Árbol Generador de Peso Mínimo (MST)

Dado un grafo, encuentra el árbol generador de peso mínimo.

Se incluyen dos algoritmos: Prim y Kruskal. En la práctica es más rápido Prim, aunque hay varios problemas que se resuelven con un algoritmo que sea una modificación de uno de ellos.

- Tiempo: $O(E\log(E))$

REQUIERE: Graph, DisjointSets (para kruskal)

```
#include "DisjointSets.hpp"
#include "Graph.hpp"

#include <cmath>
#include <queue>
#include <set>
#include <stack>

using Vertex = Graph::Vertex;
using Edge = Graph::Edge;

struct by_reverse_weight
{
    template <class T>
    bool operator()(const T& a, const T& b)
    {
        return a.weight() > b.weight();
    }
};

std::vector<Graph::Edge> prim(const Graph& G)
{
    auto n = G.num_vertices();

    std::vector<Edge> T;
    if (n < 2)
        return T;

    Vertex num_tree_edges = n - 1;

    T.reserve(num_tree_edges);

    std::vector<char> explored(n, false);

    std::priority_queue<Edge, std::vector<Edge>, by_reverse_weight>
```

```
      EdgesToExplore;

    explored[0] = true;
    for (auto v : G.neighbors(0))
    {
        EdgesToExplore.emplace(0, v, v.weight());
    }

    while (!EdgesToExplore.empty())
    {
        Edge s = EdgesToExplore.top();
        EdgesToExplore.pop();

        if (explored[s.to])
            continue;

        T.emplace_back(s);

        --num_tree_edges;
        if (num_tree_edges == 0)
            return T;

        explored[s.to] = true;
        for (auto v : G.neighbors(s.to))
        {
            if (!explored[v])
                EdgesToExplore.emplace(s.to, v, v.weight());
        }
    }
    return T;
}

std::vector<Graph::Edge> kruskal(const Graph& G)
{
    auto n = G.num_vertices();
    Vertex num_tree_edges = n - 1;

    std::vector<Graph::Edge> T;
    T.reserve(num_tree_edges);

    auto E = G.edges();

    std::sort(E.begin(), E.end(), [](const Edge& a, const Edge& b) {
        if (a.weight() != b.weight())
            return a.weight() < b.weight();
```

```
        if (a.from != b.from)
            return a.from < b.from;
        return a.to < b.to;
    });

    disjoint_sets D(G.num_vertices());

    for (auto& e : E)
    {
        Vertex a = e.from;
        Vertex b = e.to;

        if (!D.are_in_same_connected_component(a, b))
        {
            D.merge(a, b);
            T.emplace_back(e);
            --num_tree_edges;
            if (num_tree_edges == 0)
                return T;
        }
    }

    return T;
}

using namespace std;

template <class T>
std::ostream& operator<<(std::ostream& os, const std::vector<T>& A)
{
    for (const auto& x : A)
        os << x << ' ';
    return os;
}

int main()
{
    Graph G(5);
    G.add_edge(0, 1, 5);
    G.add_edge(0, 2, 2);
    G.add_edge(0, 3, 4);
    G.add_edge(1, 2, 1);
    G.add_edge(1, 3, 8);
    G.add_edge(1, 4, 7);
    G.add_edge(2, 3, 3);
```

```cpp
    G.add_edge(2, 4, 2);
    G.add_edge(3, 4, 9);

    cout << "Prim has the following edges:" << endl;
    for (auto e : prim(G))
    {
        cout << "(" << e.from << "," << e.to << "," << e.weight() << ")\n";
    }

    cout << "\nKruskal has the following edges:" << endl;
    for (auto e : kruskal(G))
    {
        cout << "(" << e.from << "," << e.to << "," << e.weight() << ")\n";
    }

    return 0;
}
```

Output:

```
Prim has the following edges:
(0,2,2)
(2,1,1)
(2,4,2)
(2,3,3)

Kruskal has the following edges:
(1,2,1)
(0,2,2)
(2,4,2)
(2,3,3)
```