Graph and Node Class Design Document

Overview

The Graph and Node classes implement a flexible graph data structure with support for weighted, labeled relationships between nodes. The design allows for dynamic graph manipulation, path finding, and complex graph operations.

Class Structure

Node Class

Attributes

- `id` : Unique identifier for the node
- `name` : Name of the node
- `type` : Type classification of the node
- `relationships` : Vector of tuples containing connected nodes, relationship labels, and weights
- `queued` : Flag for graph traversal algorithms
- `processed` : Flag for graph traversal algorithms
- `distance` : Distance metric used in path finding
- `parent` : Parent node used in path reconstruction

Key Methods

- `addRelationship()` : Add or update a relationship between nodes
- `removeRelationship()` : Remove a specific relationship
- `getRelationships()` : Retrieve all relationships for a node
- `relationshipExists()` : Check if a relationship exists

Graph Class

Attributes

- `nodes` : Vector of Node pointers representing the graph

Key Methods

- `load()` : Load graph data from a file
- `relationship()` : Create a relationship between nodes
- `entity()` : Add or update a node
- `print()` : Print relationships for a specific node
- `deleteNode()` : Remove a node and its relationships
- `path()` : Find the highest-weight path between two nodes
- `highest()` : Find the diameter of the graph
- `findAll()` : Search nodes by name or type

Function Design

`deleteNode(std::string id)`

Purpose: Remove a node and its relationships from the graph.

Key Design Points:

- Validate input
- Remove node references
- Free node memory

Algorithm Steps:

1. Validate input
2. Find target node
3. Remove node references from neighbors
4. Delete node from graph
5. Free memory

`path(std::string id1, std::string id2)`

Purpose: Find highest-weight path between two nodes.

Key Design Points:

- Modified Dijkstra's algorithm
- Maximize path weight
- Support disconnected graphs

Algorithm Steps:

1. Validate node IDs
2. Initialize path exploration
3. Use priority queue to track paths
4. Reconstruct optimal path
5. Return path details

`highest()`

Purpose: Determine graph's diameter across components.

Key Design Points:

- Handle multiple graph components
- Use maximum spanning tree
- Two-phase depth-first search

Algorithm Steps:

1. Explore graph components
2. Build maximum spanning tree
3. Perform dual DFS to find longest path
4. Track overall longest path

`findAll(std::string fieldType, std::string fieldString)`

Purpose: Search nodes by name or type.

Key Design Points:

- Flexible search criteria
- Case-sensitive matching
- Efficient linear search

Algorithm Steps:

1. Iterate nodes
2. Match against search criteria
3. Collect matching node IDs
4. Return results

`buildMaxSpanningTree(Node* startNode)`

Purpose: Construct maximum spanning tree from a node.

Key Design Points:

- Prioritize highest-weight edges
- Prevent cycles
- Partial graph traversal

Algorithm Steps:

1. Check node relationships
2. Use priority queue
3. Select highest-weight edges
4. Build tree avoiding cycles

Runtime Analysis

Runtime Analysis for `path()` Function

1. **Input Validation**
   - `validateInput()` : O(k), where k is the length of the input string
   - `findNode()` : O(V), where V is the number of nodes
2. **Initialization Loop**
   - Iterate through all nodes: O(V)
   - Setting node properties: O(1) per node
3. **Priority Queue Operations**
   - Worst-case scenario: Each node pushed into and extracted from priority queue once
   - `push()` operation: O(log V)
   - `extractMax()` operation: O(log V)
   - Total priority queue operations: O(V log V)
4. **Edge Relaxation**
   - Iterate through node relationships (edges)
   - Worst-case: Each node connected to all other nodes
   - Per relationship:
     - Distance checking: O(1)
     - Priority queue update: O(log V)

- Total edge relaxation: O(E log V), where E is the number of edges

5. **Path Reconstruction**
   - Traversing from destination to source: O(V)
   - Creating result string: O(V)

Overall Complexity Analysis

Combining the steps:

- Initialization: O(V)
- Priority Queue Operations: O(V log V)
- Edge Relaxation: O(E log V)
- Path Reconstruction: O(V)

Dominant Terms: O(V log V) and O(E log V)

Final Time Complexity: O((V + E) log V)

UML Diagram