Graph

Clase que representa un grafo. Por sí solo no hace nada.

REQUIERE: NaturalNumber

REQUERIDO POR: Bipartite, MinSpanningTree, Shortest Paths, etc.

```
#pragma once

#include <algorithm>
#include <cassert>
#include <iostream>
#include <vector>

#include "NaturalNumber.hpp"

template <class Iter, class T>
Iter find_binary(const Iter& first, const Iter& last, const T& t)
{
    auto it = std::lower_bound(first, last, t);

    if (it == last || *it != t)
        return last;

    return it;
}

// simple undirected graph
class Graph
{
public:
    using size_type = long long;

    using Vertex = std::int64_t;

    enum WORKAROUND_UNTIL_CPP17
    {
        INVALID_VERTEX = -1
    };
    //  inline static constexpr Vertex INVALID_VERTEX = -1; // Uncomment with
    // c++17

    using weight_t = std::int64_t;

    // something larger than weight_t, for when you have that weight_t doesn't
```

```
// properly hold a sum of weight_t (for example, if weight_t = char).
using sumweight_t = std::int64_t;

struct Neighbor; // Represents a half-edge (vertex,weight)

struct Edge; // (from,to,weight)

using neighbor_list = std::vector<Neighbor>;
using neighbor_const_iterator = neighbor_list::const_iterator;
using neighbor_iterator = neighbor_list::iterator;

explicit Graph(Vertex numberOfVertices = 0)
    : m_numvertices(std::max<Vertex>(0, numberOfVertices))
    , m_graph(m_numvertices)
{}

size_type degree(Vertex a) const { return m_graph[a].size(); }

// Graph modification functions
Vertex add_vertex()
{
    m_graph.emplace_back(); // empty vector
    return m_numvertices++;
}

void add_edge(Vertex from, Vertex to, weight_t w = 1)
{
    m_graph[from].emplace_back(to, w);
    m_graph[to].emplace_back(from, w);
    ++m_numedges;
    m_neighbors_sorted = false;
}

void add_edge(const Edge& e) { add_edge(e.from, e.to, e.weight()); }

template <class EdgeContainer>
void add_edges(const EdgeContainer& edges)
{
    for (auto& e : edges)
        add_edge(e);
}

void add_edges(const std::initializer_list<Edge>& edges)
{
    for (auto& e : edges)
```

```
        add_edge(e);
}

bool add_edge_no_repeat(Vertex from, Vertex to, weight_t w = 1)
{
    if (is_neighbor(from, to))
        return false;

    add_edge(from, to, w);
    return true;
}

void sort_neighbors()
{
    if (m_neighbors_sorted)
        return;

    for (auto& adj_list : m_graph)
        sort(adj_list.begin(), adj_list.end());

    m_neighbors_sorted = true;
}

// Get Graph Info
Vertex num_vertices() const { return m_numvertices; }
size_type num_edges() const { return m_numedges; }

inline const neighbor_list& neighbors(Vertex n) const { return m_graph[n]; }
inline const neighbor_list& outneighbors(Vertex n) const
{
    return m_graph[n];
}
inline const neighbor_list& inneighbors(Vertex n) const
{
    return m_graph[n];
}

using all_vertices = basic_natural_number<Vertex>;
auto vertices() const { return all_vertices(num_vertices()); }

std::vector<Edge> edges() const
{
    std::vector<Edge> total;

    for (auto u : vertices())
```

```
    {
        for (auto v : m_graph[u])
        {
            if (v > u)
                total.emplace_back(u, v, v.weight());
        }
    }

    return total;
}

bool is_neighbor(Vertex from, Vertex to) const
{
    if (degree(from) > degree(to))
        std::swap(from, to);

    auto& NF = neighbors(from);

    if (m_neighbors_sorted)
        return std::binary_search(NF.begin(), NF.end(), to);

    for (auto& a : NF)
    {
        if (a == to)
            return true;
    }

    return false;
}

weight_t edge_value(Vertex from, Vertex to) const
{
    if (degree(from) > degree(to))
        std::swap(from, to);

    auto neigh = get_neighbor(from, to);

    if (neigh == neighbors(from).end() || *neigh != to)
        return 0;

    return neigh->weight();
}

neighbor_const_iterator get_neighbor(Vertex from, Vertex to) const
{
```

```
    auto first = m_graph[from].begin();
    auto last = m_graph[from].end();

    if (m_neighbors_sorted)
        return find_binary(first, last, to);

    return std::find(first, last, to);
}

neighbor_iterator get_neighbor(Vertex from, Vertex to)
{
    auto first = m_graph[from].begin();
    auto last = m_graph[from].end();

    if (m_neighbors_sorted)
        return find_binary(first, last, to);

    return std::find(first, last, to);
}

// Start class definitions
struct Neighbor
{
    explicit Neighbor() : vertex(INVALID_VERTEX), m_weight(0) {}

    explicit Neighbor(Vertex v, weight_t w = 1) : vertex(v), m_weight(w) {}

    inline operator Vertex() const { return vertex; }

    weight_t weight() const { return m_weight; }

    void set_weight(weight_t w) { m_weight = w; }

    Vertex vertex{INVALID_VERTEX};

private:
    // comment out if not needed, and make set_weight do nothing, and make
    // weight() return 1
    weight_t m_weight{1};
};

struct Edge
{
    Vertex from{INVALID_VERTEX};
    Vertex to{INVALID_VERTEX};
```

```
        Edge() : m_weight(0) {}
        Edge(Vertex f, Vertex t, weight_t w = 1) : from(f), to(t), m_weight(w)
        {}

        Vertex operator[](bool i) const { return i ? to : from; }

        // replace by "return 1" if weight doesn't exist
        weight_t weight() const { return m_weight; }
        void change_weight(weight_t w) { m_weight = w; }

        bool operator==(const Edge& E) const
        {
            return ((from == E.from && to == E.to) ||
                    (from == E.to && to == E.from)) &&
              m_weight == E.m_weight;
        }

    private:
        weight_t m_weight{1};
    };

private:
    // Graph member variables
    size_type m_numvertices{0};
    size_type m_numedges{0};

    std::vector<neighbor_list> m_graph{};
    bool m_neighbors_sorted{false};
};
```