**Programmer Manual**

**Class Graph for Network**

1. **Problem Description**

A graph consists of a vector of data type vertex that is templated. Its purposes is to allow for a graph construction or deconstruction of any size.

1. **Class employee**

The Graph class is defined using the stl database, allowing the use of a stack, queue, deque, and vector ADT as components, or components of functions, to create a graphical representation of data. Graph is templated, allowing for flexibility of the type of vertices used to comprise.

For example, in the user’s program

Graph<V, W> graph; //declares a Graph object called graph of type Graph.

Public data members:

None

Private data members:

Vector<vertex<V, W>> G a vector array graph object named G composed of vertices of type V, and W (templated to accommodate any data types)

Public Member Functions

Graph() – this is the default constructor require by the compiler.

~Graph() – the default destructor required by the compiler.

int isVertex(V& v) – this tests whether v is a vertex in the graph and returns the index

int isUniEdge(V& v1, V& v2) – this Tests whether edge <v1,v2> in graph

int isBiDirEdge(V& v1, V& v2) – this Tests whether edge (v1,v2) in graph

int AddVertex(V& v) – this function Adds vertex with name v to the graph, if v not already in graph, and returns the index where the vertex stored.

int DeleteVertex(V& v) – this function Deletes vertex with name v from the graph, if v is in the graph. If there are any edges incident on the, these edges are deleted also.

int AddUniEdge(V& v1, V& v2, W& wt) – this function Adds the directed edge <v1,v2,wt> to the graph. adds the vertices to the graph if the vertices are already part of the graph

int DeleteUniEdge(V& v1, V& v2) – this function Deletes the directed edge <v1,v2> (any weight) from the graph, if it is in the graph. The vertices are not deleted from the graph, only the edge.

int AddBiDirEdge(V& v1, V& v2, W& wt) – this function Adds the bi-directional edge v1,v2,wt) to the graph; adds the vertices to the graph if the vertices are not already part of the graph

int DeleteBiDirEdge(V& v1, V& v2) – this function Deletes the bi-directional edge (v1,v2) (any weight) from the graph, if it is in the graph. The vertices are deleted from the graph, only the edge.

void SimplePrintGraph() – this function Prints the list of vertices in the graph, and for vertex, prints the list of edges in proper parenthesized notation, namely (v1,v2,wt) or <v1,v2,wt>

void ShortestDistance(V& v) – this function Determines the shortest paths to all other vertices from the specified vertex.

void mst() – this function Determines the minimum spanning tree using Prim's algorithm.

void BFTraversal(V& v) – this function Performs Breadth First Traversal of the graph starting at specified vertex (parameter). prints trace information.

void DFTraversal(V& v) – this function Performs (non-recursive) Depth First Traversal of the graph starting at specified vertex (parameter); prints trace information.

void DepthFirstTraversal(V& v) – this function is an auxiliary to the recursive DFT.

void DFT(V& v) – this function Performs a recursive Depth First Traversal of the graph starting at the specified vertex (parameter); prints trace information.

void GetGraph() – this function Retrieves a graph from a special file and sets up the adjacency list for the graph

1. **High Level Program Solution**

Graph() constructor

Default constructor for the compiler

~Graph() destructor

Default destructor for the compiler

isVertex(V& v)

IN: V& v – vertex to be checked for existence in graph.

FOR size of graph

If vertex is in the graph

Return success

Else return fail

isUniEdge(V& v1, V& v2)

IN: V& v1, V& v2 – vertices to be used as source and destination location.

If start or end vertex is not in the graph

Return fail

Else

FOR size of starting vertex edgelist

If end vertex is in the starting vertex edgelist

Return success

Else return fail

isBiDirEdge(V& v1, V& v2)

IN: V& v1, V& v2 – vertices to be used as source and destination location.

If start to end edge is in the graph and end to start edge is in the graph

Return success

Else return fail

AddVertex(V& v)

IN: V& v – vertex to be added to graph

Declare required data type variables

If vertex is not in the graph

Push vertex

Return index if not in the graph

Else return index

DeleteVertex(V& v)

IN: V& v – vertex to be deleted from graph

If vertex is in the graph

Erase vertex

Return success

Else return fail

AddUniEdge(V& v1, V& v2, W& wt)

IN: V& v1, V& v2, W& wt – vertices to be used as source and destination location, and the weight of its edge.

Declare required data type variables

Add vertex start and end

If edge is not in graph

Add edge to starting vertex’s edgelist

Return success

Else return fail

DeleteUniEdge(V& v1, V& v2)

IN: V& v1, V& v2 – vertices to be used as source and destination location.

If UniEdge exists from start to end

FOR length of start’s edgelist

If edgelist vertex name is equal to end vertex

Erase that edge

Return success

Else return fail

AddBiDirEdge(V& v1, V& v2, W& wt)

IN: V& v1, V& v2, W& wt – vertices to be used as source and destination location, and the weight of its edge.

If a UniEdge exists from either start to end, or end to start

Return fail

Add UniEdge from start to end

Add UniEdge from end to start

Return success

DeleteBiDirEdge(V& v1, V& v2)

IN: V& v1, V& v2 – vertices to be used as source and destination location.

If input vertices form a bidirectional edge

Delete UniEdge from each direction

Return success

Else return fail

SimplePrintGraph()

IN: None

FOR size of graph

FOR length of edgelist

Display the output message

ShortestDistance(V& v)

IN: V& v – vertex to be used as a source location.

Declare required data type variables. Initialize each node to register as not visited, each predecessor to the default name value, each cost distance to infinity minus one.

Set the cost distance starting vertex to zero.

Push to the front of deque, mark as visited, set variable to infinity for comparison

WHILE the queue is not empty

Set temp variable to front of queue, pop front, mark as visited

FOR all edges of vertex

If index of the graph is unvisited and the cost to that edge is less than infinity

Update cost to that specified edge

Update predecessor

IF graph index is not visited

Push back to queue

Print trace information after each iteration

FOR size of graph

IF vertex name does not equal input vertex

IF cost of vertex is not equal to infinity minus 1

Display output message, and initialize required variables

WHILE variable does not equal index of user inputted vertex

IF predecessor is not equal to the initialized value

Push front, update variable

ELSE update variable

Display message

WHILE queue is not empty

Set variable to the front of queue, pop front, display message

IF queue is not empty

Set variable to front of queue, output variable3

ELSE display index vertex name

ELSE display no path available

mst( :

IN: none

Declare required data type variables. Initialize each component and neighbor node to zero, each cost distance to infinity minus one.

FOR edgelist of vertex start, update cost;

Set component source to 1, set cost source to 0

FOR size of graph

Set variable to infinity

FOR size of graph;

IF component is 0

IF cost is less than variable

Update index variable

Update weight variable

IF weight is less than infinity

Set components index to 1

Add bidirectional edge to MST

FOR all edges of vertex

IF component of edge index is 0

IF edgeweight less than cost to edge

Update cost, update neighbor

ELSE break

Display distance, neighbor, and component array after each iteration

Update weight of MST

Display weight and display tree

GetGraph()

IN: None

Declare various data-types

Prompt the user to enter the name of the input file. If an incorrect file path is entered, a message saying, “Unable to open file…quitting.” appears.

Reads in vertex and edge data from input file, while the input file is valid.

Close the input file.

BFTraversal(V& v)

IN: V& v – vertex to be used as starting location.

Declare required data type variables. Initialize each node to register as not visited, then push input value vertex index to a queue.

While the vertex is not visited:

If value of visited = 0:

Set the value of visited to 1. Then, pop vertex. Push all edges of the vertex onto the queue

Else, pop

If any vertex has not been visited:

Push vertex to the queue

END WHILE

DFTraversal(V& v)

IN: V& v – vertex to be used as starting location.

Declare required data type variables. Initialize each node to register as not visited, then push input value vertex index to a stack.

While the vertex is not visited:

If value of visited = 0:

Set the value of visited to 1. Then, pop vertex. Push all edges of the vertex onto the stack.

Else, pop

If any vertex has not been visited:

Push vertex to the stack

END WHILE

DepthFirstTraversal(V& v)

IN: V& v – vertex to be used as a starting location.

Declare required data type variables

Set index of graph to visited

Display user input

FOR all edges of graph index;

Set variable to the index of edge

IF graph index visited is equal to 0

Call DepthFirstTraversal recursively, passing in current edge vertex

DFT(V& v)

IN: V& v – vertex to be used as a starting location.

FOR size of graph

Set visited of each index of graph to 0

Call DepthFirstTraversal, passing in user input

Display message

FOR size of graph

IF visited of each index of graph equals 0

Display graph index vertex name

Break

1. **Limitations and Suggestions**

The limitations of this class are there is no output file to maintain an updated list of the graph, after modifications are made to the original file. In addition, the specification of the user input file is considerable. Without a properly formatted input file, we are unable to use an option from the menu.