# 软件工程实验报告

## 实验环境

### 基础算法实现：

\1. IDE： Visual Studio 2017. VSCode

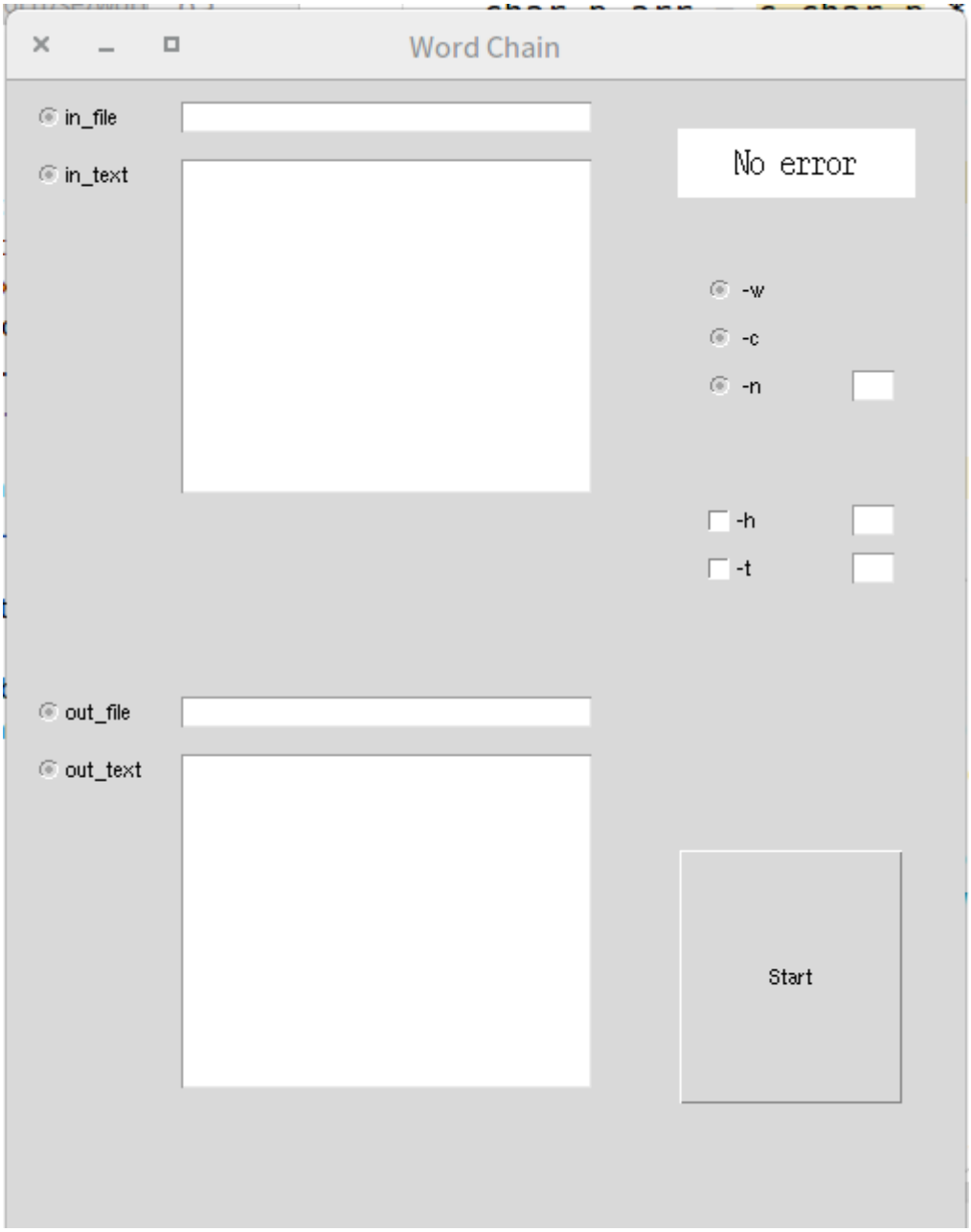
\2. 操作系统：WIndows 10. Ubuntu 16.04

\3. 运行环境:Linux Ubuntu16.04

## GUI与编译

### GUI：

**GUI界面展示:**




- GUI功能:
  - 输入:可选择文件输入,直接输入两种方式.文件输入直接选择in_file,并在in_file栏里输入待输入文件的绝对路径或者相对路径;直接输入则选择in_text,并在in_text栏中直接输入内容,如果没有选择输入方式,则默认以in_text方式输入.
  - 输出:可选择文件输出,直接输出两种方式.文件输出直接选择out_file,并在out_file栏里输入待输出文件的绝对路径或者相对路径;直接输出则选择out_text,并在out_text栏中直接输出内容,如果没有选择输出方式,则错误栏会给予报错.
  - 参数选择:'-w','-c','-n'这三种参数互斥,只可以选择一种,若都不选,则默认'-w','-h','-t'这两种参数可以任意选择,'-n','-h','-t'这三种参数若选择,则需要在后面的栏中输入相应的信息,'-n'后输入数字,'-h','-t'后输入[a-zA-z]的字母,若输入不符合要求,则会在错误栏报错
- 错误信息
  - Lack parameter: -n, -t, -h参数选择了但未输入相应信息
  - parameter error: 参数输入错误
  - in_file illegal: 输入文件不合法

- in_file not exist: 输入文件不存在
- lack input: 选择了in_text输入,但未输入内容
- input illegal: 输入内容不合法
- where to output: 选择了out_file,但未输入路径

## 编译:

- 方法一

```
\> cd wordChain/src

\> g++ -fPIC -c WordChain.cpp -std=c++11

\> g++ -shared -o libtest.so WordChain.o

\> mv libtest.so ../GUI

\> cd ../GUI

\> pyinstaller -w -F gui.py

\> mv dist/gui .

\> ./gui
```

- 方法二（推荐）

**直接进入wordChain/目录,然后输入**

```
> shell compile.sh
```

运行gui需要在GUI/目录下

若gui运行报错则,尝试按转lib/目录下的库文件

若依然报错,可直接进入GUI/目录,输入

```
> python gui.py
```

运行

# 测试文件

放置在`\test`文件目录下，主要进行了以下测试：

- testfile0：错误文档，无有效字符
- testfile1.txt：用于测试生成的图中有环的情况，其中带有重复单词。
- testfile2.txt: 用于测试生成的图中没有字符链的情况。
- testfile3.txt：用于测试有多条重复边（从字母i到字母j有多个有效单词）的情况。
- testfile4.txt: 用于测试有回边、大量环（以字母i开始字母i结束的有效单词）的情况。
- testfile5.txt: 用于测试数据规模较大的情况。

- testfile6 ~ testfile10: 随机生成的文档，用于测试程序的普遍性，单词数目不等。

# 基础算法实现

## 数据结构：

### 本次实验的数据结构采用了图的存储方式：

- 实验中，每一个单词的主要关键信息如下：
  - 单词的首尾字母
  - 单词的长度
  - 单词的内容
- 从这些信息可以看出，我们可以采用图的方式来进行存储：
  - 每一个节点表示字母，所以图的规模可以被限制。
  - 用一条有向边表示一个单词，边的相关信息为单词的长度和内容。
- 进一步抽象后可以得出，相关信息可以抽象为用矩阵（二维数组）来存储：

```
    int alphaMatrix[26][26];
```

`alphamatrix[i][j]`的值表示：以第i个字母开始、第j个字母结束的单词数量，字母的顺序按照字母表来排列。

接下来的两个vector二维数组分别存储单词的长度和内容。

数据结构和相关算法在Prefix.cpp中的类`words`实现。

类的基本存储单元如下：

```
    vector <string> wordMatrix[26][26];
    //保存字符串的向量矩阵，[i][j]中表示从i开始j结束的所有单词组成的向量
    vector <int> wordSizeMatrix[26][26];
    //保存单词长度的向量矩阵，[i][j]中表示从i开始j结束的所有单词长度组成的二向量
    int alphaMatrix[26][26];
    int total_word_num;           //保存总单词数
```

其中`wordMatrix`和`wordSizeMatrix`中的元素按顺序一一对应，且按照从小到大的顺序排列（即最大的元素在vector的末尾）。

## 算法思想：

- 由于采用了图的存储方式，因此非常自然地想到了采用深度优先搜索（DFS）的方式来遍历生成的图，从而得到符合要求的单词链。
- 根据需求描述，可以将四种不同的用户需求转变为不同约束条件下的DFS：
  - 寻找最大单词数的单词链： 可以转化为在边权为1的图上找最长路径的DFS问题，在代码中通过`findLongerest`方法实现。
  - 寻找字母数最多的单词链： 可以转化为在边权不恒为1的图上找最长路径的DFS问题。在代码中通过`findLargest`方法实现。

- 寻找指定开头字母/结尾字母的单词链： 转化为从特定行开始、以特定列结束的DFS问题，代码中综合了其他部分实现。
- 寻找所有指定长度的单词链：转化为深度限定的DFS问题。在代码中通过 `findNumH` 实现。

其中第三种要求可以和其他三种要求混合使用，而在没有第三种条件的限制下，需要考虑alphaMatrix的每一行，因为单词链可能以任何一个字母开始，所以建立一个循环来查找所有符合条件的单词链。

- 算法部分的主要实现在WordChain.cpp中。
- 在实际情况中，由于文件中单词数可能会很大，因此在实际搜索中，在单词数很大时，我们可以限定搜索时间或搜索深度，从而降低时间开销，满足用户的基本需求。

## 算法描述：

单纯的DFS较为简单，实验的主要难点集中在：

- 如何按照需求输出
- 如何找到以特定字母结尾的单词链。

### 按照要求输出

由于本次实验的要求四中，首先要输出符合条件的个数，然后依次输出符合条件的单词链。按照常规思路，可以保存所有的单词链，在程序结束时统计并输出。但是考虑到当单词链数目过多的时候，耗费的存储空间可能非常之大，因此在实际实现过程中，采用了递归输出的方式，将所有的单词链结果保存在一个临时文件中，当最后统计出单词链的个数时，再将所有的单词链拷贝到目标文件中。

```
ofstream ofile("solution.txt");
ifstream infile("solution_temp.txt"); //solution_temp中保存有所有的符合条件的单词链
ofile << num_count<<endl;
ofile << "" <<endl;
string line;
if(infile) // 有该文件
{
    while (getline (infile, line)) // line中不包括每行的换行符
    ofile << line << endl;
}
else // 没有该文件
    cout <<"no such file" << endl;
ofile.close();
infile.close();
remove("solution_temp.txt");
```

### 找到特定字母结尾的单词链

实验中的需求三需要能够找到以特定字母结尾的单词链，这一点可以通过回溯时增加判断条件来解决：当搜索完某一个分支时，通过增加判断条件，可以得到当前栈中的单词链是否符合当前的结尾字母需求。

下面的代码展示了DFS的实现和依据需求三更改的判断条件：

```
while (ThisTail.tail != 26 && alphaMatrix[ThisHead.tail][ThisTail.tail] == 0) {
    ThisTail.tail++;
}       //判断是否回溯
```

```
if (ThisTail.tail == 26) {//表明已经走到头
    //此节点处理完毕,出栈,判断是否更新结果并返回
    if (currentLength > resultLength && ((t < 0) || (st.top().tail == t))) {
        /**当t小于0时，证明对结尾字母无需求**/
        /**要求当前的值大于resultlength时，同时满足对于结尾字母的需求*/
        resultChainL = st;//保存新的最长单词栈
        resultLength = currentLength;//保存最长单词数
    }
    if (st.size() <= 1) {//如果栈中没有一个完整的单词链
        st.pop();
    }
    else {
        ThisTail.tail = st.top().tail + 1;//准备下一个要走的边
        currentLength -= st.top().length; //当前路权减去出栈的边权
        sizeTemp = st.top().length; //取出当前栈顶的元素
        st.pop();//出栈
        wordSizeMatrix[st.top().tail][ThisTail.tail - 1].push_back(sizeTemp);
                //回填出栈的边权
        alphaMatrix[st.top().tail][ThisTail.tail - 1] += 1;
                //将此边加回去，因为之后可能还会走到这条边
    }
}
```

# 其他的关键点：

## 重复单词的检测：

由于读取的文件中可能有重复的单词，因此需要在插入单词时检查是否有重复的单词，但是如果和所有已经读取的单词均进行一次比较，则时间复杂度会提升，因此这一部分需要进行优化。

考虑到本实验中采用的数据结构，因此可以将比较的范围缩小为首尾字母相同、长度相同的string类变量，从而缩小了比较范围，提高了函数效率。

以下代码段实现了按单词长度升序插入vector并进行重复单词的检测：

```
if (wordMatrix[a][b].size() == 0){  //如果此向量为空
    total_word_num ++;
    wordMatrix[a][b].push_back(wd);
    wordSizeMatrix[a][b].push_back(wd.size());
    alphaMatrix[a][b] += 1;              //此时矩阵中对应位置加一
}
else{   //如果不为空，为了做到按序存放，需要进行调整。
    auto i = wordMatrix[a][b].begin();  //得到向量头部
    auto j = wordSizeMatrix[a][b].begin();
    bool flag = false;
    while(i != endFlag){//从头扫描到尾
    if ((*i).size() > wd.size()) //找到了合适的插入位置了
    {
        total_word_num++;
        wordMatrix[a][b].insert(i, wd);  //在对应的位置插入
        wordSizeMatrix[a][b].insert(j, wd.size());
```

```
            alphaMatrix[a][b] += 1;                //此时矩阵中对应位置加一
            flag = true;
            break;
        }
        else{
            if ((*i).size() == wd.size())    //这里进行了重复单词的检测工作。
            {   //当两个单词长度一致并且首尾字母相同，则有着出现重复单词的风险。
                string temp_str = *i;
                if (*i == wd){
                    flag = true;
                    break;
                }
            }
        }
        i++;    j++;
    }
    if (!flag)  //flag为假，表示直到循环结束仍然不满足插入要求，应当插到末尾
    {
        total_word_num++;
        wordMatrix[a][b].push_back(wd);
        wordSizeMatrix[a][b].push_back(wd.size());
        alphaMatrix[a][b] += 1;                //此时矩阵中对应位置加一
    }
```

## 命令行的处理：

- 不同的用户需求体现在不同的输入参数中，但是如果输入正确（错误处理在下一部分有所提及），那么命令行的格式相对规整：
- 命令行读入的argv[]中，最后一个元素一定是文件路径，如果当前读入的命令行参数-h, -t,则接下来的参数一定是单个字母，如果输入-n，则下一个参数一定是一个数字，-w，-c可以在任意位置单独出现。
- 所以对参数的第1~argc-2个元素（第0个元素为.exe命令，最后一个元素为），进行上述检查：

```
for (int i = 1; i < argc-1;) {
    if (argv[i][0] != '-') { //如果第一个字符不是'-' 出错
        error(0);
            }
    switch (argv[i][1])   //根据参数的不同来进行选择
```

## 错误处理：

**本次实验中，需要对程序的健壮性进行考量，所以对于一些错误输入的情况，需要进行特殊处理：**

- 文件不存在或者文件为空：

  这两种情况均只用对文件进行简单的判断即可实现。

```
ifstream check_file(filePath);
if (check_file)
{
    char c;
```

```
    check_file >> c;
    if (check_file.eof())
    {
        cout << "该文件为空！" << endl;
        file_error_flag = true;
        error(0);
    }
}
else
{
    cout << "文件不存在！"<< endl;
    file_error_flag = true;
    error(0);
}
```

- 命令行格式出错或者存在非法组合：

  由于本工程的用户需求主要体现在命令行参数上，因此需要考虑命令行输入错误的情况：

  - 命令行参数组合错误：

    由于需求中有多种参数，当多种参数同时出现时，即发生需求冲突。除了第三种需求可以与其他参数组合以外，其他所有的参数均不能相互组合，因此在出现冲突时，需要进行处理。

    本次实验中，通过给不同类型的参数设置优先级，从而解决了参数冲突的问题：如果命令行中出现了-w，则自动忽略其他的信息（-c、-n），否则以-c为准，只有当没有-c和-w命令时，执行-n命令操作。

    ```
    if (n &&(w|c)||(w&&c))  //第四个需求和前两个冲突，或者前两个需求冲突
    {
        if (w == true)
        {
            cout << "命令出现冲突，以w(最多单词数)为准"<< endl;
            n = false;
            c = false;
        }
        else
        {
            cout << "命令出现冲突，以c(最多字母数)为准"<< endl;
            n = false;
        }
    }
    ```

- 命令行参数内容错误：

  对于命令行中单独出现的不以'-'开头的参数（即不是跟在 -h/-t/-n后出现的参数），均直接报错：

```
if (argv[i][0] != '-') { //如果第一个字符不是'-' 出错
    error(0);
}
```

对于-h/-t后面没有接单个字母(a~z)的情况进行处理：

```
case 'h': //这里还需要检查长度，如果说后面跟着的单词数目超过两个，为非法表示
    if (string(argv[i + 1]).size() > 1)
    {
        cout << "单词链开始字母不符合要求，请输入单个字母！" << endl;
        error_flag = true;
        error(0);
    }
    if (!isalpha(argv[i + 1][0]))
    {
        cout << "单词链开始字母非法！" << endl;
        error_flag = true;
        error(0);
    }
    h = argv[i + 1][0];        //得到头部字符
    i += 2;
    break;
```

对于-n后面没有接数字或者数字不为正数的情况进行处理：

```
case 'n':
    n = true;
    //这里还需要检查下面的字符串中是否全部都是数字
    len = string(argv[i + 1]).size();
    for (int k = 0; k < len; k++)
    {
        if (k == 0 && argv[i + 1][k] == '+') continue; //该情况合法
        if (argv[i + 1][k] > '9' || argv[i + 1][k] < '0')
        {
            error_flag = true;
            cout << "递归深度输入有误，有无法识别的字符" << endl;
            error(0);
        }
    }
    if (error_flag == false)
    {
        num = atoi(argv[i + 1]);  //得到递归深度
        if (num <= 0) //虽然估计没人这么作死但是还是写一下吧
        {
            error_flag = true;
            cout << "单词链长度（递归深度）必须是正数！" << endl;
            error(0);
        }
    }
    i += 2;
    break;
```

- 单词数为零：

这一部分通过Words类中新增加的变量total_word_num实现，在每一次向vector矩阵中插入单词时计数。在文件读取完成之后检查total_word_num，如果最终结果为0，则直接报错：此时文件中没有合法单词。

- -n 命令执行的情况下，单词数目小于规定的递归深度：

  这一部分和检查单词数类似，将total_word_num和递归深度n进行比较，如果不符合要求则直接报错。

```
if (n && (total_word_num < num))
{
    cout<<"单词链过长，超过单词总数！"<<endl;
    error_flag = true;
    error(0);
}
```