Module 11 Homework - Graphs

Overview

Implement data structure dependent methods for a non-directional, non-weighted graph in `AdjacencySetGraph` and `EdgeSetGraph` classes. Then implement a parent class `Graph` with methods that are independent of the underlying data structures:

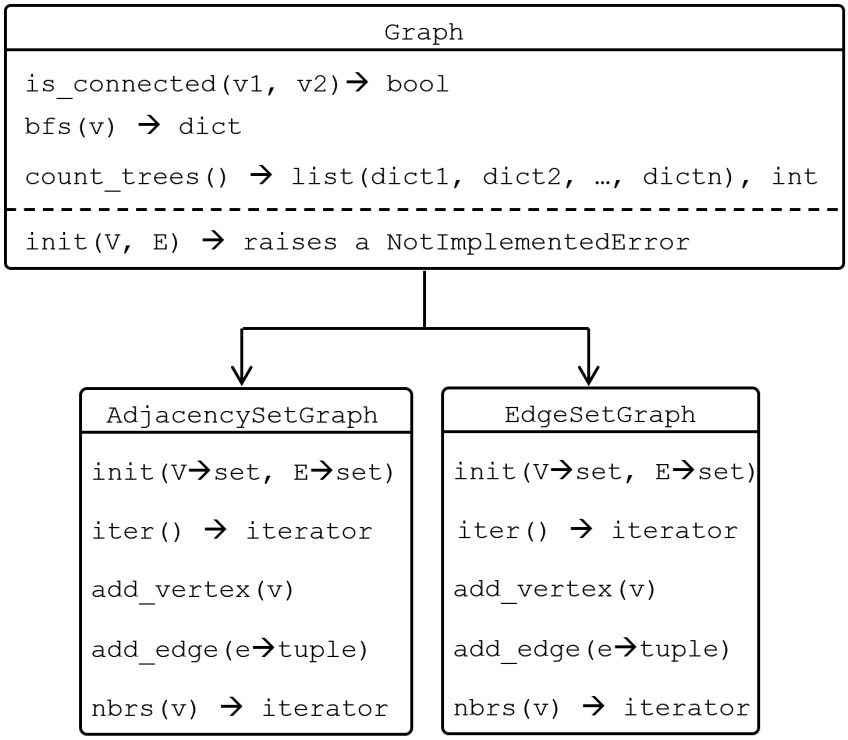

Figure 1: Class diagram showing expected output types for each method. AdjcacencySetGraph and EdgeSetGraph both inherit from Graph.

We use arrows in the figure above to denote typing of parameters (like in `add_edge`, which should expect a tuple) and return types (like `bfs()`, which should return a dictionary).

The `Graph` class is a convenient way to factor out common functionality, but should not be used on it's own - users should specify an `AdjacencySetGraph` or `EdgeSetGraph`. We expliciltly raise a `NotImplementedError` in `Graph.init` to ensure this.

AdjacencySetGraph

Store a dictionary of `vertex:set(vertices)` pairs, to allow fast iteration over neighbors.

- Inherits from `Graph`
- `__init__(V, E)` - initialize a graph with a set of vertices and a set of edges (we'll use tuples as edges, so E should be a set of tuples). Both parameters should be optional - a user should be able to use `asg = AdjacencySetGraph()` to create an empty graph.
- `__iter__()` - returns an iterator over all **vertices** in the graph
- `add_vertex(v)` - adds vertex to graph
- `add_edge(e)` - adds edge to graph

- `nbrs(v)` - returns an iterator over all neighbors of `v`.

EdgeSetGraph

As above, but use an edge set instead of an adjacency set (store a set of vertices and a set of edges intead of a dicitonary of `vertex:set(neighbors)` pairs).

Graph Methods

Methods whose implementations that do not depend on the underlying data structure (though the running times could differ) are factored out here.

- `is_connected(v1, v2)` - returns True (False) if there is (is not) a path between `v1` and `v2`.
- `bfs(v)` - returns a breadth-first search tree. You must return the tree in a dictionary, see chapter 21 in the textbook for more info.
  - *Be careful about efficiency here* - use an efficient queue.
- `count_trees()` - We will use "trees" to describe isolated structures within an overall graph, or "forest". For instance, here are 5 examples of forests with different amounts of trees:

```
1 tree      |  2 trees     |  3 trees   |  4 trees   |  5 trees
==========  |  ==========  |  ========  |  ========  |  =========
A--B  D--E  |  A--B  D--E  |  A  C  E   |  A C D E   |  A B C D E
| /      |  |  | /         |  |  |      |  |         |
C--------+  |  C           |  B  D      |  B         |
```

  `count_trees()` should return a list of trees in a graph as well as the number of distinct trees. To do this, you will need to iterate through all vertices in the graph, perform a breadth first search on them, and add any nodes in the resulting tree to a set of "visited" vertices to ignore.

```
>>> V = {'A', 'B', 'C', 'D', 'E'}
>>> E = {('A', 'B'), ('A', 'C'), ('B', 'C'), ('C', 'E'), ('D', 'E')}
>>> g = AdjacencySetGraph(V, E)
>>> trees, count = g.count_trees()
>>> print(trees)
[{A:None, B:A, C:A, E:C, D:E}]
>>> print(count)
1
>>> g2 = ... # construct the 2-tree graph above
>>> trees, count = g2.count_trees()
>>> print(trees)
[{A:None, B:A, C:A}, {D:None, E:D}]
>>> print(count)
2
```

What to unittest

A `unittest` should test a single piece of functionality. At minimum, write the following unittests (at least one test class per top level bullet point):

- Graph Construction - `init`
  - Test that you can add vertices and edges correctly. The easiest way to do this is to construct a set of vertices and edges, then test that you can:
    * construct a graph with the `init` method; e.g. `g = AdjacencySetGraph(V, E)`
    * iterate through a graph and get all the vertices you expect
    * iterate through the neighbors of every node and get the correct results
- Graph Construction - empty Graph
  - As above, but start with an empty graph, then explicilty call `g.add_vertex` and `g.add_edge`.
- `is_connected`
  - Similar to the test above, but, after constructing your graph, test that every vertex is connected to things it should be connected to and not connected to things it should not be.
  - Make sure to include tests with cycles, so you can ensure you don't fall into infinite loops
- `bfs`
  - **Do not test that you get an explicit tree back** - neighbors are stored in a set, so the order you visite them could change from run to run. Instead, take advantage of the fact that `bfs` is an optimization algorithm. It guarantees you visit each vertex in the minimum number of edges from the root, so test that: construct a dictionary of `vertex:distance_from_root` pairs you expect to get, call `bfs`, and test that every vertex in the resulting tree is the correct distance from the root.
- `count_trees` - as above, do not test that you get a specific tree back. Instead, test that each of your trees contains only the expected vertices (regardless of order) and that you have the expected number of trees.

Unittest Structring

Your unittests will be almost identical regardless of the underlying data structure - the only difference is whether you say `self.g1 = AdjacencySetGraph(V1, E1)` or `self.g1 = EdgeSetGraph(V1, E1)` in `setUp`. With this in mind, there are a few ways to structure our unittest file:

- Method 1 - Bad - Test for EdgeSet and AdjacencySet classes within the same test class, e.g.:

```
class TestAllTheGraphs(unittest.TestCase):
    def test_bfs(self):
        # graph setup
        for GraphDS in [AdjacencySetGraph, EdgeSetGraph]:
            g = GraphDS(V, E)
            # tests here
```

  The benefit here is you only write the tests once - no duplicate code! The downside is you do not know if any failures occur during just one or both data structures, making debugging more difficult.

- Method 2 - Better - Make an explicit `TestAdjacencySetGraph` and `TestEdgeSetGraph` classes that contain the same code. Gives useful debugging information, but you have a lot of duplicate code:

```
class TestAdjacencySetGraph(unittest.TestCase):
    def setUp(self):
        V1 = {...}
        E1 = {...}
        self.g1 = AdjacencySetGraph(V1, E1)
        # repeat for other graphs
```

```
    def test_bfs(self):
        # test with self.g1, self.g2, ... here

    # All other tests


class TestEdgeSetGraph(unittest.TestCase):
    def setUp(self):
        V1 = {...}
        E1 = {...}
        self.g1 = EdgeSetGraph(V1, E1)
        # repeat for other graphs

    def test_bfs(self):
         # test with self.g1, self.g2, ... here AGAIN

    # All other tests AGAIN
```

- Method 3 - Best - Factor out similar code into parent class, then use multiple inheritance to create a class of tests for AdjacencySetGraphs and a class for EdgeSetGraphs.This is the best method, since we only have to write each test once *and* we can see whether any failures happen with one or both data structures:

```
class GraphTestFactory:
    # Use setUp to create graphs for testing
    def setUp(self, GraphDS):
        # Setup vertices and edges for a few graphs to trest with
        self.g1 = GraphDS(V1, E1)
        self.g2 = GraphdS(V2, E2)
        ... # as many graphs as you need for testing

    # All your other tests go here

# Overload setUp method to pass correct data structure
class TestAdjacency(unittest.TestCase, GraphTestFactory):
    def setUp(self):
        return GraphTestFactory.setUp(GraphDS = AdjacencySetGraph)

class TestEdge(unittest.TestCase, GraphTestFactory):
    def setUp(self):
        return GraphTestFactory.setUp(GraphDS = EdgeSetGraph)
```

Whichever method you choose (*hint: choose Method 3 for full credit*), you'll also want one more test class to make sure the Graph class behaves as expected - specifically, that it raises a NotImplementedError with a helpful string to the user when you try to directly initialize an object of that type.

Imports

No imports allowed on this assignment, with the following exceptions:

- Any modules you have written yourself
- `queue.Queue` for efficient queues (you can also use a linked list you have written yourself). Be careful with this class - by default, looping over these objects pause indefinitely (instead of terminating) when the collection becomes empty.
- `typing` - this is not required, but some students have requested it
- For testing only (do not use these for functionality in any other classes/algorithms):
  - unittest
  - random

Submission

At a minimum, submit the following files:

- `Graph.py`
  - `class Graph`
  - `class AdjacencySetGraph`
  - `class EdgeSetGraph`
- `TestGraph.py`
  - see above for information on structuring your tests

Students must submit individually by 11:59 PM EST Tuesday.

Grading

This assignment is 100% manually graded.