Project 4 Design Document
Kelly Kim
ECE 250
1 December 2024
This is the design document for ECE250 Project 4 Graphs

Header Files:
Used four header files in total for this project. It keeps things organized and reduces complexity in the implementation for the Graph data structure.

graph_node.h: storing essential information about the graph node like id, name, and type
graph_edge.h: sets up a graph edge by defining its source node, destination node, label, and weight
graph.h: encapsulates the graph's functionality, enabling the creation, manipulation, and querying of a graph structure composed of nodes and edges
illegal.h: defines the custom illegal_exception class. It handles invalid input scenarios.

Public:

● Graph() ● ~Graph()	● graphNode(const std::string& id, const std::string& name, const std::string& type) ● graphEdge(const std::string& sourceId, const std::string& destinationId, const std::string& label, double weight)

Both constructor and destructor are public, allowing objects of the class to be created and destroyed outside of class. The constructor allows external code to create instances with specific parameters. The destructor cleans up the resources when they are done using. There are no destructors for graphNode and graphEdge because there is nothing to clean up.

● void load (const std::string& filename, const std::string type) ● void relationship (const std::string& sourceId, const std::string& label, const std::string& destinationId, double weight) ● void entity (const std::string& id, const std::string name, const std::string type) ● void print (const std::string id)	● void deleteID (const std::string id) ● void path (const std::string id_1, const std::string id_2) ● void highest() ● void findall (const std::string field_type, const std::string field_string) ● bool exit();

These functions are public because they provide the main ways to interact with the Graph. It allows the user to load data, create or delete components, et cetera. By keeping these functions public, it is easier to use and ensures the Graph remains functional and user-friendly while keeping internal details hidden.

load function will have a void return type, always printing out “success”.
relationship function will have a void return type, printing out “success” or “failure” depending on if the insertion was successful.
entity function will have a void return type, always printing out “success”.

print function will have a void return type, printing out all vertex IDs adjacent to the given vertex. It can also print "failure" or "illegal arguments" depending on the ID given.
deleteID function will have a void return type, printing out "success" or "failure" or "illegal arguments" depending on the ID given.
path function will have a void return type, printing out all vertex IDs on the path from the given vertices. It can also print out "failure" or "illegal arguments" depending on the given ID.
highest function will have a void return type, printing out two vertices with the highest weight path between them. It can also print out "failure" if the graph is empty or totally disconnected.
findall function will have a void return type, printing all vertex IDs with the given field_string. It can also print out "failure" if no vertices of the given field_string exist.
exit function will be used to control when the program stops. Returning true for the program to exit, returning false for the program to keep going.

Variables

graph.h	graph_edge.h	graph_node.h
● bool exitsign = true; ● bool printSuccess = true;	● std::string sourceId; ● std::string destinationId; ● std::string label; ● double weight;	● std::string id ● std::string name ● std::string type

exitsign variable changes to false when the exit function is called to exit the program.
printSuccess variable will be used so when load function calls entity function, it only prints out success once.
sourceId, destinationId, label, weight variable represent the relationship between two nodes in the graph.
id, name, type variable represent the core attributes of a node, allowing unique identification and classification within the graph.

Helper Functions

● void invalid (const std::string& input)

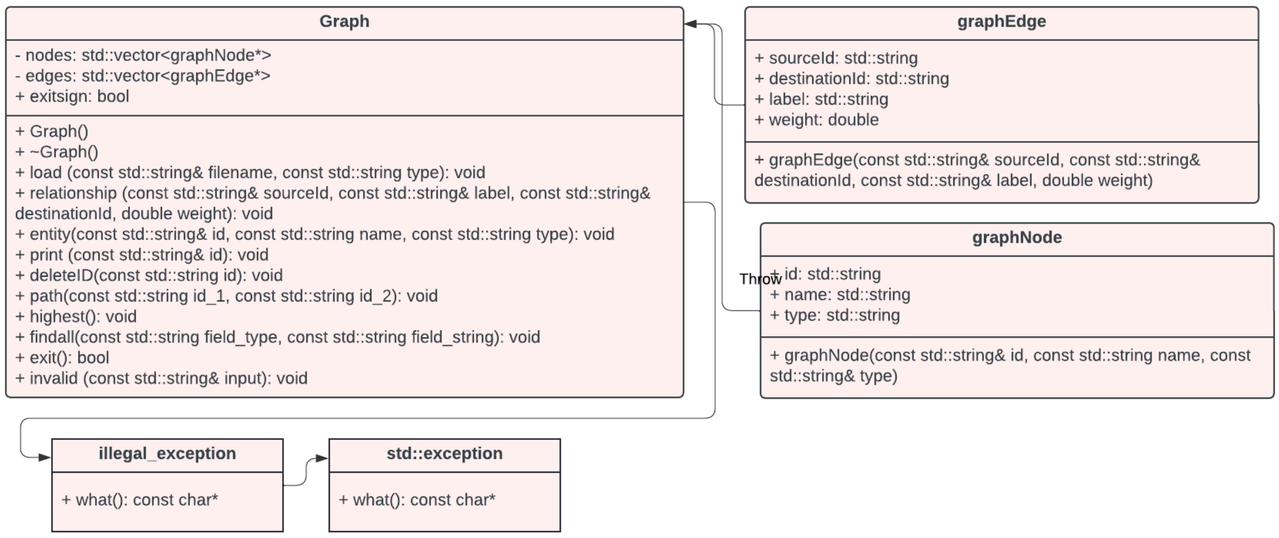
This is a helper function that will be used by main commands.
invalid function will be checking if the input contains any other characters other than uppercase/lowercase letters and numerals. If the input does contain other characters, it will throw illegal_exception. This will be called by print(), delete(), and path() to validate.

<u>Private:</u>

- std::vector<graphNode*> nodes;
- std::vector<graphEdge*> edges;

These two variables will be private because the user should not be directly accessing or modifying them. Graph class provides specific member functions (entity(), relationship(), deleteID(), etc.) to modify the graph's data**.**
nodes stores pointers to all the nodes in the graph.
edges stores pointers to all the edges in the graph.

UML DIAGRAM:


Graph
- nodes: std::vector<graphNode*>
- edges: std::vector<graphEdge*>
+ exitsign: bool
+ Graph()
+ ~Graph()
+ load (const std::string& filename, const std::string type): void
+ relationship (const std::string& sourceId, const std::string& label, const std::string& destinationId, double weight): void
+ entity(const std::string& id, const std::string name, const std::string type): void
+ print (const std::string& id): void
+ deleteID(const std::string id): void
+ path(const std::string id_1, const std::string id_2): void
+ highest(): void
+ findall(const std::string field_type, const std::string field_string): void
+ exit(): bool
+ invalid (const std::string& input): void
graphEdge
+ sourceId: std::string
+ destinationId: std::string
+ label: std::string
+ weight: double
+ graphEdge(const std::string& sourceId, const std::string& destinationId, const std::string& label, double weight)
graphNode
+ id: std::string
+ name: std::string
+ type: std::string
+ graphNode(const std::string& id, const std::string name, const std::string& type)
Throw
illegal_exception
+ what(): const char*
std::exception
+ what(): const char*

Running Time:

Worst-case runtime of PATH algorithm under the assumption that the graph is connected
$O((|E| + |V|)log(|V|))$
→ |E|: number of edges in the graph
→ |V|: number of vertices in the graph

1. First For Loop: Checking if both vertices exist in the graph
- Iterating over the size of nodes which contains all vertices
- **O(|V|)**

2. Initialize
- Initialize highestWeights, previous, and visited to size |V|
- **O(|V|)**

3. While Loop: Priority Queue
- priorityQueue is used to manage traversal order
- Algorithm sorts this queue in descending order of weights, emulating a max-priority queue
- In a graph with |V| vertices and |E| edges, the heap can hold at most |V| elements
- Each insertion and removal operation on the heap takes O(log |V|)

4. While Loop → For Loop: Edges
- Iterating over the edges to find connected vertices and update weights
- **O(|E| log (|V|))**

Total Time Complexity
1. **Checking:** O(|V|)
2. **Initialization:** O(|V|)
3. **Priority Queue:** O((|E|+|V|) log (|V|))
   a. **Edge:** O(|E| log |V|)
   b. **Push/Pop:** O(|V| log (|V|)
   c.

Therefore, the total time complexity is dominated by **O((|E|+|V|) log (|V|)).**