Part I

Labs

Lab 1

Breadth-First Search and the Kevin Bacon Problem

Lab Objective: *Learn to store graphs as adjacency dictionaries, implement a breadth-first search, learn to use NetworkX, and use the parlor game "the Six Degrees of Kevin Bacon" as an application to graph theory.*

Introduction

Graph theory has many practical applications. For example, graphs may represent a social network, a communication network, or the internet. Being able to construct, analyze, and search graphs yields critical insights into the system that the graph represents.

Graphs in Python

Graphs can be stored using various kinds of data structures. In previous labs, we stored trees as linked lists. For non-tree graphs, perhaps the most common data structure is an adjacency matrix, where each row of the matrix corresponds to a node in the graph and the entries of the row indicate which other nodes the current node is connected to. For a full treatment fo adjacency matrices, see chapter 2 of the Volume II text.

Another comon graph data structure is an *adjacency dictionary*, a Python dictionary with a key for each node in the graph. The dictionary values are lists containing the nodes that are connected to the key. For example, the following is an adjacency dictionary for the graph in Figure 1.1:

```
>>> adjacency_dictionary = {'A':['B', 'C', 'D', 'E'], 'B':['A', 'C'],
'C':['B', 'A', 'D'], 'D':['A', 'C'], 'E':['A']}

# The nodes are stored as the dictionary keys.
>>> print(adjacency_dictionary.keys())
['A', 'C', 'B', 'E', 'D']
```

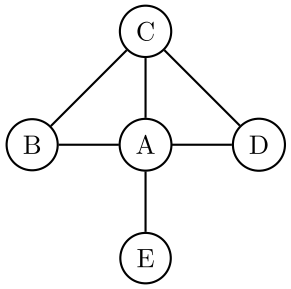

Figure 1.1: A simple graph with five vertices.

Problem 1. Implement the `__str__` method in the provided `Graph` class. Print each node in the graph in sorted order, followed by a sorted list of the neighboring nodes separated by semicolons.

(Hint: consider using the `join` method for strings.)

```
>>> my_dictionary = {'A':['C', 'B'], 'C':['A'], 'B':['A']}
>>> graph = Graph(my_dictionary)
>>> print(graph)
A: B; C
B: A
C: A
```

Breadth-First Search

Many graph theory problems are solved by finding the shortest path between two nodes in the graph. To find the shortest path, we need a way to strategically search the graph. Two of the most common searches are depth-first search (DFS) and breadth-first search (BFS). In this lab, we focus on BFS. For details on DFS, see the chapter 2 of the Volume II text.

A BFS traverses a graph as follows: begin at a starting node. If the starting node is not the target node, explore each of the starting node's neighbors. If none of the neighbors are the target, explore the neighbors of the starting node's neighbors. If none of those neighbors are the target, explore each of their neighbors. Continue the process until the target is found.

As an example, we will do a programmatic BFS on the graph in Figure 1.1 one step at a time. Suppose that we start at node C and we are searching for node E.

```
# Start at node C
>>> start = 'C'
>>> current = start

# The current node is not the target, so check its neighbors
>>> adjacency_dictionary[current]
['B', 'A', 'D']
```

```
# None of these are E, so go to the first neighbor, B
>>> current = adjacency_dictionary[start][0]
>>> adjacency_dictionary[current]
['A', 'C']

# None of these are E either, so move to the next neighbor
# of the starting node, which is A
>>> current = adjacency_dictionary[start][1]
>>> adjacency_dictionary[current]
['B', 'C', 'D', 'E']

# The last entry of this list is our target node, and the search terminates.
```

You may have noticed that some problems in the previous approach that would arise in a more complicated graph. For example, what prevents us from entering a cycle? How can we algorithmically determine which nodes to visit as we explore deeper into the graph?

Implementing Breadth-First Search

We solve these problems using a queue. Recall that a *queue* is a type of limited-access list. Data is inserted to the back of the queue, but removed from the front. Refer to the end of the Data Structures I lab for more details.

A queue is helpful in a BFS to keep track of the order in which we will visit the nodes. At each level of the search, we add the neighbors of the current node to the queue. The `collections` module in the Python standard library has a `deque` object that we will use as our queue.

```
# Import the deque object and start at node C
>>> from collections import deque
>>> current = 'C'

# The current node is not the target, so add its neighbors to the queue.
>>> visit_queue = deque()
>>> for neighbor in adjacency_dictionary[current]:
...     visit_queue.append(neighbor)
...
>>> print(visit_queue)
deque(['B', 'A', 'D'])

# Move to the next node by removing from the front of the queue.
>>> current = visit_queue.popleft()
>>> print(current)
B
>>> print(visit_queue)
deque(['A', 'D'])

# This isn't the node we're looking for, but we may want to explore its
# neighbors later. They should be explored after the other neighbors
# of the first node, so add them to the end of the queue.
>>> for neighbor in adjacency_dictionary[current]:
...     visit_queue.append(neighbor)
...
>>> print(visit_queue)
deque(['A', 'D', 'A', 'C'])
```

We have arrived at a new problem. The nodes A and C were added to the queue to be visited, even though C has already been visited and A is next in line. We can prevent these nodes from being added to the queue again by keeping a running list of nodes that don't need to be added. However, not all such nodes are created equal. For example, C has already been visited, while A is in line to be visited, but hasn't actually been visited yet. We distinguish between these two types of nodes with two additional lists: `visited` and `marked`.

Nodes that are in `visited` have already been visited by the algorithm. If we make it a Python `list`, `visited` will also keep track of the order in which we visited the nodes in the graph. Nodes that are in `marked` have been added to the queue, but have not been visited yet. Since the ordering doesn't matter for which nodes are marked, we make `marked` a Python `set`. If we check both of these groups at each step of the algorithm, the previous problems are avoided.

```
>>> current = 'C'
>>> marked = set()
>>> visited = list()
>>> visit_queue = deque()

# Visit the start node C.
>>> visited.append(current)

# Add the neighbors of C to the queue.
>>> for neighbor in adjacency_dictionary[current]:
...     visit_queue.append(neighbor)
...     # Since each neighbor will be visited, add them to marked as well.
...     marked.add(neighbor)
...
# Move to the next node by removing from the front of the queue.
>>> current = visit_queue.popleft()
>>> print(current)
B
>>> print(visit_queue)
['A', 'D']

# Visit B. Since it isn't the target, add B's neighbors to the queue.
>>> visited.append(current)
>>> for neighbor in adjacency_dictionary[current]:
...     visit_queue.append(neighbor)
...     marked.add(neighbor)
...

# Since C is visited and A is in marked, the queue is unchanged.
>>> print(visit_queue)
deque(['A', 'D'])
```

Problem 2. Implement the `search` method in the `Graph` class using a BFS. Start from a specified node and proceed until all nodes in the graph have been visited. Return the list of visited nodes. If the starting node is not in the adjacency dictionary, raise a `ValueError`.

Problem 3. (Optional) Create a new method called `DFS` in the `Graph` class that mimics the `search` method, but uses a DFS instead of a BFS.

(Hint: this can be done by changing a single line of the BFS code.)

Shortest Path

In a BFS, as few neighborhoods are explored as possible before finding the target. Therefore, the path taken to get to the target must be the shortest path.

Examine again the graph in Figure 1.1. The shortest path from C to E is start at C, go to A, and end at E. During a BFS, A is visited because it is one of C's neighbors, and E is visited because it is one of A's neighbors. If we knew programmatically that A was the node that visited E, and that C was the node that visited A, we could retrace our steps to reconstruct the search path.

To implement this idea, initialize a new dictionary before starting the BFS. When a node is added to the visit queue, add a key-value pair to the dictionary where the key is the node that is visited and the value is the node that is visiting it. When the target node is found, step back through the dictionary until arriving at the starting node, recording each step.

Problem 4. Implement the `shortest_path` method in the `Graph` class using a BFS. Begin at a specified starting node and proceed until a specified target is found. Return a list containing the node values in the shortest parth from the start to the target (including the endpoints). If either of the inputs are not in the adjacency graph, raise a `ValueError`.

Network X

NetworkX is a Python package for creating, manipulating, and exploring large graphs. It contains a graph object constructor as well as methods for adding nodes and edges to the graph. It also has methods for recovering information about the graph and its structure.

```
# Create a new graph object using networkX
>>> import networkx as nx
>>> nx_graph = nx.Graph()

# There are several ways to add nodes and edges to the graph.
# One is to use the add_edge method, which creates new edge
# and node objects as needed, ignoring duplicates
>>> nx_graph.add_edge('A', 'B')
>>> nx_graph.add_edge('A', 'C')
>>> nx_graph.add_edge('A', 'D')
>>> nx_graph.add_edge('A', 'E')
>>> nx_graph.add_edge('B', 'C')
>>> nx_graph.add_edge('C', 'D')
```

```
# Nodes and edges are easy to access
>>> print(nx_graph.nodes())
['A', 'C', 'B', 'E', 'D']

>>> print(nx_graph.edges())
[('A', 'C'), ('A', 'B'), ('A', 'E'), ('A', 'D'), ('C', 'B'), ('C', 'D')]

# NetworkX also has its own shortest_path method, implemented
# with a bidirectional BFS (starting from both ends)
>>> nx.shortest_path(nx_graph, 'C', 'E')
['C', 'A', 'E']

# With small graphs, we can visualize the graph with nx.draw()
>>> from matplotlib import pyplot as plt
>>> nx.draw(nx_graph)
>>> plt.show()
```

Problem 5. Write a `convert_to_networkx` function that accepts an adjacency dictionary. Create a `networkx` object, load it with the graph information from the dictionary, and return it.

The Kevin Bacon Problem

"The 6 Degrees of Kevin Bacon" is a well-known parlor game. The game is played by naming an actor, then trying to find a chain of actors that have worked with each other leading to Kevin Bacon. For example, Samuel L. Jackson was in the film *Captain America: The First Avenger* with Peter Stark, who was in *X-Men: First Class* with Kevin Bacon. Thus Samuel L. Jackson has a "Bacon number" of 2. Any actors who have been in a movie with Kevin Bacon have a Bacon number of 1.

Problem 6. Write a `BaconSolver` class to solve the Kevin Bacon problem.

The provided `movieData.txt` file contains data from about 13,000 movies made over the course of several years. A single movie is listed on each line, followed by a sequence of actors that starred in it. The movie title and actors' names are separated by a '/' character. The actors are listed with last name first, followed by their first name.

The provided `bacon_data` module has a function called `parse` that generates an adjacency dictionary from a specified file. Used with `movieData.txt`, `parse` generates a dictionary where each key is a movie title and each value is a list of the actors appearing in the movie.

Implement the constructor of `BaconSolver` so it accepts a filename to pull data from. Get the dictionary generated by `parse`. Store the collection of actors in the dictionary as a class attribute (avoid duplicates). Convert the dictionary to a NetworkX graph and store it as another class attribute. This graph will have actors, connected to movies, connected to other actors.

Finally, use NetworkX to implement the `path_to_bacon` method. Accept start and target values (actor names) and return a list with the shortest path from start to target. Make Kevin Bacon the default target. If either of the specified actors are not in the stored set of actors, raise a `ValueError`.

```
>>> baconator = BaconSolver("movieData.txt")
>>> baconator.path_to_bacon("Jackson, Samuel L.")
['Jackson, Samuel L.', 'Captain America: The First Avenger', 'Stark,
Peter', 'X-Men: First Class', 'Bacon, Kevin']
```

Warning

Because of the size of the dataset, **do not** attempt to visualize the graph with `nx.draw`. The visualization tool in NetworkX is only effective on relatively small graphs. In fact, graph visualization in general remains a challenging and ongoing area of research.

Problem 7. Implement the `bacon_number` method in the `BaconSolver` class. Accept start and target values and return the number of actors in the shortest path from start to target. Note that this is different than the number of entries in the shortest path list, since movies do not contribute to an actor's Bacon number.

Also implement the `average_bacon` method. Compute the average Bacon number across all of the actors in the stored collection of actors. Exclude any actors that are not connected to Kevin Bacon (their theoretical Bacon number would be infinity). Return the average Bacon number and the number of actors not connected at all to Kevin Bacon.

As an aside, the prolific Paul Erdös is the Kevin Bacon of mathematicians. Someone with an Erdös number of 2 co-authored a paper with someone who co-authored a paper with Paul Erdös.

Problem 8. (Optional) Create a `plot_bacon` method in the `BaconSolver` class that produces a simple histogram displaying the frequency of the Bacon numbers in the data set. The output should look something like Figure 1.2.

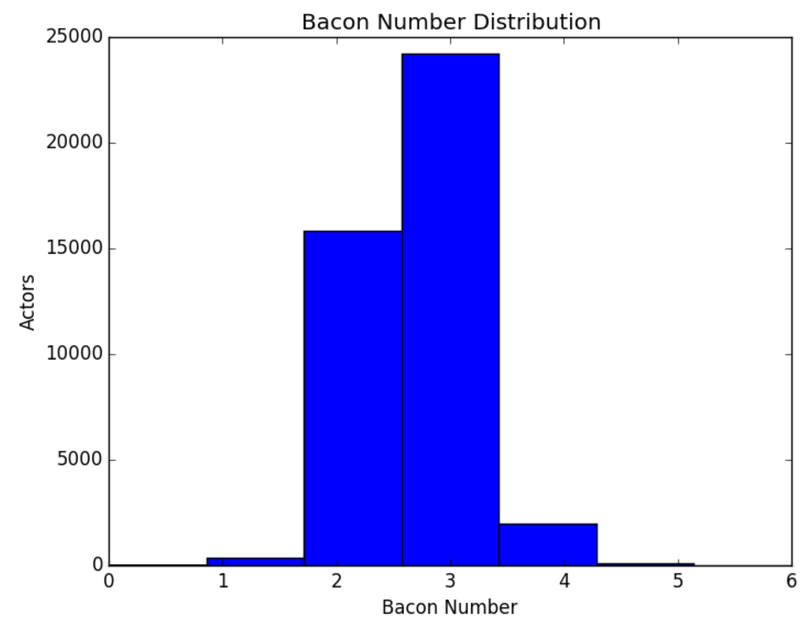

Figure 1.2: The frequency of the bacon numbers.