Adjacency Lists

A common file format for encoding graph information is called a DOT file (http://en.wikipedia.org/wiki/DOT_(graph_description_language)). We're going to write a class to read in a .dot file of an undirected, unweighted graph, store the information in an adjacency list, and then run some common operations on that graph.

Here's an example dot file (labs/example.dot). The first line names the graph. Following that is a list of all edges in the graph. You may assume the vertices are "named" using a single word, but they may not be single letters like this. You can also assume the graph is undirected and unweighted, and that there is one edge specified per line, like in our example file.

To represent a graph in code, you can use a Node class very much like we did for trees to represent each vertex. The difference here will be that instead of having a left and right child, graph vertices can have any number of neighbors, so you need to accommodate for that. This means that your Node should have a self variable that stores a list of adjacent nodes, i.e. an adjacency list from our previous lecture.

Remember, unlike trees, graphs do not have a "root". Because of this, you need to have a data structure for storing the vertices in a way that you can directly access any vertex. The best way to do this is to use a Map with the key being the vertex identifier and the value being the Node object that represents the vertex. For this lab you can use the built in Python dictionary, no need to make your own Map.

This means that your task in the lab is to maintain a Map where the keys are vertex identifiers (A, B, C, etc.) and the values are the Node objects corresponding to those identifiers. To do this, you must read the .dot file and add neighbors appropriately for every line that contains an edge between two vertices.

For example, if you see the line `A -- B`, there is a four step process:

1. Does A exist in the map yet? If not, create a new Node object with the label A and insert it into the map.
2. Does B exist in the map yet? If not, create a new Node object with the label B and insert it into the map.
3. Add A to B's adjacency list.
4. Add B to A's adjacency list.

You will create a Graph class in the file `graph.py` Your Graph class needs the following methods:

- A constuctor which accepts a file name, stores the graph name, and builds the adjacency list. You can assume a competant user is calling this, and the file is, indeed, a .dot file. (Ex: `g=Graph('example.dot')`)
- `isAdjacent(self, vertexA, vertexB):` Returns True if the two vertices are adjacent, and False if they are not. (Ex: Given our example.dot, `g.isAdjacent('B','A')` would return True, while `g.isAdjacent('D','A')` would return False.) Because the graph is undirected, the order of the arguments shouldn't matter.
- `returnAdjacent(self, vertex):` Returns a list of all vertices adjacent to `vertex` This should be a list of the identifiers, not the Node objects themselves.
- `__str__(self):` Returns a string which re-creates the .dot file from the adjacency list. It doesn't need to be in the same order as it was when it came in, but it should contain the same information. You may not just store all the text and regurgitate it for this method.

Your code should be in a file called `graph.py`. This lab will be the basis of our next project and lab, so understanding it carries some extra importance.

Important: For the `__str__` function, you have to make sure not to print out the edges twice, i.e. you should not have `A -- B` and `B -- A`.

Visualizing .dot files

There is a great open-source package for visualizing graphs caled `graphviz`. If you don't have it yet, you can install it with the command:

```
sudo apt-get install graphviz
```

Now, with example.dot, run `fdp -Tpng example.dot > example.png`. You can then look at example.png to see your graph. Run `man dot` for the full specifications.

Extra Challenge

.dot files can also handle directed graphs, though right now, your program probably can't. Fix that! If `A -> B;`, then `isAdjacent('A','B')` should return True, but `isAdjacent('B','A')` should return False.

Arrows always point right.

Directed graphs are labelled as such: in the first line, instead of "graph graphName {" it would say "digraph graphName {". In digraphs, ALL edges are directed (A -- B; is not allowed). In regular graphs, NO edges are directed.