Shortest Paths

Dado un grafo y un vértice inicial, encuentra el camino de menor peso a un objetivo.

Se incluyen dos algoritmos: Dijkstra y A*.

REQUIERE: Graph

```
#include "Graph.hpp"

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

using Vertex = Graph::Vertex;
using Edge = Graph::Edge;
using Distance = Graph::sumweight_t;

const auto INF = std::numeric_limits<Distance>::max();

// Used by both A* and Dijkstra
template <class Path = std::deque<Graph::Neighbor>>
inline Path PathFromParentsAndDistances(Vertex origin,
                                        Vertex destination,
                                        const std::vector<Distance>& distance,
                                        const std::vector<Vertex>& parent)
{
    Path P;

    if (origin == destination)
    {
        P.emplace_back(origin, 0);
        return P;
    }

    auto remaining = distance[destination];

    if (remaining == INF)
        return P;

    do
    {
        auto previous = destination;
        destination = parent[destination];
```

```cpp
        auto d = distance[previous] - distance[destination];
        P.emplace_front(previous, d);
    } while (destination != origin);

    P.emplace_front(origin, 0);

    return P;
}

//------------------ Start Dijsktra Searcher

struct DummyPath
{
    DummyPath(Vertex v, Distance d) : last(v), length(d) {}
    Vertex last;
    Distance length;
};

bool operator<(const DummyPath& a, const DummyPath& b)
{
    return a.length > b.length;
}

class DijkstraSearcher
{
public:
    // If destination is invalid, it constructs all single-source shortest
    // paths. If destination is a specific vertex, the searcher stops when it
    // finds it.
    DijkstraSearcher(const Graph& G,
                     Vertex origin_,
                     Vertex destination_ = Graph::INVALID_VERTEX)
        : origin(origin_)
        , destination(destination_)
        , distance(G.num_vertices(), INF)
        , parent(G.num_vertices(), -1)
    {
        distance[origin] = 0;

        std::priority_queue<DummyPath> frontier;

        frontier.emplace(origin, 0);

        while (!frontier.empty())
        {
```

```
            auto P = frontier.top();
            frontier.pop();

            if (P.length > distance[P.last])
                continue;

            if (P.last == destination)
                break;

            for (auto& v : G.neighbors(P.last))
            {
                auto d = P.length + v.weight();
                if (distance[v] > d)
                {
                    distance[v] = d;
                    parent[v] = P.last;
                    frontier.emplace(v, d);
                }
            }
        }
    }

    // dest might be different from destination, if and only if, either
    // destination is Graph::INVALID_VERTEX or distance from origin to dest is
    // smaller than distance to destination.
    template <class Path = std::deque<Graph::Neighbor>>
    Path GetPath(Vertex dest = Graph::INVALID_VERTEX) const
    {
        if (dest == Graph::INVALID_VERTEX)
            dest = destination;
        assert(dest != Graph::INVALID_VERTEX);
        return PathFromParentsAndDistances<Path>(
          origin, dest, distance, parent);
    }

    Vertex Origin() const { return origin; }
    Vertex Destination() const { return destination; }

    const std::vector<Distance>& Distances() const { return distance; }
    const std::vector<Vertex>& Parents() const { return parent; }

private:
    Vertex origin;
    Vertex destination;
    std::vector<Distance> distance;
```

```
    std::vector<Vertex> parent;
};

template <class Path = std::deque<Graph::Neighbor>>
Path Dijkstra(const Graph& G, Vertex origin, Vertex destination)
{
    return DijkstraSearcher(G, origin, destination).GetPath<Path>();
}

//------------------- START A* searcher

struct DummyPathWithHeuristic
{
    DummyPathWithHeuristic(Vertex v, Distance c, Distance h)
        : last(v), cost(c), heuristic(h)
    {}
    Vertex last;
    Distance cost;
    Distance heuristic;

    Distance cost_plus_heuristic() const { return cost + heuristic; }
};

inline bool operator<(const DummyPathWithHeuristic& A,
                      const DummyPathWithHeuristic& B)
{
    if (A.cost_plus_heuristic() != B.cost_plus_heuristic())
        return A.cost_plus_heuristic() > B.cost_plus_heuristic();
    if (A.heuristic != B.heuristic)
        return A.heuristic > B.heuristic;
    return A.last < B.last; // if same cost plus heuristic, I don't know.
}

class AstarSearcher
{
public:
    // Finds a path from origin to destination using heuristic h
    template <class Heuristic>
    AstarSearcher(const Graph& G,
                  Vertex origin_,
                  Vertex destination_,
                  Heuristic h)
        : origin(origin_)
        , destination(destination_)
        , distance(G.num_vertices(), INF)
```

```
        , parent(G.num_vertices(), Graph::INVALID_VERTEX)
    {
        auto obj = [destination_](Vertex v) { return v == destination_; };
        Init(G, obj, h);
    }

    // Finds a path from origin to some destination that satisfies predicte obj,
    // using heuristic h
    template <class Objective, class Heuristic>
    AstarSearcher(const Graph& G, Vertex origin_, Objective obj, Heuristic h)
        : origin(origin_)
        , destination(Graph::INVALID_VERTEX)
        , distance(G.num_vertices(), INF)
        , parent(G.num_vertices(), Graph::INVALID_VERTEX)
    {

        Init(G, obj, h);
    }

    template <class Path = std::deque<Graph::Neighbor>>
    Path GetPath() const
    {
        return PathFromParentsAndDistances<Path>(
          origin, destination, distance, parent);
    }

    Vertex Origin() const { return origin; }
    Vertex Destination() const { return destination; }

    const std::vector<Distance>& Distances() const { return distance; }
    const std::vector<Vertex>& Parents() const { return parent; }

private:
    Vertex origin;
    Vertex destination;
    std::vector<Distance> distance;
    std::vector<Vertex> parent;

    template <class Heuristic, class Objective>
    void Init(const Graph& G, Objective obj, Heuristic h)
    {
        using std::cout;
        using std::endl;
        distance[origin] = 0;
```

```cpp
        std::priority_queue<DummyPathWithHeuristic> frontier;

        frontier.emplace(origin, 0, h(origin));

        while (!frontier.empty())
        {
            auto P = frontier.top();
            frontier.pop();

            if (P.cost > distance[P.last])
                continue;

            if (obj(P.last))
            {
                destination = P.last;
                return;
            }

            for (auto& v : G.neighbors(P.last))
            {
                auto d = P.cost + v.weight();
                if (distance[v] > d)
                {
                    distance[v] = d;
                    parent[v] = P.last;
                    frontier.emplace(v, d, h(v));
                }
            }
        }
    }
};

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, 9);
    G.add_edge(1, 2, 3);
    G.add_edge(2, 3, 4);
    G.add_edge(3, 4, 5);

    Vertex from = 0;
    Vertex to = 4;

    std::vector<int> heuristic = {5, 5, 4, 4, 0};

    cout << "Dijsktra produces the following path:\n\t";
    for (auto t : Dijkstra(G, from, to))
    {
        cout << "----(w = " << t.weight() << ")----> " << t.vertex << " ";
    }
    cout << endl << endl;
    AstarSearcher A(
      G, from, to, [&heuristic](Vertex v) { return heuristic[v]; });

    cout << "A* produces the following path:\n\t";
    for (auto t : A.GetPath())
    {
        cout << "----(w = " << t.weight() << ")----> " << t.vertex << " ";
    }

    return 0;
}
```

Output:

```
Dijsktra produces the following path:
    ----(w = 0)----> 0 ----(w = 5)----> 1 ----(w = 3)----> 2 ----(w = 4)----> 3 ----(w =

A* produces the following path:
    ----(w = 0)----> 0 ----(w = 5)----> 1 ----(w = 3)----> 2 ----(w = 4)----> 3 ----(w =
```