Assignment 4: Graphs

UMaine COS 350

Assigned: 12 March 2020

Due: 27 March 2020

In this assignment, you will build a Python graph package that will support future homework assignments. The package should be named `graphs.py`; you should be able to use it in other programs by:

```
import graphs
```

or:

```
from graphs import *
```

This package must be object-oriented. In the remainder of this homework description, we'll look at the classes and methods `graphs.py` will provide.

Jupyter notebooks

You will be turning in a Jupyter notebook for this assignment. I believe most of you have used Jupyter notebooks for Python development. If not, see jupyter.org and click on the "Jupyter Notebook" link under "User Interfaces". If you need to install Jupyter, the documentation guides you through the process. I recommend installing it (and Python) using Anaconda. (You should be using Python 3.5 or greater, by the way, for this class; I'm using 3.6 for the code in this document.)

Develop all your code and documentation and show your tests of your code in a single Jupyter notebook; this is what you will turn in. I'll provide a blank notebook you can start with that contains the code described below.

Class `Graph`

You will need to create a class, `Graph`, that is the base class for graphs. A graph is, of course, just a set of vertices and edges, but there are many different ways to implement them (edge lists, etc.), just as there are different types of graphs (digraph, weighted digraph, etc.). Different implementations have different properties, for example, how much time it takes to find an adjacent vertex or incident edge. The `Graph` class won't be directly implemented. Rather, its sub-classes will implement graphs in various ways so that the best implementation can be used for the algorithm you're working on.

So let's define the `Graph` class:

```
###
### Graph class
###
### This class is meant to be specialized for whatever implementation of graphs you
### want to create.
###

class Graph ():
```

Although we don't want to necessarily instantiate this class, we'd like to be able to instantiate its subclasses all the same way unless there's a need not to. So let's create a default initialization function that takes two (optional) arguments, `graph_desc`, a Python list that describes the graph, and `properties`, that is meant to allow you to use when specializing this class:

```
    def __init__(self,graph_desc=None,properties=None):
```

The argument `graph_desc` should be a list whose first element describes the kind of graph it is and the rest describing the edges and vertices. The description of the graph should be one of `'graph'` or `'digraph'`, at least, though you may want to add additional types as you write specialized subclasses.

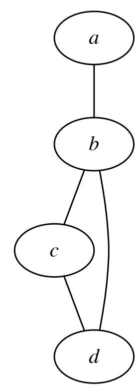

Figure 1: Example graph.

The edges and vertices are specified themselves as lists. If there's a single element, it's a vertex, else it's an edge. Edge specifications are expected to have either two or three elements, with the first two being the source and destination of the edge and the (optional) third being a weight. For example,

```
['graph' ['a', 'b'], ['b', 'c'], ['c', 'd'], ['d', 'b']]
```

would specify the graph in Fig. 1, and

```
['digraph' ['a', 'b', 3], ['b', 'c', 2], ['c', 'd', 5], ['d', 'b', 1]]
```

would specify the graph in Fig. 2.

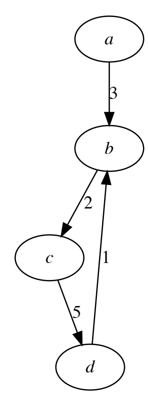

Figure 2: Example digraph.

Note that

```
['graph' ['a', 'b'], ['b', 'c'], ['c', 'd'], ['d', 'b']]
```

and

```
['graph' ['a'], ['b'], ['a', 'b'], ['b', 'c'], ['c', 'd'], ['d', 'b']]
```

are specifications of the same graph, with the latter just directly specifying two of the vertices.

For our implementation of the initialization function, we ignore weights and whether or not something is a digraph; you'll need to handle those aspects of things in your subclasses. First, set up the instance variables in the case of no description:

```
        self.properties = properties

        self.vertices = {}
        self.edges = []
```

If there *is* a description, though, we need to put the first element (e.g., `'digraph'`) onto the list of properties:

```
        if graph_desc:
            if properties == None:
                self.properties = list(graph_desc[0])
            else:
                self.properties.append(graph_desc[0])
```

Now we need to parse the rest of the description, i.e., the edge and vertex descriptions. For this base class, we ignore any weights. We define two other instance variables as we parse:

- `vertices`: This is a dictionary for the base class, with the keys being vertex names and the values being instances of `Vertex`.
- `edges`: This is just a list of instances of class `Edge`.

It's here that you'll want to make some changes about how you store vertices and edges in the subclasses; this is just a placeholder that could be useful:

```
            for ele in graph_desc[1:]:
                if ele[0] not in self.vertices.keys():
                    self.vertices[ele[0]] = Vertex(self,ele[0])
                if len(ele) > 1:              #then it's a real edge description
                    if ele[1] not in self.vertices.keys():
                        self.vertices[ele[1]] = Vertex(self,ele[1])

                    # Not checking for duplicate edges here:
                    self.edges.append(Edge(self, self.vertices[ele[0]],
                                        self.vertices[ele[1]]))
```

Just for fun, let's create some methods to get vertices, edges, etc.:

```
    def find_vertex(self,name):
        if name in self.vertices.keys():
            return self.vertices[name]
        else:
            return None
```

```

    def add_vertex(self,name):
        self.vertices[name] = Vertex(self,name)

    def find_edge(self,v1,v2):
        for e in self.edges:
            if e.v1.name == v1 and e.v2.name == v2:
                return e
        return None

    def add_edge(self,v1,v2):
        source = self.find_vertex(v1)
        dest = self.find_vertex(v2)
        if source == None:
            self.add_vertex(v1)
            source = self.vertices[v1]
        if dest == None:
            self.add_vertex(v2)
            dest = self.vertices[v2]

        self.edges.append(Edge(self,source,dest))

    def incident_edges(self,v):
        if not isinstance(v,Vertex):
            v = self.find_vertex(v)
        incident = []
        for e in self.edges:
            if e.v1 == v or e.v2 == v:
                incident.append(e)
        return incident
```

And to see what's in a graph, we'll create two other methods, `to_list` and `to_gv`. The first will return a list suitable for passing to the initialization function for `Graph`; i.e., to copy a graph `g`, you could do:

```
g1 = Graph(g.to_list())
```

The second function will print out a graphviz "dot" representation for the graph; you could, for example, cut and paste this to a file `foo.gv`, then (on a Mac or Linux box, anyway) do:

```
dot -Tpdf foo.gv > foo.pdf
```

to get a PDF diagram of the file (e.g., like those in Fig. 1 and Fig. 2).

```
    def to_list (self):
```

```
        if ’digraph’ in self.properties:
            output = [’digraph’]
        else:
            output = [’graph’]

        for vname in self.vertices.keys():
            output.append([vname])

        for e in self.edges:
            output.append([e.v1.name,e.v2.name])

        return output

    def to_gv(self):
        desc = self.to_list()

        print(desc[0], " {")

        if desc[0] == ’digraph’:
            sep = " -> "
        else:
            sep = " -- "

        for thing in desc[1:]:
            if len(thing) == 1:  # vertex
                print(’   ’, thing[0])
            elif len(thing) > 2: # edge, weight exists
                print(’   ’, thing[0], sep, thing[1], ’ ’, ’[label="’, thing[2],
                      ’"]’)
            else: #no weight
                print(’   ’, thing[0],sep,thing[1])
        print(’}’)
```

We’ll also provide two other base classes, one for vertices and one for edges, that are used in the initialization function of `Graph`:

```

class Vertex ():
    def __init__(graph,name):
        self.graph = graph
        self.name = name


class Edge ():
    def __init__(graph,v1,v2):
```

```
        self.graph = graph
        self.v1 = v1
        self.v2 = v2
```

Note that these are very minimal, and you'll need to extend them for your own use. The `Vertex` class just holds the name of the vertex and a pointer back to the graph it's part of, and the `Edge` class just holds a pointer to the graph and pointers to the two end points of the edge (expected to be instances of `Vertex`).

Your turn

Classes

Now you will need to complete the package. Your `graph.py` package should include these classes, ultimately based on the `Graph` class:

1. `AdjList`: Implements a graph as a an adjacency list.
2. `EdgeList`: Implements a graph as an edge list.
3. `AdjMatrix`: Implements a graph as an adjacency matrix.

Each of these should be able to handle undirected graphs, digraphs, and weighted or unweighted edges.

Methods

You should write all the methods needed to access vertices and edges in your subclasses (or use default methods, if those work). This includes the following (some of which are implemented already for the base class above):

- `add_vertex`, `delete_vertex`: add/delete a vertex
- `add_edge`, `delete_edge`: add/delete edge
- `all_vertices`: return all vertices; should take an optional argument to cause it to return either a list of `Vertex` instances or a list of names of the vertices.
- `all_edges`: return all edges; should take an optional argument to cause it to return a list of `Edge` instances or a list of the form

  ```
  [['a', 'b'], ['b', 'c], ...]
  ```

- `to_list`: return a description of the graph in a from that could be given to `Graph()` or similar instantiation function.

- `find_edge`: find an edge given an `Edge` instance or list of the form `[v1,v2]` of vertex names.
- `find_vertex`: find a vertex given its name.
- `incident_edges`: return a list of all edges incident on a vertex
- `adjacent_vertices`: return all vertices adjacent to another vertex
- `edge_cost`: return cost of an edge
- `path_cost`: return the cost of a path, where a path is a list of vertex names, e.g., `['a', 'b', ...]`
- `adjacent_to`: return `True` if v1 is adjacent to v2
- `incident_to`: return `True` if an edge is incident to a vertex.

In addition, write `DFT`, a depth-first traversal method for the classes that, given a vertex, does a depth-first traversal and returns a spanning tree rooted at that vertex. (Note that the spanning tree is itself a graph, and should be returned as an instance of `Graph` or one of its subgraphs.

Turn in

For this assignment, turn in a single Jupyter notebook showing your code. Comments should for the most part be in text boxes (in plain text or Markdown format) in the midst of your code, but occasional short "regular" Python comments (i.e., those starting with `#`) are reasonable, too, where needed.

The notebook should function as a user guide for your `graph.py` package (which can be exported from the notebook for use), so the notebook should include information about how to use the classes and methods developed. At the end of the notebook, you should demonstrate your code working.