Lappeenranta University of Technology
School of Engineering Science
Degree Program in Intelligent Computing

Tikhon Belousko

SOFTWARE FOR TRANSPORT ACCESSIBILITY ANALYSIS: THE CASE OF MOSCOW

Examiners: Assoc. Prof. Arto Kaarna
Sen. Lec. Vitaly Bragilevsky

Supervisors: Assoc. Prof. Arto Kaarna
Sen. Lec. Vitaly Bragilevsky

ABSTRACT

Lappeenranta University of Technology
School of Engineering Science
Degree Program in Intelligent Computing

Tikhon Belousko

Software for Transport Accessibility Analysis: The Case of Moscow

2016

19 pages, 2 thingy, 3 stuff.

Examiners: Assoc. Prof. Arto Kaarna
Sen. Lec. Vitaly Bragilevsky

Keywords: web-based maps, GIS, transport accessibility, vector tiles

Abstract here...

PREFACE

I wish to thank my supervisors ...

Lappeenranta, April 19th, 2016

Tikhon Belousko

CONTENTS

1 INTRODUCTION

1.1 Background

In recent years, there have been many papers describing ...

1.2 Objectives and Delimitations

The aim of this paper is to generalize methods ...

1.3 Structure of the Report

The paper is divided into N main sections...

2 OVERVIEW

2.1 GIS Tools for Accessibility Analysis

2.2 Modern Web-based Maps

2.3 Data Formats

2.3.1 GeoJSON

2.3.2 MBTiles

2.3.3 Protocol Buffers

3 Software Requirements Specification

3.1 Graphical User Interface

3.2 Operating Environment

3.3 System Features

4 ARCHITECTURE

4.1 Client

The first significant part of the application is the client which will be the interface for the user to manipulate map and data. The core functionality of the client is follows:

1. Panning and zooming map.
2. Selecting a point on the map to build a whole graph of the routes needed to move from selected point to all other points of the city. All of the routes are parametrized by color and width. These parameters are calculated utilizing information about transport type specified in this point and the coefficient which is defined as number of times this particular line is used in all of the directions calculated from this point.
3. Showing prevailing transport in all points of the grid. Prevailing transport is defined by calculating total time spent in each transport type. The maximum of all of the values will be taken as prevailing.
4. Showing fastest and slowest roads. The map should also have interactive slider which will be used to filter roads within particular speed range.
5. Showing most accessible points on the grid. The transport accessibility coefficient of the point can be defined as total time spent in a transport while moving from selected point to all other points. All of the values after that scaled to be between 0 and 1. This view should also have a filter by transport accessibility coefficient.

For the reason of the easy distribution it was selected to use browser based client. Thus client is implemented as single-page web application, which should support all modern browsers such as Safari, Google Chrome, Firefox, Microsoft Edge.

4.2 RESTful Server

For serving data to the client it was selected to utilize RESTful [1] architectural style, which represents HTTP approach to read, update and delete data. Due to the fact, that our application will only read data and not modify it, then we will only need to describe all

of the end-points for data access where data will be encoded in GeoJSON [2]. For our application following end-points were selected:

- **`GET /points`**
  The returned points are described as FeatureCollection of points with following properties (see also Listing 1):

  name is a string;

  id is a unique identifier of the point encoded as string;

  prevalingTransport could be “BUS”, “TROLLEYBUS”, “TRAM”, “SUBWAY”, “COMMUTER_TRAIN”, “SHARE_TAXI”, “DRIVING”, “WALKING”.

  accessibility is a floating number in range from 0 to 1.

```
{
  "type": "FeatureCollection",
  "features": [
    "type": "Feature",
    "geometry": [0,0],
    "properties": {
      "id": 0,
      "name": "Location Name",
      "prevailingTransport": "SUBWAY",
      "accessibility": 0.6
    }
  ]
}
```

Listing 1. Points response example

- **`GET /lines?point_id`**
  This endpoint is parametrized by point_id which means that if point id is presented then the client will receive only those lines which are associated with specified point. On the other hand, if no point id is not specified all lines are returned.

  The properties of the lines can be describe as follows (see also Listing 2):

  id is a unique identifier of the line encoded as string;

travelMode is on of the “BUS”, “TROLLEYBUS”, “TRAM”, “SUBWAY”, “COMMUTER_TRAIN”, “SHARE_TAXI”, “DRIVING”, or “WALKING”.

weight number of times this line was used in all of the routes across all of the directions’ set.

duration time needed to travel this line in seconds.

distance total length of this line in meters.

```
{
  "type": "FeatureCollection",
  "features": [
    "type": "Feature",
    "geometry": [[0,0], [0,0]],
    "properties": {
      "id": 0,
      "width": 1,
      "weight": 200
      "duration": 100,
      "distance": 400,
      "travelMode": "BUS"
    ]
  }
}
```

Listing 2. Lines response example

It is important to note that we could also add such filtering parameters as `speed` and `travelMode`, but it was revealed during set of experiments that filtering on server side can take significant amount of time which is up to few hundreds seconds. Hence, this parameters were dropped and filtering is performed on the client.

Suggested architecture implies that client will perform rendering using GeoJSON data, although it was investigated that for our purposes rendering becomes rather slow and after series of optimizations the best rendering time was close to 3 seconds. The next improvement to the current scheme will be adding tile server which can significantly speed up rendering time.

4.3 Tile Server

Graphical map tiles are usually rectangular images in raster or vector format. Most of the popular map library providers utilize tiles [3, 4] for rendering their maps. Raster tiles are basically images which do not allow to change color of roads and landscapes. In contrast, vector tiles are not just images but structures which contain geometries and metadata such as roads, rivers, places in special compact format. Vector tiles only rendered when requested by the client.

There are several benefits of using vector tiles. The first, advantage is small size of the tiles, which allows rendering of high resolution and caching. The second advantage is an ability to change style of the layers: adjust colors of the geometry, line width, set borders to polygons, set background patterns. Finally, maps based on vector tiles allow smooth transitions between zoom levels. Although, the cost of usage of the vector tiles is the lack of compatibility with older browsers. At the moment most of the libraries demand WebGL support and latest browser versions which are Chrome 49.0, Safari 9, Internet Explorer 10 and higher, Microsoft Edge 13 [5, 6]. Another drawback of usage tiles is that geometry can not be transformed in real-time and rendering small amounts of data is actually slower than GeoJSON approach.

Switching from GeoJSON server to vector-tile server resulted in considerable speed improvement. If previously rendering time took around 3 seconds and amount of transfered data was around 1.8 Mb, then with tile server rendering takes less than a second and size of the data transfered to the client is around 100 kb.

4.4 Summary

Although, tiles suit well for rendering considerable amounts data, for points rendering it was decided to utilize GeoJSON data. The final architecture is illustrated on Figure 1 and can be described as follows:

1. Web-based client supporting latest browsers.
2. GeoJSON RESTfull server providing points data.
3. Tile server providing data about lines.

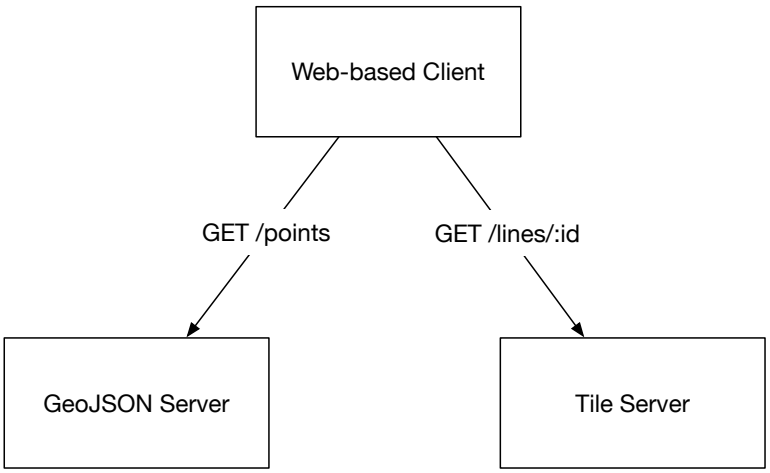

Figure 1. Final system architecture.

5 IMPLEMENTATION

5.1 Data Pre-processing

The work started by preparing the data so servers could work with it the most efficient way. This section is devoted to the description of the initial data and how it was processed. The section also involves comparison of deferent approaches of data structuring, their comparison and summarizing of the results.

5.1.1 Description of the Raw Data

The provided original data is collected using Google Directions API [7] applying following algorithm. First, on top of the city the grid of points was constructed using WGS 84 [8] coordinate system. Second, for every pair of points the directions were calculated using HTTP request to the API. Returned data is encoded in JSON format written to a text file. As result raw data is a text file where every line is a JSON object received from Google Directions API. The process of data collection is demonstrated on Figure 2.

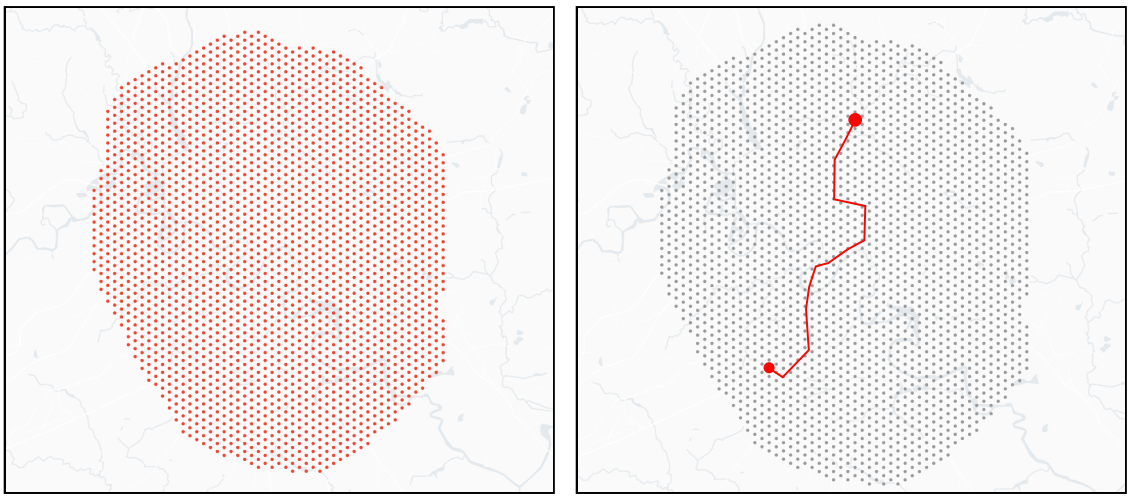

Figure 2. The process of the collecting data.

In fact there is also meta information such as object id, job status, waypoints order, warnings, but if we simplify and leave only important fields that we will need in pre-processing step, then JSON object could look like this:

```
{
  "start_lat": 55.726497,
  "start_long": 37.338183,
  "end_lat": 55.886619,
  "end_long": 37.579683,
  "data": {
    "routes": [{
      "legs": [{
        "distance": { "text": "45.1 km", "value": 45093 },
        "end_address": "Novgorodskaya ulitsa...",
        "start_address": "Razdorovskaya, Romashkovo...",
        "steps": [{
          "travel_mode": "DRIVING",
          "polyline": { "points": "wicsI{u{bFs@rDC" },
          "distance": { "text": "4.2 km", "value": 4166 },
          "duration": { "text": "7 mins", "value": 438 }
        }, ... ]
      }, ... ]
    }, ... ]
  }
}
```

Listing 3. Google Directions API simplified response example

As been be seen from simplified response example, the object contains coordinates of the start and end point, array routes which consists of `legs`. `Legs` are parts of the route between waypoints. Since in our queries we do not have middle waypoints, the routes array will have only one element in all cases. Each leg contains total distance information about start and end location. Also inside `legs` there is `steps`, each step is a part of route which can be described by single command and type of transport, for example “move forward by bus” is clearly a step. Step keeps information about its distance and duration which will be needed to complete this step. Other important things are `travel_mode` which indicates what type of transport is used in this step and `polyline` which is essentially a set points of points encoded using lossy Google’s algorithm which converts array of float numbers first to binary representation, then to decimal integers and finally to string using ASCII codes [9].

5.1.2 Extracting Points

The data collection was initiated before by Mathrioshka, thus the first task was extracting grid points from the raw data. This step was performed utilizing script written in Python, which is reading file line by line and extracting starting location point from the data. The idea of the algorithm is presented in Listing 4. As can be seen from the listing, GeoHash [REFERENCE] standard was used to prevent repetitions of the points. Thus, coordinates could be mapped to strings which can be utilized as ids in points dictionary. In our implementation for encoding PyGeohash library was used [REFERENCE].

```
1  points = []
2
3  for line in jsonfile:
4      json_obj = json.loads(line)
5      slat = json_obj['start_lat']
6      slng = json_obj['start_long']
7      point = {
8          'point_id': geohash.encode(slat, slng),
9          'lat': slat,
10         'lng': slng
11     }
12     points[point['point_id']] = point
```

Listing 4. Points extraction

5.1.3 Extracting Lines

The second main task was to extract lines from the file, but first to make experiments faster it was decided to convert data to more convenient format so it would be easier to experiment with data and process more effectively. Once we have data extracted we will convert it to the set of GeoJSON files and after that generate vector tiles which will be served by our tile server.

For intermediate lines representation the CSV file format was selected, due to its simplicity and availability of the encoders and decoders inside standard Python library. The algorithm is presented in Listing 5. First, we initialize table of lines which will contain all unique lines encoded in raw data. Second, we read file object by object. Each object is

then decomposed into set of lines using `lines_from_json()` function. Once lines are extracted the assertion is performed to check whether the line was already met before. If the answer is no, then we initialize line, otherwise we just sum distance and duration and increase weight by one. The distance, duration and weight are necessary to compute line width, speed, prevailing transport and accessibility of the point.

```
# Table of lines
T = {}

with open(DATA_PATH) as jsonfile:

    for json_str in jsonfile:
        json_obj = json.loads(json_str)

        for line in lines_from_json(json_obj):
            line_hash = line['line_hash']

            if line_hash not in T:
                del line['line_hash']
                T[line_hash] = line
                T[line_hash]['line_id'] = len(T)
            else:
                T[line_hash]['weight'] += 1
                T[line_hash]['distance'] += line['distance']
                T[line_hash]['duration'] += line['duration']
```

Listing 5. Lines conversion.

6 INTEGRATION

6.1 Development Environment

6.2 Building a Project

6.3 Deploying a Project

7 CONCLUSION

REFERENCES

[1] Representational state transfer. https://en.wikipedia.org/wiki/Representational_state_transfer, 2016. [Online; accessed 19-April-2016].

[2] Allan Doyle Sean Gillies Tim Schaub Christopher Schmidt Howard Butler, Martin Daly. The geojson format specification. http://geojson.org/geojson-spec.html, 2008. [Online; accessed 19-April-2016].

[3] Map types: Tile coordinates. https://developers.google.com/maps/documentation/javascript/maptypes?hl=en#TileCoordinates, April 14, 2016. [Online; accessed 19-April-2016].

[4] Vector tiles. https://www.mapbox.com/vector-tiles/. [Online; accessed 19-April-2016].

[5] Google maps javascript api: Browser support. https://developers.google.com/maps/documentation/javascript/browsersupport, 2016. [Online; accessed 20-April-2016].

[6] Troubleshooting: Browser support. https://www.mapbox.com/help/mapbox-browser-support. [Online; accessed 20-April-2016].

[7] Google maps directions api. https://developers.google.com/maps/documentation/directions/, 2016. [Online; accessed 20-April-2016].

[8] World geodetic system. https://en.wikipedia.org/wiki/World_Geodetic_System, 2016. [Online; accessed 20-April-2016].

[9] Google maps directions api: Encoded polyline algorithm format.