Approach Rationale

This project implements a weighted undirected graph data structure to store entities and relationships between them. It finds maximum weight paths and various other graph operations.

Implementation: Adjacency List Structure using Vectors (chosen for the following reasons):

1. Provides efficient O(|E|) edge traversal for path finding operations as required for this project, achieved through direct access to edge lists
2. Vector-based implementations allow for simple iteration through adjacent nodes and straightforward memory management
3. More space efficient than adjacency matrix for sparse graphs
4. Better performance than hash maps for moderate-sized graphs
5. Simplifies implementation of undirected edges by maintaining symmetrical edge lists

Class Design

1. Class Graph

This class is the main graph data structure implementation that manages the overall graph operations. It handles entity and relationship management and path finding using a modified Dijkstra's algorithm, etc. It makes use of helper functions that validate inputs and manage node/edge operations. This class handles errors through input validation and exception throwing for illegal arguments.

Private:

1. nodes: Vector of Node pointers
   Core data structure must be protected, access through public interface only
2. isValidId(): Helper function
   For internal validation of node IDs, ensures alphanumeric characters only
3. findNode(): Helper function
   Internal node lookup helper, should not change, used by many public methods
4. dijkstra(): Helper function
   Used internally by findPath() and findHighest() to ensure consistent path-finding behavior

Public:

constructor - destructor - loadEntities() - loadRelationships() - addRelationship() - addEntity() - deleteNode() - getAdjacentNodes() - findPath() - findHighest() - findAll()

2. Class Node

This class manages individual nodes in the graph structure. It maintains node properties (ID, name, type) and manages edges to other nodes. This class handles memory management for edge relationships and provides controlled access to node properties.

Private:

1. id: String identifier - Must be immutable after creation
2. name: Node name string - Changes must go through setName() to ensure proper validation
3. type: Node type string - Changes must be managed through setType() to ensure valid type
4. edges: Vector of edge tuples - Must maintain consistency of edge relationships

Public:

constructor - getId() - getName() - getType() - getEdges() - setName() - setType() - addEdge() - removeEdge() - hasEdge() - updateEdge()

3. Class illegal_exception

An exception class that implements a simple exception handler for dealing with illegal input arguments (inputs containing non-alphanumeric characters). It has one responsibility, it handles only input format violations. Uses const char* for message storage to minimize memory overhead.

<u>**Function Design**</u>

Function: Node* Graph::findNode(const std::string& id) const

Parameters: id

Return Type: node pointer (nullptr if not found)

Rationale: Linearly searches through graph's node vector to find node with matching ID. **Critical helper function** used by other operations to locate nodes in the graph.

Function: bool Graph::isValidId(const std::string& id) const

Parameters: id

Return Type: bool (true if valid)

Rationale: Validates that node IDs contain only alphanumeric characters. Used by all functions that take node IDs to ensure data integrity. Throws illegal_exception if valid. fails.

Function: bool Graph::deleteNode(const std::string& id)

Parameters: id

Return Type: bool (true=success)

Rationale: Removes specified node and all its relationships from graph. Validates ID, finds node, removes all connected edges, deletes node, and updates graph structure. Returns false if node !found.

Function: std::vector<std::string> Graph::getAdjacentNodes(const std::string& id)

Parameters: id

Return Type: vector of adjacent node IDs

Rationale: Returns sorted list of IDs for all adjacent nodes connected to a specific node by traversing edge list and sorting ID's in descending order. Also performs existence and connectivity checks. Returns empty vector if node !found or !connections.

Function: std::tuple<std::vector<std::string>, double> Graph::findPath(const std::string& id1, const std::string& id2)

Parameters: id1, id2

Return Type: tuple of (path node IDs, total path weight)

Rationale: Validates inputs, locates nodes, then calls modified Dijkstra's alg. to maximize path weight. Returns empty path and zero for invalid/disconnected nodes to handle edge cases.

Function: std::vector<std::string> Graph::dijkstra(Node* start, Node* end, double& pathWeight)

Parameters: start, end, pathWeight
Return Type: vector of node IDs in path
Rationale: Uses max-heap priority queue to find highest weight path, maintaining O((|E|+|V|)log|V|) runtime. Tracks visited nodes, maintains distances and previous nodes for path reconstruction, and modifies pathWeight using total path weight.

Function: std::tuple<std::string, std::string, double> Graph::findHighest()

Parameters: none
Return Type: tuple of (start ID, end ID, path weight)
Rationale: Implements HIGHEST command by checking all possible node pairs using helper dijkstra() to find path with highest total weight. Checks and returns empty strings and zero if graph is empty or disconnected.

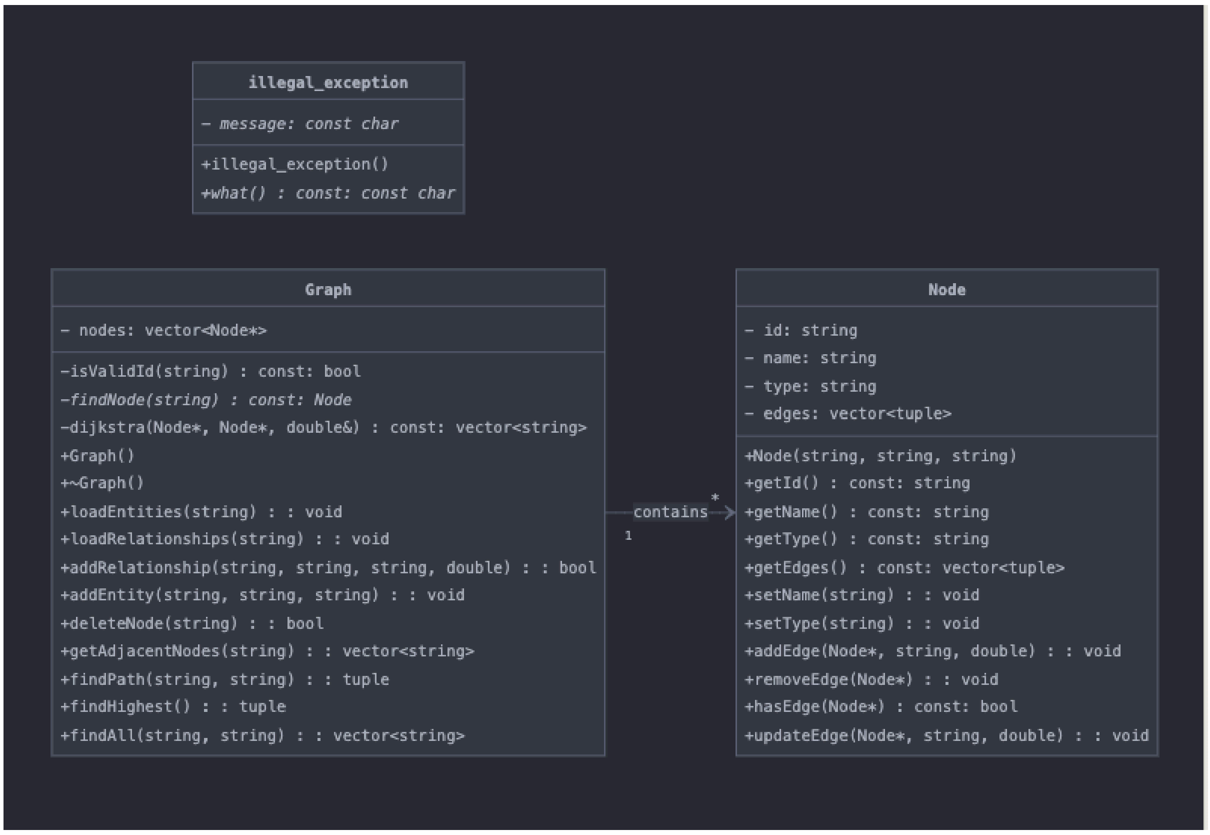
Function: std::vector<std::string> Graph::findAll(const std::string& fieldType, const std::string& fieldString)

Parameters: fieldType ("name" or "type"), fieldString (value to match)
Return Type: vector of matching node IDs
Rationale: Implements FINDALL command by linearly searching all nodes for matching name or type field. Returns empty vector if no matches found.

Function: Node::Node(const std::string& id, const std::string& name, const std::string& type)

Parameters: id, name, type
Return Type: n/a (constructor)
Rationale: Initializes new node with provided identification data in graph.

<u>UML Diagram</u>


illegal_exception
− message: const char
+illegal_exception()
+what() : const: const char
Graph
− nodes: vector<Node*>
−isValidId(string) : const: bool
−findNode(string) : const: Node
−dijkstra(Node*, Node*, double&) : const: vector<string>
+Graph()
+~Graph()
+loadEntities(string) : : void
+loadRelationships(string) : : void
+addRelationship(string, string, string, double) : : bool
+addEntity(string, string, string) : : void
+deleteNode(string) : : bool
+getAdjacentNodes(string) : : vector<string>
+findPath(string, string) : : tuple
+findHighest() : : tuple
+findAll(string, string) : : vector<string>
contains
1
*
Node
− id: string
− name: string
− type: string
− edges: vector<tuple>
+Node(string, string, string)
+getId() : const: string
+getName() : const: string
+getType() : const: string
+getEdges() : const: vector<tuple>
+setName(string) : : void
+setType(string) : : void
+addEdge(Node*, string, double) : : void
+removeEdge(Node*) : : void
+hasEdge(Node*) : const: bool
+updateEdge(Node*, string, double) : : void

<u>Runtime Analysis</u>

Preprocessing: O(|V|)
Creating distance/previous node/visited array: O(|V|) and Creating heap: O(1)

Every vertex is visited once: O(|V|) and every edge is examined once: O(|E|)
Each vertex and edge operation involves at most one heap operation: O(log|V|)
No redundant operations or nested loops over vertices or edges
Connected graph assumption ensures |E| ≥ |V|-1, O(|V|) terms can be absorbed into O(|E|)

Therefore total runtime = O(|V| + |V|log|V| + |E|log|V|)
= O((|E|+|V|)log|V|)