Project2 Analysis

This file implements a Graph class with two main algorithms: Kruskal's algorithm for finding a Minimum Spanning Tree (MST) and Dijkstra's algorithm for finding shortest paths.

Key Components

1. **Graph Class**: Represents a graph with vertices and edges
    - Supports both directed and undirected graphs
    - Uses both adjacency lists and edge lists for different algorithms
    - Maps between vertex letters (A, B, C...) and indices (0, 1, 2...)
2. **File Input**: Allows creating a graph from a text file with a specific format
3. **Algorithms**:
    - Kruskal's MST algorithm
    - Dijkstra's shortest path algorithm

Pseudocode for Key Algorithms

Kruskal's MST Algorithm

```
FUNCTION KruskalMST(Graph G):
    result = empty list
    i = 0, e = 0

    Sort all edges in G in non-decreasing order of weight

    Initialize disjoint sets for each vertex (using parent and rank arrays)

    WHILE e < G.V - 1 AND i < number of edges:
        Pick the smallest edge (u, v, w)
        i = i + 1

        x = FIND-SET(u)
        y = FIND-SET(v)

        IF x ≠ y:
            e = e + 1
            Add edge to result
            UNION(x, y)

    Calculate and print total cost of MST
    RETURN result
```

Dijkstra's Shortest Path Algorithm

```
FUNCTION Dijkstra(Graph G, source s):
    Initialize dist[v] = ∞ for all vertices v in G
    dist[s] = 0
    prev[v] = NULL for all vertices v in G

    priority_queue pq = {(0, s)}
    visited = empty set

    WHILE pq is not empty:
        current_dist, u = pq.extract_min()

        IF u in visited:
            CONTINUE

        Add u to visited

        IF current_dist > dist[u]:
            CONTINUE

        FOR EACH neighbor v with weight w:
            IF dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
                prev[v] = u
                pq.add((dist[v], v))

    RETURN dist, prev
```

Time Complexity Analysis

Kruskal's MST Algorithm

- Sorting edges: O(E log E) where E is the number of edges
- Processing edges with Union-Find: O(E log V) where V is the number of vertices
- Overall: O(E log E) or O(E log V) since E can be at most V^2

Dijkstra's Algorithm

- Using a binary heap: O((V + E) log V)
- Each vertex is inserted into the priority queue once: O(V log V)
- Each edge is examined once: O(E log V)
- Overall: O((V + E) log V)

Space Complexity Analysis

Kruskal's MST Algorithm

- Edge list: O(E)
- Disjoint-set data structure: O(V)
- Result list: O(V)
- Overall: O(E + V)

Dijkstra's Algorithm

- Distance array: O(V)
- Previous vertex array: O(V)
- Priority queue: O(V)
- Visited set: O(V)
- Overall: O(V)

Key Features and Design Choices

1. **Vertex Representation**:
    - Uses letters (A, B, C...) for user interface
    - Maps to indices (0, 1, 2...) internally for array-based operations
2. **Graph Representation**:
    - Edge list for Kruskal's algorithm (efficient for sorting edges)

    - Adjacency list for Dijkstra's algorithm (efficient for traversing neighbors)
3. **Union-Find Data Structure**:
    - Uses path compression and union by rank for efficient operations
    - Ensures near-constant time complexity for find and union operations
4. **File Input Format**:
    - First line: Number of vertices, edges, and graph type (directed/undirected)
    - Following lines: Edges with format "u v w" (vertices and weight)
    - Last line: Source vertex for algorithms
5. **Error Handling**:
    - Robust error checking for file input
    - Provides meaningful error messages

This implementation provides a comprehensive solution for graph algorithms with a focus on MST and shortest path problems, with efficient implementations and a user-friendly interface.

Data Structure Analysis

The implementation uses several key data structures to represent graphs and support the algorithms. Here's a detailed breakdown:

1. Graph Representation

Edge List

```
self.graph = []  # Stores edges as [u, v, w] where u, v are vertex indices and w is weight
```

- **Purpose**: Primary structure for Kruskal's MST algorithm
- **Implementation**: List of lists, where each inner list represents an edge [u, v, w]
- **Advantages**:
    - Easy to sort edges by weight (crucial for Kruskal's)
    - Simple to iterate through all edges
- **Disadvantages**:
    - Not efficient for checking if an edge exists
    - Not suitable for quickly finding neighbors of a vertex

Adjacency List

```
self.adj_list = {chr(65+i): [] for i in range(vertices)}  # Maps vertex letters to lists of tuples
```

- **Purpose**: Primary structure for Dijkstra's algorithm
- **Implementation**: Dictionary where keys are vertex letters (A, B, C...) and values are lists of (neighbor, weight) tuples
- **Advantages**:
    - Efficient for finding neighbors of a vertex
    - Space-efficient for sparse graphs
    - Fast iteration over adjacent vertices
- **Disadvantages**:
    - Less efficient for checking if an edge exists compared to adjacency matrix

2. Vertex Mapping

```
self.vertex_to_index = {chr(65+i): i for i in range(vertices)}  # Maps letters to indices
self.index_to_vertex = {i: chr(65+i) for i in range(vertices)}  # Maps indices to letters
```

- **Purpose**: Conversion between user-friendly vertex labels (A, B, C...) and array indices (0, 1, 2...)
- **Implementation**: Two dictionaries for bidirectional mapping
- **Advantages**:
    - Allows for human-readable vertex names in input/output
    - Enables efficient array-based operations internally
- **Usage**: Used when adding edges and displaying results

3. Disjoint-Set (Union-Find) Data Structure

Used in Kruskal's algorithm to detect cycles:

```
parent = []  # Parent pointers for disjoint-set forest
rank = []    # Ranks for union by rank optimization
```

- **Purpose**: Efficiently determine if adding an edge creates a cycle
- **Implementation**: Two arrays - parent pointers and ranks
- **Optimizations**:

    - Path compression in find() operation
    - Union by rank to keep trees balanced
- **Operations**:
    - `find(parent, i)` : Find the representative of the set containing i
    - `union(parent, rank, x, y)` : Merge sets containing x and y
- **Time Complexity**: Nearly constant time per operation (inverse Ackermann function)

4. Priority Queue for Dijkstra's Algorithm

```
pq = [(0, src)]  # Min-heap of (distance, vertex) pairs
```

- **Purpose**: Always extract the vertex with minimum distance
- **Implementation**: Binary heap via Python's heapq module
- **Operations**:
    - `heapq.heappush(pq, (dist, v))` : Add vertex with its distance
    - `heapq.heappop(pq)` : Extract vertex with minimum distance
- **Time Complexity**: O(log V) per operation
- **Advantage**: Efficiently maintains the set of vertices to process next

5. Distance and Path Tracking

```
dist = {v: float('inf') for v in self.adj_list}  # Maps vertices to shortest distance
prev = {v: None for v in self.adj_list}          # Maps vertices to predecessor in shortest path
```

- **Purpose**: Track shortest distances and reconstruct paths
- **Implementation**: Dictionaries with vertices as keys
- **Usage**:
    - `dist[v]` : Shortest distance from source to vertex v
    - `prev[v]` : Previous vertex in shortest path to v

6. Visited Set

```
visited = set()  # Tracks processed vertices in Dijkstra's algorithm
```

- **Purpose**: Prevent reprocessing vertices in Dijkstra's algorithm
- **Implementation**: Python set for O(1) lookups
- **Advantage**: Ensures each vertex is processed exactly once

7. Result Collection for MST

```
result = []  # Stores edges in the MST as [u, v, w]
```

- **Purpose**: Collect edges that form the Minimum Spanning Tree
- **Implementation**: List of edge lists
- **Usage**: Stores and displays the final MST edges

This comprehensive use of data structures enables efficient implementation of both Kruskal's MST algorithm and Dijkstra's shortest path algorithm, with appropriate structures chosen for each algorithm's specific needs.

Running Instructions

- Line 284 of the code specifies the testfile name and can be changed to test separate input files. Default is '/uwGraph1.txt'
- uwGraph1, uwGraph2, dwGraph1, and dwGraph2 have been included as sample files to run agains this code.

```
284    filename = running_directory + '/uwGraph1.txt'
```

Sample Input/Output

```
 9 12 U
A B 5
A C 3
B D 2
C D 8
D E 7
E F 4
F G 9
G H 6
H I 1
```

```
I A 10
D G 11
B H 12
A

Example: Reading a graph from a file
Graph loaded from /python/ITCS6114/Project2/uwGraph1.txt
Edges in the constructed MST
H -- I == 1
B -- D == 2
A -- C == 3
E -- F == 4
A -- B == 5
G -- H == 6
D -- E == 7
F -- G == 9
Minimum Spanning Tree 37

Running Dijkstra's algorithm from source vertex A:
Shortest paths from source vertex A:
To vertex B: A -> B, distance: 5
To vertex C: A -> C, distance: 3
To vertex D: A -> B -> D, distance: 7
To vertex E: A -> B -> D -> E, distance: 14
To vertex F: A -> B -> D -> E -> F, distance: 18
To vertex G: A -> I -> H -> G, distance: 17
To vertex H: A -> I -> H, distance: 11
To vertex I: A -> I, distance: 10



9 12 U
A B 15
B C 8
C D 7
D E 9
E F 6
F G 5
G H 10
H I 4
I A 3
B F 12
C G 11
D H 14
B

Example: Reading a graph from a file
Graph loaded from /python/ITCS6114/Project2/uwGraph2.txt
Edges in the constructed MST
I -- A == 3
H -- I == 4
F -- G == 5
E -- F == 6
C -- D == 7
B -- C == 8
D -- E == 9
G -- H == 10
Minimum Spanning Tree 52

Running Dijkstra's algorithm from source vertex B:
Shortest paths from source vertex B:
To vertex A: B -> A, distance: 15
To vertex C: B -> C, distance: 8
To vertex D: B -> C -> D, distance: 15
To vertex E: B -> F -> E, distance: 18
To vertex F: B -> F, distance: 12
```

```
To vertex G: B -> F -> G, distance: 17
To vertex H: B -> A -> I -> H, distance: 22
To vertex I: B -> A -> I, distance: 18



7 15 D
A B 3
A C 5
B C 2
B D 6
C D 4
C E 8
D E 1
D F 9
E F 7
E G 10
F G 3
G A 5
G B 8
F C 2
E A 6
A

Example: Reading a graph from a file
Graph loaded from /python/ITCS6114/Project2/dwGraph1.txt
Edges in the constructed MST
D -- E == 1
B -- C == 2
F -- C == 2
A -- B == 3
F -- G == 3
C -- D == 4
Minimum Spanning Tree 15

Running Dijkstra's algorithm from source vertex A:
Shortest paths from source vertex A:
To vertex B: A -> B, distance: 3
To vertex C: A -> C, distance: 5
To vertex D: A -> B -> D, distance: 9
To vertex E: A -> B -> D -> E, distance: 10
To vertex F: A -> B -> D -> E -> F, distance: 17
To vertex G: A -> B -> D -> E -> G, distance: 20

7 15 D
A B 12
B C 9
C D 7
D E 5
E F 3
F G 1
G A 4
A D 8
B E 6
C F 10
D G 2
E A 11
F B 14
G C 13
G E 15
B

Example: Reading a graph from a file
Graph loaded from /python/ITCS6114/Project2/dwGraph2.txt
Edges in the constructed MST
F -- G == 1
D -- G == 2
```

```
E -- F == 3
G -- A == 4
B -- E == 6
C -- D == 7
Minimum Spanning Tree 23

Running Dijkstra's algorithm from source vertex B:
Shortest paths from source vertex B:
To vertex A: B -> E -> F -> G -> A, distance: 14
To vertex C: B -> C, distance: 9
To vertex D: B -> C -> D, distance: 16
To vertex E: B -> E, distance: 6
To vertex F: B -> E -> F, distance: 9
To vertex G: B -> E -> F -> G, distance: 10
```