Analysis of graphAlgsReadFile.py

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. Let me break down the key components and provide pseudocode for the main algorithms.

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.