Homework 5

MPCS 51042 Fall 2021 – Python Programming

Due: Nov 16 2021, 5:00 pm CST

Introduction

- For this assignment, you will submit two modules:
  - `graph.py`: Contains a `Graph` class capable of performing a breadth-first search (BFS)
  - `test_graph.py`: Uses the `unittest` module to write a series of tests that verify your `Graph` class
- If you are unfamiliar with graph algorithms, these resources will bring you up to speed:
  - Chapter 22 from Cormen et al., *Introduction to Algorithms* (available here)
  - The "Graphs and BFS" video in the Week 7 module on Canvas (available here)

The `Graph` Class

- Your graph should be undirected. It can be implemented as either an adjacency list or an adjacency matrix. **Do not use the `graphlib` stdlib module nor any third-party modules.** Regardless of the underlying implementation, it should be able to recognize:
  - A node as a unique ID or object
  - An edge as a pair of nodes. Note that since the graph is undirected, the edge (u, v) is equivalent to the edge (v, u).
- Your `Graph` class should have at least the following methods.
  - The constructor takes a collection of edges, which are used to construct the graph. If given an empty collection, it constructs an empty graph.
  - `add_node` takes a single node and adds it to the graph, if not already present.
  - `add_edge` takes an edge and adds it to the graph, if not already present.
  - `bfs` takes a starting node and returns an iterable of **unique** (node, distance) pairs. It should return a pair for every other node in the graph, besides the starting node. The distance is the shortest distance between the starting node and the other node, discovered via a BFS.
  - `distance` takes two nodes and returns the distance of the shortest path between them.
  - `__iter__` returns an iterator that yields each node in the graph. The order is arbitrary.
  - `__getitem__` takes a node as a key. It returns a container with all the nodes that are adjacent to the given node.
  - `__contains__` determines if a given node is in the graph.

- You must annotate and document `Graph` as follows:
  - Add type annotations for all function arguments, function return values, and class attributes. It is optional to annotate local variables.
  - Document all functions and classes according to Google, NumPy, or Sphinx docstring style. Since you are using type annotations, you do not need include type information in the docstring.

Unit Tests

- Using the `unittest` module, create a set of unit tests that ensure your `Graph` class is working. In fact, if you want to follow "test driven development", you can write the tests prior to writing `Graph` and develop the latter incrementally.
- You must implement tests to verify at least these cases. Please use the graph shown in Figure 1 to devise test input and expected results.
  - A node can be added via `add_node`.
  - An edge can be added via `add_edge`.
  - The `in` operator works as expected.
  - The subscript operator `[]` works as expected
  - The `bfs` method works as expected. Ensure that distances are correct and that the (node, distance) pairs are unique.
  - The `distance` method works as expected.
  - The `Graph` can be correctly used as an iteratable.
- You must document your unit testing code according to Google, NumPy, or Sphinx docstring style. Type annotations are optional for the unit testing code.

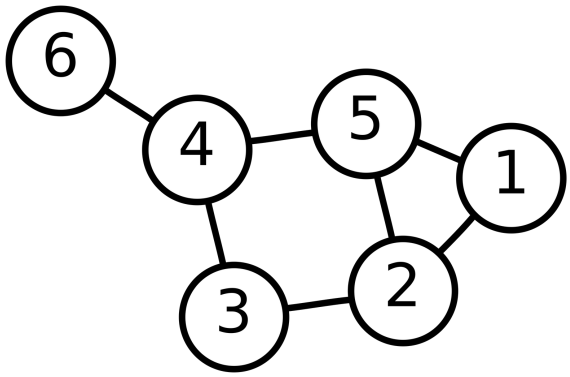

Figure 1: The graph to use for your test cases and expected results.