<u>Graphs</u>

Proficiency: Mastery with Distinction

After this semester I feel as though my knowledge of graphs, their implementations and usages is one of my strongest subjects. Due to the fact that I was concurrently enrolled in Data Structures as well as Graph Theory I was able to get much practice implementing graphs and algorithms used on graphs. The two courses approached graphs with different ideology, however. Graph Theory focused mainly on traits of graphs and what makes them unique, while Data Structures focused mainly on designing an efficient way to implement graphs and graphical algorithms. In **Project 0111** we were assigned the task of implementing a weighted, directed graph. When implementing a graph class you have two main ways to represent a graph, a collection of adjacency lists or an adjacency matrix. Each representation has its advantages and disadvantages. An adjacency list offers a faster runtime for DFS and BFS 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 $O(n^2)$.

I was able to implement a graph class in both C++ and Python this semester. I also implemented Depth First Search in both Python and C++ (see attached). a Depth First Search is used to find a minimum spanning tree in a connected graph. A minimum spanning tree of graph G is a subgraph of G that is a tree and has minimum weighted edges. A minimum spanning tree has many applications such as network design and finding the most efficient routes (traveling salesman problem). Breadth First Search accomplishes the same outcome as DFS but in a different fashion. While both are greedy algorithms, DFS spans out as far as it can and once it can travel no further it back traces and checks to see if it can find new nodes to travel to. A BFS, unlike DFS, travels to all adjacent nodes first and then spreads out. Visually a BFS looks more like an expanding circle, while DFS looks like many lines sprouting out of a root. Attached are implementations of BFS and DFS in Python using adjacency lists. Kruskal's algorithm is another graph algorithm which finds a a minimum spanning tree. The main difference between Kruskal's algorithm and DFS/BFS is the fact that BFS and DFS stay connected throughout their steps, while Kruskal's algorithm simply picks the edge of minimum weight in a graph that doesn't create a cycle. All in all, I feel like my knowledge of graphs is in the category of Mastery with Distinction due to the fact that I was able to implement graphs and their algorithms in multiple ways. I was also able to study graphs in different ways, allowing me to understand of graphs from multiple perspectives.

Algorithm 8.1 (Breadth-First Search)

1. Let $S = \{v\}$ [v is the root]; $T = \emptyset$ [initially there are no edges in the tree].
2. $C = N(v)$ [C is the current set of vertices being processed].
3. $l(v) = 0$ [label root as vertex 0]; $p(v) = v$; $b^* = v$ [keeps track of current vertex we are branching from]; $i = 1$ [initializes variable i used to help label the vertices]; remove b^* from all adjacency lists.
4. For each $w \in N(b^*)$, place w in S and place edge $b{*}w \in T$; assign successive labels $l(w) = i$ and $p(w) = p(b^*), w$ to vertices of C. Add one to i after each vertex is labeled; remove w from all adjacency lists [this ensures that a vertex w gets labels $l(w)$ and $p(w)$ just once].
5. Define a new b^* to be the vertex x in C such that $l(x)$ is minimum; remove b^* from C, and return to step 4. If, however, C is empty, stop. If every vertex of G has been labeled, a spanning tree has been found. If not, then G is disconnected, but a spanning tree of the component containing the root has been found.

```
#Breadth First Search
def BFS(self,v):
    if v in self.Graph:
        #step 1
        graph = copy.deepcopy(self.Graph)       #create copy of self.Graph
        S = [v]                              #list S
        label = []                            #list to hold labels, index refers to label
        T = []                                 #empty list of edges
        #step 2
        C = graph[v]                #list of v's neighbors
        C.sort()
        #step 3
        label.append(v)  #label v as 0
        bstar = v        #bstar = vertex we're branching to

        for item in graph:
            if bstar in graph[item]:           #remove bstar from adjacency lists
                graph[item].remove(bstar)

        #step 4
        while(True):
            neighbors = graph[bstar]        #list of adjacent vertices

            for vert in label:
                if vert in neighbors:       #neighbors = adjacent vertices that haven't been visited
                    neighbors.remove(vert)

            for item in neighbors:          #iterate through neighbors
                S.append(item)          #add vertex to S
                S.sort()
                T.append((bstar,item))  #add edge to T
                label.append(item)      #label item

            graph2 = copy.deepcopy(graph)       #create copy of graph bc can't edit something you're iterating

            for thing in C:
                for item in graph:
                    if thing in graph2[item]:       #remove bstar from adjacency lists
                        graph2[item].remove(thing)


            graph = graph2           #update graph

            #step 5
            if (len(C) == 0):           #halting condition
                return T

            else:
                bstar = C[0]            # define a new bstar to be min vertex in C
                C.remove(bstar)
```

Algorithm 8.2 (Depth-First Search)

1. Let $S = \{v\}$ [v is the root], $T = \emptyset$ [initially there are no edges in the tree]; $b^* = v$ [our initial branch vertex]; $p^* = v$ $l(v) = 0$ [label root as vertex 0]; $i = 1$ [initializes variable i used to help label the vertices]; $U = V(G) - \{v\}$ [keeps track of unlabeled vertices].
2. While $N(b^*) \cap U \neq \emptyset$ [b^* has unlabeled neighbors] do
   begin
   label the next unlabeled neighbor w of b^* by i; place b^*w in T; remove w from U; let $p^* = b^*$ [helps in backtracking]; $b^* = w$ [new branch vertex]; $i = i + 1$ [increment i for future labeling].
   end.
3. $b^* = p^*$[backtrack].
4. If $b^* = v$ and $N(b^*) \cap U = \emptyset$ [v has no unlabeled neighbors], stop. A spanning tree for the component containing the root v has been found. Otherwise, repeat step 2. If $U = \emptyset$, then the tree found is a spanning tree of all of G. If $U \neq \emptyset$, then G is disconnected.

```
#Depth First Search
def DFS(self,v):
    if v in self.Graph:     #check that vertex v is in the graph
        #step 1
        graph = copy.deepcopy(self.Graph)       #create copy of self.Graph
        T = []                                  #empty list of edges
        bstar = v                               #initial branch vertex
        pstar = v
        history = [v]                           #history of vertices that are used as bstar
        label = []                              #list to hold labels
        label.append(v)                         #label v as 0
        U1 = graph.keys()                       #keys of dict
        U = []                                  #list of unlabeled vertices

        for item in U1:
            U.append(item)
        U.remove(v)         #remove v since it is labeled
        U.sort()

        neighbors = graph[v]        #list of vertices adjacent to v
        neighbors.sort()
        intersection = list(set(U) & set(neighbors))        #U intersect neighbors
        intersection.sort()                                 #finds unlabeled niehgbors

        #step 2
        while (True):
            while (intersection):
                w = intersection[0]
                label.append(w)         #label neighbor of bstar
                T.append((bstar,w))     #add edge to T
                U.remove(w)             #remove labeled vertex from unlabeled list
                pstar = bstar           #help for backtracking
                bstar = w
                history.append(bstar)       #update history list
                neighbors = graph[bstar]    #get neighbors
                neighbors.sort()
                intersection = list(set(U) & set(neighbors))    #get intersect of U and neighbors
                intersection.sort()

            neighbors = graph[bstar]            #update neighbors
            neighbors.sort()
            intersection = list(set(U) & set(neighbors))     #used to check for halting condition
            intersection.sort()

            if (len(intersection) == 0):    #if U intersect neighbors = []
                num = history.index(bstar)
                bstar = history[num-1]      #bstar = pstar

            if (bstar == v and intersection == [] and len(U) == 0): #halting condition
                return T        #return spanning tree

            if (intersection != []):
                history.append(bstar)       #update history list
```

```
#Taylor Heilman
#an undirected simple graph
#Feb 11, 2016

import copy


class Graph(object):

    def __init__(self):
        #Graph constructor
        self.Graph = {}


    def add_vertices(self, vertices):
        #Add a list of vertices to the graph
        length = len(vertices)
        for i in range (length):
            if vertices[i] not in self.Graph:   #check if vertex exists in dictionary alr
eady
                self.Graph[vertices[i]] = []


    def delete_vertex(self, v):
        #Delete a vertex from the graph.
        if v in self.Graph:
            del self.Graph[v]           #delete key from dictionary
            for item in self.Graph:
                if v in self.Graph[item]:       #delete vertex from other keys' values
                    self.Graph[item].remove(v)




    def contract_edge(self, e):
        vertex1=e[0]        #first element of edge
        vertex2=e[1]        #second element of edge

        if vertex1 < vertex2:
            for item in (self.Graph[vertex2]):
                if item not in self.Graph[vertex1] and item != vertex1:
                    self.Graph[vertex1].append(item)    #copy vertex2's edges into vertex
1's

            for item in self.Graph:
                if vertex2 in self.Graph[item]:
                    self.Graph[item].remove(vertex2)    #remove vertex2 from other keys'
values

            del (self.Graph[vertex2])   #delete vertex2 from dictionary


        else:
            for item in (self.Graph[vertex1]):
                if item not in self.Graph[vertex2] and item != vertex2:
                    self.Graph[vertex2].append(item) #copy vertex1's edges into vertex2's

            for item in self.Graph:
                if vertex1 in self.Graph[item]:
                    self.Graph[item].remove(vertex1) #remove vertex1 from other keys' val
ues

            del (self.Graph[vertex1])  #delete vertex1 from dictionary



    def delete_edge(self, e):
        vertex1=e[0]        #first element of edge
        vertex2=e[1]        #second element of edge
```

```
        if vertex1 in self.Graph and vertex2 in self.Graph:
            if vertex1 in self.Graph[vertex2]:          #check if element1 is in element2
's values
                    self.Graph[vertex2].remove(vertex1)     #remove element1 from values
if true
            if vertex2 in self.Graph[vertex1]:          #check if element2 is in element1
's values
                    self.Graph[vertex1].remove(vertex2) #remove element2 from values if t
rue


    def vertices(self):
        #Return a list of nodes in the graph.
        return list(self.Graph.keys())

    def add_edges(self, edges):
        #Add a list of edges to the graph
        edges = list(edges)
        length1 = len(edges)
        for i in range (length1):
            temp = edges[i]
            first = temp[0]     #first vertex of pair
            second = temp[1]    #second vertex of pair

            if first not in self.Graph:         #Vertex1 is not in dictionary
                self.Graph[first] = [second]
            else:
                if second not in self.Graph[first]:     #Vertex1 in dictionary but Vertex
 2 isn't
                    self.Graph[first].append(second)

            if second not in self.Graph:        #Vertex2 is not in dictionary
                self.Graph[second] = [first]

            else:
                if first not in self.Graph[second]:     #Vertex2 is in dictionary but Ver
tex1 isn't
                    self.Graph[second].append(first)



    def edges(self):
        #Return a list of edges in the graph
        edge = []
        for vertex1 in self.Graph:
            for vertex2 in self.Graph[vertex1]:
                if (vertex2, vertex1) not in edge:      #check if inverse of edge is alre
ady in the list
                    edge.append((vertex1, vertex2))     #ex. if (u,v) is in list (v,u) wo
n't be appended


        return edge


    #Breadth First Search
    def BFS(self,v):
        if v in self.Graph:
            #step 1
            graph = copy.deepcopy(self.Graph)       #create copy of self.Graph
            S = [v]                         #list S
            label = []                       #list to hold labels, index refers to label
            T = []                            #empty list of edges
            #step 2
            C = graph[v]            #list of v's neighbors
            C.sort()
            #step 3
            label.append(v)  #label v as 0
            bstar = v        #bstar = vertex we're branching to
```

```
            for item in graph:
                if bstar in graph[item]:          #remove bstar from adjacency lists
                    graph[item].remove(bstar)

            #step 4
            while(True):
                neighbors = graph[bstar]        #list of adjacent vertices

                for vert in label:
                    if vert in neighbors:       #neighbors = adjacent vertices that haven
't been visited
                        neighbors.remove(vert)

                for item in neighbors:          #iterate through neighbors
                    S.append(item)          #add vertex to S
                    S.sort()
                    T.append((bstar,item))  #add edge to T
                    label.append(item)      #label item

                graph2 = copy.deepcopy(graph)       #create copy of graph bc can't edit s
omething you're iterating

                for thing in C:
                    for item in graph:
                        if thing in graph2[item]:       #remove bstar from adjacency list
s
                            graph2[item].remove(thing)


                graph = graph2          #update graph

                #step 5
                if (len(C) == 0):           #halting condition
                    return T

                else:
                    bstar = C[0]            # define a new bstar to be min vertex in C
                    C.remove(bstar)




    #Depth First Search
    def DFS(self,v):
        if v in self.Graph:     #check that vertex v is in the graph
            #step 1
            graph = copy.deepcopy(self.Graph)       #create copy of self.Graph
            T = []                                  #empty list of edges
            bstar = v                               #initial branch vertex
            pstar = v
            history = [v]                           #history of vertices that are used as
 bstar
            label = []                              #list to hold labels
            label.append(v)                         #label v as 0
            U1 = graph.keys()                       #keys of dict
            U = []                                  #list of unlabeled vertices

            for item in U1:
                U.append(item)
            U.remove(v)         #remove v since it is labeled
            U.sort()

            neighbors = graph[v]        #list of vertices adjacent to v
            neighbors.sort()
            intersection = list(set(U) & set(neighbors))        #U intersect neighbors
            intersection.sort()                                 #finds unlabeled niehgbor
s

            #step 2
```

```
            while (True):
                while (intersection):
                    w = intersection[0]
                    label.append(w)          #label neighbor of bstar
                    T.append((bstar,w))      #add edge to T
                    U.remove(w)              #remove labeled vertex from unlabeled list
                    pstar = bstar            #help for backtracking
                    bstar = w
                    history.append(bstar)       #update history list
                    neighbors = graph[bstar]    #get neighbors
                    neighbors.sort()
                    intersection = list(set(U) & set(neighbors))    #get intersect of U a
nd neighbors
                    intersection.sort()

                neighbors = graph[bstar]            #update neighbors
                neighbors.sort()
                intersection = list(set(U) & set(neighbors))     #used to check for halti
ng condition
                intersection.sort()

                if (len(intersection) == 0):    #if U intersect neighbors = []
                    num = history.index(bstar)
                    bstar = history[num-1]      #bstar = pstar

                if (bstar == v and intersection == [] and len(U) == 0): #halting conditio
n
                    return T        #return spanning tree

                if (intersection != []):
                    history.append(bstar)       #update history list




    def MaxDegC(self):
        order = list()      #list of vertices with degrees
        sort = list()       #sorted list of veritces based off of degrees
        colors = {}         #dictionary to keep track of color labels
        for index in range (len(self.Graph)):       #creates dictionary of colors for wor
st case scenario
            colors[index+1] = []                    #worst case = (every vertex has own c
olor)

        for vertex in self.Graph:
            order.append((len(self.Graph[vertex]), vertex))     #make list with (degree,
vertex)

        order.sort()        #sort in ascending order
        order.reverse()     #reverse order

        for index in range (len(order)):
            sort.append(order[index][1])        #remove degrees from list

        colors[1] = [sort[0]]             #color first vertex 1
        x = sort[0]
        sort.remove(x)                   #remove vertex


        while len(sort) > 0:                #while not all vertices are labeled
            j=1                              #set color = 1
            found = False
            w = sort[0]                         #set w = vertex with max degree
            while found == False:
```

```
                for label in colors:
                    if found == True:           #if a vertex can be labeled stop the for
loop
                        break
                    spot = 0        #keep track of # of neighbors checked
                    for neighbor in self.Graph[w]:
                        spot = spot+1  #checked one more neighbor
                        if neighbor in colors[label]:   #a neighbor vertex is already lab
eled with a num = label
                            j = 0
                            j = label+1                         #set j = one more than label
of neighboring vertex

                        elif spot == len(self.Graph[w]):        #all neighbors have been
checked
                            found = True
                            colors[j].append(w)     #label w with color j
                            x = sort[0]
                            sort.remove(x)      #remove from U




        ret = list()
        for i in colors:
            if (len(colors[i]) != 0):       #don't print out empy lists
                ret.append(colors[i])

        return ret



    def SeqC(self):
        colors = {}                                 #dictionary to hold colors of vertices
        for index in range (len(self.Graph)):       #creates dictionary of colors for wor
st case scenario
            colors[index+1] = []                        #worst case = every vertex gets i
ts own color


        vertices = list()
        for vert in self.Graph:             #list of vertices in self.Graph
            vertices.append(vert)

        colors[1] = [vertices[0]]       #label first vertex 1

        labeled = list()                    #list of labeled vertices
        labeled.append(vertices[0])


        for vertex in self.Graph:           #iterate though all vertices in graph
            if vertex not in labeled:
                neighbors = self.Graph[vertex]      #list of adjacent vertices
                a = neighbors
                b = labeled                         #list of labeled vertices

                important = list(set(a) & set(b))   #list of labeled adjacent vertices

                if(len(important) == 0):        #no labeled adjacent vertices
                    colors[1].append(vertex)    #label vertex with 1

                else:
                    found = False
                    for label in colors:
                        if found == True:           #if a vertex can be labeled stop the
for loop
                            break
                        spot = 0        #keep track of # of neighbors checked
                        for neighbor in important:
                            spot = spot+1           #checked one more neighbor
```

```
                            if neighbor in colors[label]:   #a neighbor vertex is already
 labeled with a num = label
                                j = 0
                                j = label+1                     #set j = one more than la
bel of neighboring vertex

                            elif spot == len(important):        #all neighbors have been
checked
                                found = True
                                colors[j].append(vertex)     #label w with color j




        ret = list()
        for i in colors:
            if (len(colors[i]) != 0):       #don't print out empy lists
                ret.append(colors[i])

        return ret


    def isTree(self):       #modified DFS to check for cycles since we assume G is connec
ted
        keys = self.vertices()
        v = keys[0]             #arbitrarily pick a vertex for v
        #step 1
        graph = copy.deepcopy(self.Graph)       #create copy of self.Graph
        S = [v]               #list
        T = []                #empty list of edges
        history = [v]         #history of vertices that are used as bstar
        bstar = v
        pstar = v
        label = []            #list to hold labels
        label.append(v)      #label v as 0
        U1 = graph.keys()    #keys of dict
        U = []               #list of unlabeled vertices
        for item in U1:
            U.append(item)
        U.remove(v)          #remove v since it is labeled
        U.sort()
        neighbors = graph[v]        #list of neighbors of v
        neighbors.sort()
        intersection = list(set(U) & set(neighbors))        #U intersect neighbors
        intersection.sort()
        #step 2
        while (True):
            while (intersection):
                w = intersection[0]
                label.append(w)          #label neighbor of bstar
                T.append((bstar,w))      #add edge to T
                U.remove(w)              #remove labeled vertex from unlabeled list
                pstar = bstar               #help for backtracking
                bstar = w
                history.append(bstar)       #update history list
                neighbors = graph[bstar]    #get niehgbors
                neighbors.sort()
                intersection = list(set(U) & set(neighbors)) #get intersect of U and neig
hbors
                intersection.sort()

        #MAIN EDIT
                #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                cycle = list(set(history) & set(neighbors)) #take the intersection of the
 bstar's neighbors and history(visited vertices)
                if (len(cycle) >= 2):                      # IF 2 OR MORE NEIGHBORS OF CURR
ENT VERTEX HAVE BEEN VISITED
                    return (False)              #A CYCLE HAS BEEN FOUND, RETURN FALSE
                #~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
            neighbors = graph[bstar]
            neighbors.sort()
            intersection = list(set(U) & set(neighbors))     #used to check for halting c
ondition
            intersection.sort()


            if (len(intersection) == 0):    #if U intersect neighbors = []
                num = history.index(bstar)
                bstar = history[num-1]      #bstar = pstar

            if (bstar == v and intersection == [] and len(U) == 0): #halting condition
                return True        #IF DFS COMPLETES NORMALLY, RETURN TRUE

            if (intersection != []):
                history.append(bstar)       #update history list






    def Center(self):
        if (self.isTree()):     #isTree returned True
            graph = copy.deepcopy(self.Graph)   #make a copy to delete vertices without c
hanging real graph
            leaves = []             #make a list of leaves found
            x = True
            while (x == True):
                for vertex in graph:       #iterate through vertices in graph
                    if (len(graph[vertex]) == 1):       #leaf found (has degree = 1)
                        leaves.append(vertex)           #get a list of leaves to delete


                for i in range (len(leaves)):   #iterate through leaves
                    del graph[leaves[i]]               #delete leaf from dict
                    for item in graph:
                        if leaves[i] in graph[item]:       #delete leaf from other keys'
values
                                graph[item].remove(leaves[i])


                leaves = []             #reset list for next iteration

                if (len(graph) <= 2):  #check if center/ centers have been found
                    for item in graph:
                        leaves.append(item) #reuse leaves list to return center/centers
                    return(leaves)

        else:
            return('Graph is not a Tree')           #isTree returned False



def main():
    #call functions from here
    G = Graph()


main()
```

```
// 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"
```

```
// 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;
}


/*-----------------------------------------------
 * 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;
        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;
}


/*=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
                    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;
}
```