Prométhée Spathis and Benjamin Baron

Visualization of transit systems in Java

Date: February 9, 2015

1 | Goals

The objective of this JAVA project is to create a program to visualize the instant positions of the vehicles in service in a transit system. An example of such visualization can be found here. Moreover, your program should exhibit features to help better understand transportation systems, how riders use public transports, and how they interact with each other.

The project is guided along four separate parts we will detail in the following. Note that, the document contains a lot of helpful (*hopefully?*) embedded links clickable directly from the PDF file. Those links are provided to help you conduct your project successfully.

2 | Turn GTFS data into time series

2.1 About the GTFS file format

Transit data made available by public transit agencies are published using a common format called GTFS *(General Transit Feed Specification)*. This format allows developers to write applications that consume that data in an interoperable way. The GTFS information combines the public transportation schedules and associated geographic information. Examples of public transit agencies who provide public GTFS feeds include MTA in New York (USA), RATP in Paris (France), or Chicago Transit Authority (USA).

Select a transit agency and get the corresponding GTFS files. (*Careful, following the links below will trigger the zipped files download!*)

New York, USA	MTA	Click here
Dublin, Ireland	Dublin Bus	Click here
Chicago, USA	Chicago Transit Authority	Click here

The format of a GTFS feed file contains the following CSV files of interest for this project:

`agency.txt` One or more transit agencies that provide the data in this feed.
Joins `route` table on `agency_id`.

`routes.txt` Transit routes. A route is a group of trips that are displayed to riders as a single service.
Joins `trips` table on `route_id`.

`trips.txt` Trips for each route. A trip is a sequence of two or more stops that occurs at specific time.
Joins `stop_times` table on `trip_id`.

`stop_times.txt` Times that a vehicle arrives at and departs from individual stops for each trip.
Joins `stops` table on `stop_id` and `trips` table on `trip_id`.

`stops.txt` Individual locations where vehicles pick up or drop off passengers.

`calendar.txt` Dates for service IDs (`service_id`) using a weekly schedule. Specify when service starts and ends, as well as days of the week where service is available. *Joins `trips` table on `service_id`.*

`shapes.txt` Rules for drawing lines on a map to represent a transit organization's routes. The shapes represent the routes of the vehicle for a given trip, however, it is independent from the location of the stops given in the `stops.txt`.
Joins `trips` table on `shape_id`.

Content of the files provided in a GTFS feed are arranged in columns with field names. The complete list and description of the CSV files can be found in the GTFS reference document available online). The reference document also provides the definition of all the fields used in each file of a GTFS feed.

Summary. The transit system consists of routes (`routes.txt`). Each route consists of trips (`trips.txt`). The trips correspond to the trajectory of the vehicles. The vehicles stop at stops (whose locations and names are in `stops.txt`) at given times (`stop_times.txt`). The vehicles follow a trajectory (`shapes.txt`). Finally, the trips are gathered in services that share the same service dates (`calendar.txt`).

2.2 Read the CSV files

The first step will consist in reading the content from the GTFS feed files. To this end, you will need to write a function named `readCSV` that reads the content of each file and returns the field values organized in a data structure. The GTFS feed files follow the CSV (comma-separated values) format and as so, can be read using the `CSVReader` class. This class provides methods to read a CSV file line by line, converting each line in a String. All lines will have to be stored in an `ArrayList`. Each element of the resulting list will contain the values of a line stored in a `Map<String,String>`. The output of the `readCSV` function is thus a data structure of type `ArrayList<Map<String, String>>`.

The expect signature of the `readCSV` function goes as follow:

```
private static ArrayList<Map<String, String>> readCSV(InputStream csv) throws IOException
```

2.3 Manipulate the GTFS format

In a second step, you will need a set of JAVA classes to manipulate the GTFS feed files. In the following, we provide a list of classes as examples you may want to follow. Feel free to implement more or other classes as needed.

`GTFSParser` This class is intended to gather all the tools you will need to parse the GTFS feed CSV files. The purpose of those tools is to parse a CSV file and output the results as instances of the `Trajectory` and `Transit` classes. This class draws on the `readCSV` function defined in Section 2.2 as follows:

- The `GTFSParser` class implements a function `ArrayList<Trajectory> parseTrips(String tripFile)` that parses the GTFS files (in particular, `calendar.txt`, `shapes.txt`, `stop_times.txt`, and `trips.txt`) and creates as an output an `ArrayList<Trajectory>` of trajectories.

`Trajectory` This class describes the complete path a vehicle takes through time and space. The `Trajectory` class includes the following attributes and methods (the list is not exhaustive):

- The `trip_id` attribute identifies an instance of the `Trajectory` class.
- The `service_id` attribute gathers multiple trips that share the same service dates (in the `calendar.txt` CSV file).
- An attribute `trajectory` of type `SortedMap<Long, Coordinates>` consists in the sequence of timestamped positions for the corresponding trajectory.
- The method `Coordinate getPosition(Long time)` returns the position (an instance of the class `Coordinate`) of the vehicle at timestamp `time` (of instance `Long`).

- The class also implements a method `boolean isActive(Long time)` that returns whether the vehicle (using a `boolean` type) is active at timestamp `time`.

`Transit` Corresponds to the global transit system. This class features the following attributes and methods (the list is not exhaustive):

- An attribute `ArrayList<Trajectory>` containing all trajectories of the transit system.
- A method `ArrayList<Trajectory> activeTrajectories(Long time)` that returns the `ArrayList` of all active trajectories at timestamp `time`.

Coordinates. In the following, a geographic location is denoted "Coordinate" and is represented by the `Coordinate` class taken from the GeoTools JTS Java library. A coordinate consists of two main attributes: a longitude (`double x`) and a latitude (`double y`). The GeoTools library also provides the `Point` class that inherits from the `Geometry` class. The `Point` class allows to do various geometric operations (*e.g.*, buffer, distance, intersect) missing from the `Coordinate` class.

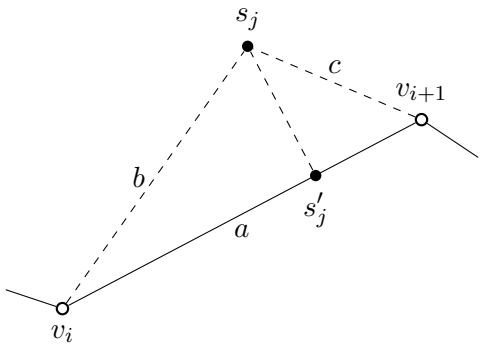
Timestamps. The timestamps are intended to represent the GTFS times. The type of a timestamp instance is a `Long`. GTFS times are provided in a `%H:%M:%S` format and thus need to be converted. We suggest the Unix time format which should ease the processing of the trajectories as instances of the `Trajectory` class. The Unix time (Unix epoch or POSIX time or Unix timestamp) is "a system for describing points in time, defined as the number of seconds elapsed since midnight (00:00:00) Coordinated Universal Time (UTC), Thursday, January 1, 1970." The timestamps of a trajectory will be abstracted as the number of seconds elapsed since `` `1970-01-01-00.00.00' ``. We also suggest you use the days of service for a given trajectory (identified by the `service_id` attribute of the `Trajectory` class. The value of this attribute is taken from the `calendar.txt` CSV file).

Therefore, you will need to implement a function `Long toElapsedTime(String time)` that turns the `String` time `%H:%M:%S` format into a `Long` number which represents the elapsed time since midnight.

Directions to implement `parseTrips()`. In the following, we give some directions on how to implement the `parseTrips()` method of the `GTFSParser` class:

1. Parse the `trips.txt` GTFS file and instantiate a new `Trajectory` object each time a new `trip_id` is encountered.
2. Parse the `stops.txt` to associate the stop identifiers `String stop_id` with their respective geographic coordinates (using a `HashMap<String,Coordinate>` data structure for instance).
3. Parse the `stop_times.txt` file to populate the stops of the vehicles in the `trajectory` attribute of the `Trajectory` class. You may use the `HashMap` structure created in the previous step to match the stop identifier `stop_id` with the coordinates of the stop. Also, you may use the `toElapsedTime()` function to store the time in the `trajectory` attribute.
4. Parse the `shapes.txt` file to add *waypoints* of the trajectory, that is, the points at which the vehicles will change directions, in between stops. The shape defines a LineString, that is, an ordered succession of geographic coordinates. We assume the vehicles move linearly from one waypoint to the next. In the following, we provide some basics to implement this step.

Knock the trajectories into shape. First, we propose to project each stop location of the `Trajectory` class onto the "closest" (from a geographic standpoint) line segment of the shape associated with the trajectory. For instance, in the figure below, we want to place a stop s_j on the shape. The stop s_j must be placed on the line segment $[v_i,\ v_{i+1}]$ such that the distance to the line segment $s_j s'_j$, equal to $(b+c)^2 - a^2$, is minimal. So, we first need to position the location of the stop s_j between the two vertices v_i and v_{i+1}. Second, we need to project the stop s_j coordinates onto the line segment $[v_i,\ v_{i+1}]$. For instance, the `project()` method from the JTS library `LocationIndexedLine` class may be used for this purpose. As an example, this link refers to an instantiation of a similar problem. Another solution would be to use the `project()` method from the `LineSegment` class, also from the JTS library.

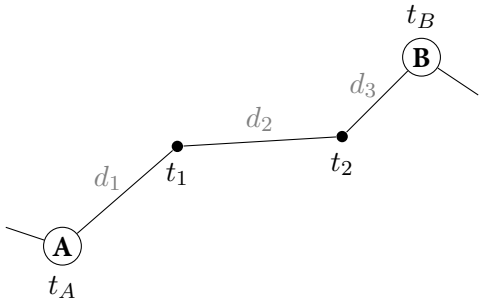


Once we have the coordinates of the correctly positioned projected stop location on the shape trajectory, we need to determine the timestamps from the vertices of the shape. We note that we have the timestamps of the stop locations. In the figure below, we have the timestamps of the stops A and B (t_A and t_B, respectively), and wish to deduce the timestamps of the shape points, t_1 and t_2. We assume a constant speed between the stops A and B. From the figure, we deduce:

$$\begin{cases} d_{AB} = d_1 + d_2 + d_3 \\ t_{AB} = t_B - t_A & \text{if } t_B > t_A \end{cases}$$

The Euclidean distance between the two points is given by the `distance()` method of the `Geometry` class. The linear temporal interpolation of the points' timestamps is as follows:

$$\begin{aligned} t_1 &= t_A + \frac{d_1}{d_{AB}} t_{AB} \\ t_2 &= t_A + \frac{d_1 + d_2}{d_{AB}} t_{AB} \end{aligned}$$



Get timestamped vehicle positions from the GTFS format. In a `Trajectory` instance, the combination between the projected stop locations (with the corresponding timestamps) and the shape points between the stop locations (with the interpolated timestamps) forms a time series. A time series is a sequence of data points over a time interval. Here, we are interested in having the successive geographical positions of the vehicles as a function of the time.

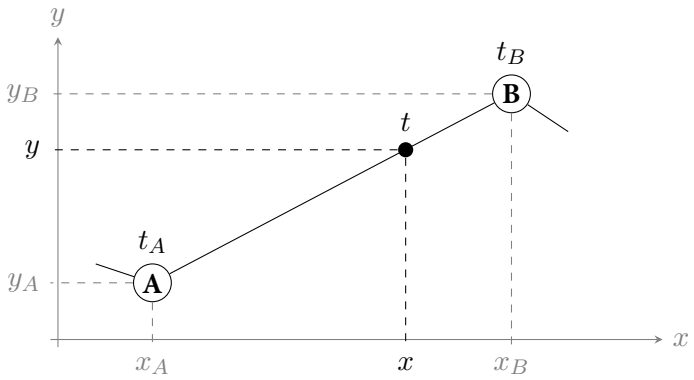
Now, we get to the last function that is of interest to us in this part. Given a `Long time` Unix timestamp, you need to define the method `Coordinate getPosition(Long time)` of the `Trajectory` class that returns the position (an instance of the class `Coordinate`) of the vehicle at timestamp `time` (of instance `Long`).

You may use some optimization algorithms, as binary search, to decrease the complexity of the function (which is likely to implement a search procedure). The position of the point in the trajectory is interpolated

linearly, in case the timestamp passed as argument does not exist in any of the successive positions of the vehicle.

The linear interpolation of a point $(x,\ y)$ corresponding to a timestamp t on line segment $[(x_A,\ y_A,\ t_A),\ (x_B,\ y_B,\ t_B)]$ with $\min(t_A,\ t_B) \leq t \leq \max(t_A,\ t_B)$ is as follows:

$$\begin{cases} x = x_A + \dfrac{t - t_A}{t_B - t_A}(x_B - x_A) \\ y = y_A + \dfrac{t - t_A}{t_B - t_A}(y_B - y_A) \end{cases}$$



3 | *Vehicles visualization*

The second part of your work will consist in visualizing the time series in a Java window. Each vehicle will be represented as a moving point on a map.

The moving vehicles are represented using the class `Trajectory` defined earlier. Their trajectories are represented by the data structure `SortedMap<Long,Coordinates>`. This data structure is composed of the vehicles' trajectories. We recall that a trajectory is a time series of the successive positions (`Coordinates` from the GeoTools library) of a moving vehicle at given timestamps (of instance `Long`). A vehicle may have multiple trajectories, as it may alternate between active and idle states during the observed time period. The trajectories describe the trips taken by a vehicle when active (on the move or at a standstill for a short period of time).

In this section, you will need Swing to implement the visualization of the bus transit system. The interactions with the GUI (Graphical User Interface) and the data are handled with an Observer pattern through a GUI action listener.

3.1 Hands-on example of the Observer design pattern

We provide a minimal example of the Observer design pattern with a Swing GUI, along with some explanations. (The example is taken from the `http://www.programcreek.com/` website.)

```
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
```

```
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JTextArea;

public class SimpleSwingExample {

    public static void main(String[] args) {
        JFrame frame = new JFrame("Frame Title");
        final JTextArea comp = new JTextArea();
        JButton btn = new JButton("click");
        frame.getContentPane().add(comp, BorderLayout.CENTER);
        frame.getContentPane().add(btn, BorderLayout.SOUTH);

        btn.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ae) {
                comp.setText("Button has been clicked");
            }
        });

        int width = 300;
        int height = 300;
        frame.setSize(width, height);
        frame.setVisible(true);
    }
}
```

First, we need a container like a Frame, a Window, or an Applet to display components such as panels, buttons, text areas etc.

```
JFrame frame = new JFrame("Frame Title");
```

Create some components such as panels, buttons, text areas etc.

```
final JTextArea comp = new JTextArea();
JButton btn = new JButton("click");
```

Add components to the container and arrange its layout using the `LayoutManagers`.

```
frame.getContentPane().add(comp,BorderLayout.CENTER);
frame.getContentPane().add(btn, BorderLayout.SOUTH);
```

Attach a listener to the button component. Interacting with a `Component` causes an `Event` to occur. To associate a user action with a component, attach a listener to it.

```
btn.addActionListener(new ActionListener(){
       public void actionPerformed(ActionEvent ae){
             comp.setText("Button has been clicked");
       }
});
```

```
public interface ActionListener extends EventListener
```

The listener interface is for receiving action events. The class (Main, in this case) that is interested in processing an action event, implements this interface, and the object created with that class is registered with a component, using the component's `addActionListener` method. When the action event occurs, that object's `actionPerformed` method is invoked.

Show the Frame.

```
int width = 300;
int height = 300;
frame.setSize(width, height);
frame.setVisible(true);
```

3.2 Get started with the GeoTools library

Although we have manipulated some components of the GeoTools library, we will delve into this library in this subsection. The GeoTools library provides tools (through API and interfaces) to manipulate and visualize geographic data. The GeoTools website proposes a tutorial to get started and familiar with the tools provided by the library. We propose that you start this tutorial to display a Shapefile in a Swing window using your favorite IDE.

Once you have successfully followed the tutorial and displayed a Swing window showing a Shapefile, we will need to represent points at the correct location. In a first step, we will only consider static points with (longitude, latitude) coordinates, before animating them.

To represent a point feature with (longitude, latitude) coordinates on a `Layer` of a `MapContent` map, we propose to use the following commented code.

To create the Point feature and display it on a Swing window, we first need to create the structure of the data type, with a `SimpleFeatureType` instance. In our context, a point represents a stop location, or a node in movement. Here, we choose to represent a stop location. The point is put in a feature that:

- Has a single property `"name"` of type `String`;
- Has a `Geometry` object `"location"` of the type `Point` in the Coordinate Reference System (CRS) WGS84;

In the GeoTools library, feature types (of instance `SimpleFeatureType`) are built using the Builder design pattern. In particular, the `SimpleFeatureTypeBuilder` is used to create an instance of `SimpleFeatureType`.

```
// Create a SimpleFeatureType builder
SimpleFeatureTypeBuilder b = new SimpleFeatureTypeBuilder();

//set the name
b.setName( "Stop" );

// Add a "name" property
b.add( "name", String.class );

//add a geometry property
b.setCRS( DefaultGeographicCRS.WGS84 ); // set crs first
b.add( "location", Point.class ); // then add geometry

//build the type
final SimpleFeatureType STOP = b.buildFeatureType();
```

Then, create the feature(s) using the feature type previously defined. The Builder design pattern is yet again used to create instances of `SimpleFeature`. However, a Factory design pattern is used to create `Geometry` instances, using a `GeometryFactory` factory.

```
// Create a SimpleFeature builder
SimpleFeatureBuilder featureBuilder = new SimpleFeatureBuilder(TYPE);
// Create a Geometry factory
GeometryFactory gf = new GeometryFactory();
// Create a point corresponds to the location of the Eiffel Tower
Coordinate coords = new Coordinate(2.294339,48.858179);
Point point = gf.createPoint( coords );

// Build the Stop feature
featureBuilder.add( "location", point );
featureBuilder.add( "name", "Tour Eiffel" );
SimpleFeature feature = featureBuilder.buildFeature( "fid.1" );
```

Next, you can create a feature collection `DefaultFeatureCollection`, using the feature(s) (instances of `SimpleFeature`) you defined before

```
// Create a FeatureCollection
DefaultFeatureCollection featureCollection = new DefaultFeatureCollection();
featureCollection.add(feature); // Add feature 1
// Add feature 2, 3, etc.
```

Finally, you can add the feature collection to a layer and add it to the map.

```
Style style = SLD.createSimpleStyle(featureCollection.getSchema());
Layer layer = new FeatureLayer(featureCollection, style);
map.addLayer(layer);
```

Some notes:

- GeoTools documentation for geographic features (the features are the main components of Shapefile): click here.
- The style of the displayed features is defined by the `Layer` class. You may have a look at the tutorial available on the GeoTools website: click here.

We note that all the coordinates of the traces given as example are generally given in WGS84 (EPSG:4326). This coordinate reference system (CRS) is *geodetic*, which means that the coordinates of a point in this system corresponds to a couple of (longitude, latitude). Besides, the line segments geocoded in this CRS are not linear, which means that all the interpolations and geometric operations would be wrong. We need to project the coordinate onto a plane to enable these operations. We propose to use the Mercator projection (EPSG:3857), which projects the surface of the Earth on a cylinder. The GeoTools library provides functions to transform geometries from one CRS to another:

```
// Create a Geometry factory
GeometryFactory gf = new GeometryFactory();

// Set the source and target CRS
CoordinateReferenceSystem sourceCRS = CRS.decode( "EPSG:4326" ); // WGS84
CoordinateReferenceSystem targetCRS = CRS.decode( "EPSG:3857" ); // Mercator
```

```
// Get the transform function
MathTransform transform = CRS.findMathTransform(sourceCRS, targetCRS, true);

// Transform the point (2.294339, 48.858179) from WGS 84 to Mercator
Geometry sourceGeometry = gf.createPoint(new Coordinate(2.294339,48.858179))
Geometry targetGeometry = JTS.transform( sourceGeometry , transform );
```

Note. The following dependencies must be added to the `pom.xml` file (in case you use Eclipse and Maven), between `<dependencies>` and `</dependencies>`:

```
<dependency>
    <groupId>org.geotools</groupId>
    <artifactId>gt-referencing</artifactId>
    <version>${geotools.version}</version>
</dependency>
<dependency>
    <groupId>org.geotools</groupId>
    <artifactId>gt-epsg-hsql</artifactId>
    <version>${geotools.version}</version>
</dependency>
<dependency>
    <groupId>org.geotools</groupId>
    <artifactId>gt-epsg-extension</artifactId>
    <version>${geotools.version}</version>
</dependency>
```

Also, you need to use the `Geometry` package from the `com.vividsolutions.jts.geom` package when importing the packages.

3.3 Visualize the vehicles' movements

Now, you need to implement a visualization of the transit system using the Observer design pattern to update the view when the timestamp increases. The window should display all active vehicles (*e.g.*, by querying all active vehicles at the current instant, using the `activeTrajectories()` method of the `Transit` class. You can add some components to the frame, for instance, one that displays the current timestamp, as well as a slider with a cursor that allows a user to choose a timestamp and display the active vehicles for this timestamp. You can also add buttons to increase or decrease the speed at which the time elapses.

To animate the movement of the **active** vehicles on the map, you may instantiate a `JMapPane` and use Swing timers to animate the movement of active vehicles on the map.

If you want to represent the points on a street view, you may download the street maps from the OpenStreetMap project. In particular, the geofabrik website proposes multiple datasets in the Shapefile format to download. For instance, you can download the streetmap of Dublin and display it as a layer, and display the vehicle movements on top of this layer, in another layer. If you chose to represent the coordinates in the Mercator projection (EPSG:3857), you need to project the whole shapefile you have downloaded with Mercator CRS. The GeoTools website has a tutorial which describes how to proceed to transform all the features of the Shapefile and save the output Shapefile in another file. Note that, the coordinates of the trajectories (longitude,

latitude) might be turn out to be reversed into (latitude, longitude), depending on the source of the streetmap data.

The website This website post provides the basic components to animate objects using Swing timers. Keep in mind that you should use the classes defined in the GeoTools library, inherited from Swing.

Another possibility would be to use another geographic library, as minigeo, which provides basic functions to display points on a window, using Swing. You may also check the source code and see whether it would be better to implement your visualization application using this framework.

4 | Tweaking and improvements

In this (final!) section, we give you *carte blanche* to improve the application you have built. We give out some possible ways to tweak it, improve it, and add functionalities to it.

First, you may have noticed that, when the number of vehicles increases, the application's response time is slower. This is due to performance issues. You may give some ways to improve the overall performance of the applications, and try to implement the changes you have considered.

Second, we can imagine that the vehicles can communicate with each other. In the Java applet, we represent the nodes communications with a transmission radius (for instance, using the `buffer()` method from the `Geometry` class of the GeoTools library. When in contact (*i.e.*, when the nodes are in each other's transmission radius), the nodes exchange data according to a forwarding algorithm. A simple algorithm, known as "epidemic", would be to transmit the data owned by a vehicle whenever there is a contact with another vehicle. You are also welcome to think and implement other transmission algorithms. Some are briefly described here.

Third, you may want to improve the user interface of the application by refining the positioning of the components on the frame, or adding some components related to new features to the frame. The components you may add should *enhance* the user experience, and add an extra value to application. For instance, you could add some tools for analysis of the transit system (*e.g.*, statistics showing how many stops are serviced par hour, or the instant number of vehicles that are in service).