Design Document Project 4 ECE 250

Class Design

Node Class

The Node class represents a node in a graph.

private variables: std::string identifier, std::string name, std::string type, std::vector<Edge*> adjacentEdges, double distance, Node* parent.

These variables store the node's identity, its connections to other nodes, its distance for algorithms like Dijkstra's, and a reference to its parent node. Storing pointers to adjacent edges directly in the Node class makes it easy to quickly access a node's connections. This avoids having to search through the whole graph or store extra information like edge IDs. By keeping these variables private, the Node class ensures that the node's data can only be changed in a controlled way, helping to maintain its integrity.

The constructor sets up the node with the given identifier, name, and type, while setting the distance to negative infinity and the parent to nullptr. The destructor clears the list of adjacent edges and sets the parent pointer to nullptr, preventing memory issues.

Key methods include getters and setters for the node's properties, such as getIdentifier(), getName(), and getType(). The addAdjacentEdge() method prevents adding the same edge more than once to the list of adjacent edges. The getAdjacentEdges() and getAdjacentEdgesEdit() methods allow reading and editing the list of edges. The reason for having two different methods is to keep the list constant when reading and only allow changes when needed.

The setDistance() method sets the node's distance, which is important for finding paths, and setParent() lets you set the parent node, which is useful for printing paths.

Edge Class

Private Variables

The Edge class contains the following private variables:

std::string sourceID and std::string destID: Identifiers for the source and destination nodes, ensuring controlled modification through methods. std::string label: Describes the relationship between nodes, with controlled updates to maintain consistency. double weight: Represents the strength of the relationship, preventing invalid weights through private access. The variables are kept private to ensure controlled access and maintain the integrity of the edge's data, preventing unintended modifications.

Public Functions

The class includes the following public methods:

Constructor (Edge(const std::string&, const std::string&, const std::string&, double)): Initializes an Edge with source, destination, label, and weight. Destructor (~Edge()): Defined for potential future modifications, such as dynamic memory allocation. Getter Methods (getSourceID, getDestID, getLabel, getWeight): Provide read-only access to private variables. Setter Methods (setLabel, setWeight): Allow controlled updates to label and weight, with potential for future validation.

Graph Class

The Graph class represents the entire graph structure, managing the nodes and edges, and providing functionality for adding, removing, and interacting with these components. The maid purpose of this class is its ability to store nodes and edges in a manageable way while allowing commands to execute operations such as graph traversal and pathfinding.

Private Variables:

std::vector<Node> nodes: This vector stores all the nodes of the graph. Each node is a unique entity identified by its ID and contains references to the edges connected to it. I chose to use a vector since it automatically manages memory by resizing itself as elements are added or removed, which can be an advantage since the number of nodes in the graph isn't fixed. It's also relatively easy to add new elements at the end of the vector, making insertion operations fast which is key since we are constantly adding to the graph.

std::vector<Edge*> edges: This vector holds pointers to Edge objects, which connect pairs of nodes in the graph. I chose to store edges as pointers instead of objects directly allows for more flexible memory management. If we were to store Edge objects directly, we would need to copy the entire edge every time we add it, which is less efficient. Using pointers means we can manage memory dynamically and only pass references to edges which reduces memory use since we are not copying values. It also makes it easier to remove edges or modify them without affecting other parts of the graph.

Public Member Functions

addNode(const Node& newNode): Iterates through the existing nodes vector to check if a node with the same identifier (getIdentifier()) already exists. If found, updates the node's name and type using the new node's properties (setName() and setType()). If not found, uses push back to add the new node to the nodes vector. The function ensures graph consistency by preventing duplicate nodes with the same identifier. Updating an existing node instead of creating a duplicate helps maintain the adjacency list and keep it updated. The function accepts a Node object by reference (const Node&) to avoid unnecessary copying, ensuring efficiency while still protecting the input object from modification.

addEdge(const std::string& sourceID, const std::string& destID, const std::string& label, double weight): Checks if an edge between the specified source and destination nodes already exists by iterating through the edges vector. The function explicitly checks both directions (sourceID → destID and destID → sourceID) to ensure edge updates work for undirected graphs. The dynamic

allocation of edges ensures shared access to the same edge object between the graph's edges vector and the adjacency lists of nodes, this is vital to ensure that the memory is properly deallocated one time.

void printAdjacent(const std::string& id) const: The function is designed to display all nodes directly connected to a specified node in the graph. The function ensures accurate results for undirected graphs by checking both the source and destination of each edge, capturing all neighbours regardless of direction.

deleteNode(const std::string& id): The deleteNode function is the primary method for removing a node and its associated edges from the graph. The function uses removeFromAdjacencyList to update all other nodes' adjacency lists, ensuring no references to the node remain. Then, it calls removeEdgesWithID to clear the graph's edge list of edges involving the target node. Finally, the node itself is removed from the node vector. This process ensures a clean deletion, and that memory is properly managed to prevent leaks.

printPath: The function utilizes the results from Path_ModifiedDijkstra to display the path and weight of the longest route from a starting node to a specified end node.

Private Member Functions

removeFromAdjacencyList and removeEdgesWithID: These functions are private helper methods used in the void deleteNode(const std::string& id) function that streamline the process of removing edges related to a specific node ID. By separating these tasks into dedicated functions, the code avoids redundancy and keeps the main logic in deleteNode concise and readable. These functions are kept private to prevent outside code from accidentally modifying the graph, which helps maintain the graph's reliability and prevents potential errors.

Path_ModifiedDijkstra: function implements a modified version of Dijkstra's algorithm for finding the longest path in a graph by assigning negative infinity to initial distances and using a max heap for traversal. Using a max-heap, adjacent nodes are updated if a longer path is found. This ensures all longest paths from the starting node are computed efficiently.

printPathRecursive: function ensures the path is displayed from start to end by using recursion. It is kept private to ensure no output is printed to the console. By ensure this function cannot be called by the user we also ensure that the Path_ModifiedDijkstra function is always called first to ensure the correct path is displayed.

Illegal_Exception Class: used to catch any uppercase letters in a string, works alongside a helper function in the tree class that detects if there is an invalid character based on ascii values. This class works in conjunction with the bool Invalid(const std::string& str) const function in the graph class.

Runtime Analysis

The runtime of the functions for the PATH command can be broken down into two main parts: the initialization operations and the edge relaxations.

First, the code uses multiple loops that iterate over the nodes in the graph. We add all nodes to the max heap, which takes O(1) time. Creating the heap with std::make_heap takes O(v) time for v nodes.

After that, to get the next node to process, we pop the heap using std::pop_heap, which takes **O(log v)** time. Since this is done for each of the **v** nodes, the total time spent is **O(v log v)**.

For each node that was popped from the heap, you look at all its adjacent edges (those in the adjacentEdges vector). In your code, you're iterating over the list of edges, which takes **O(E)** time, where **E** is the number of edges connected to the node. We can conclude that the total time spent relaxing edges is **O(E)**, where **E** is the total number of edges in the graph.

Relaxing an edge involves checking if the distance to the destination node can be updated, which happens in constant time **O(1)**. And a heap operation to update the heap after an edge is relaxed which occurs in **O(log v)**.

When you combine these parts, the total complexity is **O(v log v)** for the heap operations and initializations and **O(E log V)** for relaxing the edges. So, the overall time complexity of your code is **O((|V| + |E|) log |V|)**, where **|V|** is the number of nodes and **|E|** is the number of edges.