

SoftwareEngineeringLab1

1. 项目概述

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，去除非法字符
- 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
- 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
- 特殊需求实现

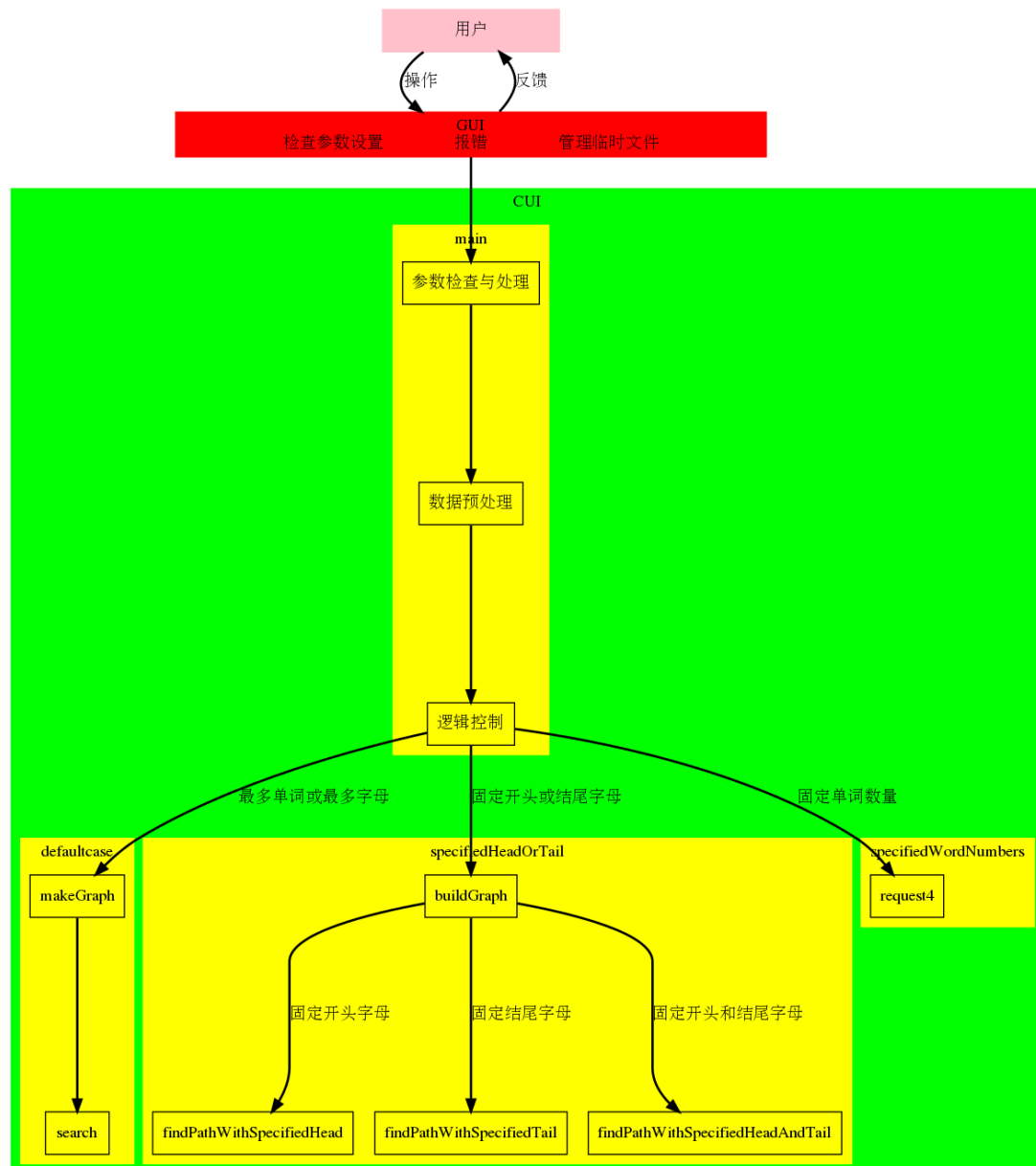
某些可能的参数组合并未直接要求，但我们也将其实现了：

- -h/-t -c: 指定开头或结尾字母的字母数最多的单词链
 - -h/-t -n: 指定开头或结尾字母，单词链长度的所有单词链
 - 输入文件为空
 - 输入文件过大时限制时间搜索
- 异常情况报错
 - 输入异常参数
 - 找不到输入文件位置
- 使用提示信息 Usage:

Usage 信息函数：

```
void usage(){
    std::string usage="Usage: Wordlist [arguments] <filename>\r\n\
Mandatory arguments:\r\n\
    -c: Output one word list with the most characters. \r\n\
    -w: Output one word list with the most words.\r\n\r\n\
Optional arguments:\r\n\
    -h <char>: Determine the head character of the word list.\r\n\
    -t <char>: Determine the tail character of the word list.\r\n\
    (Note: -h and -t can be used at the same time)\r\n\
    -n <num>: Output all word lists containing <num> words.\r\n\
    (Note: -n must be used with -w)\r\n";
    cout<<usage;
    exit(0);
}
```

项目实施框架为:



2. 编译环境

- 操作系统: Ubuntu 18.10
- 编译器版本: g++ (Ubuntu 8.2.0-7ubuntu1) 8.2.0
- C++版本: C++11

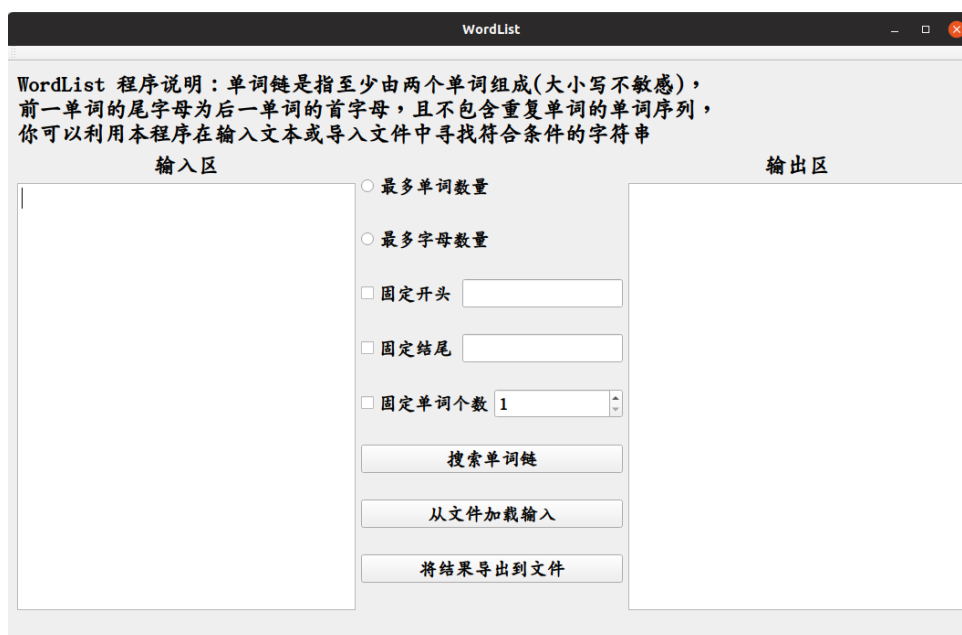
- 具体编译/使用方法请阅读 `readme.md`

3. GUI 版本使用说明

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	将命令行程序作为依赖库添加在 GUI 目录下重命名为 <code>WordlistCUI</code>

界面说明



GUI 控件与命令行版本控件的对应关系：

- 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--->

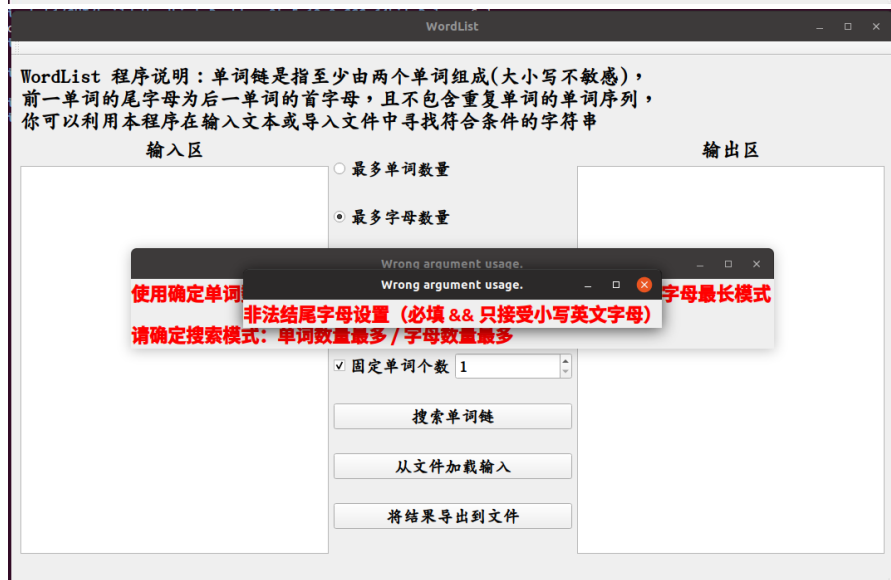
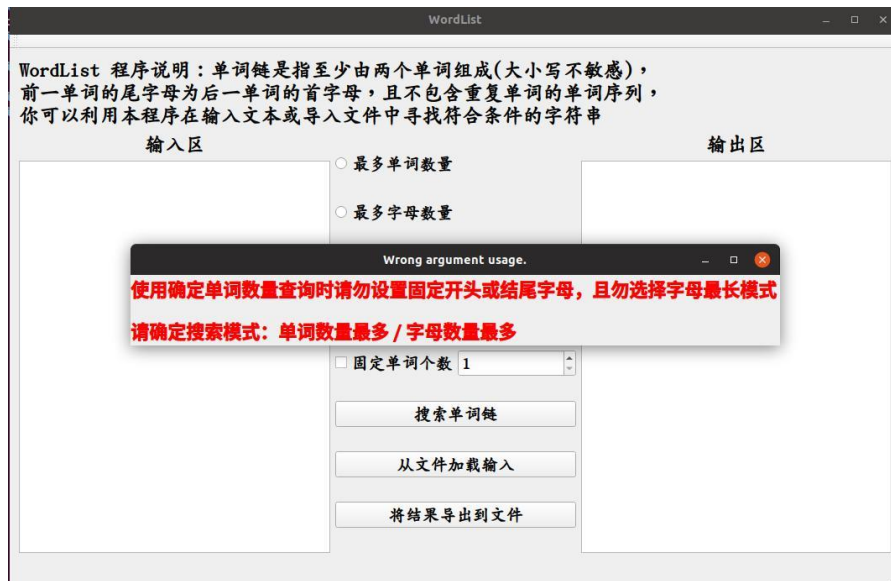
新建输出文件将输出区文本写到输出文件

- 处理命令行返回的找不到单词链的信息: `void 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. 命令行版本主函数设计

- 主函数功能
 - 读取文件
 - 数据预处理：包括去除不合法字符，大小写转换，去除重复单词。最终提取出的单词存放于 `vector<string> crudeData` 中，提供给各个函数。
 - 功能分派（**dispatch**）：将不同需求分配给不同函数执行
 - 异常报错：包括异常输入参数，异常文件位置等
 - 输出结果到文件：输出到当前目录下的 `solution.txt`。若不存在则自动创建，已存在则覆盖原文件。
- 主函数控件说明

为了使得接口更为一致和整洁，工程使用以下功能控制开关和对应变量来实现功能的分派：

功能控制开关：

```
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

本部分介绍四个需求的实现原理。其中每个需求的文档分为以下部分

- **数据结构：**介绍使用的数据结构和代码实现
- **算法原理：**介绍算法的思想和主要过程
- **核心实现：**介绍算法核心部分的代码实现

其中在需求 4：指定单词个数的单词链中，还加入了算法优化部分的说明。

在本部分的最后，加入了搜索超时处理机制的说明。

需求 1: 单词数量最多的单词链（位于 defauleCase.h）

- **数据结构：**

有向图，采用邻接链表形式存储。以字母为节点，以单词为边。

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;  
};
```

节点结构成员的解释：

1)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。

2)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。

- 核心实现

```
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（位于 defauleCase.h）

- **数据结构**
数据结构基本同需求一。
区别在于：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();

```

- **核心实现**

同需求一。

需求 3: 指定开头/结尾字母的单词链（位于 specifiedHeadOrTail.h）

- **数据结构：**

有向图，使用邻接链表表示，包括字母节点和边节点。

数据结构实现及解释如下

边节点：

```
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;
};
```

点节点：

```
struct node{
    char ch;           //节点对应的字母
    edge* adjEdges;    //邻接链表
};
typedef struct node node;
```

全局变量：

```
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：

本需求有 3 部分要求：

- 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。在我们的实现中，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：

```

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());
    }
}

```

需求 4:指定单词个数的单词链（位于 specifiedWordNumbers.h）

- 数据结构：

有向图，采用邻接表实现。

图的节点为单词（字符串）。边的定义为：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<string>` 的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
- 对于过大的输入，可以让用户选择随机方式：即采取以一定概率随机裁剪中间结果，从而降低搜索空间。其代价就是不一定能找到所有解，或不一定能找到解。

目前测试中，对于大数据集 `test_4`（经过设计的 64 个节点的有向完全图），可以 3 分钟完整输出约 500 万个符合要求的 $n=7$ 时的单词链。具体请参见报告的测试部分。

- 异常情况处理：

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 $n > \text{wordlist 长度}$ ，不报错，直接输出 0。

- 算法实现：

```
//b 的含义：初始化为单词列表
//a 与 b 是交替存储  $n=k-1$  和  $n=k$  下所有符合要求的单词链的集合
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])//尝试所有可连接单词 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
            }
    }
}
```

```
}
```

超时处理机制（针对需求 1-3：搜索最长单词链的情况实现）

由于算法的时间复杂度是指数级的，因此在对于复杂问题进行处理时可能会花费过长时间，因此我们设置了超时机制，如果搜索时间超过 15 秒，就停止搜索，输出当前得到的最长的单词链。

```
void signalHandler(int m){
    //超时处理函数：输出 DFS 当前查找到的最长单词链
    std::cout<<"Time limits. The word list output may not be the longest one."<<std::endl;
    std::ofstream mycout("solution.txt");
    for (int i=0;i<longestList.size();i++){
        mycout<<longestList[i]<<std::endl;
    }
    exit(0);
}

void initTimer(int sec){
    //设置定时器，搜索 sec 秒后停止搜索
    struct itimerval value;
    value.it_value.tv_sec=sec;
    value.it_value.tv_usec=0;
    value.it_interval=value.it_value;
    setitimer(ITIMER_REAL,&value,NULL);
}

void search(){
    signal(SIGALRM, signalHandler);
    initTimer(15);
    ...
}
```

6. 测试

测试样例设计

本部分介绍了每个测试样例的主要内容和测试功能。列出表格如下：

testcase	设计目的
test_1.txt	基本需求 1，2。该样例为题目中需求 1 给出的测试样例。可测试大小写转化和基本功能
test_2.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
test_3.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
test_4.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
test_5.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
test_6.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
test_7.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。
test_8.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。
test_9.txt	鲁棒性需求。提供了空文件。此为 side case，测试程序的鲁棒性。
test_10.txt	基本需求 4。题目中需求 4 给出的测试样例。

测试结果

本部分列出了每个需求的完成状况。结论为所有需求均完整实现。

鲁棒性需求测试结果

1. 重复单词：自动去重

示例：test_8

```
aaa
aaa
aaa
aaa
```

abb

bacon

```
.\Wordlist_win10.exe -w .\test\test_8.txt
```

输出结果：

aaa

abb

bacon

2. 空文件：输出文件为空

示例：test_9(空文件)

```
.\Wordlist_win10.exe -w .\test\test_9.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：test_2

ab bieiojiosjdfc89biufd.

cd76677667d8---

dcfaswc/add,daa

```
.\Wordlist_win10.exe -w .\test\test_2.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：test_1

```
Algebra
```

```
Apple
```

```
Zoo
```

```
Elephant
```

```
Under
```

```
Fox
```

```
Dog
```

```
Moon
```

```
Leaf
```

```
Trick
```

```
Pseudopseudohypoparathyroidism
```

```
.\Wordlist_win10.exe -w .\test\test_1.txt
```

输出结果：

```
algebra
```

```
apple
```

```
elephant
```

```
trick
```

5. 未规定的参数组合：报错

```
Wrong argument usage.
```


6. 输入非法参数：报错

例如：

```
.\Wordlist_win10.exe -h 2 -w .\test\test_1.txt
```

报错：

```
A character is needed after -h.
```

基本需求 1 测试结果(单词数量最多的单词链)

输入：test_1

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

```
.\Wordlist.exe -w .\test\test_1.txt
```

输出结果：与正确结果相同

```
algebra
apple
elephant
```

trick

基本需求 2 测试结果(字母数最多的单词链)

输入: **test_1**

Algebra

Apple

Zoo

Elephant

Under

Fox

Dog

Moon

Leaf

Trick

Pseudopseudohypoparathyroidism

```
.\Wordlist_win10.exe -c .\test\test_1.txt
```

输出结果:

pseudopseudohypoparathyroidism

moon

基本需求 3 测试结果(指定开头或结尾的字母的单词链)

输入: **test_1**

固定开头为 **e**

```
.\Wordlist_win10.exe -h e -w .\test\test_1.txt
```

elephant

trick

固定结尾为 t

```
.\Wordlist_win10.exe -t t -w .\test\test_1.txt
```

algebra

apple

elephant

固定开头为 a 和结尾为 t

```
.\Wordlist_win10.exe -h a -t t -w .\test\test_1.txt
```

algebra

apple

elephant

深入测试：特殊需求

-h/-t 与 -c 组合的情况

输入：test_6:

abcdefghhijklmn

ab bc cd dn

该测试中，规定单词数最多和字母数最多的输出不同

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\test_6.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\test_6.txt
```

abcdefgh

hijklmn

正确性测试：证明指定的开头字母不同，则输出不同

分别输出字母 a 开头和 w 开头的字符串

输入：test_3.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

若指定以字母 a 开头：

```
.\Wordlist_win10.exe -h a -w .\test\test_3.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

若指定以字母 **w** 开头：

```
.\Wordlist_win10.exe -h w -w .\test\test_3.txt
```

输出结果：

wx

xy

基本需求 4 测试结果(指定单词链个数的单词链)

由于该部分为性能考察的主要部分，因此在基本功能测试以外，添加性能测试，采用可输出 500 万个解的数据集进行测试。

基本功能测试

输入：**test_10**

```
word  
digital  
list
```

```
.\Wordlist.exe -n 2 -w .\test\test_10.txt
```

输出结果：

```
2  
digital  
list  
  
word  
digital
```

性能测试

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：

输入：**test_4(大数据集)**：

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\test_4.txt`

测试算法输出所有结果的时间。总共 5016480 个结果，约耗时 3 分钟

输出结果：

5016480

ab

ba

ac

ca

ad

da

ae

.....

完整性/正确性测试

测试输出的完整性/正确性：测试算法是否能正确输出所有单词链

输入：**test_7**

所有单词构成一个环

abc
cde
efg
ghi
ijk
klm
mna

```
.\Wordlist_win10.exe -n 25 -w .\test\test_7.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

可见其正确完整输出所有结果