ECE250 Project 4: Graphs

Due: Monday, December 2, 2024 at 11:00pm to the dropbox on Learn

Overview

In this project, you must design and implement a graph data structure to store a weighted, undirected graph, and perform various lookup operations on it.

Background: Multi-Relational Graphs

A multi-relational graph is one in which nodes represent entities and edges represent relationships between entities. For instance, nodes may represent students and edges may represent courses that they took together, where two students are connected by an edge if they took a course together. If we allow for there to be multiple types of relationships and multiple types of nodes, we get a multi-relational graph. For instance, nodes may represent students, instructors, TAs, and classrooms; edges may represent courses, clubs, and hometowns. We further enhance our graph by requiring that our edges all have weights, indicating the strength of the relationship. We can do very interesting queries on such graphs, and these queries have proven invaluable in fields as diverse as artificial intelligence, search engines, and medical diagnoses.

To create a multi-relational graph we begin with a regular graph and allow each node and edge to have a label, represented as a string, indicating its type. Nodes also have a unique identifier, which is also a string, which allows us to look up a node uniquely. Finally, nodes have names, also given by strings, which need not be unique. By using both names and identifiers we can handle the case where two nodes may have the same name (both "John Smith", say) but represent two separate people.

Input Files

In this project you will read from input files to load data into your graph. There are two types of input files: entities and relationships. They are read using separate commands in the table below.

Entity files contain only nodes for your graph and each line follows the format: string_ID string_name string_type. Note that all three fields are separated by spaces. Fields are guaranteed to not contain whitespace. Here is a sample:

```
7C7CAEED On_rank_correlation_in_information_retrieval_evaluation paper
7AEE29E3 The_Voting_Model_for_People_Search paper
7D68490B Document_clustering_with_committees paper
```

Relationship files contain only edges and follow the format: string_sourceID string_label string_destinationID double weight, where source_ID is the ID of the source node and destination_ID is the ID of the destination. The weight is always strictly positive. It is possible that either the source or the destination are not in the graph, and if so that edge should be ignored. Labels are not unique, nor should they be, but they are guaranteed to not contain any whitespace. Here is a sample:

```
7D1D0E19 paper_cite_paper 812313D9 1.2
75D262B7 paper_cite_paper 7E932EDF 43.3
```

Note that relationships are *undirected* edges, and cycles are allowed.

Program Design and Documentation

You must use proper Object-Oriented design principles to implement your solution. You will create a design using classes which have appropriately private/protected data members and appropriately public services (member functions). It is not acceptable to simply make all data members public. **Structs are not permitted.** You may notice that you are writing a generic graph implementation for this project. Write a short description of your design to be submitted along with your C++ solution files for marking according to the template posted to Learn. **You may not use the STL except for <vector>, <algorithm>,** and **<tuple>.**

Input/Output Requirements

In this project, you must create a test program that must read commands from standard input and write the output to standard output. The commands are described below.

Note: The highest-weight path may not be unique. This will affect the "PATH" command below. You may assume that any time the PATH command is called, the path between the two vertices is guaranteed to be unique. You may also assume that no self edges will be present in the input data for any command.

Command	Parameters	Description	Output
LOAD	Filename type	Load a dataset into a graph. This command may not be present in all input files or may be present multiple times. You may assume there are no illegal arguments in the datasets. All rows in these files will have the same format as above. Type is a string and is guaranteed to be one of "entities" or "relationships"	**success** Note: This command should output "success" no matter what. It should not output one "success" per node or edge. If an entity exists in the graph already with the ID being inserted, update that entity's other fields per the input file. If a relationship already exists in the graph between two nodes, update the label and weight.
RELATIONSHIP	*Source_ID* *label* *Destination_ID* *weight*	Insert a new edge into the graph from vertex *Source_ID* to vertex *Destination_ID* with label *label* and weight *weight*. If either *vertices* are not in the graph, this command will fail.	**success** if the insertion was successful. If an edge already exists between the two vertices, you must update the *label* and *weight*, and this counts as a "success". Do not allow multiple edges between vertices! **failure** if either entity does not exist in the graph
ENTITY	*ID name type*	Insert a new entity into the graph with the given ID, name, and type. If an entity with the given ID already exists, update its name and type. This command always succeeds.	**success** this command always succeeds.
PRINT	*ID*	Print all vertices adjacent to vertex with ID *ID*. The order in which you print the vertices is not important. The Autograder will be programmed to handle any ordering.	**ID_1 ID_2 ID_3 …** Print all vertex IDs adjacent to the given vertex on a single line with spaces between them, followed by a newline character. If there are no such vertices, print a blank line. **failure** if vertex with the given ID is not in the graph **illegal argument** (see below the table)
DELETE	*ID*	Delete the vertex with ID *ID* and any edges containing it. Note that this means you will need to remove the vertex from the edge set of all vertices adjacent to it if you have designed your code this way. This command may produce an unconnected graph.	**success** if the vertex is in the graph and was erased **failure** if the vertex is not in the graph, including the case where the graph is empty **illegal argument** (see below the table)
PATH	ID_1 ID_2	Print the vertices and labels along the *highest weight* path between vertices with IDs ID_1 and ID_2. You may assume that if this command is called, the highest weight path between the vertices is guaranteed to be unique if it exists. The output format is specified in the next cell in this table.	**ID_1 ID_a ID_b … ID_2 W** Print all vertex IDs on the path from the given vertices with spaces between them, followed by a newline character. Note that vertex IDs ID_1 and ID_2 must be printed as well. W is the weight of the path, computed as the sum of the weights of all edges in the path. The exact order in which you print does not matter (for instance, if you print ID_1 first or ID_2 first). The autograder will handle that. However, W must come at the end.

			failure if the graph is empty or there is no path between the vertices or any one of the vertices is not in the graph. **illegal argument** (see below the table)
HIGHEST		Determine the two vertices with the highest weight path between them.	**ID_1 ID_2 W** Where ID_1 is the source vertex and ID_2 is the destination vertex, W is the path weight between them. The order of printing ID_1 or ID_2 does not matter as long as W is at the end **failure** if the graph is empty or totally disconnected (that is, has zero edges)
FINDALL	Field_type Field_string	Output a list of unique entities with the given Field string. The order does not matter. Field_type will be one of "name" or "type", and Field_string will be the string to search for. For instance: `FINDALL name Mike_Cooper_Stachowsky` or `FINDALL type instructor`	**ID_1 ID2 …** Print all vertex IDs with the given Field_string on a single line with spaces between them, followed by a newline character. **failure** No vertices of the given Field_type exist
EXIT		Last command for all input files.	This command does not print any output

Illegal arguments: For the commands *PRINT, DELETE,* and *PATH* you must handle invalid input. An ID is invalid if it contains any characters other than upper- or lower-case English letters and numerals. Whitespace will not be tested. Your code must throw an illegal_argument exception, catch it, and output **"illegal argument"** (without quotes) if it is caught. To do this, you will need to:

a. Define a class for this exception, call it **illegal_exception**
b. Throw an instance of this class when the condition is encountered using this line:

```
throw illegal_exception();
```

c. Use a try/catch block to handle this exception and print the desired output of the command

You must analyze the **worst-case** runtime of your implementation of your PATH algorithm under the assumption that the graph is connected. Your implementation must meet the runtime of O((|E|+|V|)log(|V|)), where |E| is the number of edges and |V| is the number of vertices in the graph.

Valgrind and Memory Leaks, Formatting, and Commenting

10% of the grade of this project will be allocated to memory leaks. We will be using the Valgrind utility to do this check. The expected behaviour of Valgrind is to indicate 0 errors and 0 leaks possible, with all allocated bytes freed. A penalty of 10% will be applied for poor commenting or code organization. A penalty of 15% will be applied if non-permitted header files or structs are included, *even if they are not used*.

Test Files

Learn contains some sample input files with the corresponding output. The files are named test01.in, test02.in and so on with their corresponding output files named test01.out etc.
All test files are provided as-is. Your code must be able to parse them.

Submitting your Program

Once you have completed your solution and tested it comprehensively on your own computer or the lab computers, you should transfer your files to the eceUbuntu server and test there. We perform automated tested on this platform, so if your code works on your own computer but not on eceUbuntu it will be considered incorrect. **A makefile is required for this project since the exact source structure you use will be unique to you.** Do not submit the test files or precompiled binaries.

Once you are done your testing you must create a compressed file in the tar.gz format, that contains:

- A typed document, maximum of four pages, describing your design. Submit this document in PDF format. The name of this file should be xxxxxxxx_design_pn.pdf where xxxxxxxx is your maximum 8-character UW user ID (for example, I would use my ID “mstachow”, not my ID “mstachowsky”, even though both are valid UW IDs), and n is the project number. In my case, my file would be mstachow_design_p4.pdf.
- A test *.cpp file that contains your main function that reads the commands and writes the output
- Required header files that you created.
- Any additional support files that you created.
- A makefile, named Makefile, with commands to compiler your solution and creates an executable. Do not use the -o output flag in your makefile. The executable’s name must be a.out.

The name of your compressed file should be **xxxxxxxx_p4.tar.gz**, where xxxxxxxx is your UW ID as above.