Main

1220604 Rodrigo Correia

1220623 Carolina Sá

1220784 David Sousa

1221124 Tiago Carvalho

ISEP

RELATED TO INTEGRATED PROJET 2023/2024

Group 61

Introduction

This report serves as a complement to the project developed in Information Structures(ESINF) for the Integrated Project 3, which aims to develop a suite of classes and corresponding tests that adhere to the Graph interface, facilitating the administration of a network designed for the distribution of agricultural product baskets.

This network comprises diverse vertices, each representing locations where distribution hubs may be established. The transportation of product baskets is executed through electric vehicles operated by GFH, featuring limited autonomy. These vehicles are stationed at hubs, available for subsequent retrieval by interested parties. It is imperative to note that the distribution of products is contingent upon the operational hours of the respective hub. For the sake of simplicity, it is posited that the vehicles possess a maximum autonomy of A kilometers, maintain an average speed of V km/h, and undergo recharging exclusively at specified locations, with no provision for recharging during transit. Each battery charging session for the vehicles requires Tc minutes, while the average duration for unloading product baskets at hubs approximates Td minutes.

USEI07 uses: Hub manager class: getHubs and setHubsList methods.; Route Manager class: getRouteMaxHubs, getRouteInfo and getRouteTotalDistance methods.

USEI08 uses: Hub Manager class: setList and getHubs methods; Route Manager class: getShortestDeliveryRouteInfo method.

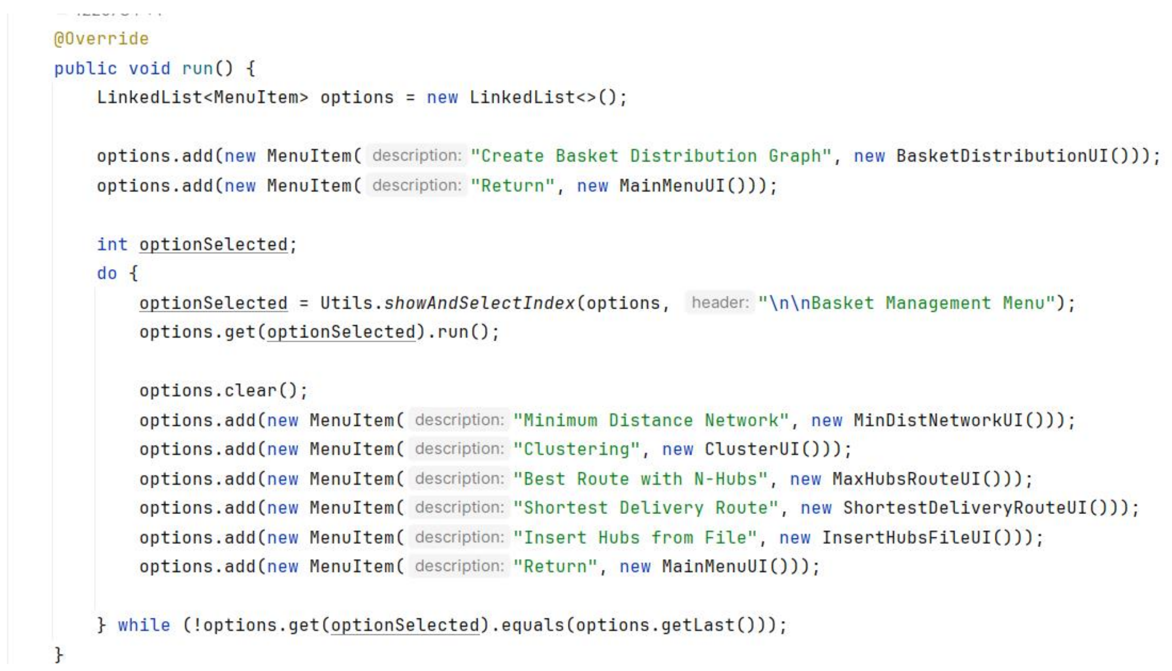
USEI09 uses: ClusterManager class: all methods.

USEI10 uses: AlgorithmsGraph class: allPathsWithAutonomy method.

Contents

Main

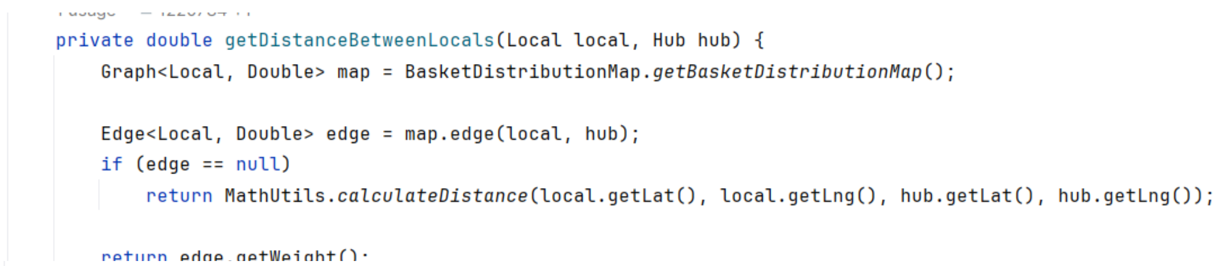
To better organize the set of exercises proposed in a single class, a menu was designed where the user can choose which functionality of the program they want to perform, allowing the application to observe more than one exercise.



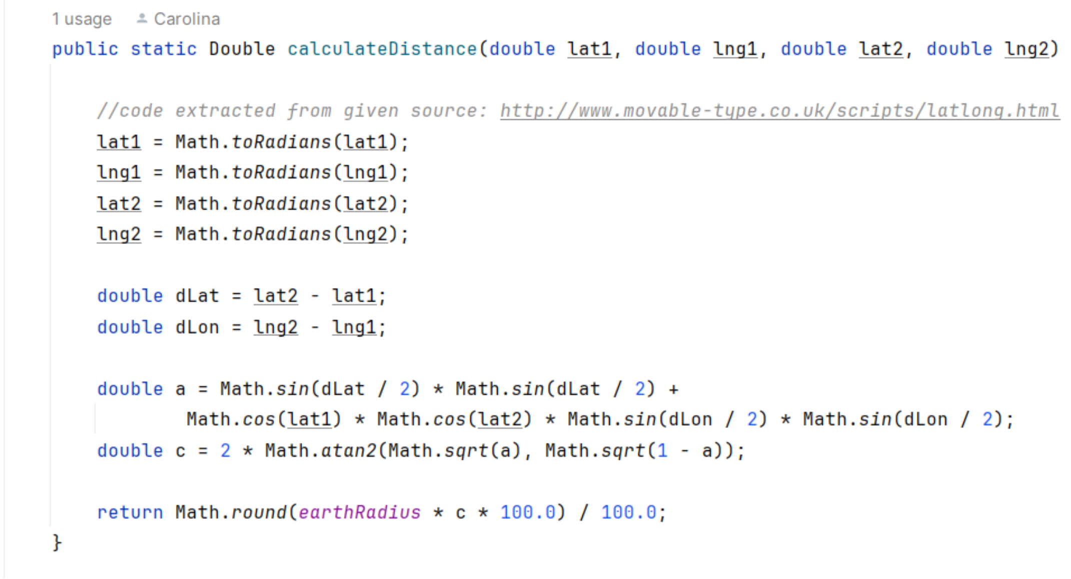
```java
@Override
public void run() {
    LinkedList<MenuItem> options = new LinkedList<>();

    options.add(new MenuItem( description: "Create Basket Distribution Graph", new BasketDistributionUI()));
    options.add(new MenuItem( description: "Return", new MainMenuUI()));

    int optionSelected;
    do {
        optionSelected = Utils.showAndSelectIndex(options,  header: "\n\nBasket Management Menu");
        options.get(optionSelected).run();

        options.clear();
        options.add(new MenuItem( description: "Minimum Distance Network", new MinDistNetworkUI()));
        options.add(new MenuItem( description: "Clustering", new ClusterUI()));
        options.add(new MenuItem( description: "Best Route with N-Hubs", new MaxHubsRouteUI()));
        options.add(new MenuItem( description: "Shortest Delivery Route", new ShortestDeliveryRouteUI()));
        options.add(new MenuItem( description: "Insert Hubs from File", new InsertHubsFileUI()));
        options.add(new MenuItem( description: "Return", new MainMenuUI()));

    } while (!options.get(optionSelected).equals(options.getLast()));
}
```

ClusterManager

The *getDistanceBetweenLocals* method in the ClusterManager class calculates the distance between a 'Local' and a 'Hub'. It attempts to find a direct connection between these two points in a predefined graph. The time complexity is O(1).



```
private double getDistanceBetweenLocals(Local local, Hub hub) {
    Graph<Local, Double> map = BasketDistributionMap.getBasketDistributionMap();

    Edge<Local, Double> edge = map.edge(local, hub);
    if (edge == null)
        return MathUtils.calculateDistance(local.getLat(), local.getLng(), hub.getLat(), hub.getLng());

    return edge.getWeight();
```

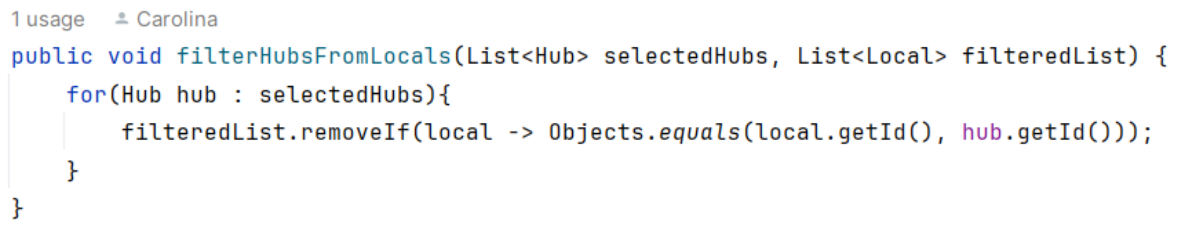


```
1 usage   Carolina
public static Double calculateDistance(double lat1, double lng1, double lat2, double lng2) {

    //code extracted from given source: http://www.movable-type.co.uk/scripts/latlong.html
    lat1 = Math.toRadians(lat1);
    lng1 = Math.toRadians(lng1);
    lat2 = Math.toRadians(lat2);
    lng2 = Math.toRadians(lng2);

    double dLat = lat2 - lat1;
    double dLon = lng2 - lng1;

    double a = Math.sin(dLat / 2) * Math.sin(dLat / 2) +
            Math.cos(lat1) * Math.cos(lat2) * Math.sin(dLon / 2) * Math.sin(dLon / 2);
    double c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));

    return Math.round(earthRadius * c * 100.0) / 100.0;
}
```

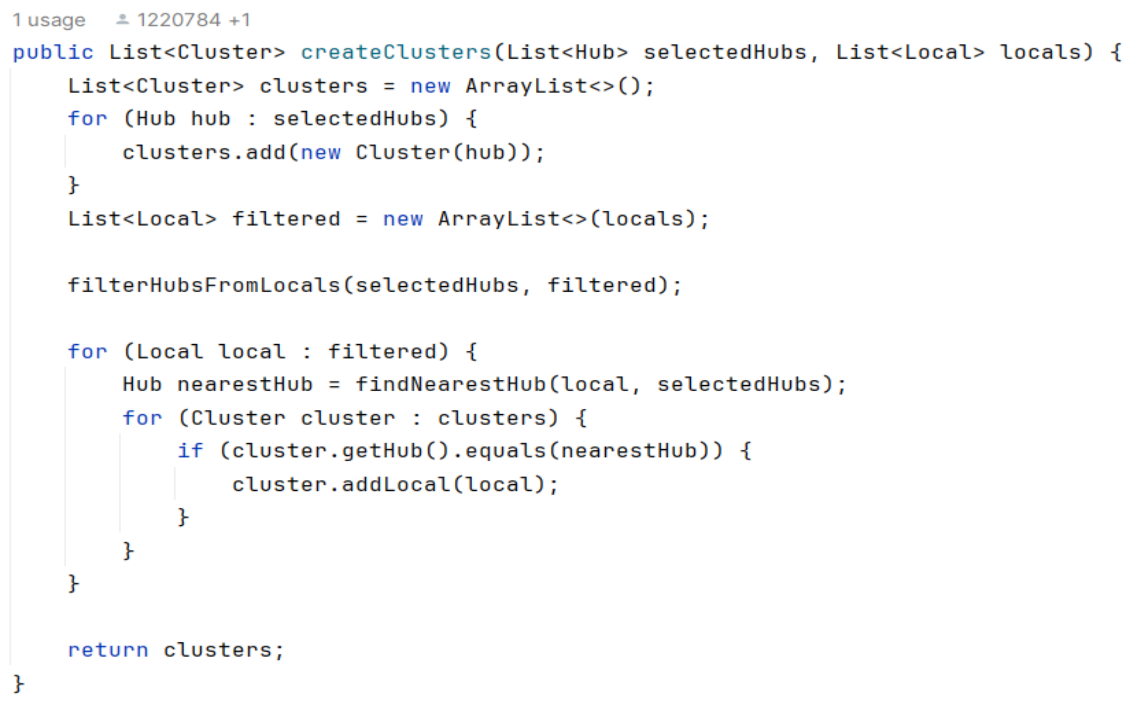
If a direct edge exists, the method returns its weight, signifying the distance. In cases where no direct edge is found, the method calculates the distance using the MathUtils.calculateDistance() function.

The calculateDistance method computes the distance between two geographical points, given their latitude and longitude coordinates. It implements the Haversine formula, a well-known method for calculating the great-circle distance between two points on the surface of a sphere, given their longitudes and latitudes: useful for finding distances on Earth. The method first converts the latitude and longitude coordinates from degrees to radians. It then calculates the differences in latitude and longitude between the two points. The Haversine formula is applied next, which involves calculating the squares of the sine of half the latitude difference and adding it to the product of the cosines of the two latitudes times the square of the sine of half the longitude difference. The central angle (c) between the two points is obtained using the *atan2* function. The final distance is the product of the Earth's radius and the central angle, with the result being rounded to two decimal places. The time complexity is O(1).

The *filterHubsFromLocals* method serves to exclude 'Hub' instances from a list of 'Local' instances. It iterates over a list of pre-selected 'Hubs' and removes any 'Local' from the 'filteredList' that shares the same ID with a 'Hub'. This method is particularly useful for segregating specific locations designated as 'Hubs' from a broader list of locations, ensuring that the resulting list contains only 'Local' instances. The time complexity is O(n*m).

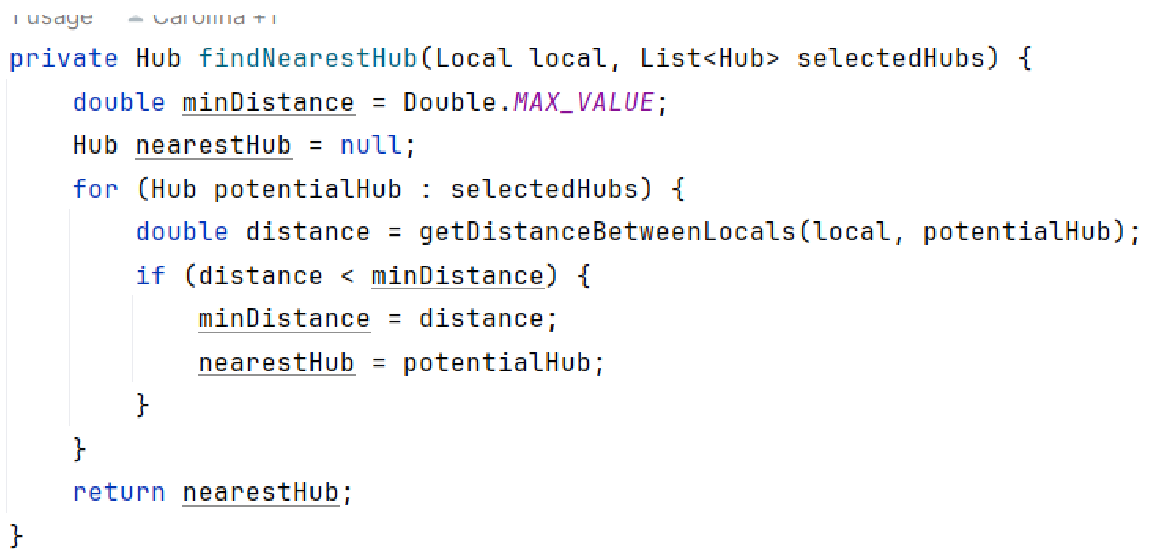

```
1 usage   Carolina
public void filterHubsFromLocals(List<Hub> selectedHubs, List<Local> filteredList) {
    for(Hub hub : selectedHubs){
        filteredList.removeIf(local -> Objects.equals(local.getId(), hub.getId()));
    }
}
```

In the createClusters(List<Hub> selectedHubs, List<Local> locals) method, the goal is to organize 'Local' instances into clusters based on their proximity to selected 'Hubs'. The method begins by creating a new 'Cluster' object for each 'Hub' in the 'selectedHubs' list. It then copies the 'locals' list and removes the 'Hubs' from it using the filterHubsFromLocals() method. Following this, the method finds the nearest 'Hub' for each 'Local' using the findNearestHub() method. Each 'Local' is then added to the cluster corresponding to its nearest 'Hub', effectively grouping the 'Locals' based on their closest 'Hub'. The time complexity is O(m + n * m^2).



```
1 usage   1220784 +1
public List<Cluster> createClusters(List<Hub> selectedHubs, List<Local> locals) {
    List<Cluster> clusters = new ArrayList<>();
    for (Hub hub : selectedHubs) {
        clusters.add(new Cluster(hub));
    }
    List<Local> filtered = new ArrayList<>(locals);

    filterHubsFromLocals(selectedHubs, filtered);

    for (Local local : filtered) {
        Hub nearestHub = findNearestHub(local, selectedHubs);
        for (Cluster cluster : clusters) {
            if (cluster.getHub().equals(nearestHub)) {
                cluster.addLocal(local);
            }
        }
    }

    return clusters;
}
```

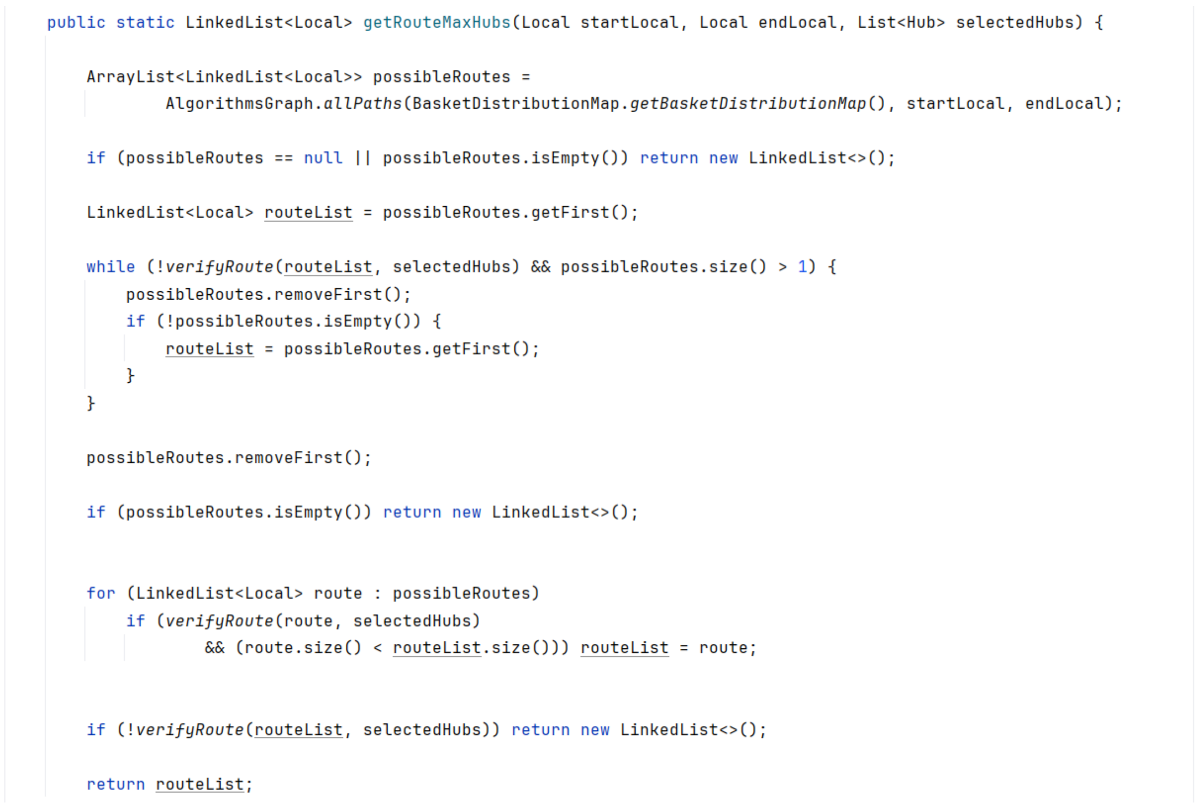
The *findNearestHub(Local local, List<Hub> selectedHubs)* method is designed to identify the nearest 'Hub' relative to a given 'Local'. It iterates through each 'Hub' in the 'selectedHubs' list, calculating the distance to the 'Local' using the getDistanceBetweenLocals() method. The method keeps track of the 'Hub' that is closest to the 'Local' by comparing distances. The 'Hub' with the shortest distance to the 'Local' is then returned as the nearest 'Hub'. This functionality is crucial for the cluster creation process, as it determines how 'Locals' are grouped into clusters based on their proximity to 'Hubs'. The time complexity is O(n).



```
private Hub findNearestHub(Local local, List<Hub> selectedHubs) {
    double minDistance = Double.MAX_VALUE;
    Hub nearestHub = null;
    for (Hub potentialHub : selectedHubs) {
        double distance = getDistanceBetweenLocals(local, potentialHub);
        if (distance < minDistance) {
            minDistance = distance;
            nearestHub = potentialHub;
        }
    }
    return nearestHub;
}
```

RouteManager

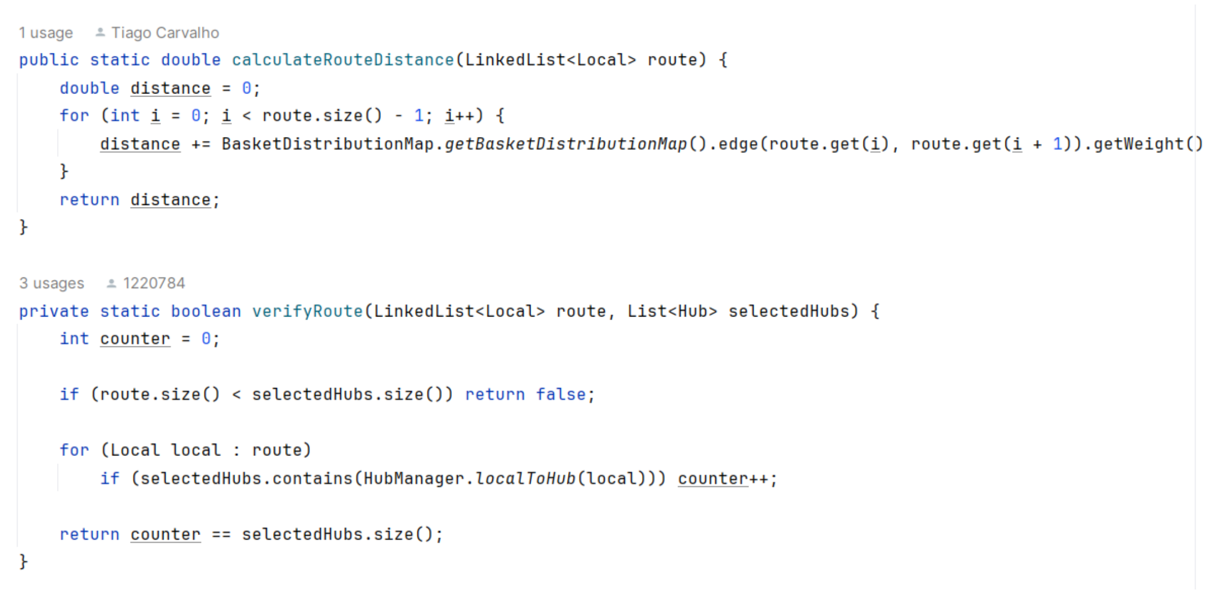
The getRouteMaxHubs(Local startLocal, Local endLocal, List<Hub> selectedHubs) method determines a route from a starting point (startLocal) to an end point (endLocal), passing through a set of selected hubs (selectedHubs). Initially, it generates all possible routes between the start and end points using AlgorithmsGraph.allPaths(). It then iterates through these routes, selecting the first route that passes through all the selected hubs. If no such route exists, an empty LinkedList is returned. This method ensures that the chosen route maximally utilizes the hubs while connecting the start and end locations. The complexity is O(2^V)



```
public static LinkedList<Local> getRouteMaxHubs(Local startLocal, Local endLocal, List<Hub> selectedHubs) {

    ArrayList<LinkedList<Local>> possibleRoutes =
            AlgorithmsGraph.allPaths(BasketDistributionMap.getBasketDistributionMap(), startLocal, endLocal);

    if (possibleRoutes == null || possibleRoutes.isEmpty()) return new LinkedList<>();

    LinkedList<Local> routeList = possibleRoutes.getFirst();

    while (!verifyRoute(routeList, selectedHubs) && possibleRoutes.size() > 1) {
        possibleRoutes.removeFirst();
        if (!possibleRoutes.isEmpty()) {
            routeList = possibleRoutes.getFirst();
        }
    }

    possibleRoutes.removeFirst();

    if (possibleRoutes.isEmpty()) return new LinkedList<>();


    for (LinkedList<Local> route : possibleRoutes)
        if (verifyRoute(route, selectedHubs)
                && (route.size() < routeList.size())) routeList = route;


    if (!verifyRoute(routeList, selectedHubs)) return new LinkedList<>();

    return routeList;
```

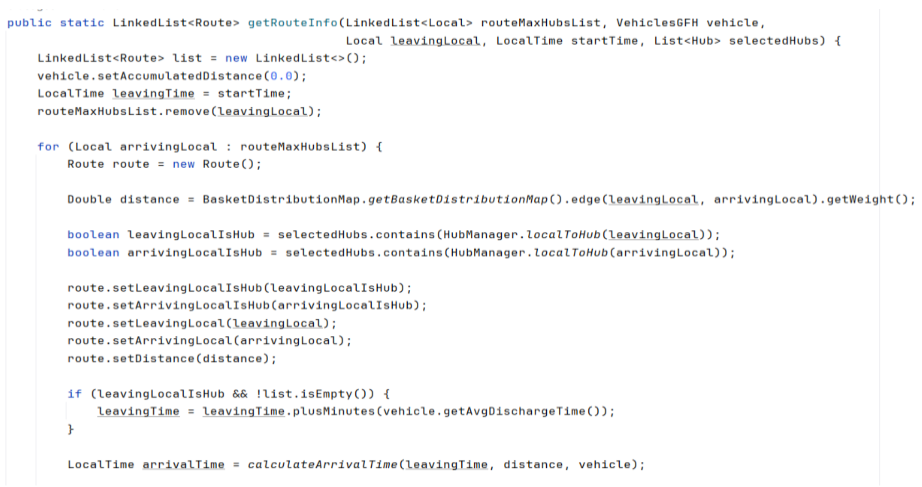
The calculateRouteDistance method calculates the total distance of a given route, represented as a LinkedList of Local points. It iterates through the route, summing up the distances between consecutive Local points. These distances are obtained from a graph structure, presumably representing a network of locations, where each edge has a weight representing the distance between points. This method provides a quantitative measure of the route's length. The time complexity is O(V).

The verifyRoute method’s purpose is to validate whether a given route (route) includes a specific set of hubs (selectedHubs). It checks if the route contains at least as many points as there are selected hubs and then verifies whether each hub is present in the route. If the route includes all selected hubs exactly once, the method returns true, indicating the route's validity with respect to the hub requirements. The time complexity is O(N*M).



```
1 usage   Tiago Carvalho
public static double calculateRouteDistance(LinkedList<Local> route) {
    double distance = 0;
    for (int i = 0; i < route.size() - 1; i++) {
        distance += BasketDistributionMap.getBasketDistributionMap().edge(route.get(i), route.get(i + 1)).getWeight();
    }
    return distance;
}

3 usages   1220784
private static boolean verifyRoute(LinkedList<Local> route, List<Hub> selectedHubs) {
    int counter = 0;

    if (route.size() < selectedHubs.size()) return false;

    for (Local local : route)
        if (selectedHubs.contains(HubManager.localToHub(local))) counter++;

    return counter == selectedHubs.size();
}
```

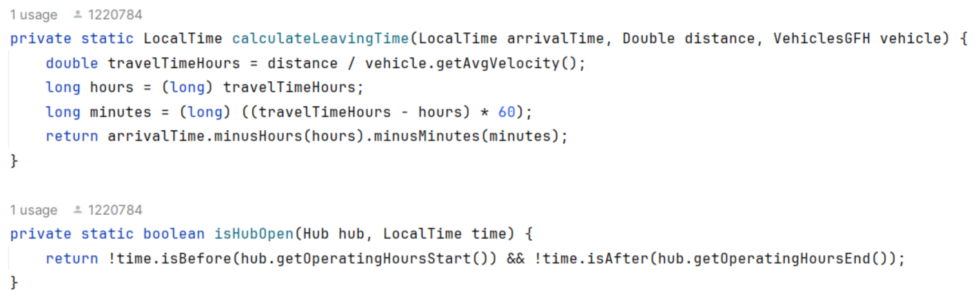
The getRouteInfo method is designed to generate detailed information about a route, including the specifics of each leg of the journey. It calculates various attributes such as distance, arrival and departure times, and hub utilization for each segment of the route. The method takes into consideration the vehicle's characteristics, operational times of hubs, and initial starting conditions to provide a comprehensive plan for navigating the route. The time complexity is O(N^2).



```java
public static LinkedList<Route> getRouteInfo(LinkedList<Local> routeMaxHubsList, VehiclesGFH vehicle,
                                             Local leavingLocal, LocalTime startTime, List<Hub> selectedHubs) {
    LinkedList<Route> list = new LinkedList<>();
    vehicle.setAccumulatedDistance(0.0);
    LocalTime leavingTime = startTime;
    routeMaxHubsList.remove(leavingLocal);

    for (Local arrivingLocal : routeMaxHubsList) {
        Route route = new Route();

        Double distance = BasketDistributionMap.getBasketDistributionMap().edge(leavingLocal, arrivingLocal).getWeight();

        boolean leavingLocalIsHub = selectedHubs.contains(HubManager.localToHub(leavingLocal));
        boolean arrivingLocalIsHub = selectedHubs.contains(HubManager.localToHub(arrivingLocal));

        route.setLeavingLocalIsHub(leavingLocalIsHub);
        route.setArrivingLocalIsHub(arrivingLocalIsHub);
        route.setLeavingLocal(leavingLocal);
        route.setArrivingLocal(arrivingLocal);
        route.setDistance(distance);

        if (leavingLocalIsHub && !list.isEmpty()) {
            leavingTime = leavingTime.plusMinutes(vehicle.getAvgDischargeTime());
        }

        LocalTime arrivalTime = calculateArrivalTime(leavingTime, distance, vehicle);
```

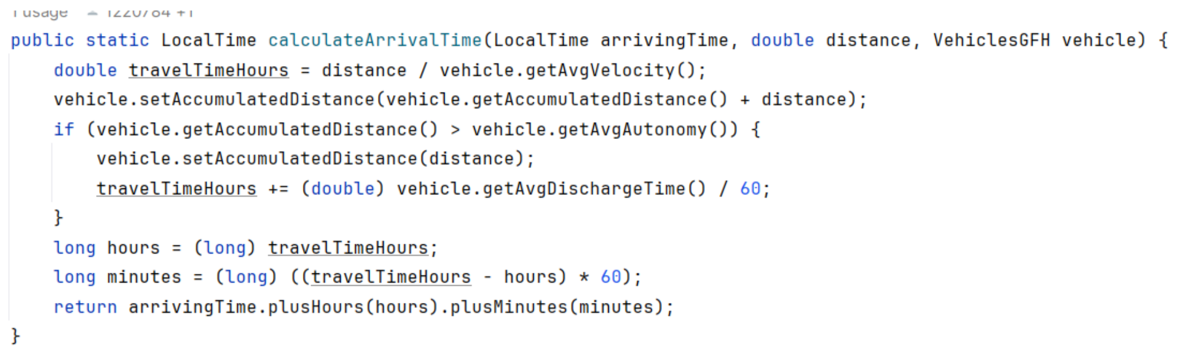
The calculateLeavingTime method calculates the departure time from a location based on the arrival time, distance to the next location, and the average velocity of the vehicle. It determines how long the journey will take and subtracts this duration from the arrival time to find the leaving time. This is useful for scheduling and logistics planning, ensuring timely departures after arrivals. The time complexity is O(1).

The isHubOpen(Hub hub, LocalTime time) method checks if a hub is operational at a given time. It compares the time parameter with the hub's operating hours. If the time falls within the hub's operating hours, the method returns true, indicating that the hub is open. This function is essential for route planning, ensuring that vehicles arrive at hubs during their operational times. The time complexity is O(1).



```java
1 usage  1220784
private static LocalTime calculateLeavingTime(LocalTime arrivalTime, Double distance, VehiclesGFH vehicle) {
    double travelTimeHours = distance / vehicle.getAvgVelocity();
    long hours = (long) travelTimeHours;
    long minutes = (long) ((travelTimeHours - hours) * 60);
    return arrivalTime.minusHours(hours).minusMinutes(minutes);
}

1 usage  1220784
private static boolean isHubOpen(Hub hub, LocalTime time) {
    return !time.isBefore(hub.getOperatingHoursStart()) && !time.isAfter(hub.getOperatingHoursEnd());
}
```

The calculateArrivalTime(LocalTime arrivingTime, double distance, VehiclesGFH vehicle) method calculates the arrival time at a destination based on the starting time, the distance to the destination, and the vehicle's average velocity. It also adjusts the vehicle's accumulated distance and accounts for the vehicle's average discharge time if necessary. This calculation is crucial for determining accurate arrival times and for managing the vehicle's operational constraints. The time complexity is O(1).


```
1 usage  ≗ 1220784 +1
public static LocalTime calculateArrivalTime(LocalTime arrivingTime, double distance, VehiclesGFH vehicle) {
    double travelTimeHours = distance / vehicle.getAvgVelocity();
    vehicle.setAccumulatedDistance(vehicle.getAccumulatedDistance() + distance);
    if (vehicle.getAccumulatedDistance() > vehicle.getAvgAutonomy()) {
        vehicle.setAccumulatedDistance(distance);
        travelTimeHours += (double) vehicle.getAvgDischargeTime() / 60;
    }
    long hours = (long) travelTimeHours;
    long minutes = (long) ((travelTimeHours - hours) * 60);
    return arrivingTime.plusHours(hours).plusMinutes(minutes);
}
```

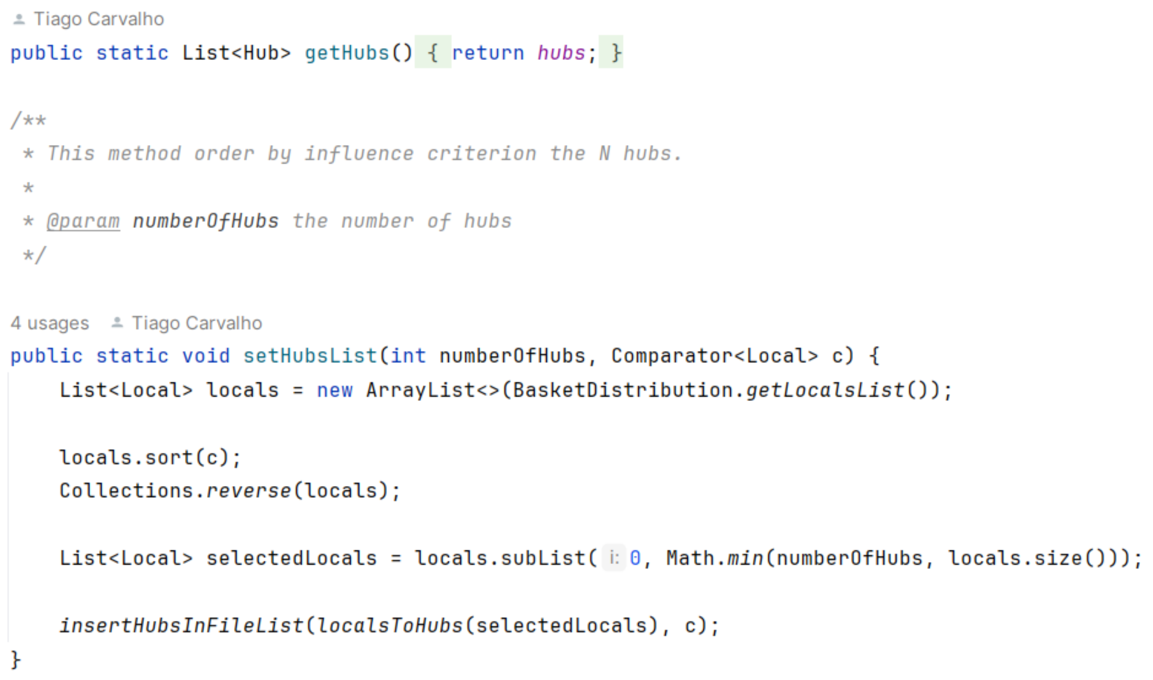
The getShortestDeliveryRoute(Local vertex, List<Hub> hubs) method finds the shortest delivery route starting and ending at a specified vertex, while visiting a list of hubs. It iteratively calculates the shortest path from the current vertex to each hub and then to the next, eventually returning to the starting vertex. The method uses an algorithm to find the shortest path between points, updating the best route and distance at each iteration. This functionality is vital for optimizing delivery routes in terms of distance. The time complexity is O(N^3).

The getShortestDeliveryRouteInfo method combines the functionality of getShortestDeliveryRoute and getRouteInfo to provide detailed information about the shortest delivery route for a vehicle. It first determines the shortest delivery route using getShortestDeliveryRoute and then generates detailed route information including timings and distances for each leg of the journey using getRouteInfo. This comprehensive approach is essential for efficient route planning and management in logistics and delivery operations. The time complexity is O(N^3).

HubManager

The getHubs method provides access to the current list of hubs managed by HubManager. It returns a list of Hub objects, enabling other parts of the program to retrieve information about all the hubs currently stored in the HubManager class. This method is critical for accessing hub data throughout the application. The time complexity is O(1).

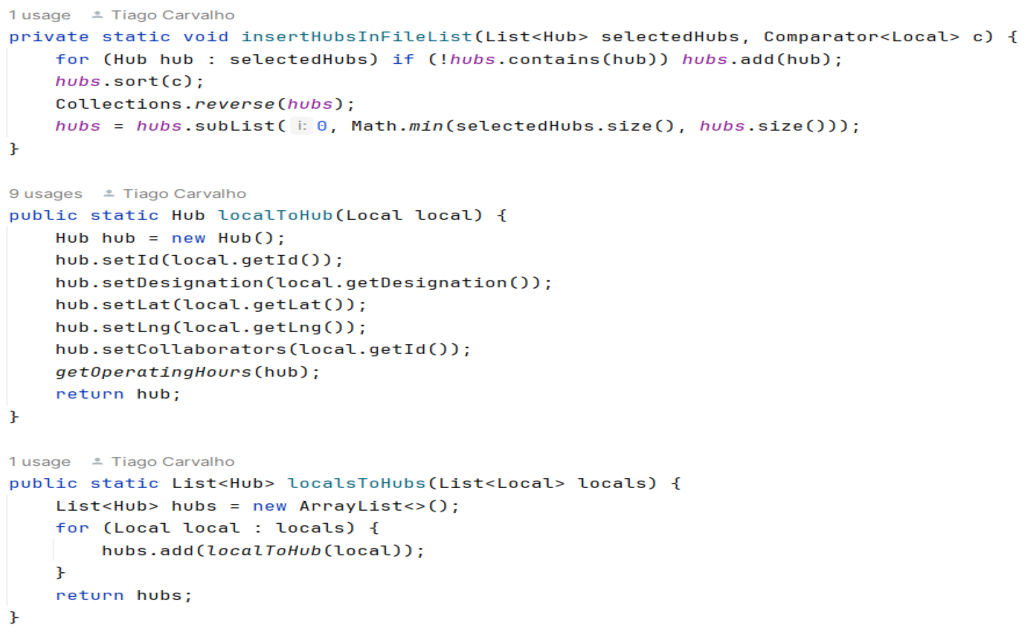
The setHubsList method is responsible for setting up and organizing the list of hubs based on specific criteria. It takes the number of hubs to be included (numberOfHubs) and a comparator (c) used for sorting. The method sorts a list of Local objects, reverses it, and selects the top elements based on the specified number. It then converts these selected Local objects into hubs and organizes them accordingly. This method is pivotal for initializing and ordering hubs based on given criteria. The time complexity is O(n*log(n)).



```
Tiago Carvalho
public static List<Hub> getHubs() { return hubs; }

/**
 * This method order by influence criterion the N hubs.
 *
 * @param numberOfHubs the number of hubs
 */

4 usages   Tiago Carvalho
public static void setHubsList(int numberOfHubs, Comparator<Local> c) {
    List<Local> locals = new ArrayList<>(BasketDistribution.getLocalsList());

    locals.sort(c);
    Collections.reverse(locals);

    List<Local> selectedLocals = locals.subList( i: 0, Math.min(numberOfHubs, locals.size()));

    insertHubsInFileList(localsToHubs(selectedLocals), c);
}
```

The insertHubsInFileList method inserts a list of selected hubs into the existing list of hubs managed by the HubManager. It ensures that the existing list does not already contain the selected hubs before adding them. The list is then sorted using the provided comparator and trimmed to the size of the selected hubs list. This method is essential for updating the internal list of hubs with new selections while maintaining a specific order. The time complexity is O(n*log(n)).

The localToHub method converts a Local object into a Hub object. It takes the properties of a Local instance, such as ID, designation, latitude, longitude, and collaborators, and assigns them to a new Hub instance. Additionally, it sets the operating hours for the hub based on its ID. This method is crucial for transforming local points into hub entities with additional properties. The time complexity is O(1).

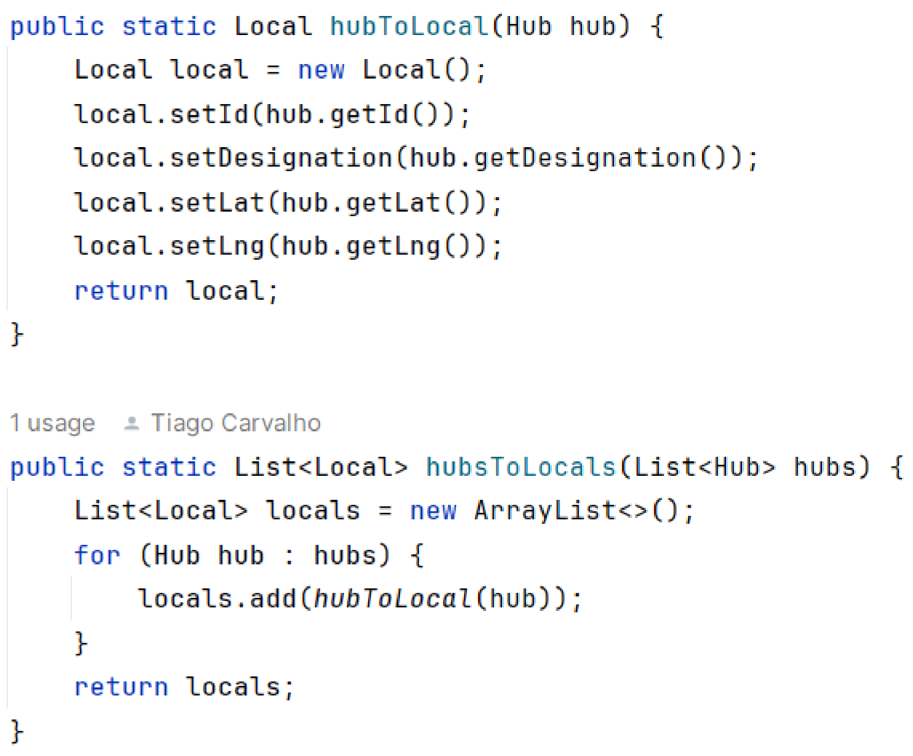


```
1 usage   Tiago Carvalho
private static void insertHubsInFileList(List<Hub> selectedHubs, Comparator<Local> c) {
    for (Hub hub : selectedHubs) if (!hubs.contains(hub)) hubs.add(hub);
    hubs.sort(c);
    Collections.reverse(hubs);
    hubs = hubs.subList( i: 0, Math.min(selectedHubs.size(), hubs.size()));
}

9 usages   Tiago Carvalho
public static Hub localToHub(Local local) {
    Hub hub = new Hub();
    hub.setId(local.getId());
    hub.setDesignation(local.getDesignation());
    hub.setLat(local.getLat());
    hub.setLng(local.getLng());
    hub.setCollaborators(local.getId());
    getOperatingHours(hub);
    return hub;
}

1 usage   Tiago Carvalho
public static List<Hub> localsToHubs(List<Local> locals) {
    List<Hub> hubs = new ArrayList<>();
    for (Local local : locals) {
        hubs.add(localToHub(local));
    }
    return hubs;
}
```

The localsToHubs method transforms a list of Local objects into a list of Hub objects. It iterates through each Local in the provided list and uses the localToHub method to convert each one into a Hub. This method is instrumental in creating a collection of hub entities from local points. The time complexity is O(1).

The hubToLocal(Hub hub), an inverse of localToHub, converts a Hub object back into a Local object. It transfers the properties from a Hub to a new Local instance, such as ID, designation, latitude, and longitude. This method is useful for operations that require treating hubs as general local points. The time complexity is O(1).

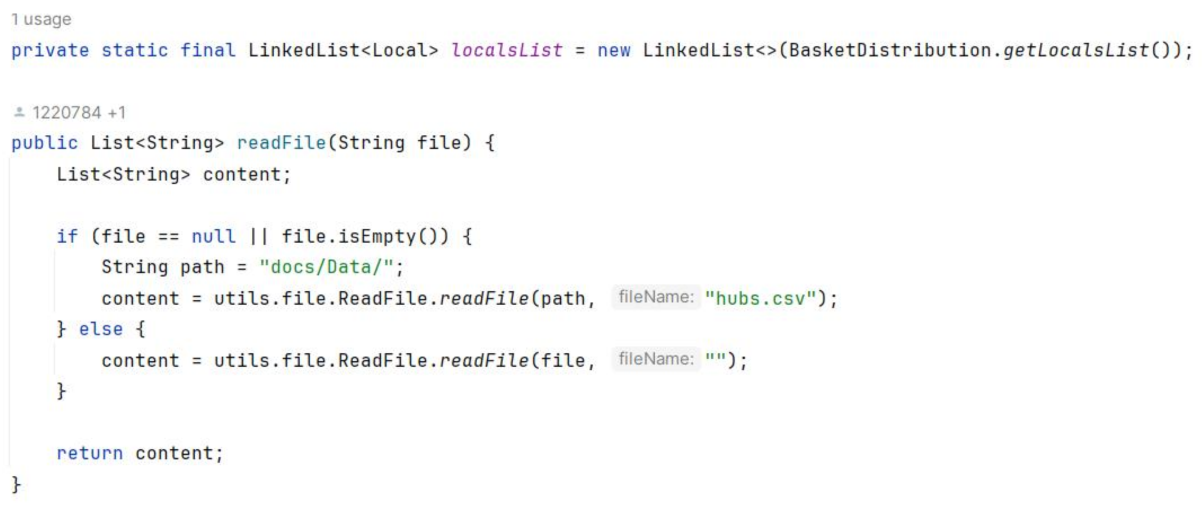
The hubsToLocals, similar to localsToHubs, method converts a list of Hub objects into a list of Local objects. It processes each Hub in the provided list, utilizing the hubToLocal method to perform the conversion. This method is beneficial for instances where hub entities need to be handled as generic local points. The time complexity is O(n).



```java
public static Local hubToLocal(Hub hub) {
    Local local = new Local();
    local.setId(hub.getId());
    local.setDesignation(hub.getDesignation());
    local.setLat(hub.getLat());
    local.setLng(hub.getLng());
    return local;
}

1 usage   Tiago Carvalho
public static List<Local> hubsToLocals(List<Hub> hubs) {
    List<Local> locals = new ArrayList<>();
    for (Hub hub : hubs) {
        locals.add(hubToLocal(hub));
    }
    return locals;
}
```

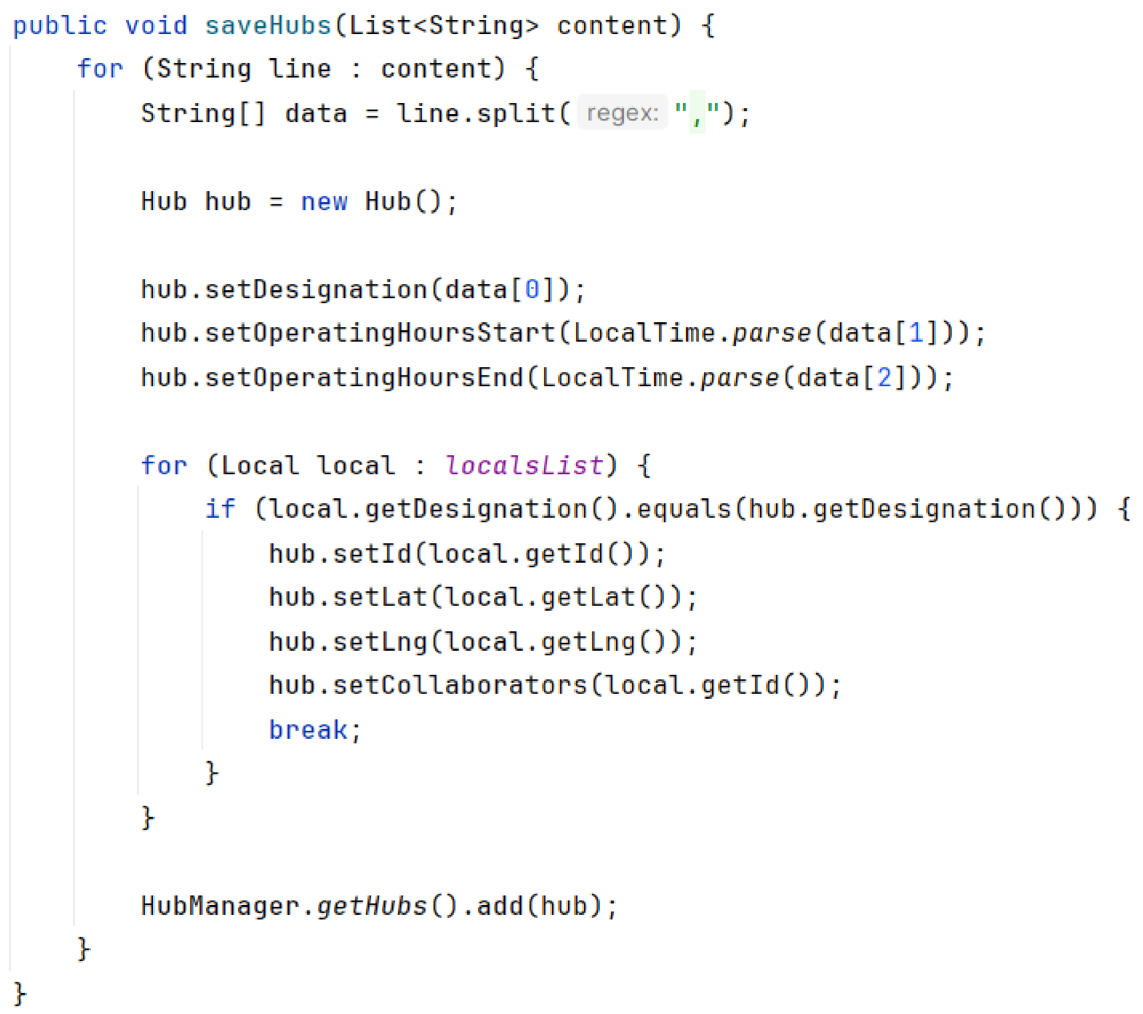
InsertHubsFile

The readFile method is responsible for reading the content of a file and returning it as a list of strings. If the file argument is null or empty, it defaults to reading a file named "hubs.csv" from a specified directory ("docs/Data/"). Otherwise, it reads the file specified by the file argument. The method leverages a utility function utils.file.ReadFile.readFile to perform the actual file reading. This design allows the method to be flexible in handling different file sources or defaulting to a pre-determined file when necessary. The time is O(n).



```
1 usage
private static final LinkedList<Local> localsList = new LinkedList<>(BasketDistribution.getLocalsList());

1220784 +1
public List<String> readFile(String file) {
    List<String> content;

    if (file == null || file.isEmpty()) {
        String path = "docs/Data/";
        content = utils.file.ReadFile.readFile(path,  fileName: "hubs.csv");
    } else {
        content = utils.file.ReadFile.readFile(file,  fileName: "");
    }

    return content;
}
```

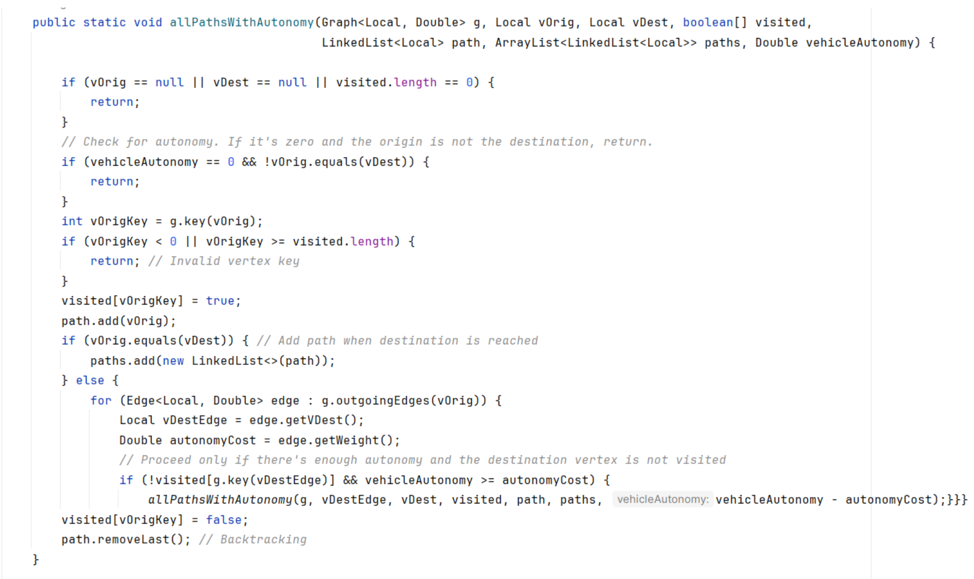
The saveHubs method processes a list of strings representing the lines of a file, presumably containing hub data. Each line is split into different data components using a comma as the delimiter. For each line, a new Hub object is created and populated with data from the line (designation, operating hours start and end times). The method then matches each hub with a Local from a pre-existing list (localsList) based on the designation. If a match is found, additional attributes like ID, latitude, longitude, and collaborators are set for the hub. Finally, the fully populated hub is added to the list of hubs managed by HubManager. This method effectively transforms raw file data into usable hub objects, integrating them into the application's data structure. The time complexity is O(n*m).



```
public void saveHubs(List<String> content) {
    for (String line : content) {
        String[] data = line.split( regex: ",");

        Hub hub = new Hub();

        hub.setDesignation(data[0]);
        hub.setOperatingHoursStart(LocalTime.parse(data[1]));
        hub.setOperatingHoursEnd(LocalTime.parse(data[2]));

        for (Local local : localsList) {
            if (local.getDesignation().equals(hub.getDesignation())) {
                hub.setId(local.getId());
                hub.setLat(local.getLat());
                hub.setLng(local.getLng());
                hub.setCollaborators(local.getId());
                break;
            }
        }

        HubManager.getHubs().add(hub);
    }
}
```

AlgorithmsGraph

The allPathsWithAutonomy method is a sophisticated pathfinding algorithm in a graph, tailored to accommodate a constraint based on the concept of 'autonomy', which could represent a quantifiable resource like fuel or energy. This method is an adaptation of the classic all-paths search algorithm, integrating an additional layer of complexity through the autonomy constraint. Upon invocation, the method first conducts preliminary validations, ensuring that the input vertices and visited array are viable for traversal. It then proceeds to traverse the graph recursively, beginning at the specified origin vertex (vOrig) and aiming for the destination (vDest).



```java
public static void allPathsWithAutonomy(Graph<Local, Double> g, Local vOrig, Local vDest, boolean[] visited,
                                        LinkedList<Local> path, ArrayList<LinkedList<Local>> paths, Double vehicleAutonomy) {

    if (vOrig == null || vDest == null || visited.length == 0) {
        return;
    }
    // Check for autonomy. If it's zero and the origin is not the destination, return.
    if (vehicleAutonomy == 0 && !vOrig.equals(vDest)) {
        return;
    }
    int vOrigKey = g.key(vOrig);
    if (vOrigKey < 0 || vOrigKey >= visited.length) {
        return; // Invalid vertex key
    }
    visited[vOrigKey] = true;
    path.add(vOrig);
    if (vOrig.equals(vDest)) { // Add path when destination is reached
        paths.add(new LinkedList<>(path));
    } else {
        for (Edge<Local, Double> edge : g.outgoingEdges(vOrig)) {
            Local vDestEdge = edge.getVDest();
            Double autonomyCost = edge.getWeight();
            // Proceed only if there's enough autonomy and the destination vertex is not visited
            if (!visited[g.key(vDestEdge)] && vehicleAutonomy >= autonomyCost) {
                allPathsWithAutonomy(g, vDestEdge, vDest, visited, path, paths,  vehicleAutonomy: vehicleAutonomy - autonomyCost);}}}
    visited[vOrigKey] = false;
    path.removeLast(); // Backtracking
}
```

The traversal is governed by the autonomy parameter, which decrements with each traversed edge, reflecting the consumption of the resource. The method carefully tracks visited vertices to avoid redundant paths and employs backtracking - a common technique in recursive algorithms - to explore all potential paths that adhere to the autonomy constraint. Upon reaching the destination vertex, it records the successfully traversed path. This algorithm is particularly adept in scenarios like route planning where resource constraints are pivotal, offering a comprehensive set of viable paths within the specified autonomy limits.

G061 – SPRINT3

Note: All Time Complexity Calculations were done in Intellij, commented in each method. The overall (result) Time Complexity was added to this report.