Problem Set One, CR15 ENS de Lyon. Course authorities : Karsai / Crespelle / Unicomb

Due date : 7/11/2016, by 11:59 p.m.

Complementary material : graph.py, *.txt data files

1 Introduction to a basic graph data structure

In this problem set we will be dealing with a graph $G = (V, E)$ where V is a set of vertices and E a set of edges. In graph theory, the adjacency list representation of a graph is a collection of unordered lists, one for each vertex in the graph. Each list describes the set of neighbours of its vertex. We define our graph with a dictionary,

```
graph = {
  "a" : ["c", "d", "g"],
  "b" : ["c", "f"],
  "c" : ["a", "b", "d", "f"],
  "d" : ["a", "c", "e", "g"],
  "e" : ["d"],
  "f" : ["b", "c"],
  "g" : ["a", "d"]
}
```

The aim of this problem set is to develop a rudimentary graph library. The skeleton of the library is provided, leaving you room to add the class *Graph* by solving the problems below. The following code is provided as a .py file, which you will edit during the problem solving session.

```
""" IXXI graph library """

class Graph(object):

    def __init__(self, graph_dict={}):
        """ initializes a graph object """
        self.__graph_dict = graph_dict

    def vertices(self):
        """ returns the vertices of a graph """
        return list(self.__graph_dict.keys())

    def edges(self):
        """ returns the edges of a graph """
        return self.__generate_edges()

    def add_vertex(self, vertex):
        """ If the vertex "vertex" is not in
            self.__graph_dict, a key "vertex" with an empty
            list as a value is added to the dictionary.
            Otherwise nothing has to be done.
        """
        """ TO COMPLETE """

    def add_edge(self, edge):
        """ assumes that edge is of type set, tuple or list;
            (no multiple edges)
        """
        """ TO COMPLETE """

    def __generate_edges(self):
        """ A static method generating the edges of the
            graph "graph". Edges are represented as sets
            two vertices (no loop)
        """
        edges = []
        """ TO COMPLETE """
        return edges

    def __str__(self):
        """ A better way for printing a graph """
        res = "vertices: "
        for k in self.__graph_dict:
            res += str(k) + " "
        res += "\nedges: "
        for edge in self.__generate_edges():
            res += str(edge) + " "
        return res

if __name__ == "__main__":
    g = {
        "a" : ["c", "d", "g"],
        "b" : ["c", "f"],
        "c" : ["a", "b", "d", "f"],
        "d" : ["a", "c", "e", "g"],
        "e" : ["d"],
        "f" : ["b", "c"],
        "g" : ["a", "d"]
    }

    graph = Graph(g)
    print("Vertices of graph:")
    print(graph.vertices())
    print("Edges of graph:")
    print(graph.edges())
```

1.1 Simple methods to get the ball rolling

Add the following methods to the *Graph* class,

- *__generate_edges*: list the edges E
- *add_vertex*: add a vertex to the graph G
- *add_edge*: add an edge to the graph G

2 Methods related to degree

In the following section we develop our library of functions, outputting characteristic properties of the instance graph. These methods may be useful in subsequent sections.

2.1 Degree and isolated vertices

Add the following methods to the *Graph* class,

- *vertex_degree*, which gives all degree for each vertix of the input graph, and
- *find_isolated_vertices*, which gives the set of zero-degree vertices.

2.2 Density calculation

The graph density is defined as the ratio of the number of edges of a given graph, to the total number of edges the graph could have. In other words, it measures how close a given graph is to a complete graph.

Create the method *density* that computes this parameter for the instance graph. A simple verification would be to test your code on a random, complete and empty graph, where you know the densities in advance.

2.3 Degree sequence

The degree sequence of an undirected graph is defined as the sequence of its vertex degrees in a non-increasing order. For our example, the degree sequence would be the tuple $(4, 4, 3, 2, 2, 2, 1)$.

Create a method *degree_sequence* that returns a tuple with the degree sequence of the instance graph.

2.4 Erdös-Gallai theorem

There is a question as to whether a given degree sequence can be realised by a simple graph. The Erdös-Gallai theorem states that a non-increasing sequence of n numbers d_i, for $i = 1, ..., n$, is the degree sequence of a simple graph if and only if the sum of the sequence is even and the following condition is fulfilled:

$$\sum_{i=1}^{k} d_i \leq k(k-1) + \sum_{i=k+1}^{n} \min(d_i, k) \text{ for } k \in \{1, ..., n\} \tag{1}$$

Develop a function *erdos_gallai* within our *Graph* class to determine whether a sequence fulfills the Erdös-Gallai theorem.

2.5 Clustering coefficient

The global clustering coefficient is based on triplets of nodes. A triplet consists of three nodes that are connected by either two (open triplet) or three (closed triplet) undirected ties. A triangle consists of three closed triplets, one centred on each of the nodes. The global clustering coefficient is the number of closed triplets, or thrice the number of triangles, over the total number of triplets (both open and closed).

$$C = \frac{3 \times \text{number of triangles}}{\text{number of connected triplets of vertices}} \tag{2}$$

Add the method *global_clustering_coefficient*, to compute this parameter for the instance graph.

3 Methods related to graph traversal

In this section we add more depth to the *Graph* class. To develop the following functions, it may be useful to call upon previously solved problems to avoid duplicating the same calculations.

3.1 Connected components

A connected component of a graph is a subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the supergraph.

Compute the sizes and number of connected components for the instance graph, defining in your class the function *connected_component*.

3.2 Shortest path

The shortest path problem concerns finding a path between two vertices in a graph such that the sum of the weights of its constituent edges is minimised. In an unweighted graph, this is the same as minimising the number of edges traversed.

For each pair of vertices (u, v), compute the distance between u and v in a method *shortest_path*.

3.3 Diameter

First, write a *diameter* method outputing the maximum value of the shortest path distances. After coding this tool, write a second method *biggest_component_diameter* that computes the diameter of the biggest component.

3.4 Spanning tree

The spanning tree T of a connected graph G is a tree that includes all of the vertices and some or all of the edges of G. For a weighted graph, a Minimum Spanning Tree, or MST, is then a spanning tree with weight less than or equal to the weight of every other spanning tree. Unweighted graphs can be viewed as having uniform edge weights, with spanning trees having a total weight of $n - 1$.

Implement a spanning tree algorithm in the method *spanning_tree*.

4 Testing on real datasets

In this section you will test your *Graph* class on three datasets, chosen from the online database http://konect.uni-koblenz.de/networks/

- the Zachary karate club
- a random graph, 100 nodes and 1000 edges
- a random graph, 1000 nodes and 4000 edges

To facilitate the computation, we have already extracted the biggest component. The input format is an unweighted edge list, where a row contains the tab separated values u and v if (u, v) belongs to the edge list E.

4.1 Importing real data

Create and test a piece of code that loads the graph from a file. Note that the graphs are supplied in the format of edge lists, whereas the *Graph* class inputs a dictionary. So, consider that you will have to first manipulate the supplied data.

4.2 Properties of supplied graphs

For each dataset, compute the number of vertices, number of edges, density and clustering coefficient with your new graph library. Then, complete the table below.

dataset	#vertices	#edges	density	diameter	clustering coefficient
Zachary					
random, $N = 100$					
random, $N = 1000$					