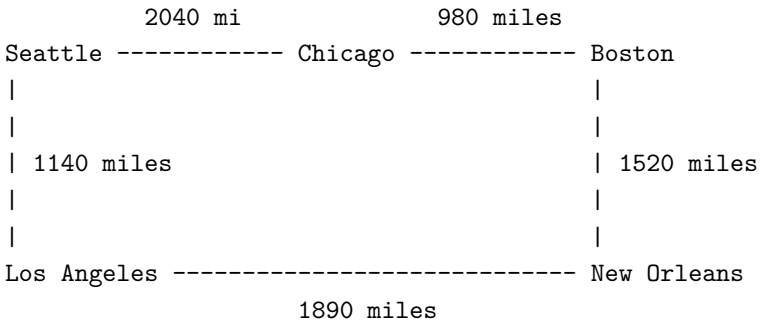
Module 11 Homework - Graphs

1) Implement a non-directional edge-weighted graph.
2) Add graph traversal algorithms to solve a few types of optimal path problems.

As always, use TDD here - write unittests first, then implement the methods in your `Graph` class.

Part 0 - Unittests

It will be helpful to have an image in mind when writing your tests. Look up 5 cities, and sketch an ascii-art graph of those 5, using distances as edge weights.



Which and how many edges you draw aren't terribly important, but make sure your map is connected - it should be possible to reach any vertex from any vertex. However, you should create your own map for testing - do not use any of the cities or the shape (a ring) given above.

It will be helpful to store this graph in a `setUp()` method in your unittest classes, so you only have to define it once (docs).

Functionality

At minimum, support:

- `add_vertex(v)`
- `remove_vertex(v)`
- `add_edge(u, v, wt)`
- `remove_edge(u, v, wt)`
- `nbrs(v)` - returns an iterator over neighbors of `v`:

```
>>> for nbr in g.nbrs('Boston'):
>>>     print(nbr)
Chicago
New Orleans
```

You don't need to worry about multiple edges between the same two vertices (e.g. two edges between `A` and `B` with different weights) - we're only considering simple graphs.

Use a data structure that will allow you to efficiently solve the problems below.

Graph Traversal Algorithms

Implement algorithms to solve each of the problems below. The unittest skeleton code includes `# TODO` comments for you to denote which algorithm you are using for each problem and why. Each algorithm takes a `city` as an input:

1. `fewest_flights(city)` - finds how to get from `city` to **any other city in the graph** with the **fewest number of flights** - this is for busy users who need to get from city to city quickly, and cannot waste time in extraneous layovers.
2. `shortest_path(city)` - finds how to get from `city` to **any other city in the graph** with the **fewest number of miles travelled** - this is for environmentally concious users who want to use the least amount of fuel to get between two cities.
3. `minimum_salt(city)` - connects `city` to **every other city in the graph** with the **fewest total number of miles** - this tree would allow us to keep all cities connected in the winter with the least amount of salt used on roads.

Each method should return two items - a dictionary-tree showing traversal order, and a dictionary of `vertex:distance` pairs that describes the relevant distance used to add that vertex to the previous tree (the distance from source in algorithm 2, and the length of the edge in algorithm 3)

What to unittest

When testing these algorithms, **don't test that they give a fixed path** - there may be multiple paths that have the same optimal soluiton.

Instead, test what the algorithms guarantee. For instance, for part 3, the algorithm you use should guarantee that you connect every city with the fewest total number of miles, so your test should look at the number of miles in the returned tree:

```
1) path_tree, vertex_weights = minimum_salt(city)
2) Sum up all weights
3) Assert sum is correct value
```

For part 2, you can test that the distance to every city is correct (not that it returns a specific path).

For part 1, you can test the number of flights to get to every city is correct.

Submission

At a minimum, submit the following files:

- `Graph.py`
- `TestGraph.py`

Students must submit individually by 11:59 PM EST **Friday, April 28**. This is longer than the normal timeline, but **do not wait until the deadline to start** - you should finish question 1 (including the unittests) over the weekend, then add the solutions to questions 2 and 3 the week of the 28th.

Grading

This assignment is 100% manually graded.