COP 4530 Assignment 4

Few important points

Students working on Code

1. The students responsible for coding part should work on Part-A implementation only.
2. The code should handle various datatypes such as characters, integers, special symbols.
3. You should report time complexity of our own implementation. It is important that you ensure, that your version of code is optimal.

Students working on Report

1. The students working on report should explain the Dijkstra's algorithm with examples.
2. Explain working of the code
3. Write C++ code for part-B. The main task is to ensure your tree is balanced or complete.
4. You will use hash table and ensure disjoint sets are represented using tree.
5. No need to provide report for Task-B, try to construct well documented code.

PART- A

Instructions

For Assignment 4, you will be implementing an undirected weighted Graph ADT and performing Dijkstra's Algorithm to find the shortest path between two vertices. Your graph can be implemented using either an adjacency list, adjacency matrix, or an incidence matrix. Your graph will implement methods that add and remove vertices, add and remove edges, and calculate the shortest path. Each vertex in your graph will have a string label that will help identify that vertex to you and the test file.

A large portion of this assignment is researching and implementing Dijkstra's Algorithm. There is information about this algorithm in your textbook, slides and widely available on the web.

Abstract Class Methods

`void addVertex(std::string label)`
Creates and adds a vertex to the graph with label. No two vertices should have the same label.

`void removeVertex(std::string label)`
Removes the vertex with label from the graph. Also removes the edges between that vertex and the other vertices of the graph.

`void addEdge(std::string label1, std::string label2, unsigned long weight)`
Adds an edge of value weight to the graph between the vertex with label1 and the vertex with label2. A vertex with label1 and a vertex with label2 must both exist, there must not already be an edge between those vertices, and a vertex cannot have an edge to itself.

`void removeEdge(std::string label1, std::string label2)`
Removes the edge from the graph between the vertex with label1 and the vertex with label2. A vertex with label1 and a vertex with label2 must both exist and there must be an edge between those vertices

`unsigned long shortestPath(std::string startLabel, std::string endLabel, std::vector<std::string> &path)`
Calculates the shortest path between the vertex with startLabel and the vertex with endLabel using Dijkstra's Algorithm. A vector is passed into the method that stores the shortest path between the vertices. The return value is the sum of the edges between the start and end vertices on the shortest path.

Examples

Below in an example of the functionality of the implemented graph:

```
std::vector<std::string> vertices1 = { "1", "2", "3", "4", "5", "6" };
std::vector<std::tuple<std::string, std::string, unsigned long>>
edges1 = { {"1", "2", 7}, {"1", "3", 9}, {"1", "6", 14}, {"2", "3",
10}, {"2", "4", 15}, {"3", "4", 11}, {"3", "6", 2}, {"4", "5", 6},
{"5", "6", 9} };

for (const auto label : vertices1) g.addVertex(label);
for (const auto &tuple : edges1) g.addEdge(std::get<0>(tuple),
std::get<1>(tuple), std::get<2>(tuple));
```

```
g.shortestPath("1", "5", path); // == 20
g.shortestPath("1", "5", path); // = { "1", "3", "6", "5" }
```

Deliverables

Please submit complete projects as zipped folders. The zipped folder should contain:

- Graph.cpp (Your written Graph class)
- Graph.hpp (Your written Graph class)
- GraphBase.hpp (The provided base class)
- PP4Test.cpp (Test file)
- catch.hpp (Catch2 Header)
- Detailed Report

And any additional source and header files needed for your project.

Hints

Though it may be appealing to use an adjacency matrix, it might be simpler to complete this algorithm using an adjacency list for each vertex.

I suggest using a separate class for your edge and vertex.

Remember to write a destructor for your graphs!

Rubric

Any code that does not compile will receive a zero for this project.

Criteria
Your code should calculate the distance of the shortest path for graph 1
Your code should have the labels for the shortest path for graph 1
Your code should calculate the distance of the shortest path for graph 2 **Your code should have the labels for the shortest path for graph 2**
Your code should calculate the distance of the shortest path for graph 3 **Your code should have the labels for the shortest path for graph 3**
Your code implements Dijkstra's Algorithm
Code uses object oriented design principles **(Separate headers and sources, where applicable)** **Code is well documented**

Part-2 (implement disjoint set using hash-table – This part will only be done by students working on report)

In part-2, students have some flexibility. You will implement disjoint-sets using hash table.

As we all know, a disjoint–set is a data structure that keeps track of a set of elements partitioned into several disjoint or non-overlapping subsets. In simple words, when we consider a disjoint set, we say it is a group of sets where no item or element or value can be in more than one set. Thus this is also known as union-find data structure as it supports union and find operations on subsets.

Let's define each operation. Your code should have all these operations.

Find: Returns the representative of the set. In simple words it basically determines in which particular subset a particular element is in and then returns the representative of that particular set.
Union: This is a simple merge operation, since it merges two different subsets into a single subset. After this the representative of set 1 will also become representative of set 2.
Makeset: Creates set based on given elements.

Let's look at an example to see how each operations work

Let's consider we have 4 sets, each set with only one element.
S1 = [1]
S2 = [2]
S3 = [3]
S4 = [4]
Since disjoint-sets are represented as trees, we will have parent node pointer pointed to themselves or NULL..

In above case or scenario, whenever we perform or call find operator, it will return following:
The find operator on element i will return its representative S^i set, where 1<=i<=4, this find(i)=I,
Now when we perform union operation we will have following:

When we call union (S2, S3), we will have new disjoint set S2 and the element for S2 will be S2(2,3),
So new sets are
S1 = [1]
S2 = [2,3]
S4 = [4]
Thus when we run find(3) on these sets, we will have 2 as the output.

Similarly when we call union (S2, S4), we will have disjoint set S2, with elements S2 = [2,3,4] and new sets are S1 = [1] and S2 = [2,3,4]

And Find(4) = Find(3) = Find(2) will return S2 as representative set.

Given disjoint set data structure is a forest or multiway tree, you might reach a condition where your trees are highly unbalanced. Thus in this assignment you have to ensure your tree is balanced or complete. More details can be found in your book, slides or even in this article (https://en.wikipedia.org/wiki/Disjoint-set_data_structure)

Rubric

Criteria
Your code uses hash table
Your code ensures tree is used to represent disjoint sets
Your code should have three basic operations (find, union, makeset)
Your code should have the labels for the shortest path for graph 2
Your code ensures tree is balanced and complete (you can use any tree, Binary tree, B+ tree, AVL tree etc) **Code uses object oriented design principles (Separate headers and sources, where applicable)**
Code is well documented

Deliverables

1. Submit all the scripts (including .cpp, .h files) in a zip folder