Graphs

Introduction

A graph is made of two parts: vertices (or nodes) and edges. Vertices hold the keys of a graph and edges connect two vertices.

In this lab, we are going to write Python code that uses different implementations of the `Graph` ADT. In particular, you will implement the `Graph` ADT using adjacency matrix and lists. The former is good for dense graphs and the latter for sparse ones.

Objectives

The purpose of this lab is to help you:

- Gain further familiarity with class inheritance
- Understand the graph ADT
- Learn about different implementations of graph ADT.
- Learn algorithms for finding paths in graphs.

Accompanied code

This lab includes the following classes:

`Graph` represents the abstract methods of `Graph` ADT. These methods are needed in every implementation that you choose. We have provided you with the code for this.

```
class Graph:
    def addVertex(self, vert):
        #Add a vertex with key k to the vertex set.
        raise NotImplemented

    def addEdge(self, fromVert, toVert):
        #Add a directed edge from u to v.
        raise NotImplemented

    def neighbors(self):
        #Return an iterable collection of the keys of all
        #vertices adjacent to the vertex with key v.
        raise NotImplemented

    def removeEdge(self, u, v):
```

```
        #Remove the edge from vertex u to v from graph.
        raise NotImplemented

    def removeVertex(self, v):
        #Remove the vertex v from the graph as well as any edges
        #incident to v.
        raise NotImplemented
```

`SimpleGraph` represents a simple directed graph. This class inherits from `Graph` and as such implements its methods. This code is given to you as well.

`SimpleGraph` is a very simple implementation of the `Graph` ADT. The goal is to describe a graph using code that is as close as possible to the mathematical notion of a graph. That is, we want to build the graph using just a set `V` of vertices, and a collection `E` of edges (ordered pairs of elements of `V`).

For example, we should be able to write the following simple code to make a graph:

```
V = {1, 2, 3, 4}
E = {(1,2), (1,3), (1,4), (2,3), (2,1)}
G = SimpleGraph(V, E)
```

The graph will be simple. This means that there will be at most one edge for any given ordered pair. There will be no self-loops. The graph will be directed. This means that the edge `(a, b)` is different from the edge `(b, a)`.

```
class SimpleGraph(Graph):
    def __init__(self, V, E):
        self._V = set()
        self._E = set()
        for v in V: self.addVertex(v)
        for u,v in E: self.addEdge(u,v)

    def vertices(self):
        return iter(self._V)

    def edges(self):
        return iter(self._E)

    def addVertex(self, v):
        self._V.add(v)

    def addEdge(self, u, v):
        self._E.add((u,v))

    def neighbors(self, v):
```

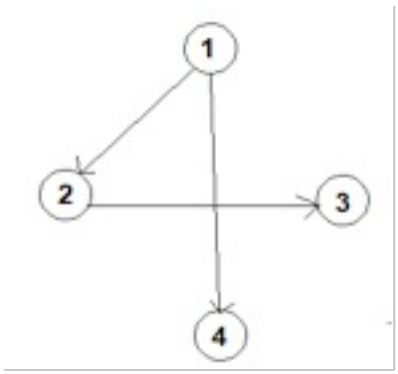
```
        return (w for u,w in self._E if u == v)

    def removeEdge(self, u, v):
        self._E.remove((u,v))

    def removeVertex(self, v):
        for neighbor in list(self.neighbors(v)):
            self.removeEdge(v, neighbor)
        self._V.remove(v)
```

Part 1 (in-lab): Implementing a very simple undirected graph

We start this part by showing you a few examples that use the `SimpleGraph` class. These examples will introduce to you the provided functionality and help you implementing the `SimpleUGraph`.

```
g1 = SimpleGraph({1,2,3,4}, {(1,2), (1,4), (2,3)})
print(list(g1.vertices()))
print(list(g1.edges()))
g1.removeVertex(1)
print(list(g1.vertices()))
print(list(g1.edges()))
```

The previous code snippet creates a simple graph g1 with 4 vertices and 3 edges. Here is how `g1` looks:



Note that `g1` is a directed graph. This means that the tuple `(1, 2)` which represents an edge between vertices `1` and `2`, is different than the tuple `(2, 1)` which represents an edge between the vertices `2` and `1` (we don't have such an edge in `g1`).

Next, the code prints the vertices of `g1` as a list and the edges as a list as well. In the fourth

line we remove vertex number `1`. That is, removing the vertex and all edges connected to it. To see the effect of this removal, we print again the vertices of `g1` then the edges. Please run this code and see what happens.

Note: In the `removeVertex` method of `SimpleGraph`, we have a loop that iterates over the neighbors of a vertex `v`. The `neighbors` method returns a generator for all those neighbors. However, the code converts this generator into a list even though we can use generator in a `for` loop. Try to remove the list conversion and see what happens.

In this part, you are required to define a class `SimpleUGraph`. This class represents a simple undirected graph. In such a graph, the vertices in every edge have no order. That is, adding an edge between `u` and `v` means that this edge exists for `v` and `u` as well. In this sense, undirected graph is a special kind of directed graph where every edge exists in both directions. In particular, that means whenever you add a new edge between `u` and `v` you have to take care of adding automatically and edge between `v` and `u`. Following is the header line of the `SimpleUGraph` class.

```
class SimpleUGraph(SimpleGraph):
    ## override addEdge and removeEdge
```

After defining the `SimpleUGraph` run the following code and see how the different functions work!

```
g1 = SimpleUGraph({1,2,3,4}, {(1,2), (1,4), (2,3)})
print(list(g1.vertices()))
print(list(g1.edges()))
g1.removeVertex(1)
print(list(g1.vertices()))
print(list(g1.edges()))
```

Part 2: The shortest path and the degrees of Kevin Bacon problem

The 'Degrees of Kevin Bacon' problem consists of determining exactly how far away someone's work is from being connected to Kevin Bacon. We can formulate the problem as stating that any actor or other production staff works with Kevin Bacon, they will have a Bacon Number of 1. Anyone who works with those people will have a Bacon Number of 2, and so on. The smallest possible number is taken, so if you end up working with Kevin Bacon later on, or if someone you've worked with does, it can lower your score. So given the name of an actor,

the Kevin Bacon game is to find some alternating sequence of actors that lead back to Kevin Bacon.

This problem is an example of path finding in a graph. In particular, given a graph `g` and a vertex `v` we want to find the paths from `v` to all vertices that can be reached from `v`. We will start by writing a method that finds just the list of vertices that can be reached from `v` then revise that method to get paths from `v` to all reachable vertices.

STEP 1: Write a method `dfs` (depth-first search) for the basic class `Graph` that receives a vertex `v` and returns a list of all vertices that can be visited from `v`.

Note: Make sure that you do not add the same vertex to this list more than once, or else your search may never terminate!

STEP 2: Revise the `dfs` method, so as to return paths from `v` to all reachable vertices, and run the `dfs` method on a graph where the vertices are actors' names and one of those names is Bacon.

The dfs will return to you all paths from Bacon to all other actors, in the form of a dictionary. For example:

```
G = SimpleGraph({'A','B','C','D','E'}, {('A','B'), ('A','C'), ('B','E'),
('E','D')})
t = G.dfs('A')
print(t)
```

The result is a dictionary that shows the shortest paths from `A` to all other vertices.

```
{'A': None,
'C': 'A',
'B': 'A',
'E': 'B',
'D': 'E'}
```

For example, the path from `A` to `D` is `A -> B -> E -> D`. This is reconstructed backwards: `D` takes you to `E`, `E` takes you to `B`, and `B` takes you to `A`.

STEP 3: Now reconstruct paths from this dictionary, as a list of nodes. This is a function that takes the dictionary for some node `X` and a node `Y`, and returns the path from `X` to `Y`. For example:

```
G = SimpleGraph({'A','B','C','D','E'}, {('A','B'), ('A','C'),
('B','E'), ('E','D')})
t = G.dfs('A')
p = getPath(t, 'D')
print(p) # prints ['A','B','E','D']
```

STEP 4: Now put it all together into a method, `findPath`, that receives two vertices and returns a list that defines the path between these two vertices or `None` otherwise. This method needs to call the `dfs` method and use its returned dictionary.

Part 3: Adjacency lists

There can be `O(n^2)` edges in the graph, whereas there are only `O(n)` neighbors for any vertex. It's expensive to look through all the edges any time we want to find neighbors. To avoid this, we will create another map implementation, using adjacency lists: `AdjacencyListGraph`. It looks A LOT like the simple graph, but instead of storing a set of edges in the graph, it stores a MAPPING from node to the list of its neighbors. This makes the `neighbors` method much simpler: we just look up the neighbors. But it makes the `edges` method more complicated: we need to traverse the map of neighbors to collect all the edges.

In this part, we will implement the `AdjacencyListGraph` as a subclass of `Graph`. We will also need to change `__init__`, `addEdge`, and `removeEdge` accordingly. But the code to find paths should not change!

You will also implement the undirected version `AdjacencyListUGraph`, as subclass of `AdjacencyListGraph`, again changing only the `addEdge` and `removeEdge` methods.

To let you check that the adjacency list is just another more efficient implementation of the same thing, we provide you a comparison method in the `Graph` class. This method will compare between two different implementations of the same graph. The function will return `True` if the two objects represent the same graph that was implemented differently and `False` otherwise. Here is the comparison method that does this work. To use this method you need to put it in the Graph class.

```
def __eq__(self, other):
    return set(self.edges()) == set(other.edges())
```

Notice that this method works when we have the same exact graph but with different implementation. Follows is a code that demonstrates the usage of this method:

```
g1 = SimpleGraph({1,2,3,4}, {(1,2), (1,4), (2,3)})
g2 = AdjacencyListGraph({1,2,3,4}, {(1,2),(2,3),(1,4)})
print(g1 == g2)
```

You should be able to find paths in `AdjacencyListGraph` exactly like in `SimpleGraph`, but faster.

Here is an example that shows the difference in time when we find all paths between one vertex to all other vertices. This example defines `1000` vertices and about `100000` edges (when we define an edge that already exists it will not be added). Then we create two graphs `g3` and `g4` of the same vertices and edges but with different implementations. Calling the `dfs` method will return a dictionary that provides us with all paths between the first vertex to all other vertices. The code measures the time for finding paths. In my run you can see the difference in time.

```
v=set()
e=set()
for x in range(1000):
    v.add(x)

for y in range(100000):
    i = random.randint(0,999)
    j = random.randint(0, 999)
    e.add((i,j))

print(len(v))
print(len(e))

g3 = SimpleGraph(v, e)
g4 = AdjacencySetGraph(v, e)

t3s = time.time()

print(g3.dfs(1))

t3e = time.time()
t4s = time.time()

print(g4.dfs(1))

t4e = time.time()

print(t3e-t3s)
print(t4e-t4s)
```

```
1000
95197
{1: None, 927: 1, 130: 927, 877: 130, 942: 877, 751: 942, 508: 751, 485: 508, 499: 485,
{1: None, 504: 1, 509: 504, 494: 509, 506: 494, 508: 506, 125: 508, 502: 125, 994: 502,
6.679204702377319
0.030982017517089844
```

Part 4: Adjacency matrix

A **dense graph** is a graph which has `O(n)` edges/neighbors per vertex. The opposite, a graph with `O(1)` edges/neighbors per vertex, is a **sparse graph**.

Whether a graph is expected to be dense or sparse usually depends on the application where it's to be used. The rule of thumb is to ask whether the number of neighbors is expected to grow as the number of vertices grows. If it does, the graph is dense. For example, if our town continues to expand and build more condos on empty land, the number of our physical neighbors does not continue to grow; however, the number of people that we pass as we walk and drive around town will continue to grow.

For path finding in dense graphs, the most efficient implementation is with adjacency matrices. An adjacency matrix `M` is an `n*n` boolean matrix, where `n` is the number of vertices. Given vertices `i` and `j`, the cell `M[i][j]` contains `1` if the corresponding graph contains the edge `(i,j)`, and `0` otherwise. For example, here is the matrix for the graph

```
({1,2,3,4}, {(1,2), (1,4), (2,3)}):

0 1 0 1
0 0 1 0
0 0 0 0
0 0 0 0
```

Note that the index of the node `4` here is actually `3`, etc. `{1,2,3,4}` are node labels rather than indices.

In this part you will be developing a class `AdjacencyMatrixGraph`. In this graph, every vertex is assigned an index from `i` to `j` (this is the index of the vertex in the vertices list), and the edge information is stored in the adjacency matrix. To find out if the graph has an edge from vertex with index `i` to the vertex with index `j`, we simply check if the cell `[i,j]` of the adjacency matrix is `1` or `0`. To get all the neighbors of vertex `i`, we simply take all the entries in row `i` of the matrix whose value is `1`.

Your job in this part is to implement this class as another subclass of Graph, so all of the Graph methods work for the AdjacencyMatrixGraph class, for example:

```
g1 = AdjacencyMatrixGraph([1,2,3,4], {(1,2), (1,4), (2,3)})
print(list(g1.vertices()))
print(list(g1.edges()))
g1.removeVertex(1)
print(list(g1.vertices()))
print(list(g1.edges()))
```

Finish by making sure that `dfs` works for `AdjacencyMatrixGraphs`!

Note: don't forget that you can still use the magic method `__eq__` to compare your graphs, for debugging.

Parts 5 and 6:

Will be released next week.