

# SoftwareEngineeringLab1

---

## 1. 项目概述

---

本次实验目的为计算最长英语单词链，具体要求：

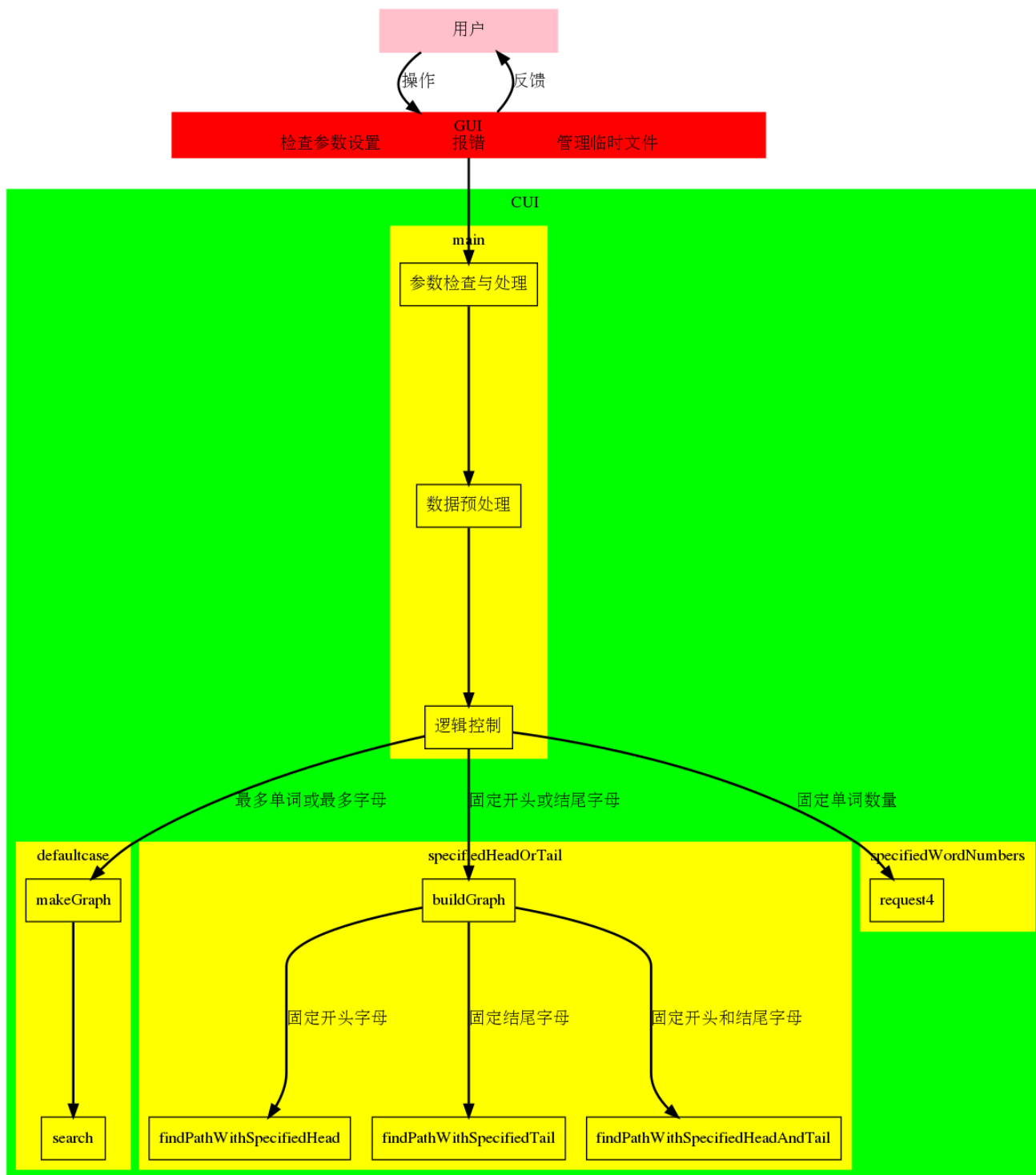
对于包含有 N 个不同的英语单词的文本，要求程序可以快速找出最长的能首尾相连的英语单词链，注意每个单词最多使用一 次，且单词大小写不敏感：**单词**：被非英文字符间隔的连续英文字符序列

**单词链**：由至少2个单词组成，前一单词的尾字母为后一单词的首字母，且不存在重复单词

针对需求及特殊情况，本次实现主要功能如下：

- 从文本/GUI中提取单词
- 对单词进行预处理，譬如：重复单词自动删除，单词统一转化为小写
- 主函数接口
- 4种基本需求：
  - 基本需求1：计算最多单词数量的单词链
  - 基本需求2：计算字母最多的单词链
  - 基本需求3：指定单词链开头或结尾字母
  - 基本需求4：指定单词链的单词个数
- 异常情况处理

工程框架为：



## 2. 编译环境

- windows 下 g++ 编译 得可执行文件
- ubuntu16.04 下通过 makefile 编译

## 3. GUI使用说明

## 开发环境

操作系统	Ubuntu 18.10
编译环境	g++ (Ubuntu 8.2.0-7ubuntu1) 8.2.0
图形库	Qt 5.12.0 GCC64bit
设计工具	Qt Creator 4.8.0

- 依赖库说明：将上述命令程序作为依赖库添加在图形化程序目录下，重命名为 `wordlistCUI`

## 界面说明及控件对应关系



- 输入区： `QPlainTextEdit InputText`
- 输出区： `QTextBrowser outputText`
- 最多单词数量： `QRadioButton mostWordCheck`
- 最多字母数量： `QRadioButton mostCharCheck`
- 固定开头： `QCheckBox fixHeadCheck`
- 固定开头输入： `QLineEdit fixHead`
- 固定结尾： `QCheckBox fixTailCheck`
- 固定结尾输入： `QLineEdit fixTail`
- 固定单词数量： `QCheckBox fixNumCheck`
- 固定单词数量选择： `QSpinBox fixNumber`
- 搜索单词链： `QPushButton convertButton`
- 从文件加载输入： `QPushButton loadFromFile`

- 将结果导出到文件: `QPushButton saveOutputFile`

## 槽函数处理

**注：（这里使用了Qt自动关联，`on_xxx_clicked()` 槽函数自动关联到xxx部件信号）**

- 处理搜索命令信号: `void MainWindow::on_convertButton_clicked()`

创建 `QProcess` 实例准备调用命令行程序 --->

将输入区文本保存为临时文件 `tempInputFile.txt` --->

检查各参数对应的部件信号，处理不合法信号组合并给出提示，根据相应的信号设置命令行参数--->

调用命令行程序--->

将结果从文件读出并打印在输出文本区

- 处理从文件加载输入信号: `void MainWindow::on_loadFromFile_clicked()`

`QFileDialog::getOpenFileName()` 函数获取选择的文件名--->

读取相应的文件并将内容打印在输入区

- 处理将结果输出到文件信号: `void MainWindow::on_saveOutputFile_clicked()`

`QFileDialog::getSaveFileName` 函数获取选择的文件名--->

新建输出文件将输出区文本写到输出文件

- 处理命令行返回的找不到单词链的信息: `oid MainWindow::Result(int a)`

将信号连接到标准输出-->

CUI退出值为1时提供提示信息

## GUI工作界面





## 4. 主函数设计

- 功能控制开关说明：

```
bool mostChar = false;           //最多字母模式开关
bool mostWord = false;           //最多单词模式开关
bool fixedWordNum = false;       //确定单词数量模式开关
bool fixedHead = false;          //确定开头字母模式开关
bool fixedTail = false;          //确定结尾字母模式开关
```

- 主要变量说明：

```
int wordNum;                      //确定单词数量模式对应的单词数量
std::string inputFile;            //数据文件名
char head,tail;                  //确定的开头字母和结尾字母
std::string crudeString;         //从输入文件读取的原始内容
std::vector<std::string> crudeData = sHoT::preprocessingData(crudeString);
//通过数据预处理得到的字符串向量，作为各功能部件的数据接口输入
```

## 5. 算法原理

### 需求1：单词数量最多的单词链（defauleCase.h）

- 数据结构：有向图，采用邻接链表形式存储。以单词为边，以字母为节点。

实现如下：

```
node *graph[26];
// 有向图的邻接链表表示
std::vector<std::string> currentList;
// 存储当前的单词链
std::vector<std::string> longestList;
// 存储至今最长的单词链
int currentweight,longestweight;
// 存储当前单词链和最长单词链的总权重
```

- 节点结构：

```
struct node{
    int weight;
    bool used;
    std::string *data;
    struct node *next;
    char lastChar;
};
```

- 成员解释：

- 对于边节点：

- `weight`：恒为1（因为不考虑单词长度）

- `used`：标记边是否被使用（用于DFS防止重复搜索）

- `data`：指向一个字符串，存放该边对应的单词

- `next`：指向链表中的下一个节点或为NULL

- `lastChar`：该单词的最后一个字母，用于指示该有向边指向的节点。

- 对于头结点：

- `next` 指向链表中的下一个节点或为NULL；其余字段无意义。

- **算法原理**：DFS（Deep First Search）

DFS可以得到从一个图节点（字母）出发导出的所有单词链。由于没有指定开头的字母，应当对所有的节点进行深度优先搜索，输出最长的单词链。在具体使用中，可以使用如下的代码：

```
def::makeGraph(s,0);
def::search();
```

其中s是一个字符串向量 `std::vector<std::string>`，而且每一个单词都应当全部由小写字母组成。最后的结果会输出到与可执行文件同目录的 `solution.txt` 文件中。

- **算法实现**

```
void DFS(node *curr){
    // 深度优先搜索，curr是一个节点的指针
    node *temp=curr->next;
    //std::cout<<*(temp->data)<<curr->lastChar;
    while(temp!=NULL){
        if (!temp->used){
            temp->used=true;
            currentList.push_back(*(temp->data));
            currentweight+=temp->weight;
            // 标记使用的边，并在当前单词链中追加单词并增加总权重
            DFS(graph[temp->lastChar-'a']);
            // 搜索当前边指向的节点
            temp->used=false;
            currentList.pop_back();
            currentweight-=temp->weight;
            // 回溯，去除标记并在当前单词链中删除单词并减少总权重
        }
        temp=temp->next;
    }
    if (currentweight>longestweight){
        //printf("\r\nbegin copy");
        longestList.assign(currentList.begin(),currentList.end());
        longestweight=currentweight;
        //printf("\r\nend");
    }
}
```



```
// 如果当前单词链长于最长的单词链，就进行把当前单词链作为最长单词链
}
```

## 需求2：字母最多的单词链 (defauleCase.h)

- 数据结构

基本同需求一。区别在于是节点权重 `weight` 设置为单词长度而不是1

- 算法原理

将节点权重 `weight` 设置为单词长度后，算法同需求1 在具体使用中，可以使用如下的代码：

```
def::makeGraph(s,1);
def::search();
```

- 算法实现

同需求一。

## 需求3：指定开头或结尾的字母的单词链 (specifiedHeadOrTail.h)

- 数据结构：有向图，使用邻接链表表示

数据结构实现及解释如下：

```
// graph datastructure
struct edge{
    std::string* data; //边对应的单词
    int weight;        //边的权重，根据模式的不同分别设置为1或单词的长度
    char destChar;     //有向边的目的节点
    char originChar;   //有向边的源节点
    bool used;         //使用标志位，避免重复搜索
    struct edge* next;
};
typedef struct edge edge;
struct node{
    char ch;           //节点对应的字母
    edge* adjEdges;    //邻接链表
};
typedef struct node node;

// global variable
std::vector<std::string> searchingPath; //当前搜索路径（单词链）
int searchingWeight;                  //当前搜索路径的权重（长度）
std::vector<std::string> resultPath;  //当前搜索过的最长路径（单词链）
int resultWeight;                     //当前搜索过的最长路径的权重（长
```

- 算法原理：通过分析需求，实现分为3部分：

- 固定开头字母搜索：DFS
- 固定结尾字母搜索：DFS。其中该方法固定开头字母搜索没有太大区别，因为对应图结构已按照要求调整，只需要将最后结果反转即可
- 同时固定开头和结尾字母搜索：DFS。区别在于不是在每条路径末尾与最长路径比较，而是当发现到达目标节点时更新最长路径

- **算法实现**

由于三部分核心搜索相同，因此这里仅展示 固定开头字母搜索 的结果：

```
//固定开头字母模式
void findPathwithSpecifiedHead(node* graph,char head){
    /* 参数说明：
        1、node* graph 要搜索的图
        2、char head 当前搜索节点
        输出：无输出，将最长路径保存在 resultPath 中
    */
    edge* fromEdge = graph[head - 'a'].adjEdges;
    while(fromEdge != NULL){
        if(fromEdge->used == false){

            fromEdge->used = true;
            searchingPath.push_back(*(fromEdge->data));
            searchingWeight += fromEdge->weight;
            //DFS递归搜索
            findPathwithSpecifiedHead(graph,fromEdge->destChar);

            searchingWeight -= fromEdge->weight;
            fromEdge->used = false;
            searchingPath.pop_back();
        }
        fromEdge = fromEdge->next;
    }
    //每条路径搜索结束后与当前搜索到的最长路径比较，如果更长则更新最长路径及最长路径权重
    if(searchingWeight > resultWeight){
        resultWeight = searchingWeight;
        resultPath.assign(searchingPath.begin(),searchingPath.end());
        //resultPath.reserve(resultPath.size());
    }
}
```

## 需求4：指定单词链个数的单词链 (specifiedWordNumbers.h)

该需求为性能主要考察部分

- **数据结构**：有向图，采用邻接表实现。

图的节点为单词（字符串）。若单词A和单词B能构成单词链，则AB之间存在一条边。

数据结构实现： `unordered_map<string, vector<string>>`

- **算法原理**：动态规划

记单词链中单词个数为 $n$ , 单词列表长度为 $d$

其最优子结构表述为:

$n = k$ 时, 所有长度满足要求的单词链存放于 $a$ 中

$n = k+1$ 时, 对 $a$ 中每个单词链, 在其末尾添加1个合法单词。将所有新的单词链存入 $a$ 中

- 算法优化:

- 采用`unordered_map` (其原理为hash表) 存储邻接表, 节省了查找邻接边的时间
- 采用2个`vector`集合交替存储上一次和当前次的结果, 相比于维护动态规划表, 极大降低了空间复杂度

- 特殊情况处理:

若输入 $n < 2$ , 命令行输出错误信息: `error: as defined, word list must have a length at least 2`

若输入

- 算法实现:

```
//b已经初始化为单词列表
for(int i=1; i<n; i++){
    a=b;
    b.clear();
    for(vector<string> v : a){//对于长度为i-1的所有单词链v进行处理
        string last = v.back();
        for(string next:adj[last])//尝试下一个单词next
            if(!used(next, v)){//检查v是否已经使用过next
                v.push_back(next);
                b.insert(v);
                v.pop_back();//向b中插入v+next,并还原v (为了尝试下一条邻边)
            }
        }
    }
}
```

## 5. 测试

### 测试样例设计

testcase	设计目的
test1.txt	基本需求1， 2。题目中需求1给出的测试样例。可测试大小写转化和基本功能
test2.txt	鲁棒性需求。提供了具有干扰字符的文件。测试程序能否分割输入字符得到单词
test3.txt	基本需求3。测试程序是否处理了不同搜索起点的情况，该样例从不同的起点开始得到的搜索结果长度不同
test4.txt	基本需求4。大数据集警告：n=64的合法单词链数目为5016480，运行时间约为3分钟。提供了一幅64个顶点的有向完全图，可以测试程序的搜索性能
test5.txt	基本需求4。提供了一幅只有两个节点的图，可以测试程序的搜索是否完全，-n参数输出的路径条数是否正确
test6.txt	基本需求3。用于测试-h/-t分别和-w，-c共同使用时的输出结果是否相同
test7.txt	基本需求4。提供了一个环。测试动态规划能否正确找出所有解。
test8.txt	鲁棒性需求。提供了具有重复单词的文件。测试程序如何处理错误输入。
test9.txt	鲁棒性需求。提供了空文件。此为side case，测试程序的鲁棒性。
test10.txt	基本需求4。题目中需求4给出的测试样例。

## 测试结果

### 鲁棒性需求

#### 1. 重复单词：自动去重

示例：test8

```
aaa
aaa
aaa
aaa
abb
bacon
```

```
.\wordlist_win10.exe -w .\test\test8.txt
```

输出结果：

```
aaa
abb
bacon
```

## 2. 空文件：输出文件为空

示例：test9(空文件)

```
.\Wordlist_win10.exe -w .\test\test9.txt
```

输出结果：

一个空的solution.txt

## 3. 干扰字符文件：自动提取符合要求的字符串

示例：test2

```
ab bieiojiosjdfc89biufd.  
cd76677667d8---  
dcfaswc/add,daa
```

```
.\Wordlist_win10.exe -w .\test\test2.txt
```

输出结果：

```
add  
dcfaswc  
cd  
daa  
ab  
biufd  
d
```

## 4. 大小写转化：自动将输入中的大写转化为小写

示例：test1

```
Algebra  
Apple  
Zoo  
Elephant  
Under  
Fox  
Dog  
Moon  
Leaf  
Trick  
Pseudopseudohypoparathyroidism
```

```
.\Wordlist_win10.exe -w .\test\test2.txt
```

输出结果：

```
algebra  
apple  
elephant  
trick
```

## 5. 未规定的参数组合：报错

```
wrong argument usage.  
Usage: wordlist [arguments] <filename>  
Mandatory arguments:  
  -c: Output one word list with the most characters.  
  -w: Output one word list with the most words.  
  
Optional arguments:  
  -h <char>: Determine the head character of the word list.  
  -t <char>: Determine the tail character of the word list.  
  (Note: -h and -t can be used at the same time)  
  -n <num>: Output all word lists containing <num> words.  
  (Note: -n must be used with -w)
```

## 6. 输入非法参数

```
.\wordlist_win10.exe -h 2 -w .\test\test1.txt
```

报错：

```
A character is needed after -h.  
Usage: wordlist [arguments] <filename>  
.....
```

## 基本需求1：单词数量最多的单词链

输入：test1

```
Algebra  
Apple  
Zoo  
Elephant  
Under  
Fox  
Dog  
Moon  
Leaf  
Trick  
Pseudopseudohypoparathyroidism
```

```
.\wordlist.exe -w .\test\test1.txt
```

输出结果：与正确结果相同

```
algebra
apple
elephant
trick
```

## 基本需求2：指定开头或结尾的字母的单词链

输入：test1

同基本需求1,略去

```
.\wordlist_win10.exe -c .\test\test1.txt
```

输出结果：

```
pseudopseudohypoparathyroidism
moon
```

## 基本需求3：指定开头或结尾的字母的单词链

示例测试：输入test1

以下测试结果均正确

固定开头为e：

```
.\wordlist_win10.exe -h e -w .\test\test1.txt
```

```
elephant
trick
```

固定结尾为t：

```
.\wordlist_win10.exe -t t -w .\test\test1.txt
```

```
algebra
apple
elephant
```

固定开头为a和结尾为t：

```
.\wordlist_win10.exe -h a -t t -w .\test\test1.txt
```

```
algebra
apple
elephant
```

### 深入测试:

1. 考虑到-h/-t可以和-c组合的隐含要求

test6:

```
abcdefghhijklmn
ab bc cd dn
```

因此可以区分单词数最多和字母树最多

若输入-h/-t -w, 则寻找固定开头/结尾的单词数最多的单词链

```
.\wordlist_win10.exe -h a -t n -w .\test\test6.txt
```

```
ab
bc
cd
dn
```

若输入-h/-t c, 则寻找固定开头/结尾的字母数最多的单词链

```
.\wordlist_win10.exe -h a -t n -c .\test\test6.txt
```

```
abcdefghh
ijklmn
```

2. 证明指定的开头字母不同, 输出不同。

分别输出字母a开头和w开头的字符串

输入: test3.txt

```
ab bc cd de
ef fg gh hi
ij jk kl lm
mn no op pq
qr rs st tu
uv vw wx xy
yz
```

```
.\wordlist_win10.exe -h a -w .\test\test3.txt
```

输出结果:



```
ab  
bc  
cd  
de  
ef  
fg  
gh  
hi  
ij  
jk  
kl  
lm  
mn  
no  
op  
pq  
qr  
rs  
st  
tu  
uv  
vw  
wx  
xy  
yz
```

```
.\wordlist_win10.exe -h w -w .\test\test3.txt
```

输出结果：

```
wx  
xy  
yz
```

## 基本需求4：指定单词链个数的单词链

示例测试：

test10

输入：

```
word  
digital  
list
```

```
.\wordlist.exe -n 2 -w .\test\test10.txt
```

输出结果：

```
2
digital
list

word
digital
```

由于基本需求4为主要性能考察部分，在此设计以下测试考察其性能：

- 大数据集：测试算法输出所有结果的时间。5016480个结果约耗时3分钟

test4: 7个节点的有向完全图

```
ab ac ad ae af ag ah
ba bc bd be bf bg bh
ca cb cd ce cf cg ch
da db dc de df dg dh
ea eb ec ed ef eg eh
fa fb fc fd fe fg fh
ga gb gc gd ge gf gh
ha hb hc hd he hf hg
```

```
.\wordlist.exe -n 7 -w .\test\test4.txt
```

输出结果：

```
5016480

ab
ba
ac
ca
ad
da
ae
.....
```

- 输出完整性：测试算法是否能输出全部符合要求的单词链

test5

输入：

```
aqb
aqqqb
aqwb
aqqqbb
aqwerrttb
```

```
bwwea
bssxxaa
baaaaa
baaa
bxsza
ba
```

```
.\wordlist_win10.exe -n 2 -w .\test\test5.txt
```

```
60
aqb
ba

aqb
baaa

aqb
baaaaa

.....
```

- 输出完整性/正确性：所有单词构成一个环，测试算法是否能输出所有单词链

test7

```
.\wordlist_win10.exe -n 25 -w .\test\test7.txt
```

输出结果：

```
7
abc
cde
efg
ghi
ijk
klm
mna

cde
efg
ghi
ijk
```

klm  
mna  
abc

efg  
ghi  
ijk  
klm  
mna  
abc  
cde

ghi  
ijk  
klm  
mna  
abc  
cde  
efg

ijk  
klm  
mna  
abc  
cde  
efg  
ghi

klm  
mna  
abc  
cde  
efg  
ghi  
ijk

mna  
abc  
cde  
efg  
ghi  
ijk  
klm