Version 2024.1

Assignment 2

Data Structures and Analysis

Chandra Gummaluru *University of Toronto*

Part 1.1: Implementing a `UnionFind` Class

When performing Kruskal's Algorithm (Part 1.2) for the the Minimum Spanning Tree (MST), the `UnionFind` is helpful for tracking, finding, and merging the vertex sets. Below you can find a skeleton implementation.

```python
class UnionFind:
    def __init__(self, elements: List[str]):
        """
        Initializes the Union-Find data structure for n elements.
        Initially, each element is in its own set (its parent is itself).
        The rank (or size) of each set is initialized to 0.

        Parameters:
        elements (List[str]): The list of elements in the Union-Find data
           structure.
        """
        self.parent = {elem: elem for elem in elements}
        self.rank = {elem: 0 for elem in elements}

    def find(self, x: str) -> str:
        """
        Find the root (or representative) of the set containing the
           element x.

        Args:
        x (str): The element whose root we want to find.

        Returns:
        str: The root of the set that contains x.
        """
        pass

    def union(self, x: int, y: int) -> bool:
        """Union (or merge) the sets containing elements x and y.
        Return True if union was successful.
        If x and y are already in the same set, do nothing (return False).

        Args:
            x (str): The first element (set to be united).
            y (str): The second element (set to be united).

        Returns:
        bool: True if x and y are successfully unioned.
              False if x and y are already in the same set (no union
                  nedded).
        """
        pass
```

Complete the `UnionFind` class provided in `a2_submission.py`.

Important: You must use the rank heuristic when performing `union`.

Part 1.2: Kruskal's Algorithm

Using Kruskal's Algorithm, find the Minimum Spanning Tree (MST) of a graph, using the `Graph` and `Vertex` classes from Assignment 1. Please refer to the following MST function:

```python
# Function to implement Kruskal's algorithm
def kruskal_mst(graph: Graph) -> List[Tuple[str, str, float]]:
    """
    Kruskal's Algorithm for Minimum Spanning Tree (MST).

    Args:
        graph (Graph): The graph for which we compute the MST.

    Returns:
        List[Tuple[str, str, float]]: A list of edges in the MST.
        Each edge is represented as a tuple (source vertex, destination
            vertex, weight).
    """

    result = [] # The final MST

    return result
```

Complete the `kruskal_mst` function provided in `a2_submission.py`. The function should return a list of edges in the Minimum Spanning Tree (MST), where each edge is represented as a tuple: `(src, dst, wgt)`.

Important: The order of the edges in the returned list is important and will be verified during testing. It is recommended that you use the `UnionFind` data structure in your implementation.

Part 1.3: Prim's Algorithm

Using Prim's Algorithm, find the Minimum Spanning Tree (MST) of a graph, using the `Graph` and `Vertex` classes from Assignment 1. Please refer to the following MST function:

```python
# Function to implement Prim's algorithm
def prim_mst(graph: Graph) -> List[Tuple[str, str, float]]:
    """
    Prim's Algorithm for Minimum Spanning Tree (MST).

    Args:
        graph (Graph): The graph for which we compute the MST.

    Returns:
        List[Tuple[str, str, float]]: A list of edges in the MST.
            Each edge is represented as a tuple (source vertex,
                destination vertex, weight).
    """

    result = [] # The final MST

    # Chosen to match the test cases
    start_vertex = graph.get_vertices()[0]

    return result
```

Complete the `prim_mst` function provided in `a2_submission.py`. The function should return a list of edges in the Minimum Spanning Tree (MST), where each edge is represented as a tuple: `(src, dst, wgt)`.

Important: The order of the edges in the returned list is important and will be verified during testing.

Tips

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

- Do **NOT** change any of the pre-existing code within `a2_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.

  ```python
  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.

  ```python
  # 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
  ```