Ruben Rehal

Project 4 : Graphs

Design Document

Key Design Goals

1. **Extensibility**
   The system is built to handle diverse entity and relationship types, allowing easy modifications and additions. By adopting a modular class structure, future expansions, such as additional query types or enhanced graph operations, can be implemented without disrupting the current architecture.
2. **Robustness**
   Error handling is integrated at both input and operational levels. Commands with invalid arguments are processed gracefully through exception handling. For example, invalid node IDs or operations on non-existent entities are intercepted with custom exception messages, ensuring the system remains stable during execution.
3. **Efficiency**
   To manage large datasets, algorithms for pathfinding and edge management are optimized for logarithmic complexity wherever applicable. The adjacency matrix, coupled with dynamic resizing, balances fast access times with efficient memory use. This ensures scalability as graph size increases.
4. **User-Centric Functionality**
   The graph is designed for ease of use, supporting intuitive commands to load data, establish connections, and perform queries. Outputs are clearly formatted, and failure cases are explicitly reported to the user, minimizing confusion and maximizing clarity.

Core Features

1. **Entity Management**
   - **Node Addition**: Nodes, identified by unique string IDs, can be added to the graph with labels and types. The graph ensures that duplicate entries are avoided by updating existing nodes if their IDs match. This approach eliminates redundant data and preserves entity uniqueness.
   - **Node Deletion**: Nodes and their associated edges are removed when a delete command is issued. The adjacency matrix is updated to maintain structural consistency.
2. **Relationship Management**
   - **Edge Addition**: Edges between nodes are defined by source and destination IDs, labels, and weights. If an edge already exists between two nodes, its weight and label are updated. This ensures the integrity of edge definitions without introducing duplicates.
   - **Dynamic Resizing**: The adjacency matrix is dynamically resized as new nodes are added, ensuring efficient use of memory while allowing the graph to grow seamlessly.
3. **Query Operations**

  - **Adjacency Listings**: For a given node, all directly connected neighbors are listed. This provides an overview of immediate relationships and serves as a foundation for more complex graph traversals.
  - **Property-Based Searches**: Users can retrieve all nodes with a specific name or type. The function iterates over all nodes, matching the given criteria to generate concise outputs.
  - **Pathfinding**: Identifies the highest-weight path between two nodes using a modified Dijkstra's algorithm. This ensures that the heaviest cumulative relationship is prioritized, aligning with graph traversal requirements in weighted systems.
  - **Highest-Weight Relationship**: Determines the pair of nodes connected by the strongest path. By iterating over all unique node pairs, the function computes cumulative weights efficiently and outputs the strongest connection.

4. **File Loading**
   - Entity and relationship data can be batch-loaded from files using predefined formats. The LOAD command processes entity files to create nodes and relationship files to establish edges. The parser ensures robust handling of data inconsistencies and guarantees that the graph reflects the input accurately.

Class Design

1. **Node Class**
   - **Attributes**: Each node is characterized by an ID, label (name), and type, providing flexibility in defining entities.
   - **Methods**: The node class includes accessors and mutators for these attributes, enabling controlled updates and queries.
2. **Edge Class**
   - **Attributes**: Contains weight and label, representing the strength and type of the relationship.
   - **Methods**: Provides accessors and mutators for weight and label, facilitating seamless updates.
3. **Graph Class**
   - **Attributes**: Maintains a dynamically resizable array of nodes and an adjacency matrix for edges. Includes metadata such as node count and graph capacity.
   - **Methods**: Implements core functionalities, including:
     - **addNode**: Adds or updates nodes.
     - **addRelationship**: Establishes or modifies edges, ensuring bidirectional consistency.
     - **path**: Computes the heaviest path between two nodes.
     - **findMaxWeightPath**: Finds the strongest connection between node pairs.

Function Design

1. **addRelationship**
   - **Description**: Establishes a weighted edge between two nodes. If an edge already exists, it updates the weight and label to reflect the latest values. Edges are bidirectional, ensuring consistency across the adjacency matrix.

   - **Implementation**: The function validates node existence before proceeding. If nodes are invalid or weights are non-positive, the operation fails gracefully. Otherwise, the edge is created or updated in O(1) time for each adjacency matrix entry.
2. **path**
   - **Description**: Finds the maximum weight path between two nodes using a modified Dijkstra's algorithm. The algorithm prioritizes paths with the highest cumulative weight, ensuring optimal traversal results.
   - **Implementation**: The function initializes a distance array with negative infinity values, representing unvisited nodes. By relaxing edges iteratively, it ensures that the heaviest path is computed efficiently. Backtracking through the predecessor array reconstructs the path. Runtime is O((|E| + |V|) log |V|), leveraging a priority queue.
3. **findMaxWeightPath**
   - **Description**: Identifies the two nodes connected by the strongest cumulative relationship.
   - **Implementation**: The function iterates over all unique node pairs, computing paths using the path function. Results are compared to track the maximum weight observed. While exhaustive, the design ensures that computations are bounded by O($|V|^2$), aligning with graph theory requirements.

Runtime Analysis

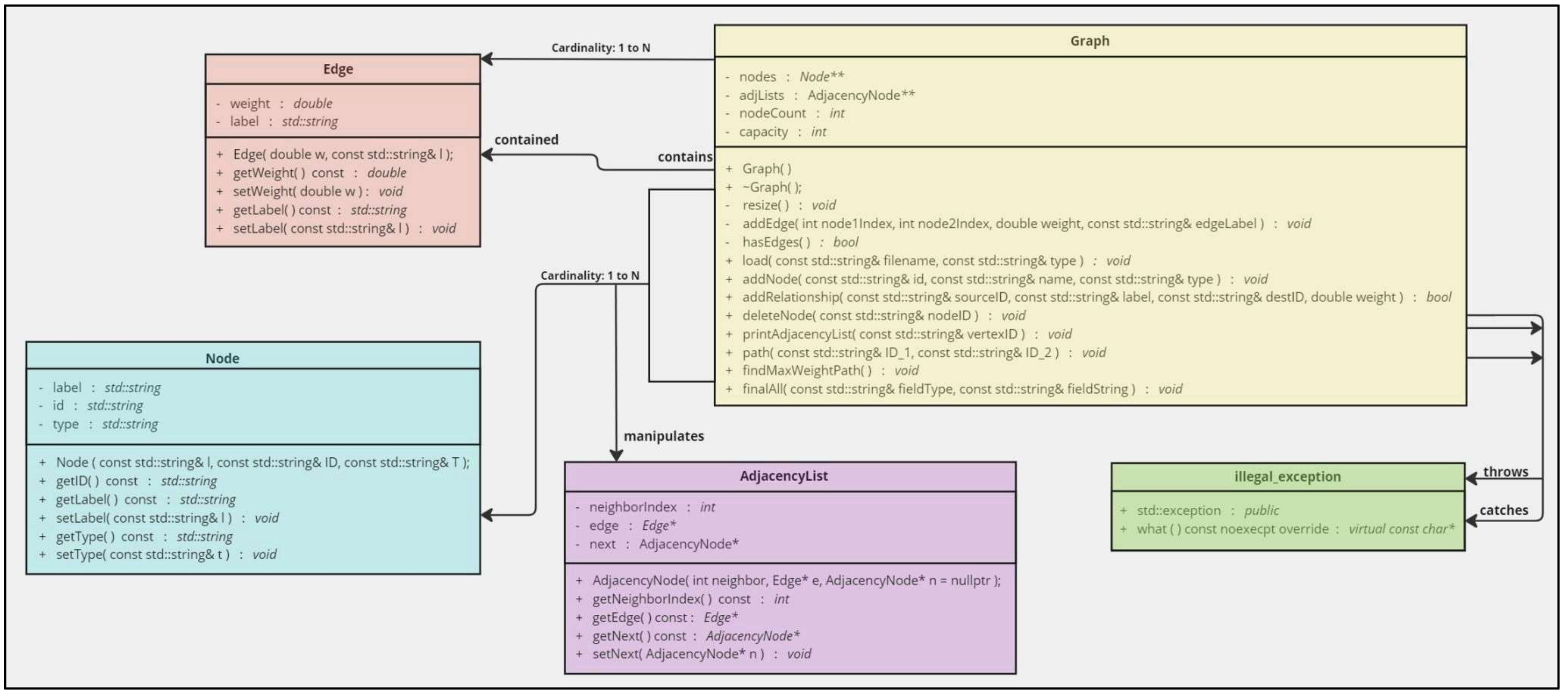
The runtime of the primary operations in this project has been carefully designed to ensure efficiency, particularly for large datasets. The **pathfinding algorithm**, a variation of Dijkstra's algorithm, achieves a worst-case complexity of O((|E|+|V|)log|V|), where |E| represents the number of edges, and |V| represents the number of vertices. This is accomplished through the use of an adjacency list for efficient edge traversal and a priority queue for managing the vertices with the highest weight paths during exploration. The adjacency list ensures that each edge is processed only once, contributing O(|E|), while the priority queue operations contribute O(log|V|) for each vertex added or removed [1].

Operations for **inserting and deleting nodes and edges** have also been optimized. Adding a node has an average complexity of O(1) but may require O(|V|) when resizing is triggered to accommodate additional capacity. Adding an edge involves traversing the adjacency list of the source node to detect duplicates, resulting in a complexity of O(d), where d is the degree of the node. Deleting a node requires removing all associated edges, which involves O(|E|) operations as every adjacency list may need to be updated.

The **graph resizing** operation, used to dynamically allocate more space for nodes and adjacency lists, has a complexity of O(|V|), as existing data must be copied to a new array. While resizing is infrequent, its cost is amortized over multiple insertions.

The **FindMaxWeightPath** command, which identifies the two nodes with the highest weight path, uses a brute-force approach by computing paths between all pairs of nodes. This results in a worst-case complexity of O($|V|^2$ (|E|+|V|)log|V|), as the pathfinding algorithm is invoked for every unique pair of nodes [1].

UML Diagram


Cardinality: 1 to N
Edge
- weight : double
- label : std::string
+ Edge(double w, const std::string& l);
+ getWeight( ) const : double
+ setWeight(double w) : void
+ getLabel( ) const : std::string
+ setLabel(const std::string& l) : void
contained
contains
Graph
- nodes : Node**
- adjLists : AdjacencyNode**
- nodeCount : int
- capacity : int
+ Graph( )
+ ~Graph( );
- resize( ) : void
- addEdge(int node1Index, int node2Index, double weight, const std::string& edgeLabel) : void
- hasEdges( ) : bool
+ load(const std::string& filename, const std::string& type) : void
+ addNode(const std::string& id, const std::string& name, const std::string& type) : void
+ addRelationship(const std::string& sourceID, const std::string& label, const std::string& destID, double weight) : bool
+ deleteNode(const std::string& nodeID) : void
+ printAdjacencyList(const std::string& vertexID) : void
+ path(const std::string& ID_1, const std::string& ID_2) : void
+ findMaxWeightPath( ) : void
+ finalAll(const std::string& fieldType, const std::string& fieldString) : void
Cardinality: 1 to N
Node
- label : std::string
- id : std::string
- type : std::string
+ Node (const std::string& l, const std::string& ID, const std::string& T);
+ getID( ) const : std::string
+ getLabel( ) const : std::string
+ setLabel(const std::string& l) : void
+ getType( ) const : std::string
+ setType(const std::string& t) : void
manipulates
AdjacencyList
- neighborIndex : int
- edge : Edge*
- next : AdjacencyNode*
+ AdjacencyNode(int neighbor, Edge* e, AdjacencyNode* n = nullptr);
+ getNeighborIndex( ) const : int
+ getEdge( ) const : Edge*
+ getNext( ) const : AdjacencyNode*
+ setNext(AdjacencyNode* n) : void
illegal_exception
+ std::exception : public
+ what ( ) const noexecpt override : virtual const char*
throws
catches

References

[1] ChatGPT, paraphrasing tool. OpenAI [Online]. https://chatgpt.com/ (accessed November 17, 2024).