum，并在搜索达到尽头时与 maxNum 进行比较，决定是否更新 maxList，而后回溯。

关键代码截图：

依旧遍历每个单词作为单词链头，对其后继采用深搜。评价标准改为 nowNum。

```
for (int i = 0; i < 26; i++) {  
    Tmp = wordList[i];  
    if (timenow - startTime > timelim) { writeResult(1); return; }  
    while (Tmp->word != "") {  
        if (timenow - startTime > timelim) { writeResult(1); return; }  
        nowLen++;  
        nowNum += Tmp->len; // nowNum: sum of char  
    }
```

搜索过程中，维护 nowNum 的值。

```
nowNum += newNode->len;  
fSearch(nowNode->tail);
```

搜索尽头，同样进行检查、回溯等处理：

```
else if ((nowNum > maxNum) && !wcFlag) { // update maxList  
    maxNum = nowNum;  
    Tmp = nowList;  
    WordNode *maxNode = new WordNode(Tmp->word);  
    maxList = maxNode;  
    Tmp = Tmp->next;  
    while (Tmp != NULL) {  
        WordNode *newNode = new WordNode(Tmp->word);  
        maxNode->next = newNode;  
        maxNode = newNode;  
        Tmp = Tmp->next;  
    }  
    nowLen--;  
    Tmp = nowList;  
    if (Tmp == NULL) return;  
    if (Tmp->next == NULL) nowList = NULL;  
    else {  
        while (Tmp->next->next != NULL)  
        {  
            Tmp = Tmp->next;  
        }  
        nowNum -= Tmp->next->len; // back search for nowNum  
        Tmp->next = NULL;  
        nowNode = Tmp;  
    }  
    return;
```

3、指定单词链开头或结尾字母（允许指定任意长度字符串作为“备选集”）：

算法思路：依照上述两种功能的搜索算法，我们都采用“遍历所有单词，将其作为链头，深度优先搜索”的方式，因此指定单词头的搜索非常容易实现：只要将遍历的单词由所有单词改为“以给定字母为开头”的单词。实际上相当于对上述两种功能的剪枝。

指定尾部单词的功能也可依照前述算法增加部分操作进行实现：在搜索到一个可更新 maxList 的词链后，检查其末尾单词是否在指定范围。若满足要求，则照常更新；若不满足要求，则回溯。

两种功能相互不冲突不矛盾，可混合使用，且支持输入一串字符作为备选。例如，含有 -h cababccc -t eggfeefg 的指令，将被理解为：找出开头为 a || b || c，且结尾为 e || f || g 的词链。相较于默认要求的只允许输入一个字母，更加灵活、适用性强。

关键代码截图：

① “-h”：直接在头部遍历过程中进行筛选即可。

```
sort(hSet.begin(), hSet.end());
hSet.erase(unique(hSet.begin(), hSet.end()), hSet.end()); //hSet去重字符
for (unsigned int i = 0; i < hSet.length(); i++) {
    if (timenow - startTime > timelim) { writeResult(1); return; }
    int num = hSet[i] - 97;
    Tmp = wordList[num]; //choose word's begin with chars in hSet
    while (Tmp->word != "") {
        if (timenow - startTime > timelim) { writeResult(1); return; }
        nowLen++;
        nowNum += Tmp->len;
        nowList = new WordNode(Tmp->word);
        nowNode = nowList;
        Tmp->uFlag = true;
        fSearch(Tmp->tail);
        Tmp->uFlag = false;
        Tmp = Tmp->next;
        nowLen = 0;
        nowNum = 0;
    }
}
writeResult(1);
```

② “-t”：在达到每次搜索尽头时，判断词链结尾是否在 tSet 范围内，决定采纳 or 回溯：

```

if (Tmp->word == "") { //end of one search
    if (!tSet.empty()) { // -ttttt
        bool tailFlag = false;
        for (unsigned int i = 0; i < tSet.length(); i++) {
            if (nowNode->tail == tSet[i] - 97) {
                tailFlag = true;
                break;
            }
        }
        if (!tailFlag) { //if tail not fit, then back search
            nowLen--;
            Tmp = nowList;
            if (Tmp == NULL) return;
            if (Tmp->next == NULL) nowList = NULL;
            else {
                while (Tmp->next->next != NULL)
                {
                    Tmp = Tmp->next;
                }
                nowNum -= Tmp->next->len;
                Tmp->next = NULL;
                nowNode = Tmp;
            }
            return;
        }
    }
}
}

```

4、指定单词链的单词个数：

算法思路：基本思路与功能 1、2 相近，依旧以每个单词依次作为链头，深搜找出其后继的词链，当满足给定长度时进行输出、回溯。

优化思路：该算法耗时随单词数增多、指定词链长度增长而呈指数级增长，**一方面**是因为指定词链长度越长，深搜的深度就越深，搜索规模呈指数级增长；**另一方面**，则是由于单词数越多，深搜需要进行的次数就越多，且满足要求的链的数量也呈指数级增长，将这些链输出到文件中的 I/O 开销不容忽视，某些情况下文件读写耗时甚至明显多于搜索耗时。由此，采用以下方法进行优化：

①减少搜索层数：为每个单词节点增设一个容器 next2，存储其后继连接的长度为 2 的所有词链。增设容器的操作在文件预处理时即完成，且耗时非常少（对于 1000 词数规模，该预处理耗时远不足 1s）。在进行搜索时，原始算法将“搜索以该词结尾的字母作为头的单词集合”，每一步搜索步长为 1；新算法将直接从单词节点中提取“2 词链”，并在其末端进行下一步搜索，每一步搜索步长为 2。理想状况下，这一操作能使深度搜索的实际操作层数减少近半。

②加快文件读写：原始算法每次获取到一条合规的单词链，便进行输出，进行了多次写文件操作；现设置一个长度为 2000*nSet 的 bufferList，其中 nSet 即为设定的词链长度。每次获取到一条合规链后，便将其加入到 bufferList 中，直到 bufferList 装满或全局搜索结束时，将 bufferList 中的链一次输出到文件中。理想状况下，这一操作能使写文件的开销减少到千分之一的程度。之所以选用 buffer，而不选择将所有获取的单词链拼起来、在最后直接输出一条链，是因为对于规模较大的数据，输出结果的存储空间极大，全部存入缓存中极易造成溢出。（经实验，对于词数 1000、词链长度为 5 的情况，输出数据文件大小为 GB

级，更多词数、更长的指定长度时，完全输出可达 TB)

③其他优化：由于大规模数据集本身搜索和输出开销过大，算法中设计了限时系统，当运行时间超出了最大允许限时，则程序自动退出，并输出当前的最优解。这部分优化能让用户在程序无力完全处理数据时，给予尽可能多的成果回馈。

关键代码截图：

①搜索思路依旧为遍历单词、分别为头、寻找尾链。由于采用了“两步搜索”，需要根据输入 n 值的奇偶决定是否需要进行一次“一步搜索”。函数中加入了 time 相关函数，用于控制程序的运行时间。

```
for (int i = 0; i < 26; i++) {
    Tmp = wordList[i];
    if (timenow - startTime > timelim) {return; }
    while (Tmp->word != "") {
        if (timenow - startTime > timelim) { return; }
        nowLen++;
        nowList->word = Tmp->word;
        nowList->tail = Tmp->tail;
        nowNode = nowList;
        Tmp->uFlag = true;
        if (nSet % 2 == 0) { //n != odd, then one step first(after that, len == 2), and two step each
            nSearch(wordList[Tmp->tail], 1);
        }
        else { //n == odd, then 2 step each time
            for (auto & i : Tmp->next2) {
                if (isRepeat(i)) continue;
                if (timenow - startTime > timelim) { return; }
                nowNode = nowList->next;
                nowNode->word = i->Word1st->word;
                nowNode->tail = i->Word1st->tail;
                nowNode->next->word = i->Word2st->word;
                nowNode->next->tail = i->Word2st->tail;
                nowNode = nowNode->next;
                nowLen += 2;
                i->Word1st->uFlag = true;
                i->Word2st->uFlag = true;
                nSearch(i->Word2st, 2);
                nowLen -= 2;
                i->Word1st->uFlag = false;
                i->Word2st->uFlag = false;
            }
        }
    }
}
```

②无论先采用一步搜索或两步搜索，之后的搜索统一采用两步搜索。一步搜索的过程与原始-w 的算法基本一致，不同之处只在于后继搜索时采用 2 步，以及长度合规时先存入 buffer 等待输出。

```

if (nowLen == nSet) { //fit len
    if (!tSet.empty()) { ... }
    nListNum++;
    buffNum++;
    if (isFull()) { ... }
    else { //insert nowList into buff
        WordNode* TnNode = nowList;
        for (int i = buffNum*nSet - nSet; i < buffNum*nSet; i++) {
            buffList[i]->word = TnNode->word;
            TnNode = TnNode->next;
        }
    }
}
else { //then 2 step
    for (auto & i : Tmp->next2) {
        if (isRepeat(i)) continue; //if repeat, then skip it
        nowNode = nowList;
        for (int K = 1; K <= nowLen; K++) nowNode = nowNode->next; //right position at list
        nowNode->word = i->Word1st->word;
        nowNode->tail = i->Word1st->tail;
        nowNode->next->word = i->Word2st->word;
        nowNode->next->tail = i->Word2st->tail;
        nowNode = nowNode->next;
        nowLen += 2;
        i->Word1st->uFlag = true;
        i->Word2st->uFlag = true;
        nSearch(i->Word2st, 2); //searching by 2
        nowLen -= 2;
        i->Word1st->uFlag = false;
        i->Word2st->uFlag = false;
    }
}
}

```

③在两步搜索中，若长度合规，则将 nowList 存入 buff，isFull 检查 buff 是否已满，若满则输出、清空，否则继续累积；若长度不合规，则继续两步搜索，其操作如上图中两步搜索部分所展示。

```

if (nowLen == nSet) {
    if (!tSet.empty()) { ... }
    nListNum++;
    buffNum++;
    if (isFull()) { //full
        buffNum = 0;
        writeResult(2); //output buff
        for (auto& i : buffList) i->word = ""; //init buff
        WordNode* TnNode = nowList;
        for (int i = 0; i < nSet; i++) {
            buffList[i]->word = TnNode->word;
            TnNode = TnNode->next;
        }
    }
    else { //not full
        WordNode* TnNode = nowList;
        for (int i = buffNum*nSet - nSet; i < buffNum*nSet; i++) { //insert into right pos
            buffList[i]->word = TnNode->word;
            TnNode = TnNode->next;
        }
    }
}
}

```


其余代码请参考附带的源文件与头文件。

测试用例：

一、命令行参数报错测试：

设计思路：应对各种非法的命令行参数输入，程序应当给予一定的反馈和合理引导，因此设计如下数个测试样例，测试程序的报错性能。（图形化后更直观）

①Worldlist.exe -w -t saf4d you_can_dddddddd (-t 后含数字)

```
PS F:\HLY2B3B\Lab1\Debug> .\Worldlist.exe -w -t saf4d you_can_input_words_by_absolute_path_of_word_list_or_only_by_keyboard
F:\HLY2B3B\Lab1\Debug\Worldlist.exe
-w
-t
saf4d
you_can_input_words_by_absolute_path_of_word_list_or_only_by_keyboard
-w
wSet!
-t
非法的命令行参数 (-h, -t 后跟的字符串中只允许包含字母)
```

②Worldlist.exe -w -n f you_can_dddddddd (-n 后不是数字)

```
PS F:\HLY2B3B\Lab1\Debug> .\Worldlist.exe -w -n f you_can_in
F:\HLY2B3B\Lab1\Debug\Worldlist.exe
-w
-n
f
you_can_input_words_by_absolute
-w
wSet!
-n
dsfsdfs 1-n参数后面需要一个数字
```

③Worldlist.exe -w -n f 1 examp.txt (-n 后数字为1)

```
D:\> Worldlist.exe -w -n 1 example.txt
Worldlist.exe
-w
-n
1
example.txt
-w
wSet!
-n
dsfsdfs 49-n后的参数至少为2!
```

更多该类测试样例请参照 GUI 界面中的报错。

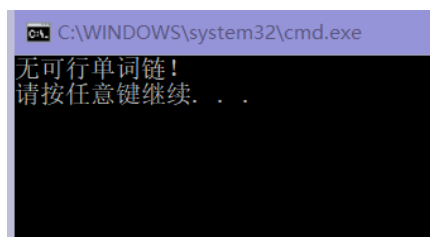
二、基础功能测试：

提供十个测试文本 test_1.txt, …… , test_10.txt，设计思路如下：（1~6 默认-w）

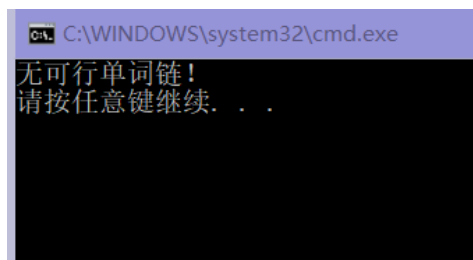
①空文本：测试报错。

```
C:\> C:\WINDOWS\system32\cmd.exe
文件为空或不存在!!!!
请按任意键继续. . .
```

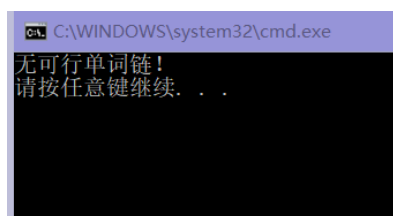
②只包含一个单词：测试是否满足“最小词链中词数>2”。



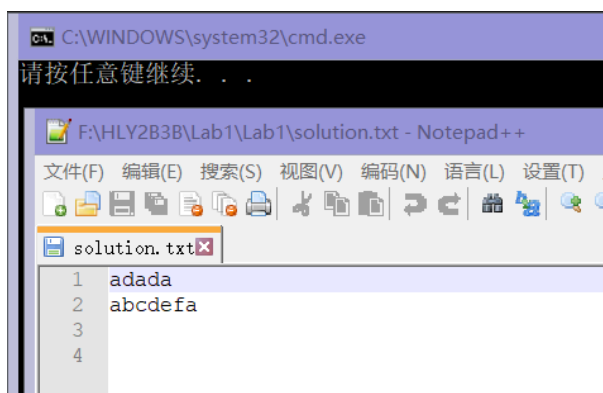
③包含数个单字母：测试识别单词的能力（单字母不算单词）。



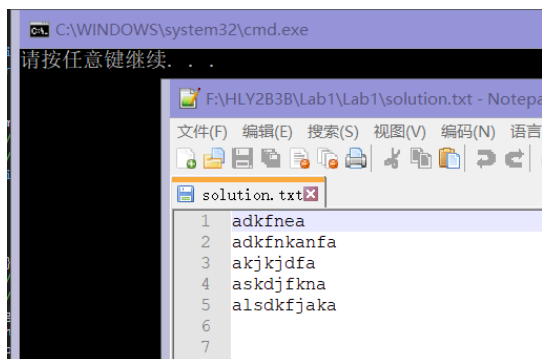
④包含数个单词，但均不可成链的文本：测试成链条件判定。



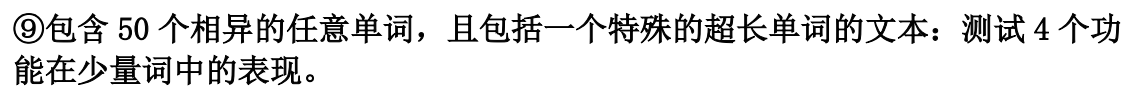
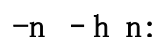
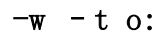
⑤包含数个相同单词的文本：测试去重能力。



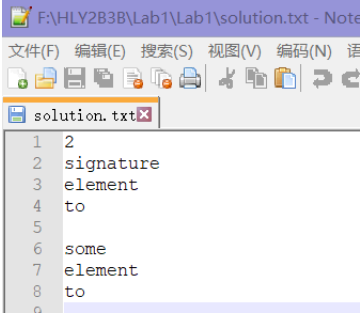
⑥包含数个同字母开头、同字母结尾单词，的文本：测试一条单词串中搜索的正确性。



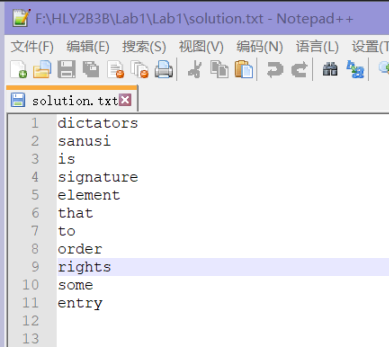
-W:



`-n -h s -t o:`

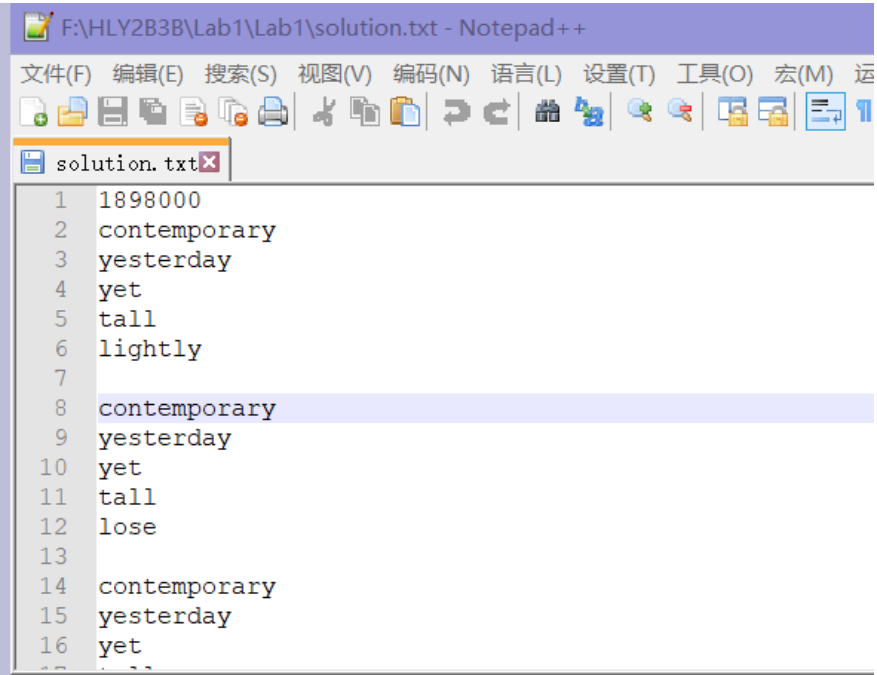


`-w -h d:`



⑩包含 700 个相异的任意单词，且包括一个特殊的超长单词的文本：测试 4 个功能在少量词中的表现。

`-n 5 -h c:`



Normal text length : 17,762,947 lines : 2,277, Ln : 8 Col : 5 Sel : 0 | 0