Project 1: PageRank in Python

Due Wednesday, Sep 19, 2018 at 8pm

Contents

In this project, you will implement a basic graph library in Python 3 and then implement a simplified version of PageRank, a famous algorithm in search-engine optimization. The primary learning goal of the project is to gain familiarity with the syntax, data structures, and idioms of Python 3. A secondary goal is to review the basics of abstract data types (ADTs) and object-oriented programming.

The project is divided into multiple suggested phases. We recommend completing the project in the order of the phases below.

Background

Review of Terminology

We start by reviewing concepts in graph data structures, which you may have seen in your prior discrete mathematics and data structures courses. We will also introduce some terminology used in search-engine algorithms.

A *graph* is an abstract data type (ADT) that consists of a set of *nodes* (or *vertices*) and a set of *edges* (or *links*) connecting pairs of nodes. Less formally, graphs model connections or interactions between entities. In this project we model both *undirected* and *directed* graphs, either of which may be optionally *attributed*.

In an attributed graph, nodes and/or edges have additional labels, or attributes, associated with them. For example, a social network is a type of attributed graph in which nodes, or users, are associated with data like age, country, and gender.

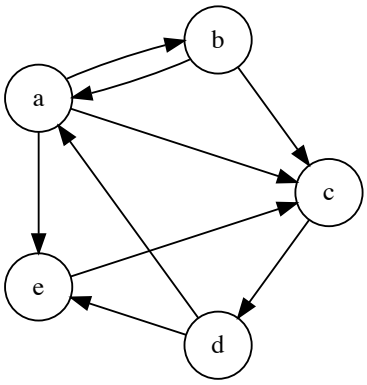

Figure 1: A directed graph.

An undirected graph is one such that an edge from node u to node v is equivalent to an edge from node v to node u, whereas in a directed graph those edges are not the same. Figure 1 shows a small directed graph.

In a directed graph, an edge leaving u and arriving at v may be called a *forward link* of u, whereas an edge leaving v and arriving at u may be called a *backlink* of u. For example, in Figure 1, node e has one forward link and two backlinks.

In an undirected graph, the *degree* of node u is the number of edges to which u is incident. In a directed graph, the *in-degree* of node u is the number of edges that arrive at u, or equivalently the number of backlinks of u; the *out-degree* of node u is the number of edges that leave u, or equivalently the number of forward links of u. In Figure 1, node e has an out-degree of 1 and an in-degree of 2. If the graph were not directed, node e would simply have a degree of 3.

PageRank

In 1998, Michigan alum Larry Page and his fellow Stanford PhD student Sergey Brin introduced PageRank, a novel web search ranking algorithm. PageRank was the foundation of what became known as the Google search engine. More generally, PageRank can be used to approximate the "importance" of any given node in a graph structure.

Intuitively, a node in a graph will have a high PageRank if the *sum of the PageRanks of its backlinked nodes* are high. Thus, a node's PageRank depends not only on the number of backlinks it has, but also the importance of those backlinked nodes. For example, if a node u only has one backlink from node v, but node v has a high PageRank, then node u will also have a relatively high PageRank.

A real-world example is the Twitter graph of who-follows-whom, where a directed edge from user A to user B exists if A follows B. Famous people like Barack Obama have millions of followers, some of whom are also famous with millions of followers. Thus, these users, when represented as nodes in a graph, will have high PageRank. Furthermore, such celebrities' high PageRank will contribute to the PageRank of the users that *they* follow. If Barack Obama follows another user and is that user's only follower, that user will still have a relatively high PageRank.

Note that this definition of PageRank is inherently recursive: computing a node's PageRank depends on other nodes' PageRanks, which in turn depend on other nodes' PageRanks, and so on. However, Page and Brin show that the PageRank algorithm may be computed iteratively until convergence, starting with any set of assigned ranks to nodes[1].

The PageRank computation models a theoretical web surfer. Given that the surfer is on a particular webpage, the algorithm assumes that they will follow any of the outgoing links with equal probability. Thus, the PageRank value of the webpage is divided equally among each of the linked webpages, raising each of their values. Using this reasoning, we end up with the following formula for computing a webpage u's PageRank:

$$PR_{k+1}(u) = \sum_{v \in BL(u)} \frac{PR_k(v)}{v^+}$$

Here, $BL(u)$ is the set of nodes that link to u (i.e. the nodes in its backlinks), and v^+ is the out-degree of node v.

The reasoning above, however, breaks down when it comes to a *sink* webpage that has no outgoing links: it assumes that the surfer gets stuck, never leaving that webpage. Instead, we will assume that the surfer restarts their session at a

[1] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst.*, 30(1-7):107–117, April 1998.

random webpage. Thus, the sink contributes $1/N$ of its PageRank value to every webpage, where N is the total number of pages. The formula then becomes:

$$PR_{k+1}(u) = \sum_{v \in BL(u)} \frac{PR_k(v)}{v^+} + \sum_{w \text{ is a sink}} \frac{PR_k(w)}{N}$$

Finally, we introduce a *damping factor* $d \in (0,1]$, which models the fact that surfers may restart their web session even if they are on a page that does have outlinks. Thus, we damp a webpage's PageRank by multiplying it by d, and then we assume that the total residual probability $1-d$ is distributed among all webpages, so that each page receives $(1-d)/N$ as its share[2]:

$$PR_{k+1}(u) = \frac{1-d}{N} + d\left(\sum_{v \in BL(u)} \frac{PR_k(v)}{v^+} + \sum_{w \text{ is a sink}} \frac{PR_k(w)}{N}\right)$$

We apply this update rule iteratively, with a uniform initial distribution of $PR_0(u) = 1/N$. This algorithm will converge after a small number of iterations, on the order of 50 for large sets of webpages. The PageRanks can then be considered the relative probability that a surfer will visit each webpage, and the PageRanks of all webpages add up to 1.

The PageRank problem can also be formulated in terms of matrix algebra. We do not use that formulation here, but you can refer to the original paper for details if you are interested.

Distribution Code

Use the following commands to download and unpack the distribution code:

```
$ wget https://eecs490.github.io/project-pagerank/starter-files.tar.gz
$ tar xzf starter-files.tar.gz
```

The distribution code consists of the following files:

`graph.py`	Definition of the graph ADTs.
`pagerank.py`	Implementation and driver for computing PageRanks.
`graph_test.py`	Basic test cases. Add your own to this file.
`graph_test.expect`	Expected output from running `graph_test.py`. Update this when you add more test cases.
`Makefile`	A Makefile for running test cases.
`characters-nodes.csv`	The nodes and their attributes for a small graph.
`characters-edges.csv`	The edges and their attributes for a small graph.
`characters-pagerank.expect`	Expected output from running `pagerank.py` on the `characters` graph.
`email-Eu-core.txt-nodes.csv`	The nodes for a medium graph from a SNAP email dataset.
`email-Eu-core.txt-edges.csv`	The edges for a medium graph from a SNAP email dataset.
`email-Eu-core.txt-pagerank.expect`	Expected output from running `pagerank.py` on the `email-Eu-core.txt` graph.

[2]The original paper mistakenly uses $1-d$ instead of $(1-d)/N$ for the PageRank formula, in which case the PageRanks sum to N rather than 1. Subsequent papers use the corrected formula.

Data Format

The distribution code includes a function `read_graph_from_csv()`, which constructs a graph from spreadsheets in CSV format. A node file must start with a header row, where one of the columns must be `Id`. The remaining columns may have arbitrary labels. Each data row must have a value for each column. The function creates a directed or undirected graph, and for each data row in the node file, it adds a node to the graph. The node's ID is the string value in the `Id` column, and the remaining columns hold the attribute values for the node, with the column labels as the attribute names. An edge file has a similar structure, except that the header must contain entries `Node_Id_1` and `Node_Id_2`. These columns contain the IDs of the the two nodes connected by the edge.

Phase I: Graph ADTs

Start by reading through the code in `graph.py`, which contains the following classes:

- `GraphError`: An exception class used by the graph ADTs, derived from the built-in `Exception`.
- `Node`: Represents a node in a graph, with an ID and attributes.
- `Edge`: Represents an edge in a graph, connecting two nodes and with its own attributes.
- `BaseGraph`: The base class for both directed and undirected graphs. Most of the functionality in the graph ADTs is implemented here.
- `UndirectedGraph`: A class representing an undirected graph, derived from `BaseGraph`.
- `DirectedGraph`: A class representing a directed graph, derived from `BaseGraph`.

`GraphError`

The `GraphError` class contains the following methods:

- `__init__`: The special method in Python for initializing an object, i.e. the constructor. For this class, the constructor takes an optional argument representing the message associated with the exception, defaulted to an empty string. This message should be saved as an *instance variable*, an attribute of the object itself using the `self` argument.
- `__str__`: The special method for retrieving the printed string representation of an object. For this class, the `__str__` method should produce the saved message.
- `__repr__`: The special method for producing a code representation of the object. Calling the built-in `eval()` on the result should produce an equivalent object.

Fill in the implementations of the first two methods.

The strings at the beginning of the class and each method are *docstrings*, serving as their documentation. They are surrounded by triple quotes, allowing the strings to span multiple lines. The docstring for the class also contains some basic test cases, called *doctests*, which can be run using the Python `doctest` module. Normally, the doctests can be invoked from the command line with the following command:

```
$ python3 -m doctest <python file>
```

However, for the `graph.py` module, we need to configure the `doctest` module to ignore exception details, since some of the doctests should produce an exception when run. As such, we have arranged for the doctests to run when invoking the `graph.py` module itself:

```
$ python3 graph.py
```

Once you have implemented `GraphError`, its doctests should pass, but the remaining doctests will fail until you implement the rest of the module.

Node

The `Node` class is used in the interface for the graph ADTs. Looking up a node ID will produce a `Node` object, with the given ID and the attributes associated with the given node. The class contains a constructor, which takes in an ID and any number of keyword arguments (also see sections 2.1.1-2.1.3 in the course notes). These are packaged into a dictionary by Python, which you should store as an instance variable.

The `attributes()` method should return a *copy* of the dictionary received by the constructor. This is so that if a user modifies the dictionary returned by `attributes()`, it does not modify the dictionary stored in the `Node` object itself:

```
>>> n = Node('foo', a=3)
>>> n.attributes()
{'a': 3}
>>> n.attributes()['a'] = 4
>>> n.attributes()
{'a': 3}
```

A shallow copy of the dictionary suffices, since the attributes we will use are immutable (strings and numbers).
The `__str__` method should produce a representation of the `Node` in the following format:

- The first line of the string should be `Node [<id>]`, where `<id>` should be the ID of the node.
- The remaining lines should the attributes of the node, one per line, in lexicographically increasing order by attribute name. Such a line should begin with exactly four spaces, then the attribute name, a space, a colon, another space, and then the value of the attribute. The line should be terminated with a newline character.

The following is an example:

```
>>> n = Node('bar', a=4, b=3)
>>> print(n)
Node [bar]
    a : 4
    b : 3

>>>
```

In order to construct the string representation, we recommend using Python 3's string formatting functionality. You may also find the built-in `sorted()` function useful for ordering attributes.

Edge

The `Edge` class is also used in the interface for the graph ADTs, representing a directed edge between two nodes. The constructor takes in the two `Node` objects and the attributes of the edge itself. The `nodes()` method returns a tuple containing the two `Node` objects, in the same order as passed to the constructor. The string representation is similar to that of `Node`, except that the initial line has the form:

```
Edge from node [<id1>] to node [<id2>]
```

Here, `<id1>` and `<id2>` should be the IDs of the two `Node`s.

BaseGraph

The `BaseGraph` class implements the core functionality of a graph ADT, and it is the base class for the other graph classes. It has the following methods:

- `__init__`: The constructor initializes the graph to be empty, so that it contains no nodes or edges.
- `__len__`: A Python special method, called by the built-in `len()` function. For `BaseGraph`, it should return the number of nodes in the graph.
- `add_node`: Takes in a node ID and keyword arguments representing the attributes of the node. Adds a corresponding `Node` to the graph. Raises a `GraphError` if the graph already contains a node with the given ID.

- `node`: Returns the `Node` object corresponding to the given ID. If the ID does not match a node in the graph, a `GraphError` is raised.
- `nodes`: Returns a list of all nodes in the graph, sorted by node ID in lexicographic order. You may find the built-in `sorted()` function useful here.
- `add_edge`: Adds an edge between the nodes corresponding to the given IDs. The keyword arguments are the attributes of the edge. If either of the given IDs does not match a node in the graph, raises a `GraphError`. Also raises a `GraphError` if the graph already contains an edge between the nodes (we are not implementing multigraphs in this project).
- `edge`: Returns an `Edge` object corresponding to the edge between the given node IDs. Order is relevant here: the edge corresponds to the call to `add_edge()` with IDs in the same order as the arguments to `edge()`. Raises a `GraphError` if no such edge is in the graph.
- `edges`: Returns a list of the edges in the graph, sorted in lexicographic order by the pair of node IDs corresponding to each edge.
- `__getitem__`: A Python special method for overloading the subscript (square-brackets) operator on an object. For `BaseGraph`, the method should return the given `Node` object if supplied with a node ID or the given `Edge` object if supplied with a pair of node IDs. If there is no corresponding node or edge, it should raise a `GraphError`.
- `__contains__`: A Python special method for overloading the `in` operator. For `BaseGraph`, it should return `True` if the the argument is the ID of a node in the graph, or if it is a pair of IDs corresponding to an edge in the graph. Otherwise, the method should return `False`.
- `__str__`: Returns a string representation of the graph. The nodes and edges are stringified in lexicographic order by ID(s).

The data representation for `BaseGraph` is up to you. Any representation is acceptable, as long as the interface is met.

`UndirectedGraph`

The `UndirectedGraph` class derives from `BaseGraph` and represents an undirected graph. Adding an edge between two nodes in an undirected graph should introduce edges in both directions between the two nodes. Looking up an edge using `edge()` should return the `Edge` object corresponding to the given direction. Retrieving all edges using `edges()` should result in a list that includes `Edge` objects in each direction for each pair of nodes connected by an edge.

We do not permit self-loops in `UndirectedGraph`. Your code should raise an exception if a self-loop is attempted to be added.

The distribution code includes the following methods:

- `__init__`: The constructor, which should delegate most of the work to the base-class constructor using `super()`.
- `degree`: Returns the degree of the node with the given ID. Raises a `GraphError` if no such node is in the graph.

You may add additional methods or override any base-class methods. Make sure that you don't repeat yourself! If the base-class method suffices, then do not override it. If you need to add additional functionality on top of what the base-class method does, then use `super()` to delegate the shared functionality to the base-class method.

`DirectedGraph`

The `DirectedGraph` class also derives from `BaseGraph`. It includes a constructor as well as `in_degree()` and `out_degree()` methods for retrieving the in- and out-degree of a node. As with `UndirectedGraph`, you may add methods or override methods from `BaseGraph`.

Unlike `UndirectedGraph`, we permit self-loops in `DirectedGraph`. Such an edge should add one to both the in-degree and out-degree of the adjacent node.

Testing

The doctests include only basic tests, and you should write further tests of your own. A handful of test cases are in `graph_test.py`, and you should add your tests to this file. Modify `graph_test.expect` as needed to contain the expected output. Use the `assert` construct to assert that a condition is true in your test cases.

As with testing any ADT, your test cases should only rely on the public interface for the `graph` module. Thus, your test cases should be able to run successfully with our implementation of `graph.py`.

Phase II: PageRank

Now that you have working graph ADTs, proceed to implement the PageRank algorithm itself. The formula you are to implement is as follows:

$$PR_{k+1}(u) = \frac{1-d}{N} + d\left(\sum_{v \in BL(u)} \frac{PR_k(v)}{v^+} + \sum_{w \text{ is a sink}} \frac{PR_k(w)}{N}\right)$$

Fill in the `pagerank()` function in `pagerank.py` to iteratively apply this formula on the given graph. The `num_iterations` argument specifies the number of iterations to run, and the `damping_factor` argument corresponds to the value of d in the formula above. Start with an initial uniform distribution of $PR_0(u) = 1/N$ for each node u, where N is the number of nodes in the graph.

The PageRank formula above specifies an *out-of-place* computation: the new values of $PR_{k+1}(u)$ should be computed in a separate data structure from the old values of $PR_k(u)$. This is to ensure that we get consistent results that are independent of the order in which the algorithm processes nodes and edges.

Read through the rest of `pagerank.py` in order to understand the command-line interface and how to run the doctests.

Since `pagerank()` is not part of the graph ADTs, it should only rely on the public interface of the latter. Any helper functions you write should go in `pagerank.py` and also rely solely on the public interface for the graph classes.

Performance

While performance is not a focus of this project, your implementation should be reasonably efficient. Running the PageRank algorithm with the default arguments on the `email-Eu-core.txt` graph should take no more than a minute on an average machine.

As an *optional* exercise, you may optimize your graph and PageRank implementations to be even more efficient. Some ways to do so are to avoid repeated computations, avoid nested loops where not necessary, and to trade memory for time by storing data that would otherwise have to be computed on the fly. Our solution takes about a second to run PageRank on the `email-Eu-core.txt` graph on a slow machine.

A larger graph dataset of Twitter connections is available as follows:

```
$ wget https://eecs490.github.io/project-pagerank/twitter_combined.tar.gz
```

As an optional challenge, try to improve the efficiency of your code so that you can compute its PageRanks within 1-2 minutes.

Grading

This project will not be hand graded, so your entire score will be determined by your results on the autograder. We will test your `graph.py` and `pagerank.py` both together as well as independently with our own implementations of the two modules, so make sure that you only use the public interfaces defined in the spec and starter code.

You are required to adhere to the coding practices in the course style guide. We will use the automated tools listed there to evaluate your code. You can run the style checks yourself as described in the guide.

Submission

Submit `graph.py`, `graph_test.py`, and `pagerank.py` to the autograder before the deadline.

Acknowledgments

The project was written by Tara Safavi and Amir Kamil in Fall 2017. The medium and large datasets for this project are from the Stanford Network Analysis Project (SNAP).