# 实验报告

## 目录



## 编译环境

### windows

- 执行命令 `g++ *.cpp -o Wordlist.exe` （需要安装 MinGW）

### linux

- 执行 `make` 编译
- 执行 `make clean` 清理

# 运行方式

## windows

```
Wordlist {-w|-c} <input_file> [-h <first_character>] [-t <last_character>] [-n <word_num>]
```

## linux

```
./Wordlist {-w|-c} <input_file> [-h <first_character>] [-t <last_character>] [-n <word_num>]
```

# 功能说明

## 功能

找到最长（字符数或单词数）单词链 / 所有固定长度（字符数或单词数）单词链

## 选项

### -w|-c <input_file>

-w 指定单词数量模，-c 指定字符数量模式，二选一，后面加上输入文件路径。可与 -h -t -n 配合使用

### -h <first_character>

指定单词链开头字母，为可选项，可与 -n -t 配合使用

### -t <last_character>

指定单词链结尾字母，为可选项，可与 -n -h 配合使用

### -n <word_num>

当 -n 存在时，会找到所有固定单词数（-w）或固定字符数（-c）的单词链。-n 不存在则找到一个最长单词/字符链。

## 输出

- 成功：在可执行文件同目录输出文件 `solution.txt` 表示搜索结果
- 失败：在命令行窗口输出错误信息

# 数据结构

## 问题抽象

- 将每个字母作为顶点，单词作为边，构造一个图，其中点的结构是维护一个它指向的边vector和指向它的边的指针vector，边的结构除了长度和string之外维护一个star标识，用于搜索时的是否已经搜索过的指示。

- 图则维护一个节点数组，将单词作为边插入到图中，对首字母作为的节点的临边向量中添加改单词，所有的边添加完之后，调用构造逆邻接表的函数，在每个结点的逆邻接边向量中插入对应边的指针，这样既可以通过临边向量对边进行操作，又可以通过指针向量通过逆邻接边对边进行查找和操作。

## 数据结构

首先考虑到算法将会使用的图搜索策略是 DFS，需要一种能很好的**支持 DFS 的有向图数据结构**，即邻**接表。**

其次，因为可能要指定首字母和尾字母，当指定尾字母时，如果可以从尾字母节点开始逆向遍历就会更快，所以需要一种存有**逆邻接表**的数据结构。综合两个，选用**十字链表的有向图结构。**在每个点的数据结构中存放他的入边和出边，且每个边只有一个对象，不重复存储。每条边上指出头点和尾点的引用，也不重复存储。用一个图的类，建立好整个图的数据结构，方便正向和逆向遍历。

### 边数据结构

```
class Arc {
    friend class WordGraph;
private:
    std::string name;
    Vertex &adjv;
    Vertex &revAdjv;
    mutable bool star = false;

public:
    Arc(const std::string &s, Vertex &v, Vertex &revv) : name(s), adjv(v), revAdjv(revv) {}
    std::string::size_type getLength() const { return name.size(); }
    const std::string &getName() const { return name; }
    char adjVexName() const { return name[name.size() - 1]; }   // arc head
    char revAdjVexName() const { return name[0]; }  // arc tail
    Vertex &adjVex() const { return adjv; } // arc head
    Vertex &revAdjVex() const { return revAdjv; }   // arc tail
    void setStar() const { star = true; }
    void clearStar() const { star = false; }
    bool isStar() const { return star; }
};
```

### 点数据结构

```
class Vertex {
    friend class WordGraph;
private:
    char name;
    std::vector<Arc> adjarcs;
    std::vector<Arc *> revadjarcs;
public:
    explicit Vertex(char n) : name(n) {}
    char getName() const { return name; }
    const std::vector<Arc> &adjArcs() const { return adjarcs; }
    std::vector<Arc> &adjArcs() { return adjarcs; }
    const std::vector<Arc *> &revAdjArcsPtr() const { return revadjarcs; }
    std::vector<Arc *> &revAdjArcsPtr() { return revadjarcs; }
};
```

### 有向图的十字链表表示

```
class WordGraph {
    friend class WordMost;

private:
    std::vector<Vertex> v;

public:
    WordGraph();
    Vertex &getVertex(char c) { return v[c - 'a']; }
    const Vertex &getVertex(char c) const { return v[c - 'a']; }
    void setRevArc();
    void clearAllStar() const;
    std::vector<Vertex> &getAllVertex() { return v; }
    const std::vector<Vertex> &getAllVertex() const { return v; }
    void createArc(const std::string &s) {
        const char b = s[0], e = s[s.size() - 1];
        getVertex(b).adjarcs.push_back(Arc(s, getVertex(e), getVertex(b)));
    }
#ifndef NDEBUG
    std::ostream &print(std::ostream &os = std::cout) const;
    std::ostream &printrev(std::ostream &os = std::cout) const;
#endif  // NDEBUG
};
```

# 算法说明

## 最长单词链搜索

### 在小规模数据下的遍历搜索

对于小规模数据（50个单词以内），可以采用深度优先搜索进行遍历，对某个点的所有未搜索过的边循环递归调用遍历搜索函数，将star标志置为true，并且在其递归调用结束后将该边的star标志清除，因为下一条边的搜索可能还会搜索到该边，由于递归调用的归纳证明可以得到改算法可以遍历所有单词链的组合。

- 遍历算法

```
void DSearch(vector<ArcNode> &qlist, int depth, Vertex &vex, ArcNode &node, WordMost
*wm)
{
    int flag = 0;
    for(int i=0; i<vex.adjArcs().size(); i++)
    {
        if(vex.adjArcs()[i].isStar() == false)
        {
            /*cout << vex.adjArcs()[i].getName() << endl;*/
            flag = 1;
            vex.adjArcs()[i].setStar();
            node.depth = depth;
```

```
            node.value = vex.adjArcs()[i].getName();
            qlist.push_back(node);
            DSearch(qlist, depth+1, vex.adjArcs()[i].adjVex() , node, wm);
            vex.adjArcs()[i].clearStar();
            qlist.pop_back();
            /*for(int i=0; i<qlist.size() ; i++)
                cout << qlist[i].value << " ";
            cout << endl;*/
        }
    }
    if(flag == 0)
    {
        /*for (int i = 0; i < qlist.size(); i++)
            cout << qlist[i].value << " ";
        cout << endl;*/
        if(depth-1 > wm->getMaxlength())
        {
            wm->setMaxLength(depth-1);
            wm->setQlist(qlist);
        }
    }
}
```

## 大规模下的爬山算法

对于大规模数据，搜索到精确最长链是不可能的，这里采用的是重新随机的爬山算法，规定一个timelimit表示最长时间，一共进行timelimit时间的爬山搜索。

主要思想是随机对一个节点进行随机的深度搜索，直到搜到节点的所有临边均被搜索过，然后对这个随机出来的状态进行变换，得到相邻状态，变换方式如下：

如果当前的链长小于20，则随机取后半部分的一个边，将其之后（包括其）的边全部删掉，从删掉边的前一个节点重新随机深度搜索，重新得到一个链，如果此链的长度大于原长，则采用并继续替换，如果没原长长，则不接受，继续进行上述变换；

如果当前的链长大于20，则对后面的10个边中随机找一个进行上述变换，这样的好处是，选取后面的边进行变换大概率可以使得整个链的长度得到增长。

当上述爬山法进行了整个时间限制的1/50时，重新随机一个初始边进行爬山，也就是随机重启的爬山法，这样不会由于特殊的前几个节点选取情况导致结果与最长链出现很大的偏差。

- 爬山法中的随机深度搜索

```
void randDSearch(vector<Arc *> &arclist,vector<ArcNode> &qlist, Vertex &vex, ArcNode
&node)
{
    vector<int> arcArray;
    for (int i = 0; i < vex.adjArcs().size(); i++)
    {
        if (vex.adjArcs()[i].isStar() == false)
        {
            arcArray.push_back(i);
        }
    }
```

```
    /*cout <<"vex " << vex.getName() << " " <<  " arcArray = " << arcArray.size() <<
endl;*/
    if (arcArray.size() == 0)
    {
        /*for (int i = 0; i < qlist.size() ; i++)
            cout << qlist[i].value << "-";
        cout << endl;*/
        return;
    }
    else
    {
        int rnum = rand() % arcArray.size();
        vex.adjArcs()[arcArray[rnum]].setStar();
        arclist.push_back(&vex.adjArcs()[arcArray[rnum]]);
        node.value = vex.adjArcs()[arcArray[rnum]].getName();
        qlist.push_back(node);
        randDSearch(arclist, qlist, vex.adjArcs()[arcArray[rnum]].adjVex(), node);
    }
}
```

- 爬山算法

```
void WordMost::wordMostLargeScaleSearch()
{
    srand(time(0));
    setMaxLength(-1);
    graph.clearAllStar();
    vector<Arc *> arclist,lastarclist;
    vector<ArcNode> qlist,lastqlist;
    ArcNode node;
    vector<Vertex> &v = graph.v;
    time_t starttime = clock(),lasttime = clock();
    int vexnum;
    int arcnum;
    int curlength;
    while (clock() - starttime < timelimit)
    {
        do
        {
            graph.clearAllStar();
            qlist.clear();
            arclist.clear();
            vexnum = rand() % 26;
            randDSearch(arclist, qlist, v[vexnum], node);
            curlength = qlist.size();
            if (curlength > getMaxlength())
            {
                setMaxLength(curlength);
                setQlist(qlist);
            }
            lastarclist.assign(arclist.begin(), arclist.end());
            lastqlist.assign(qlist.begin(), qlist.end());
        } while (curlength == 0);
```

```
        lasttime = clock();

        while (clock() - lasttime < timelimit/50)
        {
            if (curlength > 20)
                arcnum = curlength - rand() % 10 - 1;
            else if (curlength == 1)
                arcnum = 0;
            else
                arcnum = curlength - rand() % (curlength / 2) - 1;
            qlist.erase(begin(qlist) + arcnum, end(qlist));
            Arc *removearc = arclist[arcnum];
            for (int j = arcnum; j < curlength; j++)
            {
                arclist[j]->clearStar();
            }
            arclist.erase(begin(arclist) + arcnum, end(arclist));

            randDSearch(arclist, qlist, removearc->revAdjVex(), node);
            if (qlist.size() > curlength)
            {
                curlength = qlist.size();
                lastarclist.assign(arclist.begin(), arclist.end());
                lastqlist.assign(qlist.begin(), qlist.end());
            }
            else
            {
                graph.clearAllStar();
                arclist.assign(lastarclist.begin(), lastarclist.end());
                qlist.assign(lastqlist.begin(), lastqlist.end());
                for (int j = 0; j < arclist.size(); j++)
                {
                    arclist[j]->setStar();
                }
            }
        }
        if (curlength > getMaxlength())
        {
            setMaxLength(curlength);
            setQlist(qlist);
        }
    }

}
```

## 指定单词或字符数量的搜索

采用深度首先的深度优先搜索，将深度限制为指定的单词数或字符数，没有太多难度，只不过在指定头结点和尾结点时进行了优化，并优化了数据结构和操作的速度。指定尾结点将使用十字链表中逆邻接表进行反向DFS以加快速度

单词数量和字符数量差不多，都是深度受限（或字符数受限）

对于输出，首先留出第一行数量位置，最后回来输出数量

### 正向 DFS

```
static void DLS(const Vertex &v) {
    for (const auto &arc : v.adjArcs()) {
        if (arc.isStar() == false) {
            if (wordlist.size() < depthLimit) {
                arc.setStar();
                wordlist.push_back(&arc.getName());
                DLS(arc.adjVex());
                wordlist.pop_back();
                arc.clearStar();
            } else if (wordlist.size() == depthLimit) {
                writeListToFile(arc.getName().c_str());
            }
        }
    }
}
```

### 反向 DFS

```
static void revDLS(const Vertex &v) {
    for (const auto &arc : v.revAdjArcsPtr()) {
        if (arc->isStar() == false) {
            if (wordlist.size() < depthLimit) {
                arc->setStar();
                wordlist.push_back(&arc->getName());
                revDLS(arc->revAdjVex());
                wordlist.pop_back();
                arc->clearStar();
            } else if (wordlist.size() == depthLimit) {
                revWriteListToFile(arc->getName().c_str());
            }
        }
    }
}
```

## 测试用例

- `test_1.txt` 测试基本功能、重复输入、循环单词（图中的圈）
- `test_2.txt` 为空文件，测试报错
- `test_3.txt` 测试单词的分割
- `test_4.txt` 没有任何单词链，且每个边都是环（边的两端是同一个顶点）
- `test_5.txt` 测试开头为未知字符
- `test_6.txt` 为英文短文，测试基本功能
- `test_7.txt` 测试文本中含中文字符，此时 char 可能为负，可能引起 cctype 中函数报错
- `test_8.txt` 为较长的文本，测试准确搜索效率
- `test_9.txt` 为较短的文本，测试准确搜索功能
- `test_10.txt` 为很长的文本，测试局部搜索功能

所有测试用例均正确通过。