Graphs

A Graph $G = (V, E)$ is a mathematical object consisting of a finite set of vertices $V$ (sometimes called nodes) and a set of edges $E$ connecting pairs of vertices. We usually draw them as a series of dots connected by lines, but their orientation in space does not matter.

Graphs model real-world scenarios. For example, in the example above, each node might represent a computer on a network, and each edge might denote a connection between two computers. Likewise the nodes could be cities and the edges could be path. We could ask questions like: Is vertex $a$ reachable from $b$, or what is the minimum distance from $a$ to $b$?

A more abstract example is that the dots may represent parts for a computer, and the edges might represent the relationship ’is compatible with’. We might ask, what is the maximum number of parts that can be used at once?

(Very difficult problem! Best algorithm takes $O(2^{|V|})$ time. If you can do better, or show this is impossible, you would become very rich. But don’t count on it.)
Special graphs

One special type of graph is a *Labeled graph*, which has a unique label on each node, like this:

Another special type of graph is a *Directed graph*, where edges can be unidirectional or bidirectional, like this:

We can also have weighted graphs, which assign a numerical weight value (i.e. cost, profit) to each edge, like this:

Or we can have some combination of these types. Again, many practical problems can be modeled in this way. In a later class we will consider data structures for graphs.
More examples

Directed, Weighted, labeled graph - A flowchart, an automaton (node=machine states), flow network (ie water, bandwidth), A network of airline flights, a power grid, a circuit diagram.

More definitions

Path - A sequence of edges that form an unbroken line between two vertices.

deg(v) - (Degree) number of edges adjacent to v.

cycle - A path that starts and ends on the same vertex

connected component - A maximal set of vertices such that there is a path between any two vertices in the set

connected graph - A graph that contains one connected component
We can prove many relevant properties about graphs, for example:

The shortest path between two nodes contains at most $n - 1$ edges.

The sum of the degrees of the vertices of a graph is even

A graph contains no cycles iff for each pair of vertices $a$, $b$, there is at most one path from $a$ to $b$.

A connected graph that contains no cycles is called a tree. Trees make up an important class of graphs.
Trees

A very natural way of organizing things is to organize them in a hierarchy. Here are some real-world examples:

- Organizational chart for a business, government
- A Textbook (Sections, subsections, etc)
- An arithmetic expression
- Taxonomy/ Evolutionary tree

Many more examples can be seen if we consider computers.

- Org. of large scale software projects
- File directories/Databases
- Class hierarchy in Java
- The Internet (both in structure and with respect to Domain Names)
- Data structures: Search trees/heaps

So we need to be able to store information in a hierarchical way, and process it.
A *tree* is an ADT that stores elements hierarchically.

The points of the tree are called nodes (or sometimes vertices). We store data at each node. The nodes are connected in a parent-child relationship. The top node is called the root.

Each node may have any number of children. Nodes with no children are called *leaf* nodes. Nodes with at least one child are called *internal* nodes.

Every node except for the root has a parent.

We also define an *ancestor* or a *descendant* as you would expect.

We can prove properties of trees. For example:

- If there are *k* leaf nodes then there are at most *k* − 1 internal nodes.
- If there are *n* nodes then there are *n* − 1 edges.
Special trees

A tree is *ordered* if the children of each node are ordered.

A binary tree is an order tree with at most two children, which we denote the left child, and the right child. Binary trees are handy for storing items in sorted order.

The height of a binary tree is at least $\log n$ and at most $n$. 
Generic Graph ADT

There are two popular methods for representing graphs: Adjacency Matrices and Adjacency lists.

Adjacency Matrices

Fix an order on the vertices. The strategy is to maintain an $n \times n$ matrix of Boolean values. We set $A[i, j]$ to true if $v_i$ is connected to $v_j$, and 0 otherwise. Note that for simple graphs, $A$ will be symmetric.

Data:

boolean $A[][]$;

Methods:

proc addEdge(int i, int j);
begin
    if (i != j) then
    end;
end;

proc deleteEdge(int i, int j);
begin
    $A[i][j] = A[j][i] = 0$;
end;

func inspectEdge(int i, int j): returns boolean;
begin
    return $A[i][j]$;
end;
func numAdjacent(int i) returns int;
begin
    sum = 0;
    for j = 0 to A[].size-1 do
        if (A[i][j] = 1) then
            sum = sum + 1;
        end;
    end;
    return sum;
end;

// get jth vertex that is adjacent to i.
func getAdjacent(int i, int j) returns int;
    sum = 0;
    for k = 0 to A[].size-1 do
        if (A[i][k] = 1) then
            sum = sum + 1;
            if sum = j then
                return k;
            end;
        end;
    end;
    return -1;
end;

Analysis:

<table>
<thead>
<tr>
<th>Method</th>
<th>Big O</th>
</tr>
</thead>
<tbody>
<tr>
<td>addEdge()</td>
<td></td>
</tr>
<tr>
<td>deleteEdge()</td>
<td></td>
</tr>
<tr>
<td>inspectEdge()</td>
<td></td>
</tr>
<tr>
<td>numAdjacent()</td>
<td></td>
</tr>
<tr>
<td>getAdjacent()</td>
<td></td>
</tr>
</tbody>
</table>

Note that this ADT is easy to extend this to directed graphs (how?) or weighted graphs (how?) or both.
Adjacency lists

Strategy is to maintain, for each vertex, a list of adjacent vertices

Data:

int A[][];
int size[];

Methods

proc addEdge(int i, int j);
begin
    if (inspectEdge(i, j) == false) then
        A[i][size[i]] = j;
        A[j][size[j]] = i;
        size[i] = size[i] + 1;
        size[j] = size[j] + 1;
    end;
end;

proc deleteEdge(int i, int j);
begin
    for k = 0 to size[i] - 1
        if A[i][k] = j then
            A[i,k] = A[i][size[i] - 1]; // fill in blank
            size[i] = size[i] - 1;
            break;
        end;
    end;
    for k = 0 to size[j] - 1
        .... do the same for vertex j
end;
proc inspectEdge(int i, int j): returns boolean;
begin
    for k = 0 to size[i] - 1
        if A[i][k] = j then
            return true;
        end;
    end;
    return false;
end;

proc numAdjacent(int i): returns int;
begin
    return size[i];
end;

func getAdjacent(int i, int j): returns int;
begin
    return A[i][j];
end;

Analysis:

<table>
<thead>
<tr>
<th>Method</th>
<th>Big O</th>
</tr>
</thead>
<tbody>
<tr>
<td>addEdge()</td>
<td></td>
</tr>
<tr>
<td>deleteEdge()</td>
<td></td>
</tr>
<tr>
<td>inspectEdge()</td>
<td></td>
</tr>
<tr>
<td>numAdjacent()</td>
<td></td>
</tr>
<tr>
<td>getAdjacent()</td>
<td></td>
</tr>
</tbody>
</table>

So Adjacency lists are better when you need to performs traversals through the graph (As in: BFS, DFS, Minimum spanning tree, Shortest path) and you expect << n edges per vertex.

Adjacency matrices are better when you expect to perform many updates to the graph.
Generic Tree ADT

Of course, since trees are just special cases of graphs, we could use a Graph ADT to store a tree. However, because of the way that we use trees, it is convenient to have functions that perform ‘surgery' on the trees (for example, by swapping one subtree for another).

First we need a Node data type. It should contain the following data:

Object val;
private Node parent;
private Node child[]; // an array of pointers to children nodes
private int numchildren;

and the following methods:

Node(Object val, Node parent);
addChild(Node C);
removeChild(int i) returns Node;
child(int i): returns Node;
umChildren(): returns int;
getParent() returns Node;

Now our Tree Abstract Data Type

getroot(): returns Node
setRoot(Node R);
height();
depth();
// any other relevant tree operation
proc Node(Object val, Node parent)
begin
    this.val = val;
    this.parent = parent;
    numchildren = 0;
    child = {new array of 10 elements}
end;

proc addChild(Node C)
begin
    if numchildren +1> {size of child} then
        {resize child}
    end;
    child[numchildren] = C;
    C.parent = this
    numchildren = numchildren + 1;
end;

func removeChild(int i) returns Node;
begin
    Node N = child[i];
    numchildren = numchildren - 1;
    if numchildren > 0 then
        child[i]= child[numchildren];
    end;
    N.parent = null;
    return N;
end;

func child(int i): returns Node;
begin
    return child[i];
end;

the code for numChildren() and getParent() is similar...
Tree traversals

In different situations, we will want to traverse (ie inspect all of the nodes of the tree) in different ways. Three common idioms are: preorder traversal, inorder and postorder traversal. ‘Pre’, ‘in’ and ‘post’ refer to the time when the root of a subtree is inspected.

Preorder traversal:
-Inspect the root
-For each subtree, inspect the subtree
Order = 1,2,5,10,11,3,6,7,12,4,8,9,13,14

Inorder traversal: (for binary trees only)
-Inspect the left subtree
-Inspect the root
-Inspect the right subtree
Order = n-a

Postorder traversal:
-For each subtree, inspect the subtree
-Inspect the root
Order = 10,11,5,2,6,12,7,3,8,13,14,9,4,1
Example

\[ \begin{array}{ccccc}
+ & \times & \div \\
8 & 3 & 6 & 2 \\
\end{array} \]

Inorder: Could be used to write the expression in standard form:

\[(8 + 3) \times (6 / 2)\]

Preorder: Could be used to write the expression with functions in place of operators:

\[\text{Times(Add}(8,3),\text{Div}(6,2))\]

Postorder: Could be used to write code to evaluate this expression on a stack machine:

Push 8, Push 3, Add, Push 6, Push 2, Div, Times

Does anyone know of a popular modern stack machine?

More examples:

Preorder: Print a table of contents from a book organizational chart

Postorder: Compute disk space for a directory tree