Problema de la escalera de palabras

Graph.h

```
#ifndef GRAFOESCALERAPALABRAS_GRAPH_H
#define GRAFOESCALERAPALABRAS_GRAPH_H

#include <iostream>
#include <vector>

using namespace std;

template<class T>
struct Edge;
template<class T>
class Vertex;

template<class T>
class Edge{
public:
   Vertex<T>* to;
   int weight;
   friend ostream &operator<<(ostream &out, Edge<T>* edge) {
       out << "To: " << edge->to->data;
       out << ", Weight: " << edge->weight << endl;
       return out;
   }
};

template<class T>
class Vertex{
public:
   T data;
   int inDegree;
   int outDegree;
   vector<Edge<T>*> connectedTo;

   Vertex<T>* predecessor;
   int distance;
   char color;

   Vertex(const T& value);
   ~Vertex();

   void addNeighbor(Vertex<T>* to, int weight=0);
   int getWeight(const T& value);

   friend ostream &operator<<(ostream &out, Vertex<T>* vertex) {
       out << vertex->data << endl;
       out << "In degree: " << vertex->inDegree << endl;
       out << "out degree: " << vertex->outDegree << endl;
       out << "Edges: " << endl;
       vertex->connectedTo.print();
       return out;
   }
};

template<class T>
class Graph {
```

```cpp
public:
   int count;
   vector<Vertex<T>*> vertexList;
   Graph();
   ~Graph();
   Vertex<T>* addVertex(const T& value);
   Vertex<T>* getVertex(const T& value);
   void addEdge(const T& from, const T& to, int weight=0);
};


#endif //GRAFOESCALERAPALABRAS_GRAPH_H
```

Graph.cpp

```cpp
#ifndef GRAFOESCALERAPALABRAS_GRAPH_CPP
#define GRAFOESCALERAPALABRAS_GRAPH_CPP

#include "Graph.h"

template<class T>
Vertex<T>::Vertex(const T& value) {
   data = value;
   inDegree = 0;
   outDegree = 0;
   connectedTo = {};
   predecessor = 0;
   distance = 0;
   color = 'w';
}

template<class T>
Vertex<T>::~Vertex() {

}

template<class T>
void Vertex<T>::addNeighbor(Vertex<T> *to, int weight) {
   Edge<T>* temp = new Edge<T>;
   temp->to = to;
   temp->weight = weight;

   outDegree++;
   to->inDegree++;

   connectedTo.push_back(temp);
}

template<class T>
int Vertex<T>::getWeight(const T &value) {
   for(int i=0; i < connectedTo.size(); i++){
       Edge<T>* temp = connectedTo.get(i);
       if(temp->to->data == value){
           return connectedTo.get(i)->weight;
       }
   }
   return NULL;
}

template<class T>
Graph<T>::Graph() {
   count = 0;
   vertexList = {};
}

template<class T>
Graph<T>::~Graph() {
}

template<class T>
Vertex<T>* Graph<T>::addVertex(const T &value) {
   Vertex<T>* newVertex = new Vertex<T>(value);
   vertexList.push_back(newVertex);
   count++;
   return newVertex;
}

template<class T>
void Graph<T>::addEdge(const T& from, const T& to, int weight) {
   Vertex<T>* fromVertex = getVertex(from);
   if(!fromVertex){
       fromVertex = addVertex(from);
   }
   Vertex<T>* toVertex = getVertex(to);
   if(!toVertex){
       toVertex = addVertex(to);
   }
   fromVertex->addNeighbor(toVertex, weight);
}

template<class T>
Vertex<T> *Graph<T>::getVertex(const T &value) {
   for(int i=0; i < vertexList.size();i++ ){
       if(vertexList[i]->data == value) return vertexList[i];
   }
   return NULL;
}


#endif //GRAFOESCALERAPALABRAS_GRAPH_CPP
```

main.cpp

```
#include <iostream>
#include <vector>
#include <map>
#include <queue>
#include <fstream>
#include "Graph.cpp"

using namespace std;
```

```
Graph<string> buildGraph(){
   map<string, vector<string> > dict = {};
   Graph<string> g;
   string word = "";
ifstream
file("C:\\Users\\nesto\\CLionProjects\\GrafoEscaleraPalabras\\listapalabras
.txt", ifstream::in);
   if(!file){
       cout << "Error reading file" << endl;
   }
   if(file.is_open()){
       while(getline(file, word)){
           for(int i=0; i<word.size(); i++){
               string bucket = word.substr(0,i) + "_" + word.substr(i+1,
word.size());
               dict[bucket].push_back(word);
           }
       }
       file.close();
   }
   for(auto[k,v]: dict){
       for(string word1: dict[k]){
           for(string word2: dict[k]){
               if(word1 != word2){
                   g.addEdge(word1, word2);
               }
           }
       }
   }
   return g;
}

void bea(Vertex<string>* begin){
   begin->distance = 0;
   begin->predecessor = 0;
   queue<Vertex<string>*> queueVertex;
   queueVertex.push(begin);
   while(queueVertex.size() > 0){
       Vertex<string>* curVertex = queueVertex.front();
       queueVertex.pop();
       for(Edge<string>* neighbor: curVertex->connectedTo){
           if(neighbor->to->color == 'w') { // white
                neighbor->to->color = 'g'; // grey
                neighbor->to->distance = curVertex->distance + 1;
               neighbor->to->predecessor = curVertex;
               queueVertex.push(neighbor->to);
           }
       }
       curVertex->color = 'b'; //black
   }
}

void traversal(Vertex<string>* vertex){
   while(vertex->predecessor){
       cout << vertex->data << endl;
       vertex = vertex->predecessor;
   }
   cout << vertex->data << endl;
}
```

```
int main() {
   cout << "Comienza construcción del grafo" << endl;
   Graph g = buildGraph();
   cout << "Finanizó construcción del grafo" << endl;
   cout << "Realizando BEA desde vertice" << endl;
   bea(g.getVertex("lelo")); // palabra inicial
   cout << "Imprimiendo recorrido" << endl;
   traversal(g.getVertex("agil")); // palabra final
   return 0;
}
```