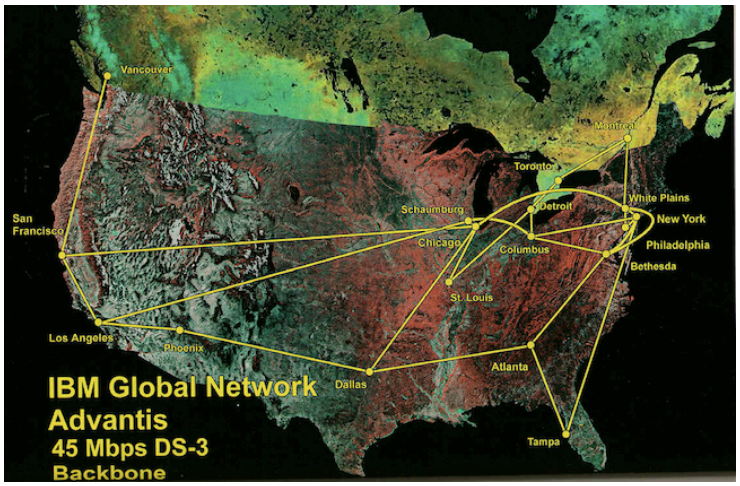
Problem-Based Intro. to Computer Science 20101
Week 9 – Message Routing

1 Problem

Imagine that we are trying to determine how to get a message from one point to another through a network. In this case, we are not worried if our path is the best one possible, we just want to find a possible path. Consider the following network as an example (IBM's network backbone, some years ago):



We wish to write a program that:

- reads a text version of some network (we need to design the input specification)
- gets the desired start and the finish location from the user, and
- tells us a path (a list of locations, in order) to get from the start to the finish, or tells us that no such path exists.

2 Solution Design and Analysis

2.1 Graphs

The network is naturally represented by a data structure called a *graph*. A graph, similar to a tree, is given by a set of *nodes* (also called *vertices*), and a set of connections, called *edges*, connecting pairs of nodes. As shown on the picture, these nodes may have some name or data associated with them. For example, the node representing San Francisco has three edges connected to it, and these edges connect to the Vancouver, Los Angeles and Chicago nodes.

2.2 Adjacency List Representation of Graphs

Graphs can be represented many different ways, partly depending on whether we wish to store data at the nodes and/or the edges, and partly on personal preference. One common approach that works here is to give a list, for every node v, of all other nodes that v is connected to. This is referred to as an *adjacency list*. In our case, we can use a small data structure to represent each node that contains the name associated with the node along with the adjacency list for the node. Each adjacency list then contains references to other nodes in the graph. The collection of all such nodes (usually in a list) defined this way is sufficient to describe the entire graph. Note that unlike a tree, there is no designated "root" to hold on to, so we need all of the nodes to be collected in a single place somehow. Also note that if one node (say node X) appears in node Y's adjacency list, that does not automatically mean that node Y is in node X's list. If our problem (such as this week's) does not specify the direction of the edge, we will want to make sure that both nodes are in each other's list for all edges.

2.3 Input Specification

Suppose we represent the network by adjacency lists. It might make sense then to have the input file contain all of the node names, followed by a description of all of the edges. In that case, we would read in the node names and make node objects, and then read in the edge descriptions and link the nodes together by adding them to each other's adjacency list. However, we can simplify things by just providing a list all of the edges. For the image above, the first few lines of such a file might look like:

```
Vancouver SanFrancisco
SanFrancisco LosAngeles
SanFrancisco Chicago
LosAngeles Schaumburg
LosAngeles Phoenix
```

In this case, when a node name is encountered that has not been seen before, we will create a new node, and then build the edges in both directions as well. This works fine as long as we do not wish to create a node with no edges coming from it (this is legal to have in a graph, just not very useful!).

2.4 Algorithm

We will use a *depth-first search* (DFS) to find a path. As an initial algorithm, we will simply decide whether a path exists between two given nodes. Note that a single graph data structure may contain more than one *component*, or group of connected nodes, and in fact it is not easy to discover whether this is the case from the adjacency lists!

We start at the starting node and store that the node has been visited. Then we move to one of its adjacent nodes and recursively visit all nodes we can possibly get to from there (except the ones we have already visited). After we return from this excursion, in similar fashion we investigate other nodes adjacent to the starting node.

This idea can be represented by the following pseudo code:

```
def DFSnode(node,visited):
    for every neighbor of node:
        if the neighbor has not been visited:
            add the neighbor to the visited list
            call DFSnode(neighbor,visited)

def DFS(start,finish):
    create an empty list called visited
    add the start node to the visited list
    call DFSnode(start,visited)
    if finish is in the visited list:
        print that it is possible to get from start to finish
    else:
        print that start and finish are not connected by a path

read the input from the file and form the nodes and their adjacency lists,
    put them in a list called nodelist
call DFS(start,finish)
```

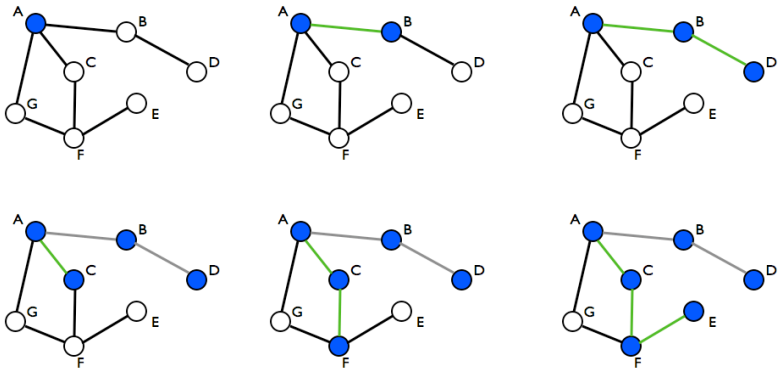
Now, if we want to actually figure out how we got from the start to the finish, it turns out we need to make only a couple of small changes. (Actually, there are several ways of doing this; here we present only one.) As we perform the recursion, we will note that the path generated when we make the recursive call consists of the current node plus the path generated in the recursive call. We then have an extra recursion base case that stops when the finish point has been reached. (Alternately, you can think of it as performing a type of calculation: path-from-node-to-finish = edge-from-node-to-neighbor + path-from-neighbor-to-finish.) The updated pseudo-code follows.

```
def DFSPnode(node,visited,finish):
    if node is the finish:
        return the path consisting only of the node
    for every neighbor of node:
        if the neighbor has not been visited:
            add the neighbor to the visited list
            if DFSPnode(neighbor,visited,finish) returns a path,
                return the path made of node followed by the
                returned path
    return no-path

def DFSpath(start,finish):
    create an empty list called visited
    add the start node to the visited list
    call DFSPnode(start,visited,finish) and call the result path
    if path is no-path:
        print that start and finish are not connected by a path
    else:
        print the names of the nodes in path

read the input from the file and form the nodes and their adjacency lists,
    put them in a list called nodelist
get the start and finish nodes
call DFSpath(start,finish)
```

We show how the traversal works on a smaller network. In this case, node A is the start and node E is the finish. For the sake of argument, in this case we assume that the neighbors of any given node are visited in alphabetical order. Visited nodes are blue while unvisited are white. The green edges show the path from start to the current node, grey edges mean that we have already gone through that edge. (Notice the subtle difference between original-black edges and edges that have been already used – they are grey.)



2.5 Implementation

See `routing.py`. Here we have chosen to use a class to represent each node, where the node contains a name (assumed to be a string) and a list of neighbors (themselves nodes). Note that the actual DFS code is much shorter than the code required to read in the graph data from the file!

2.6 Running time analysis

Typically in computer science the number of nodes is denoted by n and the number of edges is m. For our depth-first search, the `DFSnode` or `DFSPnode` gets recursively called at most as many times as is the total number of edges m (after that all the nodes will have been visited). Each time, we have to check whether the node has been visited, the time for which depends on implementation and can be anywhere from $O(1)$ to $O(n)$. Therefore, the time complexity of that part of the search is $O(mn)$. In the outer function (i.e. `DFS` or `DFSpath`), we either search the visited list (in `DFS`), or print a path (in `DFSpath`), either of which are $O(n)$ in the worst case. In our implementation, this adds up to $O(mn)$ in total, but if the recursive search takes $O(m)$, the total complexity would be $O(m+n)$. Other implementations may solve the problem differently, or solve a slightly different problem, and thus end up with different time complexity.

2.7 Testing

We should test our code incrementally, as soon as we develop a new function, we test it. We can start with the node creation and printing functions. Then, once we have the code for reading the graph from the file, we print the list of nodes – we try this with several input graphs of different complexity. We also need to test the function that finds a node for a given name, passing in both names that do correspond to nodes and names that do not. And, finally, we need to test the DFS:

- we test the case when start is the same as finish
- we test a start that is a direct neighbor of finish
- we test a start that is connected to the finish through a multi-step path
- we test a start and finish that are connected through more than one path (only one path should be generated)
- we test a start and finish that are not connected by a path