Version 2024.1

Assignment 1

Data Structures and Analysis

Chandra Gummaluru *University of Toronto*

Part 1.1: Implementing a Dynamic `Graph` Class

When a **`Graph`** needs to be dynamic (its vertices and edges may change over time), it is generally better to use an adjacency list implementation. Below you can find a skeleton implementation.

```
class Vertex:
    """
    Represents a vertex in a graph.

    Attributes:
        name (str): The label or identifier of the vertex.
        children (Dict[str, Tuple[str, str, float]]):
            A mapping between child vertex names and edges.
    """
    def __init__(self, name: str, children: Optional[Dict[str, Tuple[str,
        str, float]]] = None):
        """
        Initializes a Vertex.

        Args:
            name (str): The label or identifier of the vertex.
            children (Optional[Dict[str, Tuple[str, str, float]]]): A
                mapping between child vertex names and edges.
        """
        self.name = name
        self.children: Dict[str, Tuple[str, str, float]] = children if
            children is not None else {}

    def get_children(self) -> List[Tuple[str, str, float]]:
        """
        Returns all edges from this vertex.

        Returns:
```

```
26              List[Tuple[str, str, float]]: The list of edges from this
                    vertex.
```

```
class Graph:
"""
Represents a graph consisting of multiple vertices.

Attributes:
    vertices (List[Vertex]): The list of vertices in the graph.
"""
    def __init__(self, vertices: List[Vertex]):
        """
        Initializes a Graph.

        Args:
            vertices (List[Vertex]): The list of vertices that make up
                the graph.
        """
        self.vertices = vertices

     def get_vertices(self) -> List[Vertex]:
        """
        Returns all vertices in the graph.

        Returns:
            List[Vertex]: The list of vertices in the graph.
        """

    def is_child(self, u_name: str, v_name: str) -> bool:
        """
        Checks if vertex v_name is a child of vertex u_name.

        Args:
            u_name (str): The name of the parent vertex.
            v_name (str): The name of the potential child vertex.

        Returns:
            bool: True if the vertex v_name is a child of the vertex
                u_name, False otherwise.
        """

    def get_edge(self, u_name: str, v_name: str) -> Optional[Tuple[str,
        str, float]]:
        """
        Retrieves the edge between u_name and v_name.
```

```
40
41          Args:
42              u_name (str): The name of the parent vertex.
43              v_name (str): The name of the child vertex.
44
45          Returns:
46              Optional[Tuple[str, str, float]]: The edge if it exists,
47              or None if no such edge is found.
48          """
```

Complete the `Vertex` class and `Graph` class provided in `a1_submission.py`.

Important Note:
The `Graph` class uses `Python` dictionaries internally to store its vertices and edges. You do not need to know the details of how dictionaries work to complete this assignment. For the purpose of this task, you can think of a dictionary as being similar to a list, but instead of using only integer indices (`0`, `1`, `2`, ...), it allows you to use other types as keys, such as strings.

For example:

```
1 my_dict = {"chandra-s25": ["helen-pc", "zws-323"], "helen-pc": ["ws-14"]}
2 print(my_dict["chandra-s25"]) # prints ["helen-pc", "zws-323"]
```

This means that the graph can quickly look up information about a specific vertex by its name, without having to search through a list. All the functions you are asked to implement will work with this dictionary-based structure transparently, so you do not need to worry about how it is stored internally.

Part 1.2: Network Discovery

Consider a `Device` class, which extends the `Vertex` class with the following additional functions:

```
class Device(Vertex):
    """
    Represents a network device, extending the Vertex class with
    device-specific functionality.

    name (str): The label or identifier of the device.
    children (Dict[str, Tuple[str, str, float]]):
        A mapping between child device names and nearby devices.
    network (Graph): A graph representing this device's discovered
       network.
    """

    def __init__(self, name: str):
        """
        Initializes a Device.

        Args:
            name (str): The label or identifier of the device.
        """

    def discover_network(self, find_devices_fn: Callable[[List[str]],
       List[Tuple[str, str, float]]]) -> None:
        """
        Discovers the surrounding network starting from this device. Once
           this function is called, self.network should contain a
           representation of the device's discovered network.

        Args:
            find_devices_fn (Callable[[List[str]], List[Tuple[str, str,
               float]]]):
                A function that takes an ordered list of device names
                    (i.e., a path) and returns the edges from the last
                    device in the path to its immediate children.
        """
```

Complete the `discover_network` function within the `Device` class provided in `a1_submission.py`. When a `Device` object is initialized, its `children` attribute should be set to `{}`, and its `network` attribute should be set to `Graph([self])`. Then, when `discover_network(...)` is called `network` should be updated to accurately represent the network. The network

topology is determined by `find_devices_fn`. You will not have access to this function, but may want to mock it for the purposes of testing.

```
def find_devices_fn(path: List[str]) -> List[Tuple[str, str, float]]:
    """
    A function that takes an ordered list of device names (i.e., a path)
       and returns the edges from the last device in the path to its
       immediate children.

    Args:
        path (List[str]): The sequence of device names representing the
           discovery path. The first device is always the one initiating
           discovery.

    Returns:
        List[Tuple[str, str, float]]: A list of edges, where each tuple
           contains:
            - source device name (str),
            - child device name (str),
            - edge weight (float).

    Example:
        If `path = ['chandra-s25', 'router-051797', 'helen-pc']`:

        - The request is initiated by `chandra-s25`.
        - It is forwarded to `helen-pc`.
        - `helen-pc` then queries `router-051797`.

        Suppose `router-051797` has three children. The function returns:

        [
            ('router-051797', 'ws-102', 1.2),
            ('router-051797', 'switch-12', 0.8),
            ('router-051797', 'srv-07', 1.0)
        ]

    Notes:
        If any device in the path is not connected to the next device,
        the function returns an empty list without warning.
    """
```

Important Note:

In graph theory, a *path* is often defined as

$$\langle(\varnothing, \varnothing, v_0), (v_0, a_1, v_1), (v_1, a_2, v_2), \ldots, (v_{k-1}, a_k, v_k)\rangle,$$

where v_i are vertices and a_i are the edges (or actions) connecting them.
However, in this course we will assume that there is at most *one directed edge* from any vertex to any other vertex. Under this assumption, the specific edges a_i can be omitted without losing information, so a path can simply be written as

$$\langle(\varnothing, v_0), (v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)\rangle.$$

For simplicity, we will use this latter representation throughout the course.

However, to make our code simpler to work with, we can take it one step further, and simply write the path as an ordered list of vertices:

$$\langle v_0, v_1, \ldots, v_{k-1}, v_k\rangle.$$

We will use this representation when working with code.

Part 1.3: Cheapest-Path Problems

Complete the `find_path` function within the `Device` class provided in `a1_submission.py`. This function is meant to find the cheapest-path to the specified device. Implement the cheapest-first search (CFS) algorithm to find the cheapest path.

```
1    def find_path(self, d_name: str) -> Optional[List[str]]:
2    """
3    Finds the cheapest path from this device to the specified target
        device
4    using the Cheapest-First Search (CFS) algorithm.
5
6    Args:
7        d_name (str): The name of the destination device.
8
9    Returns:
10       Optional[List[str]]: An ordered list of device names representing
            the path
11       from this device to the target. If no path exists, returns None.
12   """
13   ...
```

Tips

Here are a few tips to help you for this assignment:

- Do **NOT** change any of the pre-existing code within `a1_submission.py`. The auto-grader will assume that the starter code remains unchanged.
- When writing tests, it's often best to test functions in isolation. If one function depends on another, you don't want failures in the inner function to cause false failures in the outer function. In such cases, you can mock (i.e., replace temporarily) the inner function so you only test the logic of the outer function.

```
def bar(x: int) -> int:
    # Imagine this is a complicated function (e.g., database call)
    return x * 2

def foo(x: int) -> int:
    # foo depends on bar
    return bar(x) + 1
```

  Normally, `foo(3)` calls `bar(3)` which returns `6`, so the result is `7`.

  But when testing `foo`, we may want to ignore the real logic of `bar` and mock it.

```
# Save the original bar function
original_bar = bar

# Define a fake version of bar for testing
def mock_bar(x: int) -> int:
    print("mock_bar called with", x)
    return 100 # always return 100, no matter what

# Replace bar with the mock
bar = mock_bar

# Now test foo in isolation
print(foo(3)) # This will print "mock_bar called with 3" and return
    101

# Restore the original bar after testing
bar = original_bar

print(foo(3)) # Back to normal: 7
```