Graphs

Overview

A graph is a collection of nodes that are connected by edges. Although trees and linked lists can be considered as graphs, in those data structures we are more interested in the objects in the links or nodes than in the connections between them. When we are looking at graphs, we are typically interested more in the interconnections than in the nodes.

Examples of graphs in the real world are the highways between cities, airline flight connections, or the fiber optic networks that we use for the internet.

Nomenclature

As stated above, a graph is a collection of ***nodes*** and the ***edges*** that connect them. A ***path*** between two nodes is a set of edges that begins at one and ends at the other. The ***degree*** of a graph is the maximum number of edges connected to a single node.

A ***connected*** graph is one where any node can be reached from the starting point. A connected graph of ***N*** nodes must have at least ***N-1*** edges.

A fully connected graph has edges between all nodes and has a total of ***N + N-1 + N-2 + … + 1 or N * (N-1) / 2*** total edges. Considering a graph with ***N*** nodes, adding one more node requires ***N*** new connections, thus proving the induction argument. The base case is a graph with one node and zero edges.

A ***minimum spanning tree*** is a subset of the edges that maintains the graphs connectivity, but has the minimal number of edges remaining after pruning. Look in the document on ***Graph Traversals*** for more information. A minimum spanning tree for a connected tree of ***N*** nodes would have ***N-1*** edges.

Graphs can be ***non-directed*** or ***directed***. Directed graphs have edges that go from one node to a second, but not necessarily ones in return. A singly linked list or a set of one-way streets is an example of a directed graph. When a non-directed graph has an edge between two nodes, it can be used for traversal in either direction. If a non-directed graph has at least ***N*** edges, then there must be at least one ***cycle*** in it (there are at least two ways to get from some node to some other node).

Graphs can also be ***weighted*** or ***un-weighted***. A weighted graph has values for the edges representing some characteristic, such as miles between cities on a map. An un-weighted graph has edges that represent connectivity, such as the edges between nodes in a tree.

Representation

Nodes can be a structure and are usually stored in an array or list. There is an identifier, a possible pointer to a list of edges (see below), and a Boolean flag ***visited*** that is starts out false and is set to true by different graph algorithms.

There are two ways that edges are represented in a program.

The first way to represent the edges in a graph is by using a ***connectivity list***. In a connectivity list, the edges are structures containing a weight (for a weighted graph), the identifier for the end of the edge, and a pointer to the next edge in the list. For a non-directed graph, there will be an edge in the list for the start and an edge in the list for the end. For a directed graph, each edge is present only once.

The second way to represent the edges is by using an ***NxN connectivity array*** where a connection between node A and node B is represented by a 1 (for a non-weighted graph) or a non-zero weight (for a weighted graph) at the location array[A][B]. For a non-directed graph, there is symmetry around the diagonal where the value at array[A][B] == array[B][A].

Example Code

Here is example pseudo code for a simple non-directed, non-weighted graph:

```
// when using an edgelist, you have a set of links
class Edge
    endIndex // the index in the Node array of the other end of the edge
    weight   // for a weighted graph, the weight of this edge
    next     // a reference or pointer to the next edge in the list

// the object used to represent a node
class Node
    name     // the name of this node
    visited  // boolean indicating it has been visited for algorithmic use
    Edges    // list of the edges starting at this node

// the graph class itself
// this one has both edgelists and a connection matrix
class Graph
    public methods
        addNode(name)
        addEdge(starting name, ending name)

        listNodes()
        displayEdges()
        displayMatrix()
    private methods and variables
        numNodes        // how many nodes are in the list
        nodeList[SIZE]  // the list of nodes
        edgeMatrix[SIZE][SIZE]

        findNode(name)  // returns index of node in list given its name
        resetFalse()    // used to reset all visted flags to false


Constructor
        // initialize number of nodes in list
    numNodes = 0

        // set up edge Matrix to start with no edges
    for(int i = 0; i < SIZE; i++)
        for(int j = 0; j < SIZE; j++)
            edgeMatrix[i][j] = false

Destructor // for C++
    // delete all connections from each node in nodeList
    for(int i = 0; i < numNodes; i++)
            // similar to destructor on linked list
        Edge * ptr = nodeList[i]->connects;
        while(ptr != nullptr)
            Edge * temp = ptr
            ptr = ptr->next
            delete temp
```

```
    // add a new node to the graph
    // only failure is if graph arrays are full
addNode(name)
        // deal with adding too many nodes
    if( numNodes >= SIZE )
        throw exception or double size of node array and connection matrix

        // create a node with this name
        // initialize it with no edges and not yet visited
    temp = new Node
    temp.name = name
    temp.visited = false
    temp.connects = nullptr

        // add to the list of nodes in graph
    nodeList[numNodes++] = temp;


    // add a new edge to the graph
    // return false and do nothing if either end is invalid
    // otherwise add to both nodes edge lists and to the matrix
addEdge(startNode name, endNode name)
       // ignore edges from a node to itself
    if starts == ends
        return false

    startIndex = findNode(starts)
    endIndex = findNode(ends)

       // if either name is not in the node array
    if startIndex == -1 or endIndex == -1
        return false

        // set both links in edgeMatrix
    edgeMatrix[startIndex][endIndex] = true
    edgeMatrix[endIndex][startIndex] = true

        // create two new edges (one for each direction)
        // and add one to each nodes list of edges
        // Note that C++ needs to use pointers for this
    Edge startEnd = new Edge
    startEnd.endIndex = endIndex
    startEnd.next = nullptr
    startEnd.next = nodeList[startIndex].connects
    nodeList[startIndex]->connects = startEnd

    Edge endStart = new Edge
    endStart.endIndex = startIndex
    endStart.next = nullptr
    endStart.next = nodeList[endIndex].connects
    nodeList[endIndex]->connects = endStart

    return true

    // linear search for a node with this name
    // return -1 if not found else return its nodeList index
findNode(name)
    for (int i = 0; i < numNodes; i++)
        if nodeList[i]->name == name
            return i
    return -1
```

```
    // listing of nodes in the order
    // they were added to graph
listNodes()
    string theList = ""
    for (int i = 0; i < numNodes; i++)
        theList += nodeList[i]->name
        theList += " "

    return theList

    // display the edgelist
    // for each node in graph, display its edges
displayEdges()
    string buffer
    for(int i = 0; i < numNodes; i++)
            // add the node name to the display
        buffer += nodeList[i]->name
        buffer += "-"

            // walk down its list of edges and add them also
        ptr = nodeList[i].connects
        while ptr != nullptr
            buffer += nodeList[ptr.endIndex].name
            buffer += " "
            ptr = ptr.next

            // end this line of output
        buffer += "\n"

    return buffer


    // display the adjacency matrix
    // as 0 for no connection and 1 for connection
displayMatrix()

        // output header line of destinations
    for(int i = 0; i < numNodes; i++)
        output nodeList[i].name

        // now output the array
    for(int i = 0; i < numNodes; i++)
            // add the starting node
        output nodeList[i]->name

            // now add its connections
        for(int j = 0; j < numNodes; j++)
            output edgeMatrix[i][j]


    // this resets the graph to no nodes visited after a traversal
resetFalse()
    for(int i = 0; i < numNodes; i++)
        nodeList[i].visited = false
```