```
//
// Created by Nikolay Yakovets on 2018-02-02.
//

#include "SimpleEstimator.h"
#include "SimpleEvaluator.h"

using namespace std;

SimpleEvaluator::SimpleEvaluator(std::shared_ptr<SimpleGraph> &g) {

    // works only with SimpleGraph
    graph = g;
    est = nullptr; // estimator not attached by default
    cache;
    estcache;
}

void SimpleEvaluator::attachEstimator(std::shared_ptr<SimpleEstimator> &e) {
    est = e;
}

void SimpleEvaluator::prepare() {
    // if attached, prepare the estimator
    if(est != nullptr) est->prepare();

    // prepare other things here.., if necessary

    createExhaustiveIndex();
}

void SimpleEvaluator::createExhaustiveIndex() {
    // exhaustive indexes: SOP, PSO, POS, OSP
    exh_indexes.POS.resize(graph->getNoLabels());
    exh_indexes.PSO.resize(graph->getNoLabels());
    for(uint32_t j = 0; j < graph->getNoVertices(); j++) {
        for (auto edge: graph->adj[j]) {
            //edge.first = edge type, edge.second = out node, j = in node
            // POS = edge type -> (out node, in node)
            exh_indexes.POS[edge.first].push_back(std::make_pair(edge.second, j));
            // PSO = edge type -> (in node, out node)
            exh_indexes.PSO[edge.first].push_back(std::make_pair(j, edge.second));
        }
    }
}

cardStat SimpleEvaluator::computeStats(std::shared_ptr<SimpleGraph> &g) {

    cardStat stats {};

    // Both of these dont seem to be checked so why bother

//    for(int source = 0; source < g->getNoVertices(); source++) {
//        if(!g->adj[source].empty()) stats.noOut++;
//    }

    stats.noPaths = g->getNoDistinctEdges();


    // This is the only use of reverse_adj, so we can get rid of it

//    for(int target = 0; target < g->getNoVertices(); target++) {
//        if(!g->reverse_adj[target].empty()) stats.noIn++;
//    }

    return stats;
}

//std::shared_ptr<SimpleGraph> SimpleEvaluator::project(uint32_t projectLabel, bool inverse, std::shared_ptr<SimpleGraph> &in) {
//
//    auto out = std::make_shared<SimpleGraph>(in->getNoVertices());
//    out->setNoLabels(in->getNoLabels());
//
//    if(!inverse) {
//        // going forward
//        for(uint32_t source = 0; source < in->getNoVertices(); source++) {
//            for (auto labelTarget : in->adj[source]) {
//
//                auto label = labelTarget.first;
//                auto target = labelTarget.second;
//
//                if (label == projectLabel)
//                    out->addEdge(source, target, label);
//            }
//        }
//    } else {
//        // going backward
//        for(uint32_t source = 0; source < in->getNoVertices(); source++) {
//            for (auto labelTarget : in->reverse_adj[source]) {
//
//                auto label = labelTarget.first;
//                auto target = labelTarget.second;
//
//                if (label == projectLabel)
//                    out->addEdge(source, target, label);
//            }
//        }
//    }
//
//    return out;
//}
```

```
std::shared_ptr<SimpleGraph> SimpleEvaluator::project_exh_index(uint32_t projectLabel, bool inverse, std::shared_ptr<SimpleGraph> &in) {
    auto out = std::make_shared<SimpleGraph>(in->getNoVertices());
    out->setNoLabels(in->getNoLabels());

    auto PSO = exh_indexes.PSO[projectLabel];
    auto POS = exh_indexes.POS[projectLabel];

    if (!inverse) {
        // forward
        for (auto edge : PSO) {
            // edge.first = in node, edge.second = out node
            out->addEdge(edge.first, edge.second, projectLabel);
        }
    } else {
        // backward
        for (auto edge : POS) {
            // edge.first = out node, edge.second = in node
            out->addEdge(edge.first, edge.second, projectLabel);
        }
    }

    return out;
}

std::shared_ptr<SimpleGraph> SimpleEvaluator::join(std::shared_ptr<SimpleGraph> &left, std::shared_ptr<SimpleGraph> &right) {

    auto out = std::make_shared<SimpleGraph>(left->getNoVertices());
    out->setNoLabels(1);

    for(uint32_t leftSource = 0; leftSource < left->getNoVertices(); leftSource++) {
        for (auto labelTarget : left->adj[leftSource]) {

            int leftTarget = labelTarget.second;
            // try to join the left target with right source
            for (auto rightLabelTarget : right->adj[leftTarget]) {

                auto rightTarget = rightLabelTarget.second;
                out->addEdge(leftSource, rightTarget, 0);

            }
        }
    }

    return out;
}

//std::shared_ptr<SimpleGraph> SimpleEvaluator::evaluate_aux(RPQTree *q) {
//
//    // evaluate according to the AST bottom-up
//
//    if(q->isLeaf()) {
//        // project out the label in the AST
//        std::regex directLabel (R"((\d+)\+)");
//        std::regex inverseLabel (R"((\d+)\-)");
//
//        std::smatch matches;
//
//        uint32_t label;
//        bool inverse;
//
//        if(std::regex_search(q->data, matches, directLabel)) {
//            label = (uint32_t) std::stoul(matches[1]);
//            inverse = false;
//        } else if(std::regex_search(q->data, matches, inverseLabel)) {
//            label = (uint32_t) std::stoul(matches[1]);
//            inverse = true;
//        } else {
//            std::cerr << "Label parsing failed!" << std::endl;
//            return nullptr;
//        }
//
//        //return SimpleEvaluator::project(label, inverse, graph);
//        //return SimpleEvaluator::project_agg_index(label, inverse, graph);
//        return SimpleEvaluator::project_exh_index(label, inverse, graph);
//    }
//
//    if(q->isConcat()) {
//
//        // evaluate the children
//        auto leftGraph = SimpleEvaluator::evaluate_aux(q->left);
//        auto rightGraph = SimpleEvaluator::evaluate_aux(q->right);
//
//        // join left with right
//        return SimpleEvaluator::join(leftGraph, rightGraph);
//
//    }
//
//    return nullptr;
//}

std::vector<RPQTree*> SimpleEvaluator::getLeaves(RPQTree *query) {
    if (query->isLeaf()) {
        return {query};
    }

    std::vector<RPQTree*> result;
    if (query->left) {
        auto rec = getLeaves(query->left);
        result.insert(result.end(), rec.begin(), rec.end());
    }
    if (query->right) {
        auto rec = getLeaves(query->right);
        result.insert(result.end(), rec.begin(), rec.end());
    }

    return result;
}

RPQTree* SimpleEvaluator::optimizeQuery(RPQTree *query) {
    std::vector<RPQTree*> leaves = getLeaves(query);

    while (leaves.size() > 1) {
        uint32_t bestScore = 0;
        RPQTree *bestTree = nullptr;
        int index = -1;

        for (int i = 0; i < leaves.size()-1; ++i) {
            std::string data("/");
            auto *currentTree = new RPQTree(data, leaves[i], leaves[i+1]);
            uint32_t currentScore = est->estimate(currentTree).noPaths;

            if (bestScore == 0 || bestScore > currentScore) {
                bestScore = currentScore;
                bestTree = currentTree;
                index = i;
            }
        }

        leaves.erase(leaves.begin() + index + 1);
        leaves[index] = bestTree;
    }

    return leaves[0];
}

//cardStatstd::shared_ptr<SimpleGraph> SimpleEvaluator::evaluate_aux(RPQTree *q) {
//
//    // evaluate according to the AST bottom-up
//
//    if(q->isLeaf()) {
//        // project out the label in the AST
//        std::regex directLabel (R"((\d+)\+)");
//        std::regex inverseLabel (R"((\d+)\-)");
//
//        std::smatch matches;
//
//        uint32_t label;
//        bool inverse;
//
//        if(std::regex_search(q->data, matches, directLabel)) {
//            label = (uint32_t) std::stoul(matches[1]);
//            inverse = false;
//        } else if(std::regex_search(q->data, matches, inverseLabel)) {
//            label = (uint32_t) std::stoul(matches[1]);
//            inverse = true;
//        } else {
//            std::cerr << "Label parsing failed!" << std::endl;
//            return nullptr;
//        }
//
//        //return SimpleEvaluator::project(label, inverse, graph);
//        //return SimpleEvaluator::project_agg_index(label, inverse, graph);
//        return SimpleEvaluator::project_exh_index(label, inverse, graph);
//    }
//
//    if(q->isConcat()) {
//
//        // evaluate the children
//        auto leftGraph = SimpleEvaluator::evaluate_aux(q->left);
//        auto rightGraph = SimpleEvaluator::evaluate_aux(q->right);
//
//        // join left with right
//        return SimpleEvaluator::join(leftGraph, rightGraph);
//
//    }
//
//    return nullptr;
//}

std::vector<std::string> SimpleEvaluator::treeToString(RPQTree *q) {
    std::vector<std::string> vec;
    SimpleEvaluator::treeToString(q, vec);
    return vec;
}

void SimpleEvaluator::treeToString(RPQTree *q, std::vector<std::string> &vec) {
    if (q->isLeaf()) {
        vec.push_back(q->data);
    } else {
        SimpleEvaluator::treeToString(q->left, vec);
        SimpleEvaluator::treeToString(q->right, vec);
    }
}

cardStat SimpleEvaluator::evaluate(RPQTree *query) {

    vector <string> paths;
    vector <shared_ptr<SimpleGraph>> projections;
    shared_ptr<SimpleGraph> result = nullptr;

    cout << endl;
    // Initalize a vector with the labels
    paths = SimpleEvaluator::treeToString(query);
    vector <string> key = paths;
    if (cache.find(key) != cache.end()) {
        return cache.find(key)->second;
    }
    // Project all the labels
    for (int i=0; i < paths.size(); i++) {
        uint32_t label = (uint32_t) std::stoul(paths[i].substr(0, paths[i].length()-1));
        bool inverse = paths[i].at(1) == '-';
        projections.push_back(project_exh_index(label, inverse, graph));
    }

    while (paths.size() > 2) {

        // Find the cheapest join
        vector <int> estimate;
        for (int i=0; i < paths.size()-1; i++) {
            string path = paths[i] + "/" + paths[i+1];
            cardStat ea;
            cout << path << " ";
            if (estcache.find(path) != estcache.end()) {
                ea = estcache.find(path)->second;
                cout << ea.noPaths << endl;
            } else {
                ea = est->estimate(RPQTree::strToTree(path));
                estcache.insert(std::pair<std::string, cardStat>(path, ea));
            }
            estimate.push_back(ea.noPaths);
        }

        int minPos = 0;
        for (unsigned i = 0; i < estimate.size(); ++i )
        {
            if (estimate[i] < estimate[minPos]) {
                minPos = i;
            }
        }

        auto merged_leafs = join(projections[minPos], projections[minPos+1]);
        paths.insert(paths.begin() + minPos, paths[minPos] + "/" + paths[minPos+1]);
        paths.erase(paths.begin() + minPos + 1);
        paths.erase(paths.begin() + minPos + 1);
        projections.insert(projections.begin() + minPos, merged_leafs);
        projections.erase(projections.begin() + minPos + 1);
        projections.erase(projections.begin() + minPos + 1);

    }
    auto last = projections[0];
    if (projections.size() > 1) {
        last = join(projections[0], projections[1]);
    }
    cardStat eval = computeStats(last);
    cache.insert(std::pair<std::vector<std::string>, cardStat>(key, eval));
    return eval;

}
```