Graphs

You are familiar with data structures such as arrays, lists, and trees. We next consider a more general data structure, known as graphs. Like the previous data structures, a graph is a set of nodes V, each node having references to other nodes. In the case of graphs, a reference from one node to another is called an edge. The set of edges is denoted E, and so the graph may be written as a pair $G = (V, E)$. In a (rooted) tree, we saw that the references were either to a child or parent node. In a general graph, there is no notion of child and parent. Every node can potentially reference every other node.

More formally, a graph consists of a set V (called "vertices") and a set $E \subseteq V \times V$, that is, a subset of all ordered pairs of vertices. In other courses (MATH 240 and 340 – Discrete Structures 1 and 2) you will work with these formal definitions. In this course, I will try to avoid these formalities and get to the basic algorithms.

Examples of graphs include transportation networks. For example, V might be a set of train stations and E might be a set of train tracks going directly from one train station to another, V might be a set of airports and E might be direct flights E. In computer system, V might be a set of computers and E might be a direct communication link between them. Or V might be a set of documents (including html) and E might be defined by URLs in the html documents.

Below is an example of two graphs. On the left, there are no arrows on the edges. When an edge is drawn without arrows, it means that there are arrows in both directions, so for each edge (v, w) there is also an edge (w, v). On the right, some of the edges have arrows. In that case, there is only one edge between them. You could, of course, use a different notation: pairs of opposite direction edges between nodes v, w could be represented with two edges with arrows, or one edge with arrows on each end. I use the notation shown here because I find it to be most simple.

Data structures

There are two very common data structures for graphs: adjacency matrices and adjacency lists. Because of time constraints, and because you should be able to provide the implementation yourself at thispoint, I will give you just the basic idea here.

Adjacency Matrix

An adjacency matrix is a $|V| \times |V|$ array of booleans, where $|V|$ is the number of elements in set V i.e. the number of vertices. The value 1 at entry $(v1, v2)$ in the array indicates that $(v1, v2) \in E$, and the value 0 indicates that $(v1, v2) \notin E$.

Examples for the above two graphs are shown below. Note that the adjacently matrix on the left is symmetric about the diagonal, i.e. it is identical to its transpose. Also note that the matrix on the right has fewer 1's in it. Obviously, the space required is $\Theta(\ |V|^2\ )$, since $|V|^2$ is the number of cells in each matrix.

```
   abcdefgh          abcdefgh
a  00110000       a  00100000
b  00001100       b  00000100
c  10010100       c  00000100
d  10100000       d  10100000
e  01000100       e  01000100
f  01101000       f  01001000
g  00000001       g  00000001
h  00000010       h  00000000
```

Suppose you wish to use one of the adjacency matrices to ask which edges a given vertex $v1$ is connected to, namely you want to know for which w's there is an edge $(v, w) \in E$. It takes time $\Theta(\ |\ V\ |\ )$ to determine this, namely you need to examine all elements in a row of the adjacency matrix and see which ones have the value 1.

Adjacency List

For many graphs, each vertex is connected to only a small number of other vertices. In this case,[1] The adjacency matrix will have mostly 0's in it. This in itself is a waste of space. It is also a waste of time to have to spend $\Theta(|V|)$ to know to which other vertices a given vertex v has an edge.

For graphs that have a relatively small number of edges, it is better to use an *adjacency list*. For each vertex v, a list of vertices w is stored such that $(v, w) \in E$. For example, here are the adjacency lists for the examples above.

```
a  -  (c,d)           a  -  (c)
b  -  (e,f)           b  -  (f)
c  -  (a,d,f)         c  -  (f)
d  -  (a,c)           d  -  (a,c)
e  -  (b,f)           e  -  (b,f)
f  -  (b,c,e)         f  -  (b,e)
g  -  (h)             g  -  (h)
h  -  (g)             h  -  ()
```

You can think of these as a map i.e. a set of (key,value) pairs. A vertex v is the key, and the list of vertices w such that $(v, w) \in E$ is the value.

The space required to store this map is $O(|V| + |E|)$. For many graphs, we have $|E| > |V|$ and in this case the space required would be $O(|E|)$. However, it can easily happen that $|V| \ll |E|$ and yet the graph is still very interesting. For example, consider the set of all files that are available on the worth wide web. While many of these (such as html files) have references to other files, many other files don't. e.g. this pdf file doesn't have a reference to any other file. It would be a vertex with an empty list of (outgoing) edges in the adjacency list.

[1] the graph is said to be *sparse*

Graph traversal

One problem that often needs to be solved is whether there is a sequence of edges (a path) from one vertex to another. More generally, we can ask for the set of all vertices that can be reached from a given vertex, that is, the set of all vertices w for which there is a path from v to w. By a *path* here, I mean a sequence of vertices $v_1 v_2 \dots v_k$ such that $(v_i, v_{i+1}) \in E$ for all $i = 1, \dots, k-1$.

Depth First Search

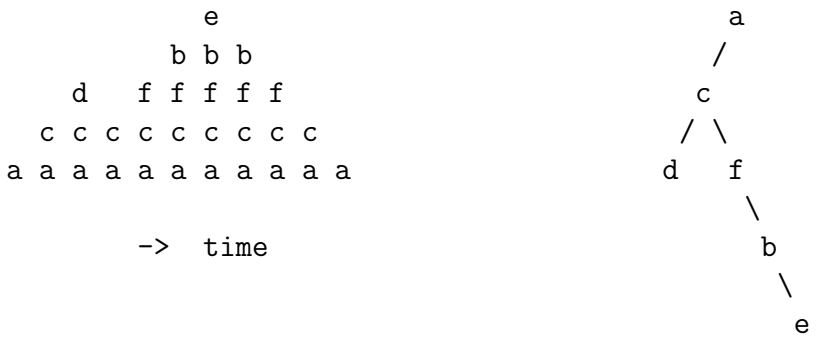
Our first solution is recursive. For any node, you visit a root first and then you visit its children. This is similar to preorder traversal of a tree, but there is a difference. You need to ensure that you don't visit a node more than once (which could send you into an infinite loop). So you need to have a boolean variable at each node, called `v.visited` which, is initialized to false for all nodes.

```
DepthFirstSearch(v){
   v.visited = true
   for each w such that (v,w) is in E
     if !w.visited
        DepthFirstSearch(w)
}
```

After running the algorithm, the nodes that have `v.visited = true` are the nodes that can be reached by a path from a given node.

Example

Take the graph shown earlier (on the left). Suppose we do depth first search in the graph starting at `a`. The order in which nodes are visited is `a, c, d, f, b, e`. The call stack (over time) and call tree are shown below. (Only the call tree was discussed in class.)



Verify for yourself that if you run the algorithm with the directed graph i.e. the graph which has some directed edges between nodes (the one on the right) then a depth first search from `a` will not reach the same set of nodes as before. In particular, it will not reach node `d`.