Module 11 Homework - Graphs

We can separate the abstract interface a user should work with (e.g. a "Graph") from the underlying data structure using object-oriented programming - this is an extremely helpful design pattern, since we can use the abstract interface as a sort of short-hand documentation for what a user should be able to do while leaving data structure specific operations to subclasses.

Here, we create a Graph class which defines the **interface** (methods and their corresponding parameters/return values) for its subclasses, with the subclasseses implementing more data-structure specific operations.

Note: Do not use any built-in modules (e.g. the `collections` module) in this assignment. If you want access to another data structure, write it yourself.

Part 1 - `class Graph`

Create a class `Graph` for an undirected graph that defines the full interface users should be able to use. The actual implementation of several attributes will be deferred to subclasses, but we want templates for them in the parent class.

Deferred Methods

These methods should be defined in the Graph class, but their only behavior is to raise a `NotImplementedError` - programs should fail if these methods have not been overridden by the corresponding data structure.

- `init(V, E)` - initializes a graph with a set of vertices `V` and edges `E`
- `add_vertex(v)` - adds vertex `v`
- `remove_vertex(v)` - removes vertex `v`
- `add_edge(u, v, w)` - adds an edge between `u` and `v` with optional weight `w`
- `remove_edge(u, v, w)` - adds an edge between `u` and `v` with optional weight `w`
- `neighbors(v)` - returns an iterator over all neighbors of `v`.
- `weight(u, v)` - returns the weight of the edge connecting `u` and `v`. If multiple edges connect `u` and `v`, return the smallest weight.

Implemented Methods

Some graph algorithms are data-structure agnostic, so they can be implemented in the parent `Graph` class for subclasses to inherit. You should only use methods in the public interface of the `Graph` class (the methods defined here and above) to do so.

All methods below return a dictionary-tree representing traversal (see "depth-first search tree" in the textbook for an example) using the corresponding algorithm starting at node `v`. `dijkstra` additionally return a dictionary of `vertex:weight` pairs, where the weight is the weight of the path to reach that vertex in that algorithm.

- `bfs(v)` - breadth-first search
- `dfs(v)` - depth-first search
- `dijkstra(v)` - dijkstra's algorithm
  - returns `(tree, D)`
- `primm(v)` - primm's algorithm

Part 2 - Data Strucutres

Implement Edge and Adjacency set graphs that inherit from the parent `Graph` class. These classes should override all methods in the "deffered methods" section above, and should implement no other methods.

- `class AdjacencySetGraph`
- `class EdgeSetGraph`

Part 3 - Graph Applications

Inherit from one of the `Graph` classes implemented in Part 2 to solve the following problems. Choose the data structure that is better for the given problem, and leave a comment about why you selected the data structure you did.

Application 1 - `FlightMap`

Design a class `FlightMap` which initializes with a set of cities (vertices) and available flights (edges). Add a function `reachable`:

- `reachable(city, n)` - returns a set of all cities reachable from `city` with `n` or fewer flights. Include `city` in this set.

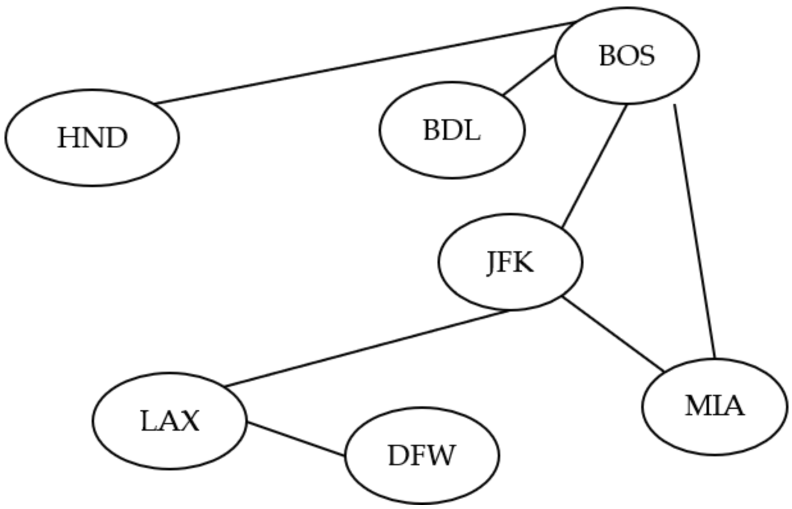

Figure 1: An example flight map

```
>>> cities = {'BOS', 'BDL', 'HND', 'JFK', 'MIA', 'LAX', 'DFW'}
>>> flights = {('BOS', 'HND'), ('BOS', 'BDL'), ('BOS', 'JFK'),
('BOS', 'MIA'), ('JFK', 'LAX'), ('JFK', 'MIA'), ('LAX', 'DFW')}
>>> fm = FlightMap(cities, flights)
>>> fm.reachable('BOS', 1)
{'MIA', 'JFK', 'BOS', 'BDL', 'HND'}
>>> 'DFW' in fm.reachable('BOS', 2)
False
>>> 'DFW' in fm.reachable('BOS', 3)
True
```

Application 2 - `SnowMap`

Design a class `SnowMap` which initializes with a set of cities (vertices) and roads (edges). Add a function `plow_city` that returns a map (dictionary-tree) that connects all cities such that a certain city (say, Hartford) has a clear path to any other city with the fewest number of miles traveled (e.g. you should be able to get from Hartford to Storrs, Hartford to New Haven, *or* Hartford to Stamford in the fewest miles possible).

- `plow_from(city)` - finds a path that connects cities as above. Returns a two-tuple:
  - dictionary of `city: distance_from_start` pairs
  - dictionary-tree giving a map connecting all cities with the minimum distance from the starting `city` specified

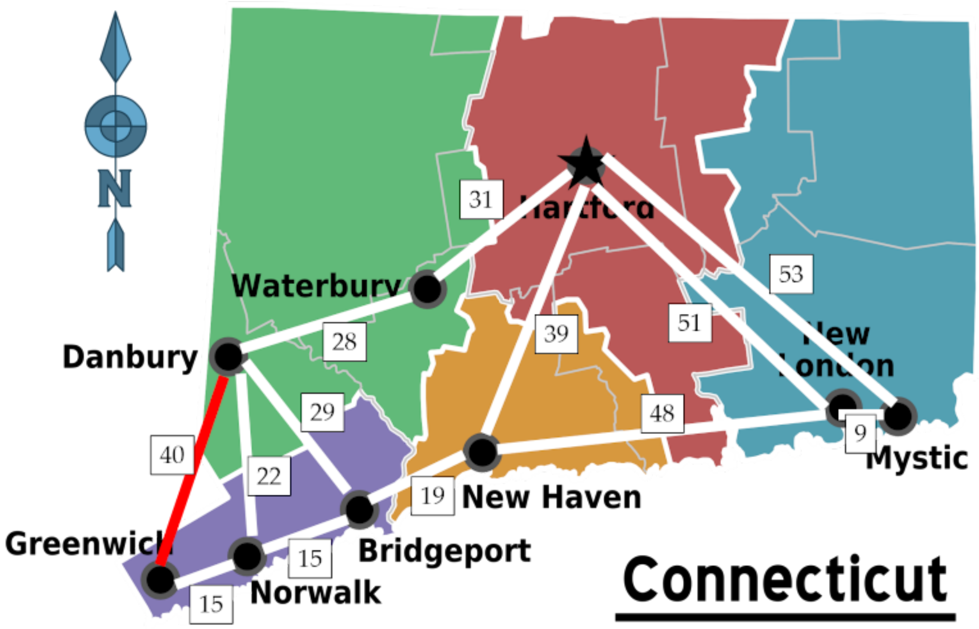

Figure 2: An example snow plowing map. Numbers give distance in miles. Image adapted from wiki commons.

```
>>> cities = {"Hartford", "Waterbury", "Danbury", "Greenwich", "Norwalk", "Bridgeport",
 "New Haven", "New London", "Mystic"}
>>> roads = {("Hartford", "Waterbury", 31), ("Hartford", "New Haven", 39),
("Hartford", "New London", 51), ("Hartford", "Mystic", 53), ("Waterbury", "Danbury", 28),
("New Haven", "Bridgeport", 19), ("New Haven", "New London", 48),
("New London", "Mystic", 9), ("Danbury", "Greenwich", 40), ("Danbury", "Norwalk", 22),
("Danbury", "Bridgeport", 29), ("Bridgeport", "Norwalk", 15), ("Norwalk", "Greenwich", 15)}
>>> sm = SnowMap(cities, roads)
>>> D, tree = sm.plow_from("Hartford")
>>> for city, miles in D.items(): print(f"{city:10}{miles:10}")
Hartford           0
Waterbury         31
New Haven         39
New London        51
Mystic            53
Bridgeport        58
Danbury           59
Norwalk           73
Greenwich         88
```

Submission

At a minimum, submit the following files:

- `Graph.py`
- `FlightMap.py`
- `SnowMap.py`

It may be helpful to include other files - if you import Stacks, Queues, and Priority Queues from a file called `ADTS.py`, for instance, include `ADTS.py` in your submission.

Students must submit to Mimir **individually** by the due date (typically, the second Wednesday after this module opens at 11:59 pm EST) to receive credit.

Grading

Some of the grading for this assignment is manual, and will be graded after the deadline.

Note: We will be manually checking the Edge and Adjacency set implementations in Part 2. Obvious bad-faith attempts (e.g. using the same code for both because only one of them passes the test cases) is considered academic misconduct.

Automated

- 45 - Part 1 (some of these require Part 2 to be finished to pass)
- 25 - Part 2
- 20 - Part 3

Manual

- 10 points - choose correct data structure (and explain that choice!) for Part 3.

Feedback

If you have any feedback on this assignment, please leave it here.