BipartiteGraph

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

REQUIERE: Graph, NaturalNumber

REQUERIDO POR: BipartiteMatcher

```cpp
#pragma once

// maybe not needed, only "Neighbor" and "Edge" are needed.
#include "Graph.hpp"

class BipartiteGraph
{
public:
    using size_type = std::int64_t;
    using Vertex = std::int64_t;
    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;

    using Neighbor = Graph::Neighbor; // Represents a half-edge (vertex,weight)

    using Edge = Graph::Edge; // (from,to,weight)

    using neighbor_list = std::vector<Neighbor>;
    using neighbor_const_iterator = neighbor_list::const_iterator;
    using neighbor_iterator = neighbor_list::iterator;
    /****** END using definitions *****/

public:
    BipartiteGraph(size_type x, size_type y) : m_X(x), m_Y(y) {}

    size_type degreeX(Vertex x) const { return m_X[x].size(); }
    size_type degreeY(Vertex y) const { return m_Y[y].size(); }

    size_type num_verticesX() const { return m_X.size(); }
    size_type num_verticesY() const { return m_Y.size(); }

    size_type num_vertices() const { return num_verticesX() + num_verticesY(); }

    using all_vertices = basic_natural_number<Vertex>;
    auto verticesX() const { return all_vertices(num_verticesX()); }
```

```
auto verticesY() const { return all_vertices(num_verticesY()); }

const auto& X() const { return m_X; }
const auto& Y() const { return m_Y; }

const neighbor_list& neighborsX(Vertex a) const { return m_X[a]; }
const neighbor_list& neighborsY(Vertex a) const { return m_Y[a]; }

void add_edge(Vertex x, Vertex y, weight_t w = 1)
{
    m_X[x].emplace_back(y, w);
    m_Y[y].emplace_back(x, 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);
}

void FlipXandY() { std::swap(m_X, m_Y); }

void sort_neighbors()
{
    if (m_neighbors_sorted)
        return;

    for (auto& x : m_X)
        std::sort(std::begin(x), std::end(x));

    for (auto& y : m_Y)
        std::sort(std::begin(y), std::end(y));

    m_neighbors_sorted = true;
```

```
    }

    Graph UnderlyingGraph() const
    {
        Graph G(num_vertices());

        for (Vertex v = 0; v < num_verticesX(); ++v)
        {
            for (auto u : neighborsX(v))
            {
                G.add_edge(v, u + num_verticesX(), u.weight());
            }
        }

        return G;
    }

private:
    std::vector<neighbor_list> m_X{};
    std::vector<neighbor_list> m_Y{};
    size_type m_numedges{0};
    bool m_neighbors_sorted{false};
};
```