Proj 7

Clay Sarafin & Taylor Heilman

April 14, 2016

1. See attached.

2. See attached.

3. Why is an adjacency list representation used in BFS? (Consider the running time of the algorithm if an adjacency matrix is used instead.)

The runtime of a Breadth First Search depends upon how the graph is represented. Because of this it is important to choose the data structure which offers the fastest runtime. An adjacentcy list offers a faster runtime compared to an adjacency matrix because of how the list stores data. An adjacency list is efficient with the data it stores in the fact that only edges found in the graph are stored in the list. An adjacency matrix on the other hand stores every possible connection, with a nonzero number denoting the connection with a weight represented by the number, and the number 0 denoting the connection to not exist. Because of these different representations, the traversal time of each implementation varies. To traverse an adjacency list you will look at each vertex in the graph and each edge of the graph. Thus the runtime of the list totals to $O(n+m)$, where n is the number of vertices and m is the total number of edges. To traverse an adjacency matrix you look at $n * n$ elements, since you will look at every row and column in the matrix. The adjacency matrix has dimensions of $n * n$, hence the runtime for the matrix totals to $O(n^2)$.

```
// test_ds.cpp
// Disjoint Set header file
// Clay Sarafin & Taylor Heilman

#ifndef DS_H
#define DS_H

#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

template <class T>
class DSNode
{
public:
        DSNode();
        DSNode(T* contents);

        T* data;
        DSNode<T>* parent;
        int dsrank;

        bool operator!=(const DSNode<T>& ds){
                return *(data) != *(ds.data);
        }
};


template <class T>
class DisjointSets
{
        public:
                DisjointSets();                                 // default constructor
                DisjointSets(int size);                         // constructor with given capacit
y
                DisjointSets(const DisjointSets<T>& ds);        // copy constructor
                ~DisjointSets();                                // destructor
                DSNode<T>* makeSet(T* x);                       // make a new singleton set conta
ining data x
                void unionSets(DSNode<T>* x, DSNode<T>* y);  // union the disjoint sets contai
ning data x and y
                DSNode<T>* findSet(DSNode<T>* x);               // return the representative of t
he set containing x
                DisjointSets<T>& operator=(const DisjointSets<T>& ds);  // assignment operator
                std::string toString();            // return a string representation of the di
sjoint set forest
        private:
                void link(DSNode<T>* x, DSNode<T>* y);
                DSNode<T> **elements;   // array of nodes in the forest
                int capacity;           // size of elements array
                int length;             // number of elements in the forest

                void copy(const DisjointSets<T>& ds);
                void dealloc();

                bool inForest(DSNode<T>* x);
};

class FullError { };      // full exception
class NotFoundError { };  // element not found exception

#endif
```

```
#include "ds.cpp"
```

```
// test_ds.cpp
// Disjoint Set code
// Clay Sarafin & Taylor Heilman

#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sstream>

using namespace std;

/*=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
                     DSNode - (DE)CONSTRUCTORS
=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+*/
/*-----------------------------------------------
 * Deafult Constructor
 * PreConditions:       n/a
 * PostConditions:      empty DSNode is created
---------------------------------------------*/
template<class T>
DSNode<T>::DSNode(){
        data = NULL;
        parent = NULL;
        dsrank = 0;
}
/*-----------------------------------------------
 * Construct with data
 * PreConditions:       pointer to the contents
 * PostConditions:      data of node class points to desirec contents
                        otherwise, just like an empty node
---------------------------------------------*/
template<class T>
DSNode<T>::DSNode(T* contents){
        data = contents;
        parent = NULL;
        dsrank = 0;
}



/*=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
                  Disjoint Set - (DE)CONSTRUCTORS
=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+*/

/*-----------------------------------------------
 * Default Constructor
 * PreConditions:       n/a
 * PostConditions:      empty DS class created
---------------------------------------------*/
template<class T>
DisjointSets<T>::DisjointSets(){
        capacity = 0;
        length = 0;
}

/*-----------------------------------------------
 * Construct with capacity
 * PreConditions:       size of array
 * PostConditions:      empty DS class created with an array of size elements
---------------------------------------------*/
template<class T>
DisjointSets<T>::DisjointSets(int size){
```

Use a default size > 0.

```
        capacity = size;
        length = 0;
        elements = new DSNode<T>*[size];
}

/*---------------------------------------------
 * Copy Constructor
 * PreConditions:       a source DS object
 * PostConditions:      a copy of the DS object is created
                        all items point to the correct values, with no aliasing
---------------------------------------------*/
template<class T>
DisjointSets<T>::DisjointSets(const DisjointSets<T> &ds){
        copy(ds);
}

/*---------------------------------------------
 * Destructor
 * PreConditions:       n/a
 * PostConditions:      DS object is deconstructed
---------------------------------------------*/
template<class T>
DisjointSets<T>::~DisjointSets(){
        dealloc();
}




/*=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
                     Disjoint Set - FUNCTIONS
=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+*/

/*---------------------------------------------
 * makeSet()
 * PreConditions:       a pointer to data x
 * PostConditions:      a node that points to x is created
                        user expected to keep track of the returned Node
---------------------------------------------*/
template<class T>
DSNode<T>* DisjointSets<T>::makeSet(T* x){
        if (length == capacity)
                throw FullError();

        //initialize the new node, make it point to itself
        DSNode<T>* node;
        node = new DSNode<T>(x);
        node->parent = node;

        //add the node to the array of elements
        elements[length] = node;
        length++;


        return node;
}

/*---------------------------------------------
 * unionSets()
 * PreConditions:       pointer to nodes x and y
 * PostConditions:      a union of roots in nodes x and y are created
---------------------------------------------*/
template<class T>
```

```
void DisjointSets<T>::unionSets(DSNode<T>* x, DSNode<T>* y){
        link(findSet(x), findSet(y));
}

/*---------------------------------------------
 * findSet()
 * PreConditions:       pointer to a node in the DS tree
 * PostConditions:      the root of the tree where the node is located is returned
---------------------------------------------*/
template<class T>
DSNode<T>* DisjointSets<T>::findSet(DSNode<T>* x){
        if (inForest(x) == false)
                throw NotFoundError();
        if (x->parent != x)
                x->parent = findSet(x->parent);
        return x->parent;
}

/*---------------------------------------------
 * operator=
 * PreConditions:       a source DS object
 * PostConditions:      current DS object is destroyed is replaced with the contents of the so
urce
                        no aliasing between the two objects will occur; both unique from each
other
---------------------------------------------*/
template<class T>
DisjointSets<T>& DisjointSets<T>::operator=(const DisjointSets<T>& ds){
        dealloc();
        copy(ds);
}
```

-3 (only if ds != this) return *this;

```
/*---------------------------------------------
 * toString()
 * PreConditions:       n/a
 * PostConditions:      string representation of the DS will be created
                        created in the form "data:rank -> parent.data:parent.rank -> ..." unti
l the root has been reached
---------------------------------------------*/
template<class T>
std::string DisjointSets<T>::toString(){
        DSNode<T> * ptr;
        std::ostringstream stream;
        for (int i=0; i<length; i++){
                ptr = elements[i];
                //find the parent of each node until it is the parent is itself & put it in th
e appropriate format
                while (ptr != ptr->parent){
                        stream << *(ptr->data) << ":" << ptr->dsrank;
                        stream << " -> ";
                        ptr = ptr->parent;
                }
                stream << *(ptr->data) << ":" << ptr->dsrank << '\n';
        }
        std::string str = stream.str();
        str.pop_back();         //deletes stray '\n' character at the end

        return str;
}

/*=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
                  Disjoint Set - PRIVATE FUNCTIONS
=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+*/
```

```
/*--------------------------------------------
 * link()
 * PreConditions:       pointers to nodes x and y
 * PostConditions:      a link between the nodes x and y is formed
--------------------------------------------*/
template<class T>
void DisjointSets<T>::link(DSNode<T>* x, DSNode<T>* y){
        if (inForest(x) == false)
                throw NotFoundError();
        if (inForest(y) == false)
                throw NotFoundError();
        //compare ranks of x and y; connect based on rank (smaller rank gets connected to larg
er rank)
        //if ranks are the same, make add 1 to x's rank
        if (x->dsrank < y->dsrank)
                x->parent = y;
        else{
                y->parent = x;
                if (x->dsrank == y->dsrank)
                        x->dsrank = x->dsrank + 1;
        }
}

/*--------------------------------------------
 * copy()
 * PreConditions:       source DS object
 * PostConditions:      copy of the source object is formed; completely unique form each other
--------------------------------------------*/
template<class T>
void DisjointSets<T>::copy(const DisjointSets<T>& ds){
        capacity = ds.capacity;
        length = ds.length;

        int parent[ds.length];

        //look through each element, find the parent of each element,
        //and put the index of each pointer in the array
        for (int i=0; i<length; i++){
                DSNode<T>* p = ds.elements[i]->parent;
                for (int j=0; j<length; j++){
                        if (p == ds.elements[j]){
                                parent[i] = j;
                        }
                }
        }


        elements = new DSNode<T>*[capacity];
        //create the new elements for the new DS
        for (int i=0; i<length; i++){
                elements[i] = new DSNode<T>(ds.elements[i]->data);
                elements[i]->dsrank = ds.elements[i]->dsrank;
        }
        //after all of the Nodes have been placed, assign the parents to each node
        //based off of the index value in the parent array
        for (int i=0; i<length; i++)
                elements[i]->parent = elements[parent[i]];



}
```

```
/*-----------------------------------------------
 * dealloc()
 * PreConditions:       n/a
 * PostConditions:      elements in the DS is deleted, including the array itself
-----------------------------------------------*/
template<class T>
void DisjointSets<T>::dealloc(){
        for (int i=0; i<length; i++)
                delete elements[i];
        delete[] elements;
}

/*-----------------------------------------------
 * inForest()
 * PreConditions:       pointer to node
 * PostConditions:      true if in DSF, false if not
-----------------------------------------------*/
template<class T>
bool DisjointSets<T>::inForest(DSNode<T>* x){
        for(int i=0; i<length; i++)
                if (x == elements[i])
                        return true;
        return false;
}
```

```
// test_ds.cpp
// Disjoint Set tests
// Clay Sarafin & Taylor Heilman
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

#include "ds.h"

using namespace std;

int * zero  = new int(0);
int * one   = new int(1);
int * two   = new int(2);
int * three = new int(3);
int * four  = new int(4);
int * five  = new int(5);
int * six   = new int(6);
int * seven = new int(7);
int * eight = new int(8);
int * nine  = new int(9);


void insertTest(){
        DisjointSets<int> ds0(64);

        DSNode<int>* node00 = ds0.makeSet(zero);
        DSNode<int>* node01 = ds0.makeSet(one);
        DSNode<int>* node02 = ds0.makeSet(two);

        assert(ds0.toString() == "0:0\n1:0\n2:0");

        DSNode<int>* node01_1 = ds0.makeSet(one);

        assert(ds0.toString() == "0:0\n1:0\n2:0\n1:0");

        //test for the FullError exception in makeSet()
        DisjointSets<int> ds1(2);
        DSNode<int>* node10 = ds1.makeSet(zero);
        DSNode<int>* node11 = ds1.makeSet(one);
        try{
                DSNode<int>* node12 = ds1.makeSet(two);
                assert(false);
        }
        catch(FullError exception){
        }



}

void unionTest(){
        DisjointSets<int> ds(64);

        DSNode<int>* node0 = ds.makeSet(zero);
        DSNode<int>* node2 = ds.makeSet(two);

        ds.unionSets(node0, node2);

        assert(ds.toString() == "0:1\n2:0 -> 0:1");
```

```
        DSNode<int>* node1 = ds.makeSet(one);

        assert(ds.toString() == "0:1\n2:0 -> 0:1\n1:0");

        DisjointSets<int> ds1(64);
        DSNode<int>* node3 = ds1.makeSet(three);
        try{
                ds.unionSets(node0, node3);
                assert(false);
        }
        catch(NotFoundError exception){
        }

}

void findTest(){
        DisjointSets<int> ds(64);

        DSNode<int>* node0 = ds.makeSet(zero);
        DSNode<int>* node2 = ds.makeSet(two);

        assert(ds.findSet(node0) == node0);
        assert(ds.findSet(node2) == node2);

        ds.unionSets(node0, node2);

        assert(ds.findSet(node0) == node0);
        assert(ds.findSet(node2) == node0);

        //test for NotFoundError exception, where the node is not in the requested DSF
        DisjointSets<int> ds1(64);
        DSNode<int>* node1 = ds1.makeSet(one);
        try{
                ds.findSet(node1);
                assert(false);
        }
        catch(NotFoundError exception){
        }
}

void copyTest(){
        DisjointSets<int> ds(64);


        DSNode<int>* node0 = ds.makeSet(zero);
        DSNode<int>* node2 = ds.makeSet(two);

        DisjointSets<int> ds1(ds);

        assert(ds1.toString() == "0:0\n2:0");

        ds.unionSets(node0, node2);

        DisjointSets<int> ds2(ds);
        assert(ds2.toString() == "0:1\n2:0 -> 0:1");

        DisjointSets<int> ds3 = ds;
        assert(ds3.toString() == "0:1\n2:0 -> 0:1");

}


int main(){
```

```
        insertTest();
        unionTest();
        findTest();
        copyTest();


        delete one;
        delete two;
        delete three;
        delete four;
        delete five;
        delete six;
        delete seven;
        delete eight;
        delete nine;
        return 0;
}
```

```
// graph.h
// Graph header file
// Clay Sarafin & Taylor Heilman

#ifndef GRAPH_H
#define GRAPH_H

#include "list.h"
#include "ds.h"
#include "pq.h"



class Vertex{
public:
        Vertex();
        Vertex(int contents);                   //construct with value
        int data;                               //value of vertex
        char color;                             //w=white, g=grey

        Vertex* pred;                           //predecessor of node, for dfs()

        List<Vertex> connections;               //list of connections, for both dfs() and Krus
kal()
};


class Edge{
public:
        Edge();
        Edge(Vertex* vertexU, Vertex* vertexV, int w);

        Vertex* u;
        Vertex* v;

        DSNode<Vertex>* nodeU;
        DSNode<Vertex>* nodeV;

        int weight;


        bool operator<(const Edge& e){  //need this for selection sort
                return weight < e.weight;
        }
};

std::ostream& operator<< (std::ostream& os, const Edge& e){
        os << e.weight << "(" << e.u->data << "," << e.v->data << ")";
        return os;
}


class Graph{
public:
        Graph();
        Graph(std::string file);
        ~Graph();

        void dfs();
        void dfsVisit(Vertex * u);
        void Kruskal();
```

```
private:
        Vertex ** vertices;
        Edge ** edges;
        int capacity;
        int capacityEdge;
        int lengthEdge;

        void dealloc();
        //disallow copying
        Graph(const Graph& g) {};
        Graph& operator=(const Graph& g) {};
};

#endif

#include "graph.cpp"
```

✓

OK

```cpp
// graph.cpp
// Graph class code
// Clay Sarafin & Taylor Heilman

#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fstream>


/*=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
                     Edge - CONSTRUCTORS
=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+*/

/*---------------------------------------------
 * Default Constructor
 * PreConditions:       n/a
 * PostConditions:      empty Edge object created
 ---------------------------------------------*/
Edge::Edge(){
        weight = 0;
}
```

u = v = NULL;

```cpp
/*---------------------------------------------
 * Construct with all properties of the edge
 * PreConditions:       pointers to both vertices on the edge, and its weight
 * PostConditions:      Edge object with above properties will be created
 ---------------------------------------------*/
Edge::Edge(Vertex* vertexU, Vertex* vertexV, int w){
        u = vertexU;
        v = vertexV;
        weight = w;


}


/*=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
                     Vertex - CONSTRUCTORS
=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+*/
/*---------------------------------------------
 * Default Constructor
 * PreConditions:       n/a
 * PostConditions:      "empty" Vertex object created
                        color 'w' denotes it has not been visited in dfs()
 ---------------------------------------------*/
Vertex::Vertex(){
        color = 'w';
        pred = NULL;
}

/*---------------------------------------------
 * Construct with contents
 * PreConditions:       integer denoting its contents
 * PostConditions:      creates vertex object with appropriate contents
 ---------------------------------------------*/
Vertex::Vertex(int contents){
        data = contents;
        color = 'w';
        pred = NULL;
}
```

```
/*=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
                     Graph - (DE)CONSTRUCTORS
=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+*/
/*----------------------------------------------
 * Default Constructor
 * PreConditions:       n/a
 * PostConditions:      creates empty Graph object
                        nothing dynamically allocated
----------------------------------------------*/
Graph::Graph(){
        capacity = 0;
        capacityEdge = 0;                vertices = edges = NULL;
        lengthEdge = 0;
}

/*----------------------------------------------
 * Construct with file
 * PreConditions:       a string denoting the name of the text file
                        file has to be in the format of:
                                (number of vertices)
                                (adjacency matrix of wieghts)
                        ex:
                        5
                        0 1 2 0 0
                        1 0 1 1 0
                        2 1 0 0 3
                        0 1 0 0 1
                        0 0 3 1 0
 * PostConditions:      creates a graph based on the relationships defined in the text file
----------------------------------------------*/
Graph::Graph(std::string file){
        std::ifstream text(file, std::ifstream::in);
        std::string line;
        std::string str;
        std::string token;

        //read 1st line for capacity
        std::getline(text, str);
        capacity = stoi(str);
        vertices = new Vertex*[capacity];

        for (int i=0; i<capacity; i++)                  //never forget that you need to initia
lize every object in an array of pointers
                vertices[i] = new Vertex(i);
        capacityEdge = (capacity*capacity)-capacity;
        edges = new Edge*[capacityEdge];
        //matrix has total elements capacity^2,
        //nodes cannot be connected to itself (hence the diagonal of 0's) so capacity is also
subtracted

        lengthEdge = 0;

        //read rest of the file
        //read by line, then by character
        for (int i=0; i<capacity; i++){
                std::getline(text,line);
                std::istringstream iss(line);
                for (int j=0; j<capacity; j++){
                        std::getline(iss,token, ' ');
                        int weight = stoi(token);
                        if (weight != 0){
                                //create new edge based off of the value being read in, and th
```

```
e iterations it is at
                                edges[lengthEdge] = new Edge(vertices[i],vertices[j],weight);
                                vertices[i]->connections.append(vertices[j]);
                                lengthEdge++;
                        }
                }
        }
        text.close();
}

/*-----------------------------------------------
 * Deconstructor
 * PreConditions:       n/a
 * PostConditions:      everything allocated in the Graph object will be deallocated
---------------------------------------------*/
Graph::~Graph(){
        dealloc();
}


/*=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
                    Graph - FUNCTIONS
=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+*/
/*-----------------------------------------------
 * dfs()
 * PreConditions:       a valid graph, can be empty
 * PostConditions:      vertices visited will be printed out
                        will explore all of the elements of the spanning tree
                        if empty, nothing will bre printed out
---------------------------------------------*/
void Graph::dfs(){
        for (int i=0; i<capacity; i++)
                if (vertices[i]->color == 'w')
                        dfsVisit(vertices[i]);
        if (capacity != 0)
                cout << endl;
}

/*-----------------------------------------------
 * dfsVisit()
 * PreConditions:       pointer to a vertex in the graph
 * PostConditions:      vertex will be colored in "grey", denoting that it has been visited
                        will go through all connections that haven't been visited
---------------------------------------------*/
void Graph::dfsVisit(Vertex * u){
        u->color = 'g';                                 //mark vertex as visited
        cout << u->data << " ";                         //print out vertex visited
        for(int i=0; i<u->connections.length(); i++)
                if (u->connections[i]->color == 'w'){   //check if vertex has not been visited
                        u->connections[i]->pred = u;
                        dfsVisit(u->connections[i]);    //recursivley do this if it hasn't bee
n visited
                }

}



/*-----------------------------------------------
 * Kruskal()
 * PreConditions:       a valid graph
 * PostConditions:      will find and print out all edges in the Miniumum Spanning Tree (MST)
---------------------------------------------*/
```

```
void Graph::Kruskal(){
        List<Edge> a;                                           //contains all edges in MST
        DisjointSets<Vertex> vertexSet(capacity);
        DSNode<Vertex>* nodes[capacity];                        //all nodes created with all v
ertices
        for (int i=0; i<capacity; i++)
                nodes[i] = vertexSet.makeSet(vertices[i]);

        MinPriorityQueue<Edge> queue(lengthEdge);

        //go through every value in the nodes, and try to find the vertex each edge points to
        for (int i=0; i<capacity; i++){
                for (int j=0; j<lengthEdge; j++){
                        if (edges[j]->u == nodes[i]->data)
                                edges[j]->nodeU = nodes[i];
                        if (edges[j]->v == nodes[i]->data)
                                edges[j]->nodeV = nodes[i];
                }
        }
        //insert all edges into an Min Priority Queue, so all edges will be sorted by weight
        for(int i=0; i<lengthEdge; i++)
                queue.insert(edges[i]);
        //dequeue each element, check if
        for(int i=0; i<lengthEdge; i++){
                Edge * e = queue.extractMin();
                if (vertexSet.findSet(e->nodeU) != vertexSet.findSet(e->nodeV)){
                        a.append(e);
                        vertexSet.unionSets(e->nodeU, e->nodeV);
                }
        }

        //print out all edges in the list
        cout << a << endl;
}
```

-5 Use adj list instead. This makes the alg $\Theta(nm)$!

```
/*=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
                   Graph - PRIVATE FUNCTIONS
=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+*/


/*-----------------------------------------------
 * dealloc()
 * PreConditions:       n/a
 * PostConditions:      will deallocate all memory in the Graph object
-----------------------------------------------*/
void Graph::dealloc(){
        for (int i=0; i<capacity; i++)
                delete vertices[i];
        delete[] vertices;
        for (int i=0; i<lengthEdge; i++)
                delete edges[i];
        delete[] edges;
}
```

```
// test_graph.cpp
// Graph class tests
// Clay Sarafin & Taylor Heilman
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

#include "graph.h"

using namespace std;


void mainTest(){
        Graph graph("test.txt");

        cout << "Output for Kruskal's Algorithm:" << endl;
        graph.Kruskal();

        cout << "Output for Depth First Search:" << endl;
        graph.dfs();
}


int main(){
        mainTest();
        return 0;
}
```

-3

Insufficient testing!
more graphs!