### 第11题：词梯【2】

#### 数据结构：

存储单词表为了可以快速通过单词字符串查找到是否存在相对应的单词，故使用STL的map容器，即map<string, string>。

为了存储词梯路径边两点的信息，用pair<string,string>

同时，为了存储从起点或终点开始搜索的边，同样使用map<string, string>

两个方向的访问辅助哈希表同样使用map<string, string>

在双向BFS时要使用队列，即queue<string>

由于词梯路径数量不定，故用vector<string>存储可以用来输出路径的中间单词

#### 算法设计思想：

先读取文件用CreatDict()函数创建字典，在获取起点词和终点词时同时判断输入是否合法。

然后用WorldLadder函数找到并输出词梯路径。

可以对起点词根据改变规则得到单词进行BFS搜索，搜索到终点词即可获得最短路径。单词的获得可以通过对要变换的词每个位置逐个变换一个字母，然后判断是否在单词表里获得。

由于随着终点词所在的搜索层次的深度增加，所对应的宽度也可能增加，且该层次除了终点词，其他词的被搜索都是没有必要且浪费时间的。同时也有可能面临着搜索空间的爆炸问题。

因为我们同时知道起点词和终点词，可以对BFS进行优化为**双向BFS**。同时从起点和终点两个方向开始搜索，一旦搜索到相同的值，意味着找到了一条联通起点和终点的最短路径。

这样可以起到剪枝和提高效率的作用。

双向 BFS的基本实现思路如下：

1. 创建「两个队列」分别用于两个方向的搜索；
2. 创建「两个哈希表（map<string, string>）」用于「解决相同节点重复搜索」和「记录转换次数」；
3. 为了尽可能让两个搜索方向“平均”，每次从队列中取值进行扩展时，先判断哪个队列容量较少；
4. 如果在搜索过程中「搜索到对方搜索过的节点」，说明找到了最短路径。

输出路径则借助「搜索到对方搜索过的节点」（简称mid），找到mid到start的路径并反转，与mid到end的路径拼接即为整条路径。

#### 源程序：

#include <bits/stdc++.h>

using namespace std;

map<string, string> dict,startSet,endSet; //单词表，起点出发搜索的边集合 终点出发搜索的边集合

vector<string> midPath; //存储中间点以便找到路径

int len; //字符串 长度

typedef pair<string,string> line;

ifstream in;

void CreatDict (){

in.open("words.txt");

if (!in){

cout << "打开文件 words.txt 失败";

exit(1);

}

string temp;

while ( getline(in,temp) ){

dict.insert(make\_pair(temp,temp));

}

in.close();

cout <<"已读取建立单词表"<<endl;

}

bool Check(string isExit){

if (dict.count(isExit)) return true;

else return false;

}

//flag : 0为从起点开始搜索，1为从终点开始搜索

int update(queue <string> &q,map<string, string> &visit,map<string, string> &other,int count,int flag){

int isFind = 0;

for(int i = 0;i < count;i++){

string st = q.front();

q.pop();

for (int j = 0;j < len;j++){

string temp = st;

for (char ch = 'a';ch <='z';ch++){

temp[j] = ch;

if (dict.count(temp) && (!visit.count(temp))){

//若单词存在且没在当前方向被访问过

q.push(temp);

//存出边以便找寻路径

if(flag == 0) startSet[temp] = st;

else{

endSet[temp] = st;

}

visit[temp] = temp;

//若单词被另一方向搜索过，则找到路径，返回值为-1

if (other.count(temp)) {

midPath.push\_back(temp);

isFind = -1;

}

}

}

}

}

if (isFind == -1) return isFind;

else return q.size();

}

//双向BFS

void WorldLadder(string start,string end){

//初始化

queue <string> fromStart,fromEnd;

map<string, string> visitS,visitE;

//从起点或终点开始的搜索队列访问数组

visitS[start] = start;

visitE[end] = end;

//从起点或终点开始的搜索队列

fromStart.push(start);

fromEnd.push(end);

int countS = 1,countE = 1;//当前搜索层的元素个数

int isExitRoad = 0;

//双向BFS主要过程

while (!fromStart.empty() && !fromEnd.empty()){

if (fromStart.size() <= fromEnd.size()){

countS = update(fromStart,visitS,visitE,countS,0);

}

else{

countE = update(fromEnd,visitE,visitS,countE,1);

}

//返回值为-1则说明词梯找到，终止搜索

if (countS == -1 || countE == -1) {

cout << "词梯已找寻到\n";

isExitRoad = 1;

break;

}

}

if (!isExitRoad) cout <<"词梯不存在"<<endl;

//输出路径

cout <<"开始输出词梯\n"<<endl;

for (auto it = midPath.begin(); it != midPath.end(); it++)

{

vector<string> path;

string tmp = \*it;

string tmp1 = \*it;

path.push\_back(tmp);

//根据中间词找到该词到起点的路径并转换

while (tmp != start){

tmp = startSet[tmp];

path.push\_back(tmp);

};

reverse(path.begin(),path.end());

//根据中间词找到该词到终点的路径

while (tmp1 != end){

tmp1 = endSet[tmp1];

path.push\_back(tmp1);

}

for (auto it = path.begin(); it != path.end(); it++)

{

cout <<(\*it);

if (it != path.end()-1){

cout<<"->";

}

}

cout <<endl;

}

}

int main(){

CreatDict();

string start,end;

//输入

while (1){

cout <<"请输入起点词\n";

cin >>start;

if (Check(start)) break;

else {

cout <<"该词不存在，请重新输入\n";

continue;

}

}

while (1){

cout <<"请输入终点词\n";

cin >>end;

if (start.length() != end.length()){

cout <<"两词长度不等，请重新输入终点词\n";

continue;

}

if (Check(end)) break;

else {

cout <<"该词不存在，请重新输入\n";

continue;

}

}

//处理输出

if (start == end){

cout << "两词相同，无需转换，词梯长度为0\n";

}

else {

len = start.length();

WorldLadder(start,end);

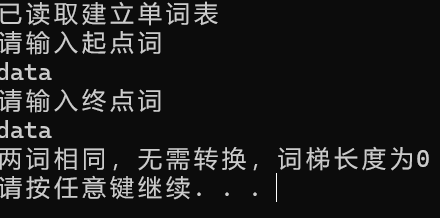
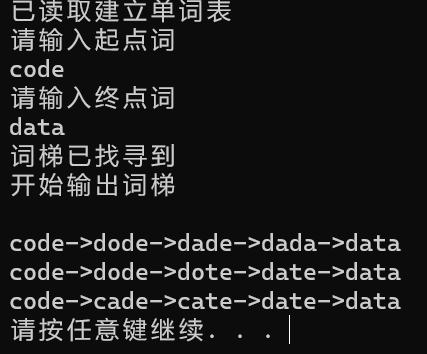
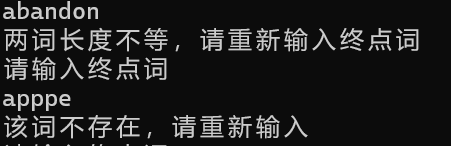
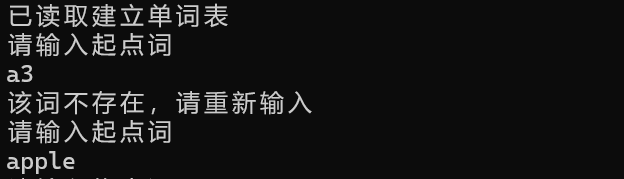
}

system("pause");

return 0;

}

#### 测试数据和结果





#### 时间复杂度：

朴素BFS的时间复杂度大概为O(n^2),双向BFS算法时间复杂度大概为O (（n^2）/2)。

#### 反思：

函数间的耦合度较高，且每个函数所拥有的功能超过两个，不便于后续的维护和复用。

判断是否被访问过的哈希表和存储单词之间的边的信息的map容器实际上可以合并在一起，减少空间的使用。

函数里的嵌套较多，可以尝试将其中的一些嵌套封装为函数来被调用。

#### 改进：

#### 该题代码行：

152行