Visualizer Instructions

August 11, 2018

**1. Steps to run the visualizer:**

• Configure the required running environment: GTFSinstructions.txt tcan be followed to install the PostgreSQL and Ruby environment:

**– Install PostgreSQL**

$ /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"

$ brew install postgresql

$ pg\_ctl -D /usr/local/var/postgres start && brew services start postgresql

$ postgres -V

$ brew install postgis

**– Install Ruby**

$ \curl -sSL https://get.rvm.io | bash -s stable --ruby

$ source /User/iniesdu/.rvm/scripts/rvm

$ ruby -v

$ gem install pg

• Generate the .jsonfiles: Place all the files in the jsonGenBatch folder into the directory which contains the original data (in .txt form). Then, run the ruby scripts by following commands in terminal:

$ ruby busJsonGen.rb

$ ruby cabJsonGen.rb

$ ruby carJsonGen.rb

$ ruby userJsonGen.rb

After running those commands, these .json files are created:

**–** routes\_bus.json

**–** DyPosition\_bus.json

**–** DyPosition\_cab.json

**–** DyPosition\_car.json

**–** DyPosition\_user.json

• Run the visualizer app: Place all the obtained .json files into the json folder. Run the canvas.html.

**2. Implementation of the data transformation:**

For the visualizer, we need to convert the original data from a .txt format into some useful and convenient formats, such as Json.

Before going to the implementation of those ruby scripts, we should first look at the data structure of the .json file.

**2.1 Data structure of the .json files**

Based on their functionalities, the five .json files can be separated into two categories: static routes (routes\_bus.json) and dynamic positions (DyPosition\_bus.json/DyPosition\_cab.json/ DyPosition\_car.json/DyPosition\_user.json). They also have different data structures.

• **static routes:** Note that in a bus system, all the buses should have very specific stops and those stops shape the fixed routes for each bus. In this sense, all the bus routes should be static (independent with time), so is the data structure of routes\_bus.json. In fact, the data structure of routes\_bus.json can be regarded as a **FeatureCollection** which collects some features. Each **Feature**, corresponding to a bus route, contains some **properties** (provide information like route ID, route name, route color, and etc) and a **geometry** (provides the coordinates of the route).

• **dynamic positions:** Different from the static bus routes, there are no specific stops or fixed routes for a cab, a private car or a user. Therefore, in order to obtain the routes for those vehicles, we need to access the dynamic positions and use the discrete sequences to shape a route for cab, car or user. In this sense, the data structure of those dynamic positions should be time-dependent. In fact, each DyPosition.json generally contains two variables: **start\_time** and **positions**. While **start\_time** (which is set as 180 by default) defines the starting time of the following position sequence, **positions** provides the exact coordinates of each active vehicle at that time step. Since the positions are defined for every minute and the visualize simulates an entire day, we should have 1440 (24\*60) entries for **positions** starting from **start\_time**. Furthermore, to show a running bus, we also need the dynamic positions for buses.

**2.2 Implementation of ruby scripts**

Once we have the data structure of those .json files defined, we can use a ruby script to convert the data from the original .txt format into a .json format. **The scripts are already written in the jsonGenBatch folder; to work with a new .txt dataset, all you have to do is edit some of the scripts**.

No matter for bus, cab, private car or user, in order to do the data transformation via ruby script, the very first thing we need to do is always to create databases and also import the data into created databases. Since these steps are pretty much the same for all vehicles, we only use buses as an example here.

• **create database:** By running the following ruby scripts, create a database (named as

database\_name) with the owner (named as owner\_name).

#define the owner\_name and database\_name owner\_name = "gtfs\_t"

database\_name = "bus\_db\_t"

#drop the database if already exists

cmd\_drop\_db\_if = "psql -c \"DROP DATABASE IF EXISTS \"databaseName\";\"" cmd\_drop\_db\_if["databaseName"] = database\_name

value = `#{cmd\_drop\_db\_if}`

#drop the role if already exists

cmd\_drop\_role\_if = "psql -c \"DROP ROLE IF EXISTS \"ownerName\";\"" cmd\_drop\_role\_if["ownerName"] = owner\_name

value = `#{cmd\_drop\_role\_if}`

#create role named as owner\_name with login

cmd\_create\_role = "psql -c \"CREATE ROLE \"ownerName\" WITH LOGIN;\"" cmd\_create\_role["ownerName"] = owner\_name

value = `#{cmd\_create\_role}`

puts "\nrole created\n"

#create database named as database\_name with owner owner\_name

cmd\_create\_db = "psql -c \"CREATE DATABASE \"databaseName\" OWNER \"ownerName\";\"";

cmd\_create\_db["ownerName"] = owner\_name cmd\_create\_db["databaseName"] = database\_name value = `#{cmd\_create\_db}`

puts "\ndatabase created\n"

#alter user with superuser

cmd\_alter\_superuser = "psql -c \"ALTER USER \"ownerName\" WITH SUPERUSER;\"" cmd\_alter\_superuser["ownerName"] = owner\_name

value = `#{cmd\_alter\_superuser}`

puts "\nsuperuser altered\n"

• **create tables:** The SQL for creating tables is defined in the separate file createTables\_bus.sql.

**Note that in order to have all the columns in the .txt files and tables matched, you may**

**need to modify the SQL in this file when applying a new dataset.**

#connect to the created database

conn = PG.connect :dbname => database\_name, :user => owner\_name

#get the current directory dirname = Dir.pwd

#run the sql in the "createTables\_bus.sql" file create\_tables = "" File.open(dirname+"/createTables\_bus.sql", "r") do |f|

f.each\_line do |line|

create\_tables += line end

conn.exec(create\_tables)

puts "\ntables created\n"

• **import data:** The SQL for importing data is defined in the separated file importData\_bus.sql.

#run the sql in the "createTables\_bus.sql" file import\_data = "" File.open(dirname+"/importData\_bus.sql", "r") do |f|

f.each\_line do |line|

line["dir"] = dirname import\_data += line

end end conn.exec(import\_data) puts "\ndata imported\n"

• **process the data:** Since steps to process the data for all vehicles are different, each is shown one by one.

**–** routes\_bus.json: As mentioned before, the data structure of routes\_bus.json is time independent.

Also, thanks to the GTFS data format, all the bus routes are already defined in the table shapes

(shapes.txt file). All we need to do is to find all the routes in the table shapes which have the

route\_ids that match the ones in the table trips. This is implemented by the SQL in

processDara\_bus.sql.

#run the sql in the "processData\_bus.sql" file process\_data = "" File.open(dirname+"/processData\_bus.sql", "r") do |f|

f.each\_line do |line|

process\_data += line end

end conn.exec(process\_data) puts "\ndata processed\n"

**–** DyPosition\_bus.json: To find a dynamic position of a bus, we consider two separated cases.

First, when a bus is at a stop at a given time step, we can easily tell the position of the bus

by querying the table stop\_times and stops. This is implemented by the SQL in exact\_routes.sql. Second, when a bus is not at stop at a given time step, it must be between two stops. Then we can use simple linear interpolation to calculate the position of the bus. This procedure is implemented by the SQL in interpolated\_routes.sql. Next, all we need to do is just to consider these two cases for every minute of the entire day. Note that the start\_time is set as 180 by default, so the time starts from 3:00 AM for the day. You may want to refer to the Pittsburgh Medium article for more detailed information.

#run the sql in the "exact\_routes.sql" file exact\_routes = "" File.open(dirname+"/exact\_routes.sql", "r") do |f|

f.each\_line do |line|

exact\_routes += line end

end

#run the sql in the "interpolated\_routes.sql" file interpolated\_routes = "" File.open(dirname+"/interpolated\_routes.sql", "r") do |f|

f.each\_line do |line|

interpolated\_routes += line end

end

#query every minute, starting from 3:00 AM by default positions = []

0.upto(23) do |i|

h = (i + 3) % 24

0.upto(59) do |m|

t = "#{ h.to\_s.rjust(2, '0') }:#{ m.to\_s.rjust(2, '0') }" p = conn.exec(exact\_routes.gsub('09:00',t)).values

p += conn.exec(interpolated\_routes.gsub('09:00',t)).values positions << p

end end

**–** DyPosition\_cab.json Since the cab information (in cab\_state, cab\_loc, cab\_busy) is only given for registered cabs (without IDs, but following lexicographic order), one can no longer access this information by the IDs directly. So the first thing we may need to do is decide the registered interval for each cab in the system.

#obtain the time step when a cab just registered

sql = "select t, cab\_id from cab\_list where status = 1 order by cab\_id asc;" p = conn.exec(sql).values

cab\_reg = []

p.each do |i|

cab\_reg << [i[1],i[0]]

end

#obtain the time step when a cab just exit

sql = "select t, cab\_id from cab\_list where status = 0 order by cab\_id asc;" p = conn.exec(sql).values

p.each do |i|

cab\_reg.each\_with\_index do |j, index|

if i[1] == j[0]

cab\_reg[index][1] += "." + i[0]

end end

end

Next, we need to do find the min and max time steps for the entire system. Also, note that different from the time steps for buses which are defined by exact time, the time steps for cabs are defined by integers (starting from 0). We need to also convert these integers into exact time, by assuming starting from 8:30 AM (0 means 8:30 AM) by default.

#obtain the min and max time steps for the cab system

time\_min = conn.exec("select min(t) from cab\_list").values.join.to\_i time\_max = conn.exec("select max(t) from cab\_list").values.join.to\_i

#set the time offset 330, meaning time starts from 8:30 AM

time\_offset = 330

Now, we can define the state for each cab (0: unregistered; 1: registered, busy&unmoved; 2: registered, busy&moved; 3: registered, idle&moved; 4: registered, idle&unmoved), and assign the locations (coordinates and distance) and finance (revenue and cost) when they are updated or keep unchanged.

for t0 in 0..1439 cab\_infos = []

if t0 - time\_offset >= time\_min && t0 - time\_offset < time\_max t = t0 - time\_offset

cab\_index = 0

cab\_reg.each\_with\_index do |cab, index|

cab\_info = []

if cab[1].split('.')[0].to\_i > t || cab[1].split('.')[1].to\_i <= t

#unregistered else

#registered cab\_info[0] = cab[0]

sql = "select state, moved from cab\_state where t = time".gsub('time', "#{t}")

p = conn.exec(sql).values

sql = "select lat, lon, distance from cab\_loc where t = time".gsub('time', "#{t}")

loc = conn.exec(sql).values

sql = "select revenue, cost from cab\_busy where t = time".gsub('time', "#{t}")

busy = conn.exec(sql).values

if p[0][0].split(//)[cab\_index] == '1' && p[0][1].split(//)[cab\_index] == '0'

#registered, busy and unmoved cab\_info[1] = "1"

cab\_info += loc[cab\_index] cab\_info << busy[cab\_index][1] cab\_info << busy[cab\_index][0]

elsif p[0][0].split(//)[cab\_index] == '1' && p[0][1].split(//)[cab\_index] == '1'

#registered, busy and moved cab\_info[1] = "2"

cab\_info += loc[cab\_index]

cab\_info << busy[cab\_index][1]

cab\_info << cab\_list[t0-1][cab\_index][6]

elsif p[0][0].split(//)[cab\_index] == '0' && p[0][1].split(//)[cab\_index] == '1'

#registered, idle and moved cab\_info[1] = "3"

cab\_info += loc[cab\_index]

cab\_info << busy[cab\_index][1]

cab\_info << cab\_list[t0-1][cab\_index][6]

else

#registered, idle and unmoved cab\_info[1] = "4"

cab\_info += cab\_list[t0-1][cab\_index][2..4]

cab\_info << busy[cab\_index][1]

cab\_info << cab\_list[t0-1][cab\_index][6]

end

cab\_index += 1 cab\_infos << cab\_info

end end

end

cab\_list << cab\_infos end

**–** DyPosition\_car.json: As same as what we did for the cabs, we first find the min and max time steps, and convert the integer time steps into exact times.

#obtain the min and max time steps for the car system

time\_min = conn.exec("select min(t) from car\_state").values.join.to\_i time\_max = conn.exec("select max(t) from car\_state").values.join.to\_i

time\_offset = 330

Notice that the data for private cars is provided quite simply. Since there is no car IDs defined in the system, it seems to always have only one private car. So, all we need to do next is just to query the car\_locto obtain the locations of cars for every minute. Note that this is implemented by assuming simple case. When it comes to the case with multiple private cars, you can generate the DyPosition\_car.json by following the steps for cabs only with some trivial modifications.

for t0 in 0..1439 car\_infos = []

if t0 - time\_offset >= time\_min && t0 - time\_offset <= time\_max t = t0 - time\_offset

sql = "select lat, lon, distance from car\_loc where t = time;".gsub('time', "#{t}");

car\_loc = conn.exec(sql).values car\_infos = car\_loc

end

car\_list << car\_infos end

**–** DyPostion\_user.json Again, we first decide the registered interval for each user in the system as before.

#obtain the time step when an user just registered

sql = "select t, user\_id from user\_list where status = 1 order by user\_id asc;" p = conn.exec(sql).values

user\_reg = []

p.each do |i|

user\_reg << [i[1],i[0]]

end

#obtain the time step when an user just exit

sql = "select t, user\_id from user\_list where status = 0 order by user\_id asc;" p = conn.exec(sql).values

p.each do |i|

user\_reg.each\_with\_index do |j, index|

if i[1] == j[0]

user\_reg[index][1] += "." + i[0]

end end

end

Also obtain the min and max time steps.

#obtain the min and max time steps for the user system

time\_min = conn.exec("select min(t) from user\_list").values.join.to\_i time\_max = conn.exec("select max(t) from user\_list").values.join.to\_i

time\_offset = 330

Similar as what we did for cabs, we now define the state for each user (0: unregistered; ”bus.bus\_id”: inside the bus with bus\_id; ”cab.cab\_id”: inside the cab with cab\_id; ”wb”: waiting before confirmation; ”wa”: waiting after confirmation; ”wk”: walking; ”pc”: inside the private car). Since the location information for a bus, a cab or an user has been defined in the previous .json files, we can easily obtain this information by matching the time steps when an user has the state

”bus”/”cab”/”pc”. Also, when an user’s state is ”wb” or ”wa”, we can keep the user’s location as unchanged as the previous time step. The only thing we need to do here is just to query the user\_loc when the user has a state ”wk”. For the utility information, we can use a same approach as what we did for cabs.

for t0 in 0..1439 user\_infos = []

if t0 - time\_offset >= time\_min && t0 - time\_offset < time\_max t = t0 - time\_offset

user\_index = 0

user\_reg.each\_with\_index do |user, index|

user\_info = []

if user[1].split('.')[0].to\_i > t || user[1].split('.')[1].to\_i <= t

#unregistered else

#registered user\_info[0] = user[0]

sql\_update = "select state from user\_update where t = time;".gsub('time', "#{t}")

user\_state\_update = conn.exec(sql\_update).values

if user\_state\_update[0][0].split(//)[user\_index] == "1"

#user's state updated

sql\_state = "select state from user\_state where t = time;".gsub('time', "#{t}")

user\_state = conn.exec(sql\_state).values if user\_state[user\_index][0] == "4"

#user's state is "4"

sql\_vehicle\_update = "select vehicle from user\_update where t = time;".gsub('time',"#{t}")

user\_vehicle\_update = conn.exec(sql\_vehicle\_update).values if user\_vehicle\_update[0][0].split(//)[user\_index] == "1"

#user's vehicle updated

sql\_vehicle = "select vehicle\_id from user\_vehicle where t = time;".gsub('time',"#{t}")

user\_vehicle = conn.exec(sql\_vehicle).values user\_info << user\_vehicle[user\_index][0]

else

user\_info << user\_list[t0-1][user\_index][1]

end else

if user\_state[user\_index][0] == "1" user\_info << "wb"

elsif user\_state[user\_index][0] == "2" user\_info << "wa"

elsif user\_state[user\_index][0] == "3" user\_info << "wk"

###### query the walking locations ######

else

user\_info << "pc" end

end else

user\_info << user\_list[t0-1][user\_index][1]

end

sql\_utility\_update = "select utility from user\_update where t = time;".gsub('time', "#{t}")

user\_utility\_update = conn.exec(sql\_utility\_update).values

if user\_utility\_update[0][0].split(//)[user\_index] == "1"

#user's utility updated

sql\_utility = "select utility from user\_utility where t = time;".gsub('time',"#{t}")

user\_utility = conn.exec(sql\_utility).values user\_info << user\_utility[user\_index][0]

else

user\_info << user\_list[t0-1][user\_index][2]

end

user\_index += 1 user\_infos << user\_info

end end

end

user\_list << user\_infos end

• **export the data:**

**–** routes\_bus.json: Export the routes\_bus.json as a **FeatureCollection**.

cmd = "ogr2ogr -f GeoJSON routes\_bus.json \"PG:host=localhost dbname=databaseName user=ownerName\"

-sql 'select route\_id, shape\_id, geom from gtfs\_shape\_lines;'" cmd["databaseName"] = database\_name

cmd["ownerName"] = owner\_name value = `#{cmd}`

puts "\nroutes.json generated\n"

**–** DyPosition\_bus.json: Export the DyPosition\_bus.jsonwith the **key** defined as trip\_id.route\_id

and the content defined as lon, lat.

positions.map! do |p|

t = {}

p.each do |trip|

t[ "#{ trip[0] }.#{ trip[1] }" ] = [ trip[3].to\_f, trip[2].to\_f ]

end t

end

File.open('DyPosition\_bus.json','wb') do |f|

f.write( { start\_time: 180, positions: positions } .to\_json)

end

puts "\nDyPosition\_bus.json generated\n"

**–** DyPosition\_cab.json: Export the DyPosition\_cab.json with the **key** defined as cab\_id and the content defined as lon, lat, distance, status, cost, revenue.

cab\_list.map! do |p|

t = {}

p.each do |trip|

t[ trip[0] ] = [ trip[2].to\_f, trip[3].to\_f, trip[4].to\_f, trip[1].to\_i, trip[5].to\_f, trip[6].to\_i ]

end t

end

File.open('DyPosition\_cab.json','wb') do |f|

f.write( { start\_time: 180, positions: cab\_list } .to\_json)

end

puts "DyPosition\_cab.json Generated"

**–** DyPosition\_car.json: Export the DyPosition\_car.json with the **key** defined as pc and the content defined as lon, lat, distance.

car\_list.map! do |p|

t = {}

p.each do |trip|

t["pc"] = [ trip[0].to\_f, trip[1].to\_f, trip[2].to\_f]

end t

end

File.open('DyPosition\_car.json','wb') do |f|

f.write( { start\_time: 180, positions: car\_list } .to\_json)

end

puts "DyPosition\_car.json Generated"

**–** DyPosition\_user.json: Export the DyPosition\_user.json with the **key** defined as user\_id

and the content defined as status, utility.

user\_list.map! do |p|

t = {}

p.each do |trip|

if trip[1] == "wk"

t[ "u" + trip[0]] = [trip[1]+"."+trip[1], trip[2].to\_i, trip[3].to\_f, trip[4].to\_f]

elsif trip[1].split(//)[0] == "c"

t[ "u" + trip[0]] = ["cab."+trip[1], trip[2].to\_i]

elsif trip[1].split(//)[0] == "b"

t[ "u" + trip[0]] = ["bus."+trip[1], trip[2].to\_i]

else

t[ "u" + trip[0]] = [trip[1]+"."+trip[1], trip[2].to\_i]

end end

t end

File.open('DyPosition\_user.json','wb') do |f|

f.write( { start\_time: 180, positions: user\_list } .to\_json)

end

puts "DyPosition\_user.json Generated"

**3. Implementation of the visualizer:**

Once having all the .json files generated, we can now implement the visualizer via html and javascript. In general, this visualizer can be separated into several layers, including a **tile layer**, a **svg layer** and a **canvas layer**. While the tile layer defines a basic map of the city, svg layer draws routes of all the vehicles and canvas layer draws dynamic locations of all the vehicles. In this sense, the implementation of visualizer can be also documented as the implementation of these three layers.

**3.1 Tile Layer**

The tile layer is quite easy and implemented by using a Google street map by default.

//initialize the map setting, modify the center coordinates when applying a new dataset var map = new L.Map('map', {

center: [42.45267, -76.49766], zoom: 13

});

window.map = map;

//initialize the tilelayer, a street map is shown by default

var tileLayer = new L.tileLayer('http://{s}.google.com/vt/lyrs=m&x={x}&y={y}&z={z}',{

maxZoom: 20, subdomains:['mt0','mt1','mt2','mt3']

});

window.tileLayer = tileLayer; tileLayer.setOpacity(0.6); map.addLayer(tileLayer);

We can also switch the tile layer between a street map and a satellite map via a selection box.

//monitor the change event of map selection (satellite map vs street map)

$(document).on('change', '.current-map', function(e){

var map\_type\_sel = $('#map\_selected').val();

map.removeLayer(tileLayer);

if (map\_type\_sel == "satellite")

window.tileLayer = L.tileLayer('http://{s}.google.com/vt/lyrs=s&x={x}&y={y}&z={z}',{

maxZoom: 20, subdomains:['mt0','mt1','mt2','mt3']

});

else

window.tileLayer = L.tileLayer('http://{s}.google.com/vt/lyrs=m&x={x}&y={y}&z={z}',{

maxZoom: 20, subdomains:['mt0','mt1','mt2','mt3']

}); window.tileLayer.setOpacity(0.6); map.addLayer(tileLayer);

});

Note that the center coordinates may need to be modified manually when applying a new dataset.

**3.2 SVG Layer**

The svg layer draws all the routes for buses, cabs, private cars and users. Since the configuration of the

SVG layers for all vehicles are quite similar, here we only take a cab layer as an example.

//svg layer of cab routes function svgCab(cab\_routes) {

var svg = L.d3SvgOverlay(function(selection, projection) { var routes\_g = selection.selectAll('.routes'); if(routes\_g.empty()){

routes\_g = selection.append('g')

.attr('class','routes')

var route\_paths = routes\_g.selectAll('path')

.data(cab\_routes.features); ##use different variable (cab\_routes) for different routes

route\_paths.enter().append('path')

.attr('class', 'route')

.attr('d', d3.geo.path()

.projection(function(l){

var p = projection.latLngToLayerPoint({ lon: l[0], lat: l[1] });

return [p.x,p.y];

})

)

.attr('stroke', function(d){ var color = "#fa0000" return color;

}

});

});

route\_paths.attr('stroke-width', 4.5);

return svg;

}

The hard part here is the update of variables (cab\_routes for instance) for different routes. For bus routes, since it is static and pre-defined, the variable bus\_routes is fixed and read from routes\_bus.json. For cab routes or private car routes, we can no longer use the pre-defined routes as for buses. Instead, we need to update the variable cab\_routes/car\_routestimely, based on the dynamic positions of cabs/cars. Therefore, we need to always update cab\_routes(or car\_routes) immediately once obtaining a new location of cab or car.

//update the cab routes, for dynamically showing the cab routes var fea\_index = parseInt(t.split("")[1]);

if (!cab\_routes.features[fea\_index]) {

var cab\_feature = { "type": "Feature",

"geometry": {"type": "LineString", "coordinates": []}

};

cab\_routes.features[fea\_index] = cab\_feature;

}

cab\_routes.features[fea\_index].geometry.coordinates.push([cab\_d[1],cab\_d[0]]);

For user\_routes, basically we need to do a same step as for cabs and cars. However, obtaining a location when the user is inside a bus/cab/car needs to query the bus\_routes/cab\_routes/car\_routes. The step gets more complicated, and depends on the user’s current status.

// assign the location information according to user's status var vehicle\_id = [];

var status\_color = [];

if (user\_status == "cab") {

vehicle\_id = user\_trips\_t1[t][0].split(".")[1]; user\_dot = cab\_dots[vehicle\_id]; user\_routes.features[fea\_index].geometry.coordinates.push([cab\_ds[vehicle\_id][1],cab\_ds[vehicle\_id][0]]); status\_color = "#faeb00";

}

else if (user\_status == "bus") {

status\_color = "#0057fa";

}

else if (user\_status == "wa") {

user\_dot = user\_dots[t];

vehicle\_id = "waiting after confirmation" status\_color = "#ffffff";

}

else if (user\_status == "wb") {

status\_color = "#ffffff";

}

else if (user\_status == "wk") {

#using the locations in user\_trips status\_color = "#111";

}

else if (user\_status == "pc") {

vehicle\_id = user\_trips\_t1[t][0].split(".")[1]; user\_dot = car\_dots[vehicle\_id]; user\_routes.features[fea\_index].geometry.coordinates.push([car\_ds[vehicle\_id][1],car\_ds[vehicle\_id][0]]); vehicle\_id = "private car";

status\_color = "#feab00";

}

Note that the ”walking” section still needs to be implemented, when the data for walking case is provided.

**3.3 Canvas Layer**

Different from the svg layer, the canvas layer shows the dynamic points of all the vehicles. Since we only show the currently moving points, we do not need to memorize any locations like what we did for the svg layer. However the use of canvas introduces another complication: the animation will run too fast. (See the Medium post for more information). Here, we add 10 more interpolates within each minute. To make sure the interpolate is calculated properly between every two minutes, we use one more condition: only if the vehicle is still active at next time step, we show it in the canvas. Here, we take the canvas layer for cabs as an example.

//obtain the active cab trips

var cab\_trips\_t1 = data\_cab.positions[ times[current\_index] ], cab\_trips\_t2 = data\_cab.positions[ times[current\_index + 1]];

if (cab\_trips\_t1 && cab\_trips\_t2){

var cab\_trip\_keys = Object.keys(cab\_trips\_t1);

cab\_trip\_keys.forEach(function(t){

if(cab\_trips\_t2[t]){

var cab\_d = interpolate( cab\_trips\_t1[t], cab\_trips\_t2[t], f), cab\_dot = canvasOverlay.\_map.latLngToContainerPoint(cab\_d);

if (layer\_type\_sel == "all" || layer\_type\_sel == "cab") {

context.fillStyle = "#faeb00";

context.beginPath();

context.arc(cab\_dot.x, cab\_dot.y, window.params.zoom - 9 + 3, 0, Math.PI \* 2);

context.lineWidth = 4; context.fill(); context.strokeStyle = '#FFFFFF'; context.stroke();

if (parseInt(cab\_trips\_t2[t][3]) <= 2)

context.strokeStyle = '#f1270b';

else

context.strokeStyle = '#b3cf3c'; context.lineWidth = 4; context.beginPath();

context.arc(cab\_dot.x,cab\_dot.y,window.params.zoom - 9 + 3, -0.5\*Math.PI, 1.5\*Math.PI, false);

context.stroke(); context.fillStyle = "#111"; context.fillText(t.split('.')[0],cab\_dot.x+6,cab\_dot.y-6); context.closePath();

}

}

});

}

Note that among those three layers, the tile layer is static and only needs to be executed once (in the script.js). The svg layer and canvas layer show the dynamic routes and circles, so they are placed in the draw() function (in the draw.js) which is executed 10 times for every simulated minute.

In addition, to implement some functionalities, such as vehicle selection, cab click, user dashboard and

etc, we put some UI elements on the website and also add the corresponding events.