Projeto DA

up202108820 – David Cordeiro

up202108803 – Diogo Viana

up202108707 – Gonçalo Martins

Code Implementation

```
void Manager::readStations() {
    for (CsvLine& line : CsvReader::read("../data/stations.csv", true)) {
        string name = line.getString(0), dist = line.getString(1), muni = line.getString(2);
        Station station = Station(name, dist, muni, line.getString(3), line.getString(4));
        networkGraph.addVertex(station);

        stationsMap.insert({clearString(name), station});

        if (districtMap.contains(dist)) {
            if (districtMap[dist].contains(muni)) {
                districtMap[dist][muni].push_back(station);
            } else {
                vector<Station> stations;
                stations.push_back(station);
                districtMap[dist].insert({muni, stations});
            }
        } else {
            vector<Station> stations;
            stations.push_back(station);
            unordered_map<string, vector<Station>> muniMap;
            muniMap.insert({muni, stations});
            districtMap.insert({dist, muniMap});
        }
    }
}
```

/*Percorre o ficheiro "stations.csv" linha a linha, parseia as linhas através do uso das classes auxiliares "CsvReader" e "CsvLine", cria um objeto do tipo "Station" e guarda a estação e informações relativas a esta, nas principais estruturas de dados usadas no projeto (grafo que representa a rede de estações, hashtable que associa o nome "limpo" (sem acentos, espaços ou letras capitalizadas) de uma estação á mesma e uma hashtable de distritos associados a uma hastable de municipios, que por sua vez guardam um array de estações localizadas nesse munnicipio (usado em "busiestMunisAndDist"))*/

/* Percorre o ficheiro "network.csv" linha a linha e usando as funções auxiliares "CsvReader" e "CsvLine", adiciona edges entre as estações (nodes) do grafo. Se o grafo for usado para problemas de max-flow, para cada segmento da rede de estações é adicionada também uma edge residual no sentido contrário e com capacidade 0 */

```
void Manager::readNetwork(bool isFlowGraph) {
    if (isFlowGraph) {
        for (CsvLine& line : CsvReader::read("../data/network.csv", true)) {
            networkGraph.addEdgeAndResidual(line.getString(0), line.getString(1), line.getInt(2), line.getService(3), INF);
        }
    } else {
        for (CsvLine& line : CsvReader::read("../data/network.csv", true)) {
            networkGraph.addEdge(line.getString(0), line.getString(1), line.getInt(2), line.getService(3), INF);
        }
    }
}
```

Code Implementation

```
int Graph::maxFlow(const string &srcName, const string &sinkName) {
    Vertex* src = findVertex(srcName);
    Vertex* sink = findVertex(sinkName);
    vector<Edge*> usedEdges;
    int maxFlow = 0;
    int currFlow;

    do {
        resetVisitedAndPath();
        currFlow = maxFlowBfs(src, sink, usedEdges);
        maxFlow += currFlow;

    } while (currFlow != 0);

    resetFlow(usedEdges);

    return maxFlow;
}
```

/* Função que controla o loop de execução de uma operação de max-flow, utilizando o algoritmo de Edmonds-Karp. A cada iteração, dá reset das propriedas "visited" e "path", usadas pelos nodes do grafo para determinar se estes já foram visitados durante a bfs e qual o seu "antecessor"(usado para depois reconstruir o path), respetivamente. Chama a função "maxFlowBFS", que cria um "augmenting path" e retorna um "bottleneck" (menor capacidade ao longo do path). O valor do bottleneck retornado é igual á quantidade de flow que passa nesse augmenting path, e é adicionado a uma variável "maxFlow" que mantém a quantidade total de flow registada até ao momento. Quando "currFlow" é igual a 0 (não é possível criar mais augmenting paths, ou seja, já não é possível enviar flow entre os nodes), o loop termina, usa a função resetFlow que volta a colocar a 0 a variável "flow" de todas as edges para determinar o max-flow e retorna o valor da variável "maxFlow"*/

/* BFS usada para determinar um augmenting path entre "src" e "sink". Percorre edges cuja capacidade restante (diferença entre o flow que passa nessa edge e a sua capacidade) é maior que 0 e em que o node de destino ainda não tenha sido visitada. Marca o node de destino como visitado e guarda um pointer para a edge usada para atravessar, na variável "path" desse mesmo node. Percorre as edges usadas nesse caminho ao contrário (fazendo uso da variável "path") e determina o "bottleneck" (capacidade restante minima) ao longo do percurso. Volta a percorrer o percurso e a aumentar o flow de cada edge com o valor do bottleneck e guarda pointers para as edges modificadas no vetor "edgesToReset", para dar reset ao flow, depois do algoritmo de max-flow terminar. Retorna o valor do bottleneck.*/

```
int Graph::maxFlowBfs(Vertex* src, Vertex* sink, vector<Edge*>& edgesToReset) {
    queue<Vertex*> q;
    src->setVisited(true);
    q.push(src);

    while (!q.empty()) {
        Vertex* currNode = q.front();
        q.pop();
        if (currNode == sink) break;

        for (Edge* edge : currNode->getAdj()) {
            int cap = edge->remainingCapacity();
            if (cap > 0 && !edge->getDest()->isVisited()) {
                edge->getDest()->setVisited(true);
                edge->getDest()->setPath(edge);
                q.push(edge->getDest());
            }
        }
    }

    if (sink->getPath() == nullptr) return 0;

    int bottleNeck = INF;
    for (Edge* edge = sink->getPath(); edge != nullptr; edge = edge->getOrig()->getPath()) {
        bottleNeck = min(bottleNeck, edge->remainingCapacity());
    }

    for (Edge* edge = sink->getPath(); edge != nullptr; edge = edge->getOrig()->getPath()) {
        edge->augment(bottleNeck);
        edgesToReset.push_back(edge);
    }

    return bottleNeck;
}
```

Code Implementation

```cpp
void Manager::busiestPairsOfStations() {
    cout << "CURRENTLY COMPARING THE MAX-FLOW BETWEEN EVERY PAIR OF STATIONS. YOUR RESULTS WILL BE SHOWED SHORTLY :)\n" << endl;

    int largestMaxFlow = 0, currMaxFlow;
    vector<pair<Station, Station>> stationPairs;

    for (auto it1 = stationsMap.begin(); it1 != stationsMap.end(); it1++) {
        for (auto it2 = next(it1); it2 != stationsMap.end(); it2++) {
            currMaxFlow = networkGraph.maxFlow(it1->second.getName(), it2->second.getName()) + networkGraph.maxFlow(it2->second.getName(), it1->second.getName());

            if (currMaxFlow == largestMaxFlow) {
                stationPairs.emplace_back(it1->second, it2->second);
            } else if (currMaxFlow > largestMaxFlow) {
                stationPairs.clear();
                stationPairs.emplace_back(it1->second, it2->second);
                largestMaxFlow = currMaxFlow;
            }
        }
    }

    cout << "THE PAIR(S) OF STATIONS THAT REQUIRE THE MOST AMOUNT OF TRAINS, WHEN MAKING USE OF THE FULL NETWORK CAPACITY, IS(ARE):\n" << std::endl;
    for (const auto& stations : stationPairs) {
        cout << "[" << stations.first.getName() << ", " << stations.second.getName() << "]" << " WITH AN ADDED MAX-FLOW (TRAVELING BOTH WAYS) OF " << largestMaxFlow << endl;
    }
}
```

/*Função que percorre todos os pares de estações, e determina qual é o par de estações com o maior valor total de max-flow entre elas (considerando os 2 sentidos, ou seja, o max-flow da estação A para a estação B, mais o da estação B para a estação A).*/

/* Função que calcula o max-flow total de cada municipio e cada distrito, somando o max-flow entre todos os pares de estações desse municipio/distrito, guarda num multiset (ordenado por ordem decrescente de flow no momento de inserção), um par que associa o flow de um municipio ao nome desse municipio e o mesmo com os distritos. Dá display dos 10 distritos e 50 municipios com mais flow.*/

```cpp
void Manager::busiestMunisAndDist() {
    multiset<pair<int, string>, greater<>> flowOfMunicipalities;
    multiset<pair<int, string>, greater<>> flowOfDistricts;

    for (const auto& district : districtMap) {
        int distFlow = 0;
        for (const auto& municipality : district.second) {
            int muniFlow = 0;
            vector<Station> muniStations = municipality.second;
            for (auto it1 = muniStations.begin(); it1 != muniStations.end(); it1++) {
                for (auto it2 = it1+1; it2 != muniStations.end(); it2++) {
                    muniFlow += networkGraph.maxFlow(it1->getName(), it2->getName());
                    muniFlow += networkGraph.maxFlow(it2->getName(), it1->getName());
                }
            }
            distFlow += muniFlow;
            flowOfMunicipalities.emplace(muniFlow, municipality.first);
        }
        flowOfDistricts.emplace(distFlow, district.first);
    }

    cout << "THE TOP 10 DISTRICTS WHERE RESOURCES SHOULD BE ALLOCATED BASED ON THEIR TRANSPORTATION NEEDS ARE:\n" << endl;

    int cnt1 = 0;
    for (const auto& flowDist : flowOfDistricts) {
        cout << flowDist.second << " | Flow: " << flowDist.first << endl;
        cnt1++;
        if (cnt1 == 10) break;
    }

    cout << "\nTHE TOP 50 MUNICIPALITIES WHERE RESOURCES SHOULD BE ALLOCATED BASED ON THEIR TRANSPORTATION NEEDS ARE:\n" << endl;

    int cnt2 = 0;
    for (const auto& flowMuni : flowOfMunicipalities) {
        cout << flowMuni.second << " | Flow: " << flowMuni.first << endl;
        cnt2++;
        if (cnt2 == 50) break;
    }
}
```

Code Implementation

```
void Manager::maxSimultaneousTrainsToAStation() {
    vector<Vertex*> sourceVertexes;
    string station;
    string clearStation;

    while (true) {
        cout << "\nWRITE THE NAME OF THE STATION YOU DESIRE: ";

        std::getline( &: cin,  &: station);

        clearStation = clearString( str: station);
        if (stationsMap.count( x: clearStation)) {
            station = stationsMap.at( k: clearStation).getName();
            break;
        } else {
            cout << "THE STATION YOU CHOSE DOES NOT EXIST. PLEASE TRY AGAIN" << endl;
            continue;
        }
    }

    for (Vertex* vertex : networkGraph.getVertexSet()) {
        if ((vertex->getIndegree() == 0) && (vertex->getName() != "START")) {
            sourceVertexes.push_back(vertex);
        }
    }

    networkGraph.addVertex( station: Station( name: "START",  dist: "ETERNITY",  muni: "BEYOND",  tnship: "START",  line: "LINE OF INFINITY"));

    for (Vertex* sourceVertex : sourceVertexes) {
        networkGraph.addEdgeAndResidual( sourc: "START",  dest: sourceVertex->getName(),  capacity: INF,  service: SERVICE_ENUM::STANDARD,  w: INF);
    }

    int maxSimulTrains = networkGraph.maxFlow( srcName: "START",  sinkName: station);
    if (maxSimulTrains == INF) maxSimulTrains = 0;

    cout << "\nTHE MAXIMUM AMOUNT OF TRAINS THAT CAN SIMULTANEOUSLY ARRIVE AT " << station << " IS " << maxSimulTrains << endl;
```

/* Função usada para remover os acentos, espaços e letras maiúsculas do nome de uma estação. Estas strings são depois utilizadas como key, para uma hashtable de estações, permitindo assim que o programa funciona, apesar de pequenas diferenças na maneira como o utilizador escreva o nome de uma estação.*/

Code Implementation

```
void Graph::dijkstraModif(const string &srcName, const string &sinkName) {
    Vertex* src = findVertex(srcName);
    Vertex* sink = findVertex(sinkName);
    unordered_map<string, double> shortestDistances(vertexSet.size());
    unordered_map<string, string> parents(vertexSet.size());
    unordered_map<string, int> edges_cap;

    for (Vertex* vertex : vertexSet) {
        vertex->setDist(std::numeric_limits<double>::max());
        vertex->setVisited(false);
    }
    src->setDist(0);
    parents[src->getName()] = "";
    set<pair<double, Vertex*>> q;
    q.insert({0, src});
    while(!q.empty()) {
        auto tmp = q.begin();
        Vertex* u = tmp->second;
        q.erase(tmp);
        u->setVisited(true);
        shortestDistances[u->getName()] = u->getDist();
        if (u->getName() == sinkName) {break;}
        for (auto edge : u->getAdj()) {
            edge->setWeight((edge->getService() == STANDARD) ? 2 : 4);
            double cost = edge->getWeight();
            Vertex* v = edge->getDest();
            if (!v->isVisited() && u->getDist() + cost < v->getDist()) {
                q.erase({v->getDist(), v});
                v->setDist(u->getDist() + cost);
                q.insert({v->getDist(), v});
                parents[v->getName()] = u->getName();
                edges_cap[v->getName()] = edge->getCapacity();
            }
        }
    }
    printPath(src->getName(), sink->getName(), shortestDistances, parents);
    printMaxFlow(src->getName(), sink->getName(), parents, edges_cap);
}
```

/* Percorre o shortest path de uma estação para a outra, em que o weight do caminho é calculado consoante o SERVICE de cada edge (se é STANDARD ou ALFA PENDULAR). De seguida calcula o max flow, isto é, o número máximo de comboios que passa por esse path */

/* Gerar um subgraph de forma aleatória da network original */

```
Graph Manager::generateSubGraph(vector<pair<string, int>> &countFails) {
    Graph subGraph = networkGraph;
    vector<Vertex*> aux = networkGraph.getVertexSet();
    pair<string, int> push;

    for(int i = 0; i < aux.size(); i++){
        push.first = aux[i]->getName();
        push.second = 0;
        countFails.push_back(push);
    }

    srand(time(NULL));

    int inbetween = rand() % subGraph.getVertexSet().size()/2;
    int count = 0, countEdge = 0, edgeN = 0;

    for(auto& help: subGraph.getVertexSet()){
        if(count == inbetween){
            edgeN = rand() % help->getAdj().size()/2 + 1;
            vector<Edge*> adj = help->getAdj();
            for(int j = 0; j < adj.size(); j++){
                if(countEdge == edgeN){
                    addFailCount(help->getName(), adj[j]->getDest()->getName(), countFails);
                    subGraph.removeEdge(help, adj[j]);
                    countEdge = 0;
                }
                countEdge++;
            }
            count = 0;
            continue;
        }
        count++;
    }

    return subGraph;
}
```

Highlights

```
string Manager::clearString(string str) {
    static const std::unordered_map<char, char> accent_map = {
            {'Á', 'A'}, {'É', 'E'}, {'Í', 'I'}, {'Ó', 'O'}, {'Ú', 'U'},
            {'À', 'A'}, {'È', 'E'}, {'Ì', 'I'}, {'Ò', 'O'}, {'Ù', 'U'},
            {'Â', 'A'}, {'Ê', 'E'}, {'Î', 'I'}, {'Ô', 'O'}, {'Û', 'U'},
            {'Ã', 'A'}, {'Õ', 'O'},
            {'á', 'a'}, {'é', 'e'}, {'í', 'i'}, {'ó', 'o'}, {'ú', 'u'},
            {'à', 'a'}, {'è', 'e'}, {'ì', 'i'}, {'ò', 'o'}, {'ù', 'u'},
            {'â', 'a'}, {'ê', 'e'}, {'i', 'i'}, {'ô', 'o'}, {'û', 'u'},
            {'ã', 'a'}, {'õ', 'o'}
    };
    std::transform( first: str.begin(), last: str.end(), result: str.begin(), unary_op: [](char c) -> char {
        auto it : const_iterator<...> = accent_map.find( x: c);
        return it != accent_map.end() ? it->second : c;
    });
    str.erase( first: std::remove_if( first: str.begin(), last: str.end(), pred: [](char c) -> bool { return !std::isalnum(c); }), last: str.end());
    std::transform( first: str.begin(), last: str.end(), result: str.begin(), unary_op: [](unsigned char c) -> int { return std::tolower(c); });
    return str;
}
```

/* Função usada para remover os acentos, espaços e letras maiúsculas do nome de uma estação. Estas strings são depois utilizadas como key, para uma hashtable de estações, permitindo assim que o programa funciona, apesar de pequenas diferenças na maneira como o utilizador escreva o nome de uma estação.*/

Main Difficulties

- Data processing
- How to remove accents from a string
- How to best organize stations by municipalities and districts
- Finding ways to filter the edges used in a specific path
- Algorithm modifying