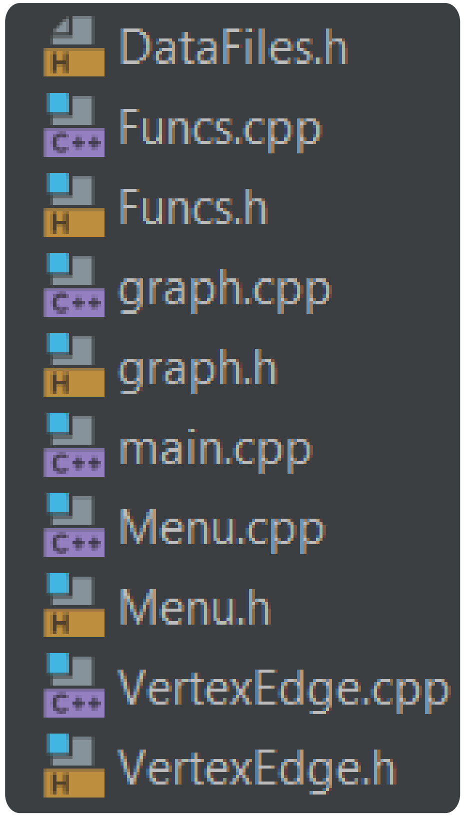
Programming Project 2- Routing Algorithm for Ocean Shipping and Urban Deliveries

Done by:
Eduardo José Neto Oliveira
Maria Abreu da Costa
Máximo Brandão Pereira

Classes' diagram

- DataFiles: has the data files' path;
- Funcs: the information extracted from the files is processed and used to solve the problem;
- Graph: represents the graph and has the functions that execute the algorithms;
- Menu: manages the menus that allow the user to choose what he wants to do in the system;
- VertexEdge: contains all the information regarding vertexes and edges.



```
void Funcs::readFiles(int option, int *RWfilesread, int *EFCGfilesread) {
    cout << "Reading files...\n" << endl;
    if (EDGES_FILE != "") {
        if (NODES_FILE != "") customGraph.readFiles( nodes: NODES_FILE,  edges: EDGES_FILE);
        else customGraph.readEdgesFile( edges: EDGES_FILE);
    }
    switch (option) {
        case 1:
            TG_shippingGraph.readEdgesFile( edges: TG_SHIPPING_FILE);
            TG_stadiumsGraph.readEdgesFile( edges: TG_STADIUMS_FILE);
            TG_tourismGraph.readEdgesFile( edges: TG_TOURISM_FILE);
            break;
        case 2:
            *RWfilesread = 1;
            TG_shippingGraph.readEdgesFile( edges: TG_SHIPPING_FILE);
            TG_stadiumsGraph.readEdgesFile( edges: TG_STADIUMS_FILE);
            TG_tourismGraph.readEdgesFile( edges: TG_TOURISM_FILE);
            RW_graph1.readFiles( nodes: RW_G1_NODES,  edges: RW_G1_EDGES);
            RW_graph2.readFiles( nodes: RW_G2_NODES,  edges: RW_G2_EDGES);
            RW_graph3.readFiles( nodes: RW_G3_NODES,  edges: RW_G3_EDGES);
            break;
        case 3:
            *RWfilesread = 1;
            *EFCGfilesread = 1;
            TG_shippingGraph.readEdgesFile( edges: TG_SHIPPING_FILE);
            TG_stadiumsGraph.readEdgesFile( edges: TG_STADIUMS_FILE);
            TG_tourismGraph.readEdgesFile( edges: TG_TOURISM_FILE);
            RW_graph1.readFiles( nodes: RW_G1_NODES,  edges: RW_G1_EDGES);
            RW_graph2.readFiles( nodes: RW_G2_NODES,  edges: RW_G2_EDGES);
            RW_graph3.readFiles( nodes: RW_G3_NODES,  edges: RW_G3_EDGES);
            EFCG_graph25.readEdgesFile( edges: EFCG_EDGES_25);
            EFCG_graph50.readEdgesFile( edges: EFCG_EDGES_50);
            EFCG_graph75.readEdgesFile( edges: EFCG_EDGES_75);
            EFCG_graph100.readEdgesFile( edges: EFCG_EDGES_100);
            EFCG_graph200.readEdgesFile( edges: EFCG_EDGES_200);
            EFCG_graph300.readEdgesFile( edges: EFCG_EDGES_300);
            EFCG_graph400.readEdgesFile( edges: EFCG_EDGES_400);
            EFCG_graph500.readEdgesFile( edges: EFCG_EDGES_500);
            EFCG_graph600.readEdgesFile( edges: EFCG_EDGES_600);
            EFCG_graph700.readEdgesFile( edges: EFCG_EDGES_700);
            EFCG_graph800.readEdgesFile( edges: EFCG_EDGES_800);
            EFCG_graph900.readEdgesFile( edges: EFCG_EDGES_900);
            break;
        default:
            exit( Code: 1);
    }
}
```

Reading the dataset

- We have different cases to read the files according to the option that the user chooses when running the code

```
void Graph::readEdges(string edges) {
    string path = edges;
    string line;
    string orig, dest, dist;
    ifstream file( s: path);

    if (file.is_open()) {
        getline( &: file,  &: line);
        while (getline( &: file,  &: line)) {
            stringstream ss( str: line);
            getline( &: ss,  &: orig,  delim: ',');
            getline( &: ss,  &: dest,  delim: ',');
            getline( &: ss,  &: dist,  delim: ',');
            if (hasDir) {
                if (!addEdge( source: stoi( str: orig),  target: stoi( str: dest),  distance: stod( str: dist))) continue;
            } else {
                if (!addBidirectionalEdge( origin: stoi( str: orig),  destination: stoi( str: dest),  distance: stod( str: dist))) continue;
            }
        }
        file.close();
    }
    else {
        cout << "Error opening edges file." << endl;
        exit( Code: 1);
    }
}
```

```
void Graph::readNodes(string nodes) {
    string path = nodes;
    string line;
    string id, lat, lon;
    ifstream file( s: path);

    if (file.is_open()) {
        getline( &: file,  &: line);
        while (getline( &: file,  &: line)) {
            stringstream ss( str: line);
            getline( &: ss,  &: id,  delim: ',');
            getline( &: ss,  &: lat,  delim: ',');
            getline( &: ss,  &: lon,  delim: ',');
            if (!addVertex( id: stoi( str: id))) continue;
            Vertex *v = findVertex( id: stoi( str: id));
            v->setLatitude( lat: stod( str: lat));
            v->setLongitude( lon: stod( str: lon));
        }
        file.close();
    }
    else {
        cout << "Error opening nodes file." << endl;
        exit( Code: 1);
    }
}
```

Reading the dataset

We have a function to read the edges of each file and another to read the nodes

```cpp
void Graph::readEdgesFile(string edges) {
    string path = edges;
    string line;
    string orig, dest, dist;
    ifstream file( s: path);

    if (file.is_open()) {
        getline( &: file,  &: line);
        while (getline( &: file,  &: line)) {
            stringstream ss( str: line);
            getline( &: ss,  &: orig,  delim: ',');
            getline( &: ss,  &: dest,  delim: ',');
            getline( &: ss,  &: dist,  delim: ',');
            addVertex( id: stoi( str: orig));
            addVertex( id: stoi( str: dest));
            if (hasDir) {
                if (!addEdge( source: stoi( str: orig),  target: stoi( str: dest),  distance: stod( str: dist))) continue;
            } else {
                if (!addBidirectionalEdge( origin: stoi( str: orig),  destination: stoi( str: dest),  distance: stod( str: dist))) continue;
            }
        }
        file.close();
    }
    else {
        cout << "Error opening edges file." << endl;
        exit( Code: 1);
    }
}
```

Reading the dataset

We have a function that is used only when an edges' file is given.

Graph’s description

The graph has:

- attribute hasDir that is false since our graph is not directed;
- attribute vertexMap which is a map with all the vertexes where the key is the vertex's id and the value is a pointer to the Vertex.

```
protected:
    bool hasDir;
    unordered_map<int, Vertex *> vertexMap;    // vertex map
```

VertexEdge's description

Vertex's description

The vertex has:

- attribute id that corresponds to the vertex's identifier;
- attribute adj which is a vector that has all the outgoing edges;
- attributes latitude and longitude that correspond to the vertex's latitude and longitude
- attribute visited which indicates if the vertex was visited or not;
- attribute dist which represents the distance;
- attribute path is a pointer to the path;
- attribute incoming which is a vector that has all the edges which's destination is this vertex.

```
protected:
    int id;          // identifier
    vector<Edge *> adj;  // outgoing edges

    double latitude = -1;
    double longitude = -1;

    // auxiliary fields
    bool visited = false; // used by DFS, BFS, Prim ...
    int dist = -1;
    Edge *path = nullptr;

    vector<Edge *> incoming; // incoming edges
```

VertexEdge's description

Edge's description

```
protected:
    Vertex *dest = nullptr; // destination vertex
    double distance;

    // used for bidirectional edges
    Vertex *orig = nullptr;
```

The edge has:

- attribute dest that corresponds to destination vertex;
- attribute distance
- attribute orig which is a pointer to the origin's vertex.

```cpp
pair<vector<Vertex *>, double> Graph::utilBacktrack(Vertex *currentNode, vector<Vertex *> &visited, double totalDistance, pair<vector<Vertex *>, dou
    visited.push_back(currentNode);  // Add the current node to visited list

    if (visited.size() == getNumVertex()) {
        // Check if there is an edge from the current node to the starting node (labeled zero)
        Edge* returnEdge = currentNode->getIncoming().front();
        if (returnEdge->getOrig()->getId() == 0) {
            totalDistance += returnEdge->getDistance();
            res.first = visited;
            res.second = totalDistance;
        }
        visited.pop_back();  // Remove the current node from visited list
        return res;
    }

    // Initialize the best result to infinity
    pair<vector<Vertex *>, double> bestRes;
    bestRes.second = INT16_MAX;

    // Iterate over the neighbors of the current node
    for (Edge *edge : currentNode->getAdj()) {
        Vertex *nextNode = edge->getDest();
        if (find( first: visited.begin(),  last: visited.end(),  val: nextNode) == visited.end()) {
            double edgeDistance = edge->getDistance();
            totalDistance += edgeDistance;

            pair<vector<Vertex *>, double> tempRes = utilBacktrack( currentNode: nextNode,  &: visited, totalDistance,  &: res);
```

```cpp
            // Update the result if a valid tour with smaller total distance is found
            if (!tempRes.first.empty() && tempRes.second < bestRes.second) {
                bestRes = tempRes;
            }

            totalDistance -= edgeDistance;
        }
    }

    visited.pop_back();  // Remove the current node from visited list
    return bestRes;
}


/*!
 * Backtracking algorithm
 * @return pair with a list of nodes of the best tour and the total distance
 */
pair<vector<Vertex *>, double> Graph::backtracking() {
    vector<Vertex *> visited;
    double totalDistance = 0.0;
    pair<vector<Vertex *>, double> res;

    // Start the TSP algorithm from the node labeled zero
    Vertex *startingNode = findVertex( id: 0);
    return utilBacktrack( currentNode: startingNode,  &: visited, totalDistance,  &: res);
}
```

Backtracking Algorithm

Triangular Approximation Heuristic

```cpp
pair<vector<pair<Vertex *, Vertex *>>, double> Graph::triangularApprox() {
    // Create the minimum spanning tree (MST)
    vector<pair<Vertex *, Vertex *>> mstEdges = primMST();
    double totalDistance = 0.0;

    // Check if the MST is empty
    if (mstEdges.empty()) {
        return make_pair(mstEdges, totalDistance);
    }

    // Get the root vertex of the MST (first key in the first element)
    Vertex *root = mstEdges[0].first;

    // Perform a preorder traversal of the MST and calculate the total distance
    unordered_set<Vertex *> visited;
    totalDistance = preorderTraversal(root, visited, mstEdges);
    return make_pair(mstEdges, totalDistance);
}
```

```cpp
double Graph::preorderTraversal(Vertex *root, unordered_set<Vertex *> &visited, vector<pair<Vertex *, Vertex *>> &mstEdges) {
    double totalDistance = 0.0;
    visited.insert(root);

    for (auto &vp : mstEdges) {
        if (visited.find(vp.second) != visited.end()) continue;
        bool edgeExists = false;
        for (Edge *edge : vp.first->getAdj()) {
            if (edge->getDest() == vp.second) {
                edgeExists = true;
                totalDistance += edge->getDistance();
                break;
            }
        }
        if (!edgeExists) totalDistance += HaversineDistance(vp.first, vp.second);

        totalDistance += preorderTraversal(vp.second, visited, mstEdges);
    }

    return totalDistance;
}
```

```cpp
vector<pair<Vertex *, Vertex *>> Graph::primMST() {
    // Create a map to store the MST edges
    vector<pair<Vertex *, Vertex *>> mstEdges;

    // Create a priority queue to store the edges based on their weights
    auto cmp = [](const Edge *e1, const Edge *e2) {
        return e1->getDistance() > e2->getDistance(); // Min-heap based on edge weight
    };
    priority_queue<Edge *, vector<Edge *>, decltype(cmp)> pq(cmp);

    // Create a set to keep track of visited vertices
    unordered_set<Vertex *> visited;

    // Choose any vertex as the starting vertex
    Vertex *startVertex = findVertex(0);
    visited.insert(startVertex);

    // Initialize the priority queue with the edges if the starting vertex
    for (Edge *edge : startVertex->getAdj()) {
        pq.push(edge);
    }

    // Perform Prim's algorithm until all vertices are visited
    while (!pq.empty()) {
        Edge *curr = pq.top();
        pq.pop();

        Vertex *src = curr->getOrig();
        Vertex *dest = curr->getDest();

        // Skip the edge if the destination vertex is already visited
        if (visited.find(dest) != visited.end()) {
            continue;
        }

        // Add the edge to the MST
        mstEdges.push_back({src, dest});

        // Mark both vertices as visited
        visited.insert(src);
        visited.insert(dest);

        // Add the adjacent edges of the newly visited vertex to the priority queue
        for (Edge *edge : dest->getAdj()) {
            if (visited.find(edge->getDest()) == visited.end()) {
                pq.push(edge);
            }
        }
    }

    return mstEdges;
}
```

```cpp
pair<vector<Vertex *>, double> Graph::approxByVertexes() {
    vector<Vertex *> visited;
    double totalDistance = 0.0;

    // Start the tour from the vertex with id = 0
    Vertex *startVertex = findVertex(0);
    visited.push_back(startVertex);

    // Create a set to keep track of unvisited vertices
    unordered_set<Vertex *> unvisited;
    for (auto &v : vertexMap) {
        if (v.second != startVertex) {
            unvisited.insert(v.second);
        }
    }

    while (!unvisited.empty()) {
        Vertex *currVertex = visited.back();
        Vertex *nextVertex = nullptr;
        double minDistance = INT16_MAX;

        // Find the nearest unvisited vertex based on the triangular inequality
        for (Vertex *v : unvisited) {
            double distance = HaversineDistance(currVertex, v);
            if (distance < minDistance) {
                minDistance = distance;
                nextVertex = v;
            }
        }

        // Add the nearest vertex to the tour
        visited.push_back(nextVertex);
        totalDistance += minDistance;

        // Remove the visited vertex from the unvisited set
        unvisited.erase(nextVertex);
    }

    // Add the distance from the last vertex back to the start vertex (id = 0)
    totalDistance += HaversineDistance(visited.back(), startVertex);

    // Return the result
    return make_pair(visited, totalDistance);
}
```

Other Heuristics

- This is a greedy algorithm which iteratively selects the next nearest neighbour, using the haversine formula to calculate distances, until all vertexes have been visited, giving us a very quick way to calculate the shortest path traversing all vertexes.
- Since it doesn't look for edges connected, and instead uses the haversine formula, it only works when we have the coordinates for every vertex.

Thank you