Bipartite Matching

Encuentra el apareamiento máximo en una gráfica bipartita.

REQUIERE: BipartiteGraph

- Tiempo de ejecución: $O(VE)$, pero en general es muuucho más rápido que eso.

Nota: Encuentra el apareamiento de cardinalidad máxima, no el de peso máximo. Si se requiere max weight matching, mejor usar max flow con el truco de agregar dos vértices fantasmas.

```
#include "BipartiteGraph.hpp"
#include <deque>
#include <queue>
#include <stack>

class BipartiteMatcher
{
public:
    using Vertex = BipartiteGraph::Vertex;
    using Edge = Graph::Edge;

    BipartiteMatcher(const BipartiteGraph& G)
        : m_Xmatches(G.num_verticesX(), -1), m_Ymatches(G.num_verticesY(), -1)
    {
        CreateInitialMatching(G);
        Augment(G);
    }

    Vertex MatchX(Vertex x) const { return m_Xmatches[x]; }
    Vertex MatchY(Vertex y) const { return m_Ymatches[y]; }

    int size() const { return m_size; }

    std::vector<Edge> Edges() const
    {
        std::vector<Edge> matching;
        matching.reserve(size());

        for (auto x : indices(m_Xmatches))
        {
            auto y = MatchX(x);
            if (y >= 0)
                matching.emplace_back(x, y);
        }
```

```
        return matching;
    }

private:
    void CreateInitialMatching(const BipartiteGraph& G)
    {
        m_unmatched_in_X.reserve(G.num_verticesX());

        for (auto x : G.verticesX())
        {
            for (auto y : G.neighborsX(x))
            {
                if (m_Ymatches[y] < 0)
                {
                    m_Xmatches[x] = y;
                    m_Ymatches[y] = x;
                    ++m_size;
                    break;
                }
            }
            if (m_Xmatches[x] < 0)
                m_unmatched_in_X.emplace_back(x);
        }
    }

    // returns false if no augmenting path was found
    void Augment(const BipartiteGraph& G)
    {
        size_t num_without_augment = 0;
        auto it = m_unmatched_in_X.begin();

        while (num_without_augment < m_unmatched_in_X.size())
        {
            if (it == m_unmatched_in_X.end())
                it =
                  m_unmatched_in_X.begin(); // Imagine this a circular buffer.

            if (FindAugmentingPath(G, *it))
            {
                *it = m_unmatched_in_X.back();
                m_unmatched_in_X.pop_back();
                num_without_augment = 0;
            }
            else
```

```
        {
            ++it;
            ++num_without_augment;
        }
    }
}

bool FindAugmentingPath(const BipartiteGraph& G, Vertex x)
{
    const Vertex not_seen = -1;
    // In order to reconstruct the augmenting path.
    std::vector<Vertex> parent(G.num_verticesY(), -1);

    std::queue<Vertex> frontier; // BFS
    frontier.emplace(x);

    while (!frontier.empty())
    {
        auto current_x = frontier.front();
        frontier.pop();

        for (Vertex y : G.neighborsX(current_x))
        {
            if (parent[y] != not_seen)
                continue;

            parent[y] = current_x;

            auto new_x = m_Ymatches[y];
            if (new_x == -1)
            {
                ApplyAugmentingPath(y, parent);
                assert(m_Xmatches[x] != -1);
                return true;
            }

            frontier.emplace(new_x);
        }
    }

    return false;
}

void ApplyAugmentingPath(Vertex y, const std::vector<Vertex>& parent)
{
```

```
            ++m_size;

            Vertex x = parent[y];
            do
            {
                auto new_y = m_Xmatches[x]; // save it because I'll erase it

                // new matches
                m_Ymatches[y] = x;
                m_Xmatches[x] = y;

                y = new_y;
                x = parent[y];
                assert(x != -1);
            } while (y != -1);
        }

        int m_size{0};
        std::vector<Vertex> m_Xmatches{}; // -1 if not matched
        std::vector<Vertex> m_Ymatches{}; // -1 if not matched

        std::vector<Vertex> m_unmatched_in_X{};
};

using namespace std;

int main()
{
    BipartiteGraph G(4, 4);
    G.add_edge(0, 3);
    G.add_edge(0, 1);
    G.add_edge(1, 0);
    G.add_edge(1, 1);
    G.add_edge(2, 0);
    G.add_edge(2, 1);
    G.add_edge(3, 0);
    G.add_edge(3, 2);
    G.add_edge(3, 3);

    BipartiteMatcher BM(G);
    cout << "Best match has size " << BM.size() << ", which is:" << endl;
    for (auto edge : BM.Edges())
    {
        cout << '(' << edge.from << ',' << edge.to << ')' << endl;
    }
```

```
    return 0;
}
```

Output:

```
0 0 1 1 2
```