FACULDADE DE ENGENHARIA
DA UNIVERSIDADE DO PORTO

G16_03

Railway Network

Manuel Alves
up201906910

Luís Contreiras
up202108742

Afonso Poças
up202008323

2022/23

Problem Description

Creating a graph using a collection of stations and the links that connect them to decide wisely regarding how to most effectively utilize its resources, both material and financial

Dataset reading

Both files csv (Stations.csv, network.csv) are stored in the graph using functionalities called readStations and readNetwork.
We used the function parse_csv_line to parse a line in each method.

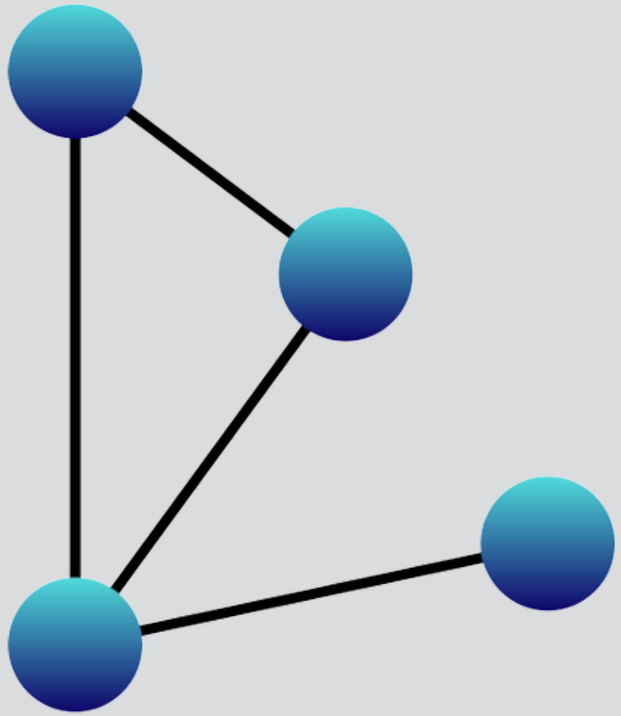
```
vector<string> parse_csv_line(const std::string& line) {
    vector<std::string> fields;
    string field;
    bool in_quotes = false;

    for (size_t i = 0; i < line.length(); ++i) {
        if (line[i] == '"') {
            in_quotes = !in_quotes;
            continue;
        }
        if (line[i] == ',' && !in_quotes) {
            fields.push_back(field);
            field.clear();
        } else {
            field += line[i];
        }
    }
    fields.push_back(field);
    return fields;
}

unordered_map<string, Vertex *> readStations(Graph& railway) {
    ifstream fin( s: "../dataset/stations.csv");
    string line;
    unordered_map<string, Vertex *> stations;
    if (!fin.is_open()) {
        throw runtime_error("Error reading stations.csv");
    }
    getline( &: fin,  &: line);
    while (getline( &: fin,  &: line)) {
        if (line[line.size() - 1] == '\r') {
            line.pop_back();
        }
        vector<string> curr = parse_csv_line(line);
        Vertex *vertex = new Vertex( name: curr[0],  district: curr[
        railway.addVertex(vertex);
        stations.insert( x: make_pair( x: curr[0], vertex));
    }
    fin.close();
    return stations;
```

GRAPH STRUCTURE

A biderectional graph with stations as vertices and each edge having a capacity and a service attached to it was used in the implementation.



Functionalities

- Operations between two specific stations
- Pairs of stations that require the most trains
- Stations by municipalities and districts.
- Max number of trains that can arrive at a station

```
void Graph::MaxFlowBetweenPairs() {
    auto start = std::chrono::high_resolution_clock::now();
    double maxflow = -1;
    std::vector<std::pair<Vertex *, Vertex *>> result;

    for (int i = 0; i < vertexSet.size(); i++) {
        for (int j = i + 1; j < vertexSet.size(); j++) {
            Vertex *s = vertexSet[i];
            Vertex *t = vertexSet[j];
            double m = EdmondsKarp(s, t);

            if (m > maxflow) {
                maxflow = m;
                result.clear();
                result.emplace_back(s, t);
            } else if (m == maxflow) {
                result.emplace_back(s, t);
            }
        }
    }

    for (const auto &pair : result) {
        std::cout << pair.first->getName() << " / " << pair.second->getName() << std::endl;
    }
    auto end = std::chrono::high_resolution_clock::now(); // Fim do temporizador
    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count();
    std::cout << "Tempo de execução: " << duration << "ms" << std::endl;
}
```

Functionalities

- Top-k districts and municipalities

- Add segment failure
- Most affected stations

```cpp
void Graph::topDistricts(int k){
    std::map<std::string, double> districtMaxFlows;

    for (int i = 0; i < vertexSet.size(); i++) {
        for (int j = i + 1; j < vertexSet.size(); j++) {
            Vertex *s = vertexSet[i];
            Vertex *t = vertexSet[j];
            if(s->getDistrict() != t->getDistrict()){ // Verificar se os distritos são diferentes
                double maxFlow = EdmondsKarp(s, t);
                districtMaxFlows[s->getDistrict()] += maxFlow;
                districtMaxFlows[t->getDistrict()] += maxFlow;
            }
        }
    }

    std::vector<std::pair<std::string, double>> sortedDistricts(districtMaxFlows.begin(), districtMaxFlows.end());
    std::sort(sortedDistricts.begin(), sortedDistricts.end(), comp: [](const std::pair<std::string, double>& left, const st
        return left.second > right.second;
    });

    // Print top districts
    std::cout << "Top districts: \n";
    for (int i = 0; i < k && i < sortedDistricts.size(); i++) { // Verificar o índice para evitar acessar um índice fora d
        std::cout << "District: " << sortedDistricts[i].first << ", Max Flow: " << sortedDistricts[i].second << std::endl;
    }
}
```

USER INTERFACE

- A menu with the principal functionalities
- Submenus for origin and destination input.
- The user can return to the menu and select the preferred option.

```
--- Railway Management System Interface Menu ---

1. - Basic Service Metrics
2. - Operations Cost Optimization
3. - Reliability and Sensitivity to Line Failures
4. - Exit

Enter your option:1

--- Basic Service Metrics ---

1. - Calculate the maximum number of trains that can simultaneously trav
2. - Determine which stations require the most amount of trains
3. - Assign larger budgets for the purchasing and maintenance of trains
4. - Report the maximum number of trains that can simultaneously arrive
5. - Return

Enter your option:1
 Enter source station name:Tunes
 Enter destination station name:Viana do Castelo
 Max trains between Tunes and Viana do Cast
elo is 4
```

Featured Algorithm

- Edmond Karp Algorithm

```
double Graph::EdmondsKarp(Vertex* s, Vertex* t) {
    double maxFlow = 0.0;
    auto bfs = [this, &s, &t]() -> double {
        for (auto v: vertexSet) {
            v->setVisited( visited: false);
        }
        std::queue<Vertex *> q;
        s->setVisited( visited: true);
        q.push(s);

        std::unordered_map<Vertex *, Edge *> prev;
        while (!q.empty()) {
            Vertex *currVertex = q.front();
            q.pop();
            if (currVertex == t) break;
            for (auto adj: currVertex->getAdj()) {
                if (adj->getDest()->isVisited() || adj->getWeight() - adj->getFlow() <= 0.0) {
                    continue;
                }
                adj->getDest()->setVisited( visited: true);
                prev[adj->getDest()] = adj;
                q.push(adj->getDest());
            }
        }
        if (prev.find(t) == prev.end()) return 0.0;
        double bottleNeck = std::numeric_limits<double>::max();
        for (auto e = prev[t]; e != nullptr; e = prev[e->getOrig()]) {
            bottleNeck = std::min(bottleNeck, e->getWeight() - e->getFlow());
        }
        for (auto e = prev[t]; e != nullptr; e = prev[e->getOrig()]) {
            e->setFlow( flow: e->getFlow() + bottleNeck);
            e->getReverse()->setFlow( flow: e->getReverse()->getFlow() - bottleNeck);
        }
        return bottleNeck;
    };
    double flow;
    while ((flow = bfs()) > 0.0) {
        maxFlow += flow;
    }
    return maxFlow;
```

Main difficulties

- Dealing with various flows types.
- Reduced connectivity operations.