Programming Assignment 10

Minimum Spanning Tree (Prim's Algorithm)

Approved Includes

```
<cassert>
<cmath>
<cstddef>
<iostream>
<list>
<sstream>
<stack>
<queue>
<unordered_map>
<unordered_set>
<vector>
"graph.h"
```

Code Coverage

You must submit a test suite for each task that, when run, covers at least 90% of your code. You should, at a minimum, invoke every function at least once. **Best practice is to also check the actual behavior against the expected behavior, e.g. verify that the result is correct. You should be able to do this automatically, i.e. write a program that checks the actual behavior against the expected behavior.**

Your test suite should include ALL tests that you wrote and used, including tests you used for debugging. You should have MANY tests.

Starter Code

```
graph.h
graph_compile_test.cpp
graph_tests.cpp
Makefile
```

Files to Submit

```
graph.h
graph_tests.cpp
```

Task 1: Undirected Graph

Implement a data structure to store an **undirected** graph. You should modify your digraph code.

Requirements

Files

`graph.h` - contains the Graph class definition (define the methods **inside** the class)
`graph_tests.cpp` - contains the test cases and test driver (main)

Class

```
class Graph;
```

You can represent the Graph internally however you want. This could be adjacency lists, an adjacency matrix, sets of Vertex and Edge objects, linked Vertex and/or Edge objects, or even some combination of methods. In class we learned the adjacency list and adjacency matrix representations, so I would encourage you to use one of those.

Functions

Constructors

`Graph()` – makes an empty graph.

`Graph(const Graph&)` - constructs a deep copy of a graph

`Graph& operator=(const Graph&)` - assigns a deep copy of a graph

`~Graph()` - destructs a graph (frees all dynamically allocated memory)

Capacity

`size_t vertex_count() const` - the number of vertices in the graph

`size_t edge_count() const` - the number of edges in the graph

Element Access

`bool contains_vertex(size_t id) const` - return `true` if the graph contains a vertex with the specified identifier, `false` otherwise.

`bool contains_edge(size_t u, size_t v) const` - return `true` if the graph contains an edge with the specified members (as identifiers), `false` otherwise.

`double cost(size_t u, size_t v) const` - returns the weight of the edge between u and v, or `INFINITY` if none exists.

Modifiers

`bool add_vertex(size_t id)` - add a vertex with the specified identifier if it does not already exist, return `true` on success or `false` otherwise.

`bool add_edge(size_t u, size_t v, double weight=1)` - add an **undirected** edge between u and v with the specified weight if there is not one already, return `true` on success, `false` otherwise. If you use adjacency lists, make sure to update both u's and v's lists.

`bool remove_vertex(size_t id)` - remove the specified vertex from the graph, including all edges of which it is a member, return `true` on success, `false` otherwise.

`bool remove_edge(size_t u, size_t v)` - remove the specified edge from the graph, but do not remove the vertices, return `true` on success, `false` otherwise.

Optional

`Graph(Graph&&)` - move constructs a deep copy of a graph

`Graph& operator=(Graph&&)` - move assigns a deep copy of a graph

Task 2: Prim's Algorithm

Implement Prim's Algorithm as a method of the Graph class from Task 1.

Requirements

Files

`graph.h` - contains the Graph class definition (define the methods **inside** the class)
`graph_tests.cpp` - contains the test cases and test driver (main)

Functions

`std::list<std::pair<size_t,size_t>> prim()` - compute a minimum spanning tree using Prim's algorithm. Return a list of edges. There may be more than one possible spanning tree depending on the starting vertex. Any correct (minimum weight) tree will be recognized as such. If a MST does not exist, then the return value should be an empty list.

`double distance(size_t id) const` - assumes Prim has been run, returns the cost of the edge that connects this vertex to the minimum spanning tree, or 0 is the vertex is the root of the tree, or `INFINITY` if the vertex is not part of the tree (i.e. the graph is not connected).

Visualization (optional)

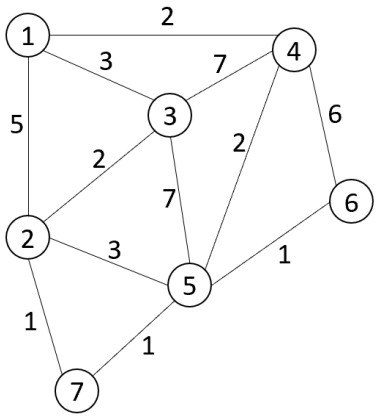
`void print_minimum_spanning_tree(std::ostream& os=std::cout) const` - assumes Prim has been run, pretty prints the minimum spanning tree as a sequences of lines with the format <vertex> --{<edge weight>} <vertex>

Example (for Tasks 1 and 2)

```
std::cout << "make an empty graph" << std::endl;
Graph G;

std::cout << "add vertices" << std::endl;
for (size_t n = 1; n <= 7; n++) {
      G.add_vertex(n);
}

std::cout << "add undirected edges" << std::endl;
G.add_edge(1,2,5);  // 1 --{5} 2; (edge between 1 and 2 with weight 5)
G.add_edge(1,3,3);
G.add_edge(2,3,2);
G.add_edge(2,5,3);
G.add_edge(2,7,1);
G.add_edge(3,4,7);
G.add_edge(3,5,7);
G.add_edge(4,1,2);
G.add_edge(4,6,6);
G.add_edge(5,4,2);
G.add_edge(5,6,1);
G.add_edge(7,5,1);
```




```
std::cout << "G has " << G.vertex_count() << " vertices and ";
std::cout << G.edge_count() << " edges" << std::endl;

std::cout << "compute a minimum spanning tree" <<std::endl;
std::list<std::pair<size_t,size_t>> = G.prim();

std::cout << "print minimum spanning tree" <<std::endl;
double tree_cost = 0;
for (const std::pair<size_t,size_t>& edge : mst) {
    std::cout << edge.first << " --{"<<G.cost(edge.first,edge.second)<<"} " <<
edge.second << ";" << std::endl;
    tree_cost += G.cost(edge.first,edge.second);
}
std::cout << "tree cost = " << tree_cost <<std::endl;
```

Example Output

```
make an empty graph
add vertices
add undirected edges
G has 7 vertices and 12 edges
compute a minimum spanning tree
print minimum spanning tree
2 --{1} 7;
5 --{1} 7;
6 --{1} 5;
3 --{2} 2;
4 --{2} 5;
1 --{2} 4;
tree cost = 9
```