MCL summer workshop in graph theory
Lab 1

6/21/16

Programming exercises

We will be using Python for most of our programming in this workshop. If you've never used Python before, take a look at this tutorial: `https://www.codecademy.com/learn/python`.

On the other hand, if you are already a Python pro, then try your had at these exercises!

Problem 1: a simple graph class

Let's create a simple `Graph` class. Your representation of a graph G will be based on its *adjacency matrix*, a 2D array whose (i, j)th entry is `True` if G has an edge connecting vertices i and j, `False` otherwise.

Your class should contain the following methods (test each one as you go along!):

- A *constructor* which takes a parameter n and constructs a new graph with n vertices (labeled 0 through $n - 1$) and no edges. Your constructor should initialize an $n \times n$ array `A` whose entries are all `False`, and an integer `numEdges` initially set to 0.

- `hasEdge(`*self*, i, j`)`: return `True` if the graph has an edge connecting vertices i and j, `False` otherwise.

- `addEdge(`*self*, i, j`)`: add an edge connecting vertices i and j, i.e. set the (i, j)th entry of the adjacency matrix to `True`. `hasEdge(i, j)` should always return the same thing as `hasEdge(j, i)` (why?), so make sure you set the (j, i)th entry of the adjacency matrix to `True`, too!

  Before setting the (i, j)th entry of the matrix to `True`, you should check its current value. If it's already `True`, then there's already an edge there, so there's nothing to do. But if you are changing it from `False` to `True`, then you really are adding an edge, so you should add 1 to *self*.`numEdges`.

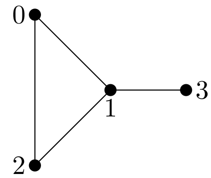

Figure 1: Use this graph to test your program!

.

- `removeEdge(`*self*, i, j`)`: you're a smart person; you can figure out what this is supposed to do!

- `neighbors(`*self*, i`)`: return an array of vertex i's *neighbors*, i.e. the other vertices connected to vertex i. For example, for the graph in Figure 1 `neighbors(1)` should return `[0, 2, 3]`, and `neighbors(3)` should return `[1]`.

- `degree(`*self*, i`)`: return the *degree* (number of neighbors) of vertex i. For the graph in Figure 1, `degree(1)` should return 3, and `degree(3)` should return 1. Hint: you've already done most of the work!

- `__str__(`*self*`)`: return a string containing each vertex and its list of neighbors. For the graph in Figure 1, this should look something like

```
0:  1 2
1:  0 2 3
2:  0 1
3:  1
```

Problem 2: special types of graphs

Add the following functions to your program (outside the `Graph` class):

- `CompleteGraph(`n`)`: return a `complete graph` with n vertices, i.e. a graph containing all possible edges. This does *not* included edges from a vertex to itself. Thus, `hasEdge(i, i)` should always be `False`. See Figure .

- `Path(`n`)`: return a *path* of length n, a graph with n vertices and edges from vertices 0 to 1, 1 to 2, . . ., and $n-2$ to $n-1$ (see Figure 3).

- `Cycle(`n`)`: return a *cycle* of length n, a graph with n vertices and edges from vertices 0 to 1, 1 to 2, . . ., $n-2$ to $n-1$ *and* $n-1$ to 0 (see Figure 3). Hint: a cycle is a path with one extra edge!

- `RandomGraph(`n`)`: create a graph with n vertices. Then, for each pair of vertices flip a coin; if heads, add an edge between those vertices. You can

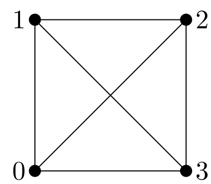

Figure 2: A *complete graph* on 4 vertices has 6 edges: 01, 02, 03, 12, 13, and 23.

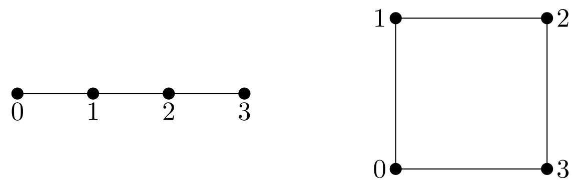

Figure 3: A *path* of length 4 (left) and a *cycle* of length 4 (right).

do this by importing the module `random` and using `random.randint(0, 1)`, with 0 corresponding to tails and 1 to heads.

You should *not* add edges from a vertex to itself, and be careful not to flip a coin for both (i, j) and (j, i) (since these are the same pair of vertices).

Each of these functions should create a new `Graph` object and add the appropriate edges with `addEdge()`.

Math exercises

Let's say you generate a random graph on 4 vertices by flipping a fair coin for each pair of vertices and adding an edge between that pair if it comes up heads. Compute the following probabilities:

1. $\Pr(G = K_4)$ (i.e., all possible edges)
2. Pr(12 and 23 are edges)
3. Pr(12 and 23 are the *only* edges)
4. Pr(G has *exactly* 2 edges)

How do your answers change if the coin is not fair (say $\Pr(\mathrm{H}) = 1/3$)?