FYP-II Chapter

5

Group # 5

Routing Optimization System

**Introduction:**

In This chapter we are discussing about the flow and step of algorithm that we have designed. We are also discussing about the database queries which are used in Sql Server and we are showing screenshots of our application that shows functionalities. We have briefly provided a few clarifications about the sort of functionalities available on the system and source code.

**Algorithm:**

* The basic structure of the algorithm would be as follows:
  + **Input:** Order, Order\_Location, Restaurant\_Location, sigma(radius) value, max\_orders\_allowed
  + **Output:** Rider, Optimized\_Route
  + **Steps:**
    1. Filtering(Restaurant\_Location, sigma, max\_orders\_allowed): possible\_riders
    2. Foreach possible\_rider:
       1. Calculate the fitness\_function (Rider, Order\_Location).
    3. Rank the rider or arange the possible\_riders array on the basis of their fitness value.
    4. Select the top most rider.
* **Filtering** is the step where we select the riders which could possible minimize the total time taken to deliver a parcel.
* Following are the steps for the Filteringprocess:
  + **Filtering(Restaurant\_Location, (σ)sigma, max\_order\_allowed):**
  + **Input:** Restaurant\_Location
  + **Output:** Possible\_riders
  + **Steps:**
    1. Select all the riders present in the same area.
    2. Initialize an empty array of riders; assign to possible\_riders
    3. Foreach rider:
       1. Calculate the distance of the rider from the restaurant location; assign to rider\_Distance
       2. If (ride\_Distance <= sigma and rider\_order\_count < max\_order\_allowed):
          - Add rider to possible\_riders
    4. Return possible\_riders.
* **Step 1 (Filtering):** This step selects the riders from the overall population and filters out only those fewer riders which could possible minimize the delivery time.
* The above function works by first selecting the riders which are present in the area of the riders. Then, out of those selected riders, it selects the riders present in the radius as specified by **σ** which in our case we have fixed it to 5 km. The value of **σ** can be increased or decreased if we are not getting the enough numbers of riders in a particular area. An empty array of possible\_riders is also initialized in the beginning. At a given particular time, we should have at least five riders to compare and find the best possible rider. While filtering out the riders, we also check if the rider is allowed to take more order or not i.e. if the rider has exceeded its max order limit or not which in our case we have decided as 5 i.e. a rider is allowed to have maximum 5 orders at a time. For the time being, we are assuming the values (such as σ, max\_orders and the number or riders to search for) as per used by most of the real world companies such as Uber Eats and foodpanda.
* If a rider fits the above criteria, it is added to the possible\_riders array.
* The final possible\_riders array is returned in the end to be used for the calculation of fitness values which is the step 2 of the basic structure of the algorithm.
* **Step 2 (Calculation of fitness function of every rider):** This part of the algorithm is not finalized yet in a proper format but the discussed points are mentioned below.
* In this step, we will use the array of riders got in step1 and **for each rider** we will do the following:
  + All the possible networks of routes will be created that a rider can opt for at a particular time. For this purpose, we might use voronoi diagram.

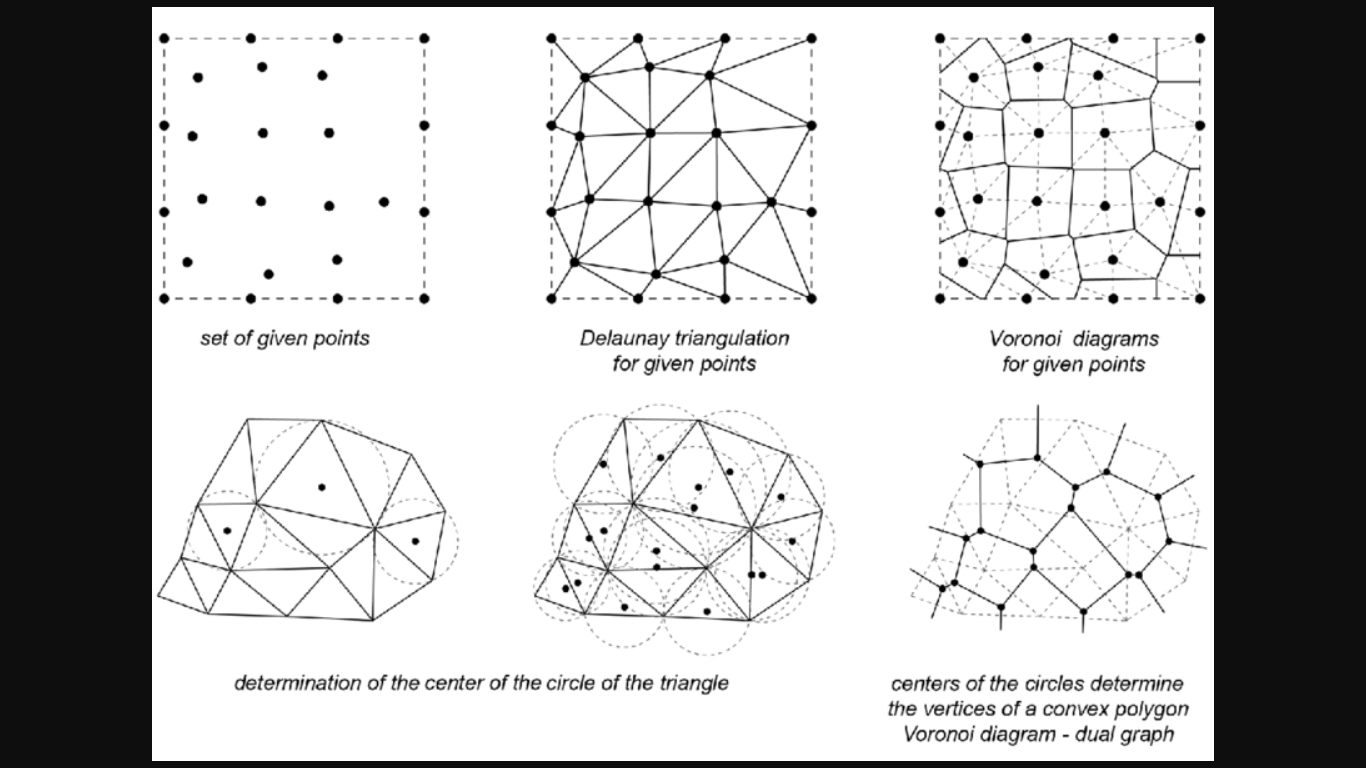


Figure : A simple Voronoi Diagram showing the network of possible routes.

* + After finding all the possible routes, a local search will be performed to find the best possible network that would get the job done in the least time. This part would require the parameters about the rider such as the orders it already has and the order for which we are finding the best rider, the delivery locations where the rider has to make the deliveries and other route related data such as the path length etc. For finding the best route for a rider we have decided to use A\* algorithm to find the shortest path among all the possible paths.
  + In the next step, we have to calculate the fitness value of every rider using the route time got from the above step. We will also use the order\_count(number of orders a rider has) and the rank (rating) of the rider to calculate its fitness value.
  + We will form a scalarization function to calculate the fitness value.
  + Order\_count and the performance (rating) of the rider will be used as the tie breaker rule in case if two or more riders come equal after calculating the fitness value.
  + Still, if two riders come equal, we will choose a random rider.
* **Step 3 & 4 (Rider Selection):** After getting the fitness value we will rank the riders according to their fitness values.
* After the ranking step, the top most rider will be selected and suggested as the **best possible rider**.

**Back-End Design:**

using RouteOptimization.App\_Files;

using RouteOptimization.Models;

using System;

using System.Collections.Generic;

using System.Data.SqlClient;

using System.Linq;

using System.Web;

namespace RouteOptimization

{

public static class Optimization

{

// location format used for this project is [lat, long]

// constant for changing the status of order to "ASSIGNED"

private const string ORDER\_STATUS\_ASSIGNED = "ASSIGNED";

// number of kms upto which the program should search for the riders.

private const int SIGMA = 5;

// maximum number of orders a rider can have.

private const int MAX\_ORDERS = 5;

// main optimization algorithm.

public static void Optimize(orders\_processed order)

{

// get restaurant location from the order

double[] pickup\_location = new double[] { (double)order.hub\_latitude, (double)order.hub\_longitude };

// get the rides that could possible be the best rider for the job.

var possible\_riders = Filtering(pickup\_location);

// empty array for storing fitness values of the riders.

var rider\_fitnesses = new List<RiderFitness>();

// for every rider...

foreach (var item in possible\_riders)

{

// get the shortest route length after this order.

var shortestRouteLength = GetShortestRoute(item, order);

// get the fitness value for this rider with this order.

var item\_fitness = GetFitnessValue((short)item.driver\_rating, shortestRouteLength.Sum(s => s.temp\_distance));

// add to the fitness values array for comparison.

rider\_fitnesses.Add(new RiderFitness() { driver\_id = item.driver\_id, rider\_fitness = item\_fitness });

}

// sort the array of fitness values according to their fitness values and select the first one.

var rider\_id = rider\_fitnesses.OrderBy(o => o.rider\_fitness).FirstOrDefault().driver\_id;

// assign the current order to the selected rider and change the order status to "ASSIGNED".

order.driver\_id = rider\_id;

order.order\_status = ORDER\_STATUS\_ASSIGNED;

}

// Filter out the rider that could possibly be the best rider.

private static List<driver> Filtering(double[] pickup\_location)

{

// empty list for riders to be returned

List<driver> possible\_riders = new List<driver>();

// empty list of riders for all riders got from the database.

List<driver> all\_riders = new List<driver>();

using (var db = new Optimization\_RWEntities())

{

// define parameters for the stored procedure to be called.

SqlParameter[] param = new SqlParameter[] {

// new SqlParameter("@lat", -30.0474147), // hard coded referene points

// new SqlParameter("@long", -51.2135086), // for testing only.

new SqlParameter("@lat", pickup\_location[0]),

new SqlParameter("@long", pickup\_location[1])

};

// call stored procedure that will return all the riders within an area of 5 km in the form of a list.

all\_riders = db.Database.SqlQuery<driver>("GetRidersIn5KmRadius @lat,@long", param).ToList();

// filter only the riders having orders less than 5

for (int i = 0, j = 0; i < all\_riders.Count; i++)

{

var rider\_id = all\_riders[i].driver\_id; // get rider id

//// get the order count for this rider

//var order\_count = db.orders\_processed.Where(w => w.driver\_id == rider\_id &&

//!w.order\_status.Equals("FINISHED")).Count();

// define parameters for the stored procedure to be called.

SqlParameter[] param2 = new SqlParameter[] { new SqlParameter("@rider\_id", rider\_id) };

// call stored procedure that will return all the riders within an area of 5 km in the form of a list.

// var order\_count = db.Database.SqlQuery<Item>("GetRidersCurrentOrderCount @rider\_id", param2);

var data = db.Database.SqlQuery<int>(@"declare @num int exec @num = GetRidersCurrentOrderCount @rider\_id select @num", param2);

var order\_count = data.First();

// if the orders of this rider is less than 5, then add to the possible riders array

if (order\_count < MAX\_ORDERS)

{

// add current rider to the possible riders array.

possible\_riders.Add(all\_riders[i]);

j++;

}

// j is used to limit the number of rider that are going to be returned.

if (j >= 10)

break;

}

}

// return the possible riders.

return possible\_riders;

}

// get the shortest path/sequence of deliveries for a rider

public static List<Location> GetShortestRoute(driver rider, orders\_processed new\_order = null)

{

// empty list for all the locations to be searched.

List<Location> locations = new List<Location>();

// empty list to keep the sequence of locations for the shortest path.

List<Location> path\_sequence = new List<Location>();

// index variable for path\_sequence

int i = 0;

// current location of this rider. (format => lat, long)

double[] ref\_location = new double[2] { (double)rider.current\_latitude, (double)rider.current\_longitude };

// get all the current orders of this rider.

// var current\_orders = db.orders\_processed.Where(w => !w.delivery\_status.Equals("FINISHED")).ToList();

List<orders\_processed> current\_orders = UtilityClass.GetUnfinishedOrdersForRider(rider.driver\_id);

// if there is any new order, add it to the list of current orders

if (new\_order != null)

{

current\_orders.Add(new\_order);

}

// add all the pickups of the orders to the locations list

foreach (var item in current\_orders)

{

locations.Add(new Location()

{

orderId = item.order\_id,

locationType = 'P',

location = new double[2] { (double)item.hub\_latitude, (double)item.hub\_longitude }

});

}

// loop on until location array is empty

while (locations.Count != 0)

{

// find distance from the reference point for every location.

foreach (var item in locations)

{

item.temp\_distance = Coordinates.distance(ref\_location[0], item.location[0], ref\_location[1], item.location[1]);

}

// get the location with the minimum location from the reference point.

var min\_location = locations.OrderBy(o => o.temp\_distance).FirstOrDefault();

if (min\_location != null)

{

// if the last minimum location is a pickup location, then add its respective delivery location.

if (min\_location.locationType == 'P')

{

// get the respective delivery location data.

var order\_delivery = current\_orders.FirstOrDefault(f => f.order\_id == min\_location.orderId);

// add the delivery location data to the locations array to be evaluated.

locations.Add(new Location()

{

orderId = order\_delivery.order\_id,

locationType = 'D',

location = new double[2] { Convert.ToDouble(order\_delivery.store\_latitude),

Convert.ToDouble(order\_delivery.hub\_longitude) }

});

}

// change the reference location to the last minimum location.

ref\_location = min\_location.location;

// add the current minimum location with its serial number to the path\_sequence array.

min\_location.index = i;

path\_sequence.Add(min\_location);

// remove the current minimum from locations array.

locations.Remove(min\_location);

// increment index variable.

i++;

}

}

return path\_sequence;

}

// get the fitness value for a rider.

private static double GetFitnessValue(short driver\_rating, double shortestRouteLength)

{

// we are going to minimize the following function.

// f = total route length / rider rating

// -- where total route length = current route lenght of all orders +

// route lenght of pickup +

// route length of delivery location.

return (shortestRouteLength / driver\_rating);

}

}

}

**DATABASE QUERIES:**

-- formula to convert degrees to radians

-- deg \* π/180

-- query to find points within a distance from the given points.

-- -- this query accepts inputs in radians

-- -- reference points = (lat, lon) =(1.3963 rad, -0.6981 rad)

-- -- Lat and Lon are the column names

-- SELECT \* FROM Places WHERE acos(sin(1.3963) \* sin(Lat) + cos(1.3963) \* cos(Lat) \* cos(Lon - (-0.6981))) \* 6371 <= 1000;

DECLARE @ref\_lat AS float = -30.0374145507813 \* (PI()/180)

DECLARE @ref\_long AS float = -51.2035217285156 \* (PI()/180)

SELECT \* FROM drivers

WHERE acos(sin(@ref\_lat) \* sin(current\_latitude \* (PI()/180)) +

cos(@ref\_lat) \* cos(current\_latitude \* (PI()/180)) \* cos(current\_longitude \* (PI()/180) - (@ref\_long))) \* 6371 <= 5;

--select \* from drivers

-- create stored procedure for getting the riders

CREATE PROCEDURE GetRidersIn5KmRadius

-- Add the parameters for the stored procedure here

@\_lat AS float = -30.0374145507813,

@\_long AS float = -51.2035217285156

AS

BEGIN

-- SET NOCOUNT ON added to prevent extra result sets from

-- interfering with SELECT statements.

SET NOCOUNT ON;

DECLARE @ref\_lat AS float = @\_lat \* (PI()/180);

DECLARE @ref\_long AS float = @\_long \* (PI()/180);

-- Insert statements for procedure here

SELECT \* FROM drivers

WHERE acos(sin(@ref\_lat) \* sin(current\_latitude \* (PI()/180)) +

cos(@ref\_lat) \* cos(current\_latitude \* (PI()/180)) \* cos(current\_longitude \* (PI()/180) - (@ref\_long))) \* 6371 <= 5;

END

-- query to call the stored procedure

EXEC GetRidersIn5KmRadius @\_lat = -30.0374145507813, @\_long = -51.2035217285156;

-- query to get riders with another hub

EXEC GetRidersIn5KmRadius @\_lat = -23.592004776001, @\_long = -46.6365051269531;

-- for all orders before now

select \* from orders where order\_created\_day <= 07 and order\_created\_hour <= 12 and order\_created\_minute <= 34

-- for all orders of today till now

select \* from QryOrders

where order\_created\_day = 13 and

order\_created\_hour <= 22 and

not (order\_created\_hour = 22 and order\_created\_minute > 45)

-- for all orders of last hour

select \* from orders where order\_created\_day = 07 and order\_created\_hour = 12 and order\_created\_minute <= 34

-- Query for View (QryOrders)

SELECT dbo.orders.order\_id, dbo.orders.store\_id, dbo.stores.store\_name, dbo.stores.store\_latitude, dbo.stores.store\_longitude, dbo.stores.hub\_id, dbo.hubs.hub\_name, dbo.hubs.hub\_city, dbo.hubs.hub\_state, dbo.hubs.hub\_latitude,

dbo.hubs.hub\_longitude, dbo.deliveries.delivery\_id, dbo.deliveries.delivery\_distance\_meters, dbo.deliveries.delivery\_status, dbo.deliveries.driver\_id, dbo.drivers.current\_latitude, dbo.drivers.current\_longitude,

dbo.drivers.driver\_rating, dbo.drivers.current\_orders, dbo.orders.order\_status, dbo.orders.order\_created\_hour, dbo.orders.order\_created\_minute, dbo.orders.order\_created\_day, dbo.orders.order\_created\_month,

dbo.orders.order\_created\_year, dbo.orders.order\_moment\_created, dbo.orders.order\_moment\_accepted, dbo.orders.order\_moment\_ready, dbo.orders.order\_moment\_in\_expedition, dbo.orders.order\_moment\_collected,

dbo.orders.order\_moment\_delivering, dbo.orders.order\_moment\_delivered, dbo.orders.order\_moment\_finished, dbo.orders.order\_metric\_collected\_time, dbo.orders.order\_metric\_paused\_time,

dbo.orders.order\_metric\_production\_time, dbo.orders.order\_metric\_walking\_time, dbo.orders.order\_metric\_expediton\_speed\_time, dbo.orders.order\_metric\_transit\_time, dbo.orders.order\_metric\_cycle\_time

FROM dbo.orders INNER JOIN

dbo.stores ON dbo.orders.store\_id = dbo.stores.store\_id INNER JOIN

dbo.drivers INNER JOIN

dbo.deliveries ON dbo.drivers.driver\_id = dbo.deliveries.driver\_id ON dbo.orders.order\_id = dbo.deliveries.delivery\_order\_id INNER JOIN

dbo.hubs ON dbo.stores.hub\_id = dbo.hubs.hub\_id

-- current query to get orders of the last hour.

select \* from orders\_processed where

order\_created\_month = 1 and order\_created\_day = 2 and

((order\_created\_hour = 21 and not (order\_created\_minute > 30)) or

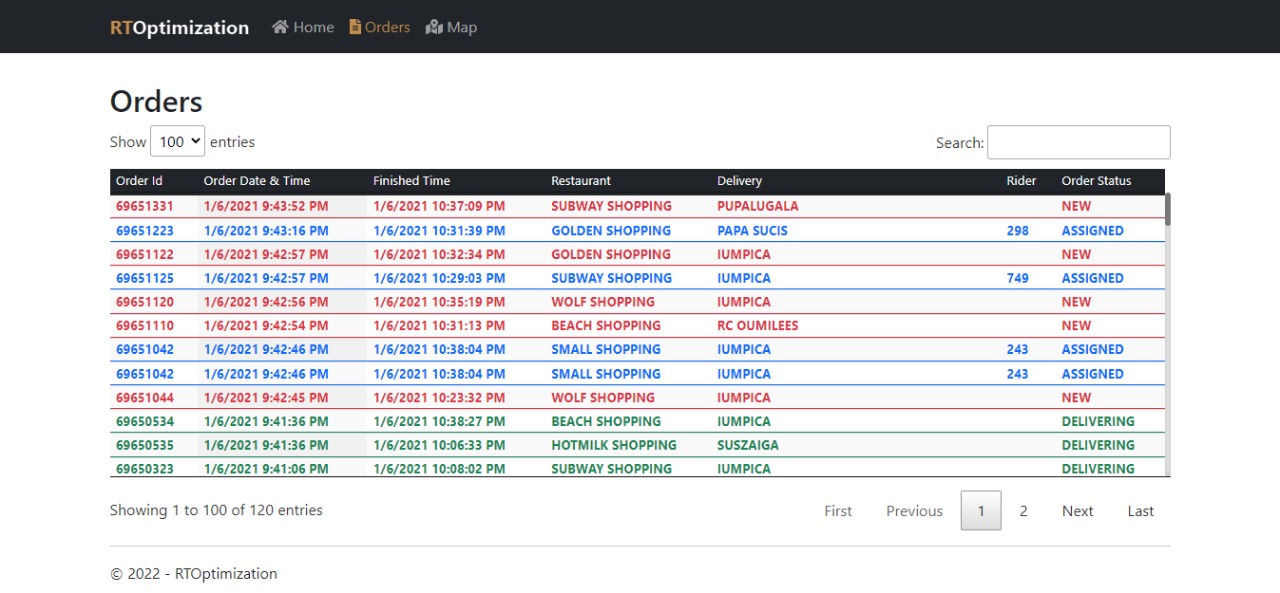
(order\_created\_hour = 20 and not (order\_created\_minute < 30))) and

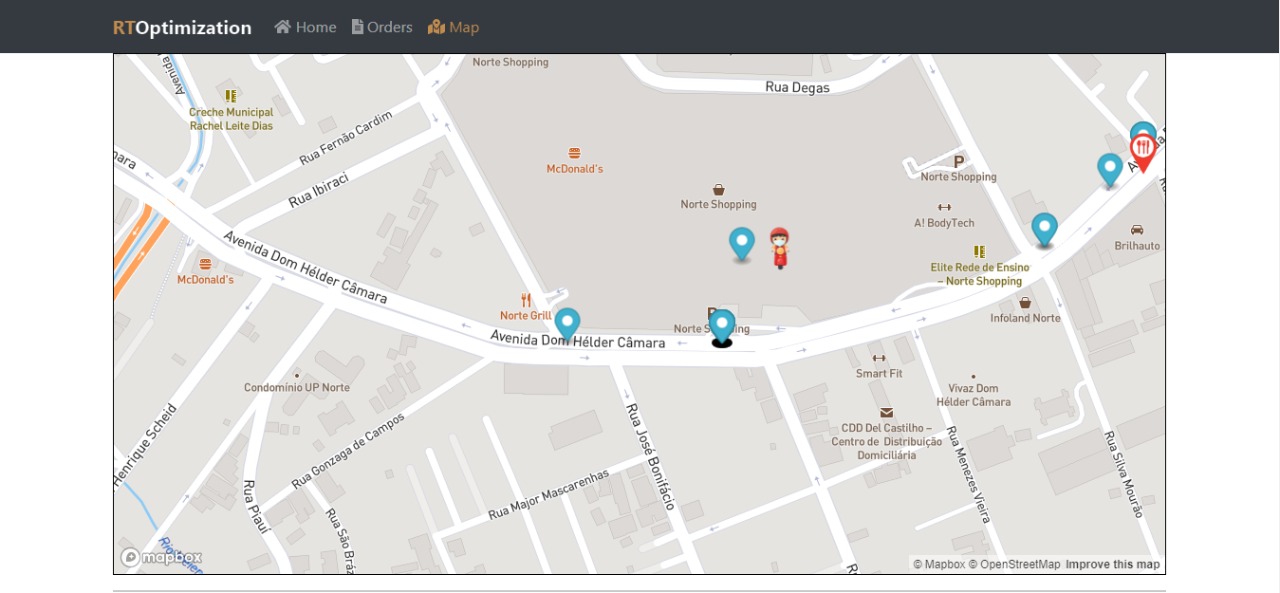
not order\_moment\_created = ('') and

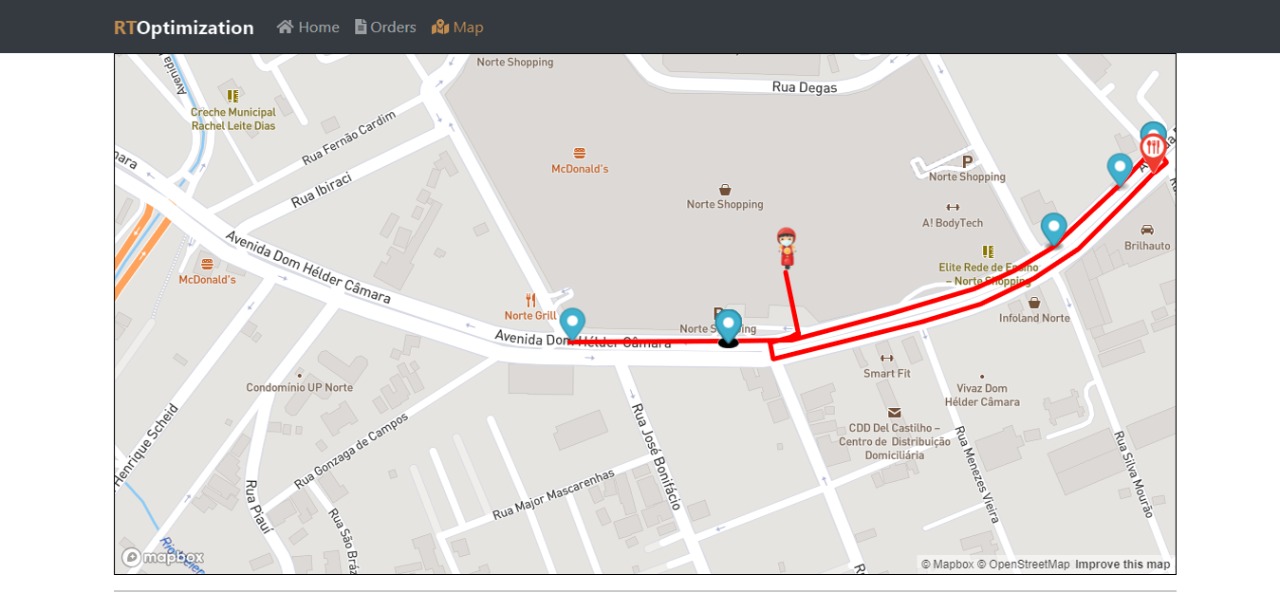
not order\_moment\_accepted = ('') and

not order\_moment\_ready = ('')

**SCREEN SHOTS:**

****

****

****

**SUMMARY:**

This Chapter consist of system prototype and development in which we explained about the frontend and backend design some database queries and some screenshot of our application. We have also attach details of algorithm step by step, attached screen shot of front-end, queries of database and back-end. The database queries are also provided to show how the system retrieve the data while functioning.