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

38 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

This section is devoted to the description of the available geo-spatial data analysis tools. Also, modern web-based maps libraries are considered. Finally, data formats used for storing geographical data are reviewed.

2.1 Geographical Analysis Software

There are many tools available for analysis of the geo-spatial data. However, in this section three different tools such as GRASS, QGIS, CartoDB will be reviewed. Although, mentioned tools may be considered as GIS, each of them provides different functionality and oriented on different user audience.

2.1.1 GRASS GIS

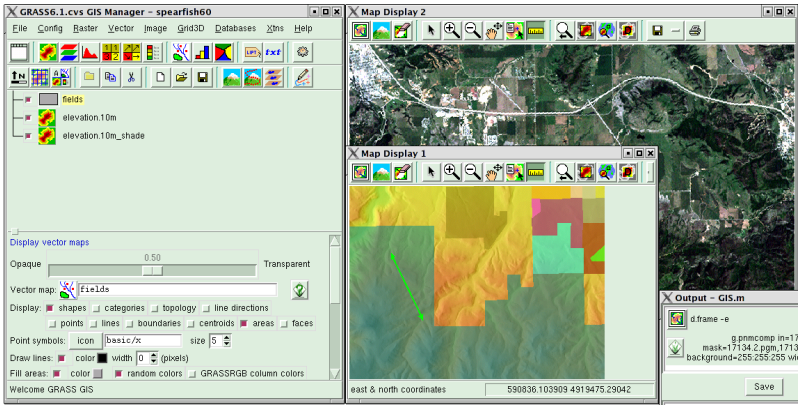

Figure 1. GRASS GIS interface.

Originally GRASS GIS [1] (see Figure 1) was developed by US Army Construction Engineering Research Laboratories and has a history of 30 years of the development. It is a powerful and open- source tool with large amount of capabilities. First, GRASS supports raster, voxel and vector analysis and visualization. In addition to that, there is a toolbox for image processing, allowing to apply variety of filters, adjust histograms and perform segmentation tasks. Finally, the program supports numerous formats such as SQLite databases, PostgreSQL, mySQL, ODBC, CSV, satellite data, UAV images.

The application of the GRASS GIS includes various fields such as archeology, cartography, geology, geophysics and meteorology, although in our case it does not suit for the task. On of the biggest drawbacks is steep learning curve. The program requires extended knowledge from the user in the field of the geographical analysis and programming. The target audience are scientists and engineers. Another problem is complex installation and distribution process. On Mac OS X, the installation requires six dependent libraries. Moreover, the last version of the Mac OS X (10.11) is not supported.

2.1.2 QGIS

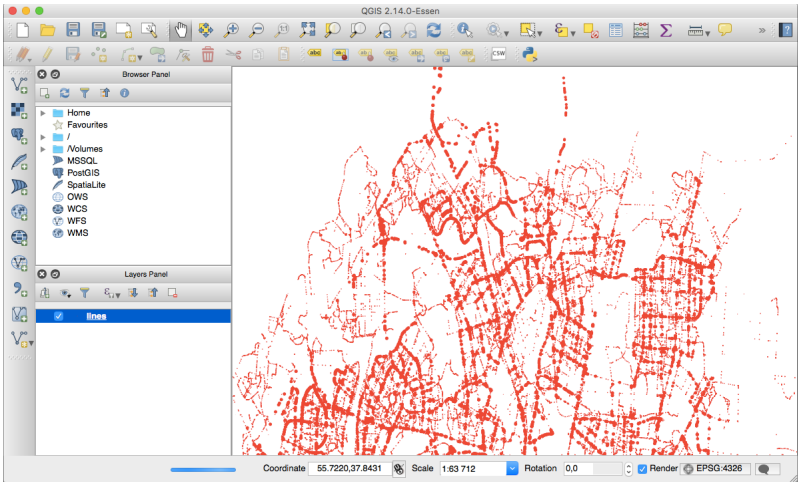

Figure 2. QGIS interface.

QGIS is a free software for geographical data viewing, editing and analysis. The software is licensed under GNU GPL. The development process started in 2002 with extensive use of the Qt library. QGIS runs on multiple operating systems including Mac OS X, Linux and Windows. The functionality of the QGIS can be extended via plug-ins written in Python.

Although QGIS lacks some features of the GRASS GIS, it has simpler UI (see example on Figure 2) and does not have support of latest Mac OS X (10.11). Moreover, there are web based implementations of the QGIS such as QGIS Server and QGIS Web Client. On the other hand, the interface is still requires some programming and geographical background.

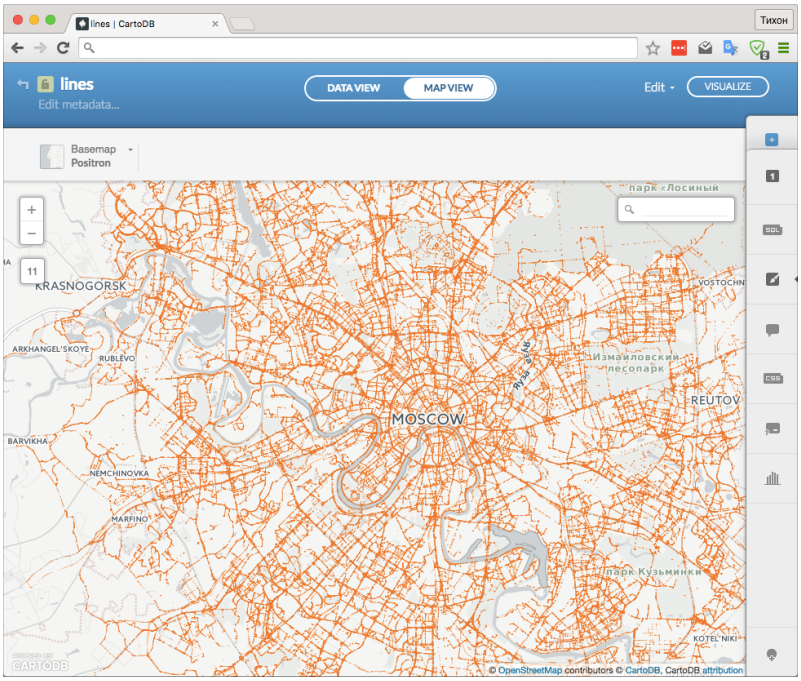

Figure 3. CartoDB interface.

2.1.3 CartoDB

CartoDB is a browser based proprietary solution for mapping and analysis, which provides simplest interface in comparison to GRASS GIS and QGIS. Indeed, CartoDB is a great tool for creating interactive maps for the user without particular background in the field. Another huge plus of the platform is simplicity of the distribution. Basically anyone with access to the Internet and modern browser can view interactive map. However, for our purpose the performance was not sufficient and basic filtering over the dataset was taking up to 10 seconds.

2.1.4 Summary

Among all of the presented solutions simple user interface and access with browser makes CartDB a good candidate for the task. Yet, there is an issue with rendering performance which can hardly be solved. Second problem is the restrictions of the API making impossible to implement some of the functionality. Finally, it should be underlined that CartoDB free plan does not support datasets larger than 250 MB.

2.2 Modern Web-based Maps

In our application the choice of maps library should be made. In this section several libraries for maps drawing are reviewed. The pluses and minuses of each of the library are considered.

Google Maps Javascript API – on of the most popular libraries on the list [2].

Pluses:

- Maps styling.
- Supports different localization options.
- The most detailed map across the world.
- StreetView.
- Support for mobile devices.

Minuses:

- Does not support vector tiles.
- Limited support for custom tiles.
- Free until exceeding 25,000 map loads per 24 hours for 90 consecutive days.

Leaflet.js – lightweight mobile-friendly interactive maps library [3].

Pluses:

- Small size – only 33 KB of JavaScript code.
- Relatively simple API.
- Support for custom tile sources.
- Support for multilayer maps.
- Considerable amount of plug-ins.
- Free and open-source.

Minuses:

- Does not support vector tiles.
- Lack of Russian localization.

Yandex Maps – best maps fidelity in Russia [4].

Pluses:

- Highly detailed maps of Russia.
- Support of Russian localization.
- No restrictions for map views.

Minuses:

- Does not support vector tiles.
- Does not support map styles.
- Does not support custom tiles rendering.
- Proprietary.

Mapbox GL JS – only library with whole vector tiles support [5].

Pluses:

- Supports vector tiles.
- Supports real-time maps styles.
- Supports custom tile source.
- Based on Leaflet API.
- Open source.

Minuses:

- Standard tile source has limitations on views per month.
- Not fully detailed map of Russia.
- Lack of Russian localization.

2.3 Data Formats

In the process of the development of transport accessibility analysis tool several particular data formats were used. In this section utilized formats will be described.

2.3.1 GeoJSON

One of the most extensively used formats for encoding various types of geographical data is GeoJSON which is actually a subset of JSON format [6]. GeoJSON can store

information about such objects as Point, MultiPoint, Line, LineString, MultiLineString, Polygon. Geometric objects which have some additional properties are stored as Feature objects. Moreover, Features can be organized in feature collection. The example of the location which represents center of the Moscow can be encoded as described on Listing 1.

```
{
  "type": "Feature",
  "geometry": {
    "type": "Point",
    "coordinates": [37.61581, 55.74489]
  },
  "properties": {
    "name": "The Center of Moscow"
  }
}
```

Listing 1. GeoJSON data example.

The GeoJSON is text data format which results in larger size of the transfered data in comparison with binary formats. Although, usually on the server it can be compressed using gzip [7] which has native support in modern browsers. Compression results in dramatical size reduction up to 5 times in general case. However, it should be mentioned that archiving adds small overhead in coding and decoding steps.

2.3.2 Geobuf

Geobuf is a compact data format for encoding geographical data. The format is based on Protocol Buffers – language-neutral mechanism developed by Google for serializing structured data [8]. Geobuf allows losslessly transformation of the GeoJSON data into protocol buffers. There are several important advantages like faster compression in comparison to even native JSON `parse` and `stringify` methods. Another valuable attribute of this format is compact size, which is 8 times smaller than raw GeoJSON and 2 times smaller than GeoJSON after applying gzip compression. The comparison of GeoJSON and Geobuf in terms size is presented on Table 1.

Table 1. Sample compression sizes [9].

	normal	**gzipped**
us-zips.json	101.85 MB	26.67 MB
us-zips.pbf	12.24 MB	10.48 MB
idaho.json	10.92 MB	2.57 MB
idaho.pbf	1.37 MB	1.17 MB

2.3.3 MBTiles

Web maps may contain million of tiles, hence there is clearly a problem in storing and managing such enormous amount of data. To overcome the problem of handling so many tiles data Mapbox team has developed open-source data format for storing tiles in a single SQLite data base.

SQLite is claimed to be ideal for the purpose of storing tiles since it is available on all of the platforms including mobile devices. Each `.sqlite` data base is self-contained and does not require any special setup, which results in high portability.

Another great feature of the MBTiles format is the ability to effectively store duplicate tiles. As an example the tile located in the middle of the ocean can be considered (see Figure 4). It is clear that there is considerable amount of tiles containing solid blue color. Taking in account all of the zoom levels it can lead to millions of the duplicates. However, MBTiles can reference thousands of tiles to the same image without the need for loading all look-alike pictures.

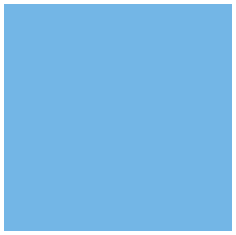

Figure 4. Example of the tile duplicate.

Important to note that MBTiles may be used to store both image-based tiles encoded in PNG or vector tiles encoded in Protocol Buffers. This is a crucial detail since vector tiles allow real-time styling and provide smoother zooming.

3 SOFTWARE REQUIREMENTS SPECIFICATION

The software development process starts with writing a software requirements specification (SRS) so that it serve as an agreement between client and contractor about the set of the features which need to be implemented. Another important purpose of this document is to create understanding of the features with higher and lower priority, thus developer would know what should be implemented in the first version of the product and what could be added later.

In this section the simplified version of the SRS will be presented which includes description of the user interface, operating environment, and set of system features with detailed explanation.

3.1 Graphical User Interface

The interface design was provided by Mathrioshka LLC, which includes set of different states of the application with small description provided. The interface should have been implemented to work in browser with use of HTML5 standard. The whole program should be implemented as single page application, which means that all changes in the state of the application happen without page reload. The interface have to be responsive and support screen sizes starting from 800×600 pixels.

The manipulation of the data within the application is performed utilizing main control elements. Most important elements are listed below:

1. Mode selector is needed for switching from general overview to location-wise analysis and located in the left bottom corner of the screen.

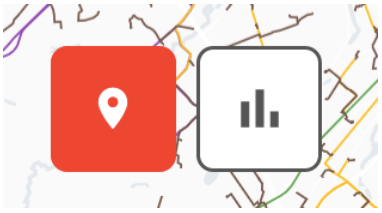

Figure 5. Mode selector.

2. Overview mode selector switches different types of overview analysis tools. For some views it has embedded slider used for filtering. The component is positioned on the right side of the screen.

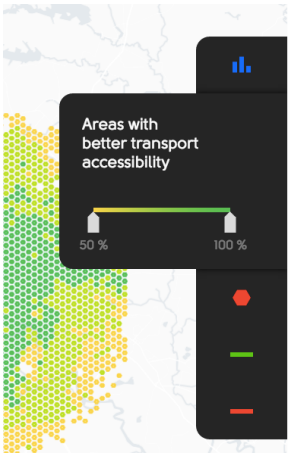

Figure 6. Overview mode selector.

3. Transport filter is a set of checkboxes where every checkbox represent certain type of transport. This component is placed on top of the screen.

Figure 7. Transport filter.

4. Current location indicator shows what point is selected on the map at this moment and also allows to clear current selection. The component is located on the right side of the window.

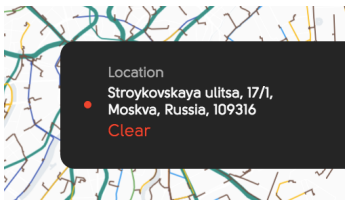

Figure 8. Current location indicator.

5. Map is the most important element of the interface which is presented on every screen of the application the only thing that changes is the content of the map. This component can show points, areas, routes and also provides functionality for selecting particular locations for analysis.

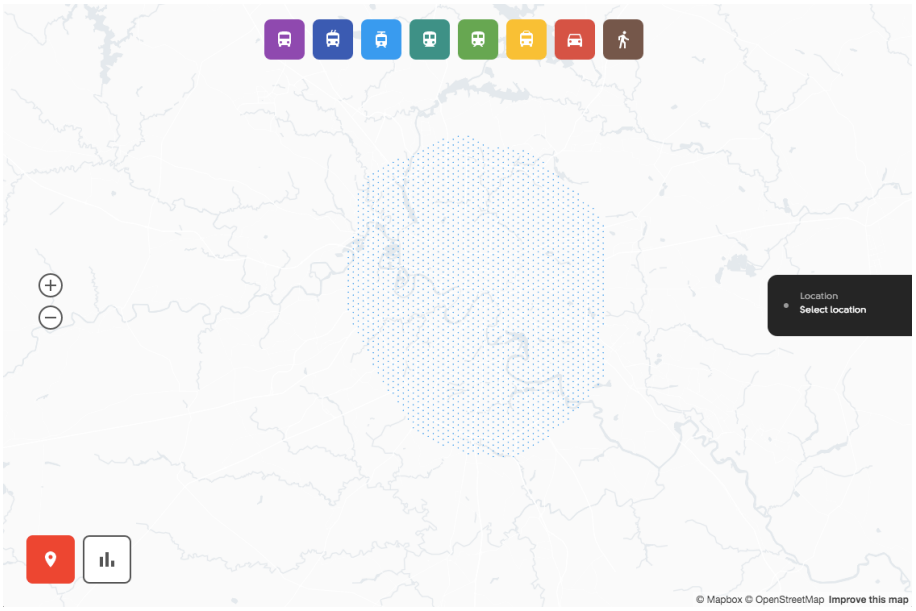

Figure 9. Map.

3.2 Operating Environment

The software is developed on x86 based computer using Mac OS X 10.11 operating system. The computer hardware is featured 8 GB 1600 MHz DDR3 RAM and 1.6 GHz Intel Core i5 processor. The client should operate on any system which is able to install browsers like Google Chrome 49.0, Safari, Firefox or Microsoft Edge. Hence it works well on Windows, Mac OS X and Linux. As for the server, it was tested to work on Ubuntu 14.04 and 512 Mb RAM. Running the server side on Windows was not tested.

3.3 System Features

Use case 1: Show all directions from specific location to all other points.

Primary actor: User

Main scenario:

1. User enters “Location analysis” mode.
2. User selects specific point on the grid.
3. The system displays all routes coming from selected point to all other points on the grid. The routes are colored by the type of transport. The line which represents part of the route has a width depending on weight of current line. The weight is calculated as occurrence of this line in all possible routes, which means if two different routes are going over particular line it means that the weight of this line will be equal to two. In other words the wider the line, the more this line is used in all other directions calculated for all points on the grid to all points.
4. The user is able to filter lines by transport type.
5. The user is able to switch color mode from coloring by transport to coloring by speed.

Use case 2: Show prevailing transport

Primary actor: User

Main scenario:

1. User enters “General overview” mode.
2. User selects “Prevailing transport” option.
3. The grid of markers is displayed. Each marker is associated with one point on the grid. The markers are colored accordingly to certain transport type. Prevailing transport is defined by calculating total time spent in particular transport when moving from current points to all other points on the grid.
4. The user is able to filter markers by transport type.

Use case 3: Most accessible locations

Primary actor: User

Main scenario:

1. User enters “General overview” mode.
2. User selects “Most accessible locations” option.
3. The grid of markers is displayed. Each marker is associated with one point on the grid. The markers are colored accordingly to accessibility level of the certain point. Accessibility is defined as total time spent in transport when user moves from current points to all other points on the grid. Once accessibility is calculated for all points the value is scaled to 0 to 100%.

4. The user is able to filter markers by accessibility level from 50% to 100%.

Use case 4: Least accessible locations

Primary actor: User

Main scenario:

1. User enters “General overview” mode.
2. User selects “Least accessible locations” option.
3. The grid of markers is displayed. Each marker is associated with one point on the grid. The markers are colored accordingly to accessibility level of the certain point. Accessibility is defined as total time spent in transport when user moves from current points to all other points on the grid. Once accessibility is calculated for all points the value is scaled to 0 to 100%.
4. The user is able to filter markers by accessibility level from 0% to 50%.

Comments: This feature is very similar to 3rd use case except the color scheme is different and filtering is preformed within other percentage range.

Use case 5: Fastest parts of the routes

Primary actor: User

Main scenario:

1. User enters “General overview” mode.
2. User selects “Fastest parts of the routes” option.
3. All of the lines (routes) are displayed which has speed in a range from 20 km/h to 40 km/h. The lines are colored differently from yellow to green depending on the speed.
4. The user is able to filter lines by speed using slider control.

Use case 6: Slowest parts of the routes

Primary actor: User

Main scenario:

1. User enters “General overview” mode.
2. User selects “Slowest parts of the routes” option.
3. All of the lines (routes) are displayed which has speed in a range from 0 km/h to 20 km/h. The lines are colored differently from dark red to light red depending on the speed.
4. The user is able to filter lines by speed using slider control.

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 [10] 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 [6]. 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 2):

  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 2. 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 3):

  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 3. 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 [11, 12] 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 [13, 14]. 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 10 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.

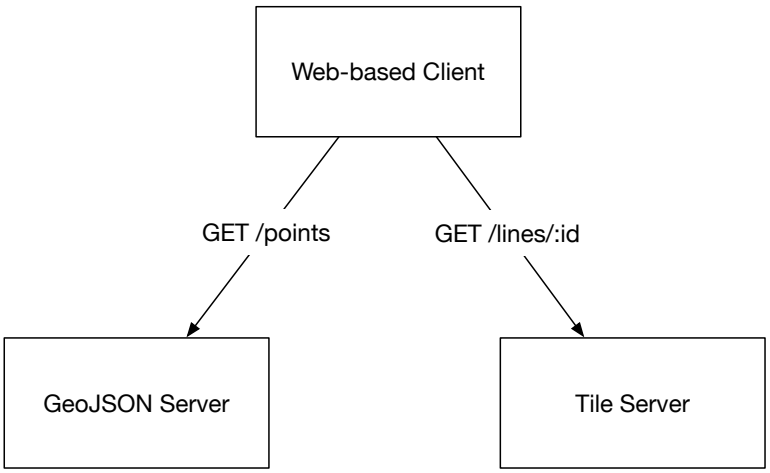

Figure 10. Final system architecture.

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 [15] applying following algorithm. First, on top of the city the grid of points was constructed using WGS 84 [16] 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 11.

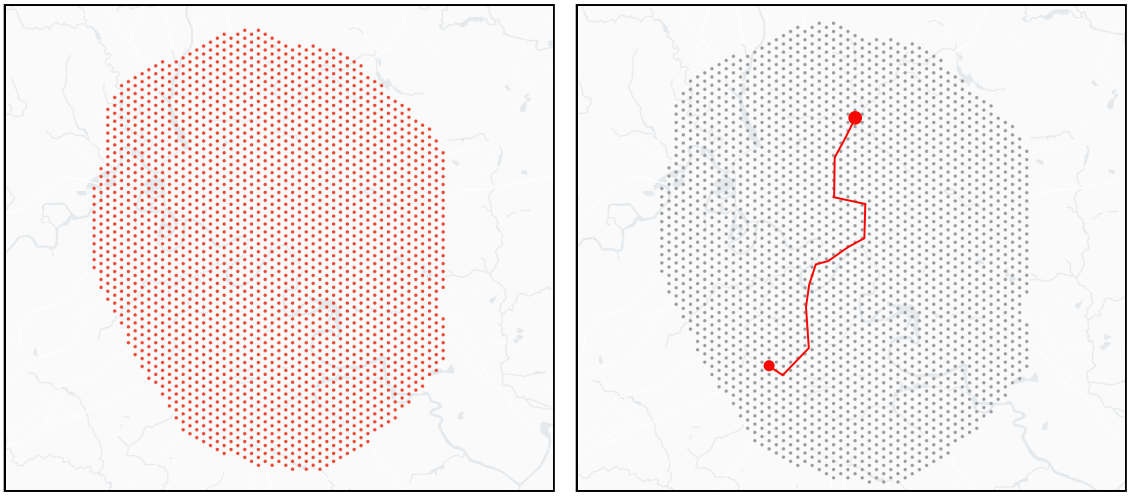

Figure 11. The process of the collecting data.

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 4. 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 [17].

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 set 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 5. As can be seen from the listing, Geohash [18] 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 `python-geohash` library was used [19]. For storing, the result of the algorithm the pickle [20] format was selected, since it has quite simple interface and built in inside Python standard library.

```
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 5. Points extraction

5.1.3 Extracting Lines

Another important task was to extract lines from the file, but first to make experiments faster it was decided to convert data to more convenient format, hence it would be easier to experiment and process data 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 6. 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 6. Lines conversion.

5.1.4 Link Lines to Points

Since we have our data extracted now it is important to have references from lines to points. The first approach was to store in lines CSV file also point identifiers so in case we would need to get all lines for given points we would just go through the lines table an select only those lines which have ID of the point. It was revealed that this approach is quite slow since each line can be associated with up to 2000 points and it would take $O(N \times M)$ time complexity to get all lines where N is the size of the lines table and M the length of the array points. The second approach was to store lines identifiers inside points table, which resulted in significant improvement and we could access all lines for $O(N)$. In combination with Pandas library [21], which is used for storing lines table in memory, method with storing lines in points table results in even more considerable speed

improvement. On the other hand, the points dictionary grew in size and started to occupy more than 8 Gb of the disc space. To overcome this issue the shelve [22] data type was chosen. Shelve is a simple file-based database with object-like interface, which is fully compatible with pickle. Thus without serious code modifications the memory usage was dropped from 8 Gb to 200 Mb.

5.1.5 Transformation to Vector Tiles

Once the data is transformed to convenient form, the next step is convert it to vector tiles. For this purpose the tippecanoe [23] tool was selected which is developed by Mapbox team as open source project. The drawback of this instrument is that to generate tiles with several layers one would need to have separate GeoJSON file for each tile layer. In our application we will need 128 tile layers to render all weights, colors and also apply filtering. Thus, we first need to generate 128 GeoJSON files for each point. Then we could easily convert set of GeoJSON files to tiles.

5.1.6 Results

The resulting preprocessing scheme is demonstrated on Figure 12. First, we transform raw data to intermediate state which is shelve of points with links to lines and table of lines with properties. Next, for each point we generate a folder which contains GoeJSON file describing set of lines associated with given point (all directions from one point on the grid to others), then GeoJSON files are transformed to vector tiles. In course of the preprocessing there were other approaches, for instance when rendering was not utilizing vector tiles, but just GeoJSON files the was improvement which allowed merging adjacent lines together to reduce the size of the occupied space, but once the the implementation started to be based in MBTiles lines merging was rejected. It is important to note that initially the tiles generation was planned to implement in real time, but it was discovered that real-time processing takes significant amount of time which results in delays in server responses.

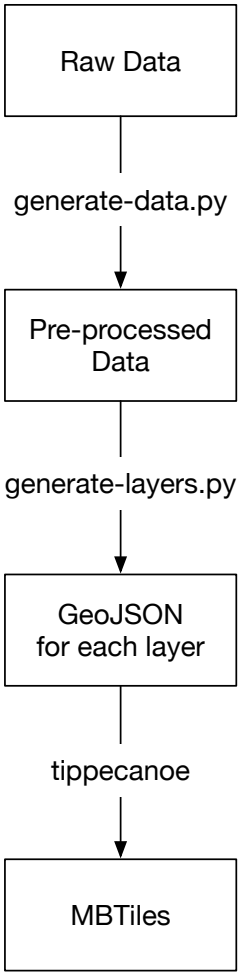

Figure 12. The process of the data transformation.

5.2 Web Interface

In this section the main architectural pattern used for building the interface is described including diagrams showing hierarchy of components, the selection of map drawing library is made.

5.2.1 FLUX Architecture

In the process of the development of the web interface the crucial thing is the state management. In 2010 the Backbone.js [24] library was introduced which was supposed to solve the problem of the state management often applying MVC [25] pattern. Although, on large projects there where a lot of cross-dependencies this paradigm resulted to be hard to maintain. Later in 2013 Facebook Inc. introduced component based React [26] library,

which allowed to think of the interface as function which maps state to representation. Although, React was implemented only as library for creating views and was not meant to solve problem of data state management in the application it still had significant success. In 2014 Facebook Inc. suggested FLUX [27] pattern which gained significant popularity. The idea behind FLUX can be described as follows: data comes from the store to the view which renders it, the user can fire an action, for example by pressing a button, the action is goes to the dispatcher which tells to all of the stores that certain action was fired. The stores modify the data and broadcast it to all of the views which are subscribed to this store. All of the views that received new data gets re-rendered. The illustration of this process can be seen on Figure 13.

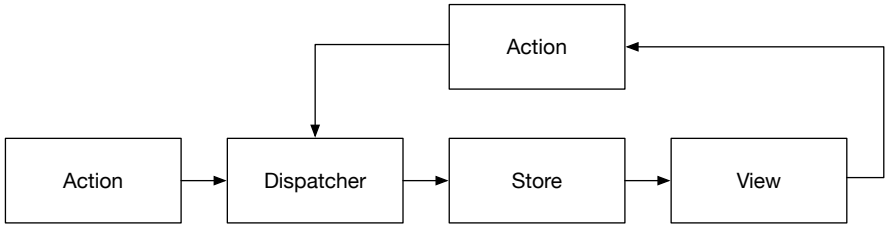

Figure 13. FLUX architecture.

Eventually FLUX evolved in library called Redux [28], which introduced few important modifications such as suppressing dispatcher, using pure functions for data modifications, using only single store where all of the data is accumulated, introducing middlewares for managing side effects. All this features create easier debugging experience and clear separation of concerns. Thus, for the development Redux and React libraries were chosen.

React components were organized accordingly to Figure 14 accordingly to the approach of presentational and container components [29]. The main presentational component is `App` which is the only one connected to the store directly. All other components are "dumb" and receive data as well as actions which are needed to be fired from `App`. The `SpeedMapButton` represents a switch for choosing between coloring lines by speed and by the type of transport. The `Map` components accumulates all of the logic related to drawing maps. The `TransportFilter` shows what kind of transport should be displayed on the map. The `LocationSelector` shows information about selected location. The `ModeSelector` switches between global overview and location information. The positioning of all of the components is illustrated by Figure 15.

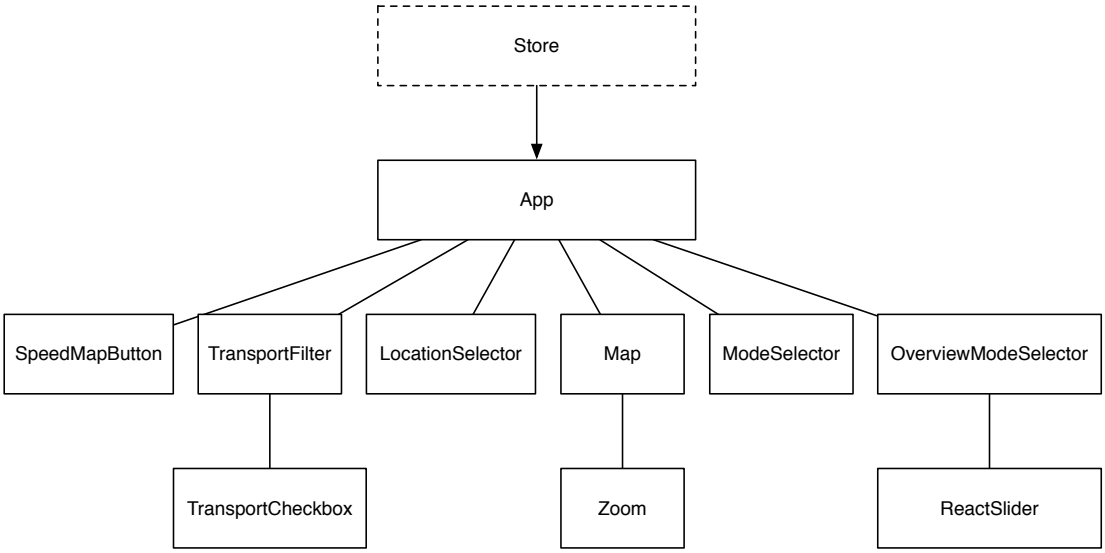

Figure 14. Structure of the components

5.2.2 Drawing Maps

As it was described before there are several the most famous libraries for drawing maps such as Yandex Mapx, Google Maps, Leaflet.js and Mapbox GL JS. To determine which library will suite better for our program, the needed functionality should be defined. The most important features are:

1. Panning and zooming.
2. Support vector tiles.
3. Rendering map from external data source.
4. Support of the vector tiles styling of such properties as line widths, line colors.
5. Ability to render icons at certain location.
6. Support for map styling.
7. Uses open standards.

Among all of the libraries the most suitable is Mapbox GL JS which supports all of the listed features. Although some part of the free Mapbox toolkit has limitations, for example free subscription plan allows not more than 50000 map views per month, but for

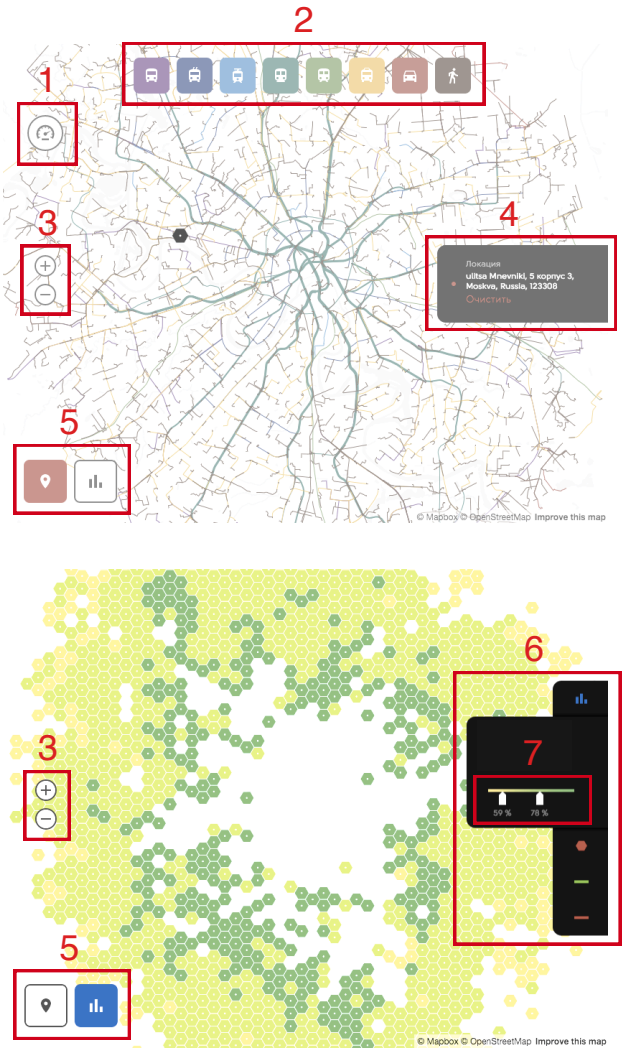

Figure 15. Components positioning. 1 – SpeedMapButton; 2 – TransportFilter; 3 – Zoom; 4 – LocationSelector; 5 – ModeSelector; 6 – OverviewModeSelector; 7 – ReactSlider.

our project that was enough. Important to note that Mapbox uses OpenStreeMap for geographical data provider which is free to use. As for Google Maps and Yandex Maps, they were not supporting maps from external data sources which is needed for rendering custom vector tiles. Although Leaflet is a very popular choice for map drawing, it is not suitable for our particular application since it does not support vector tiles rendering.

5.3 Tile Server

For rendering all of the directions the tile server would need to be chosen. The tasks in our application for this component are quite trivial:

1. Fast.
2. Support of the MBTiles format.
3. Easy to setup and configure.
4. Free and open-source.

The great candidate for this role was Tessera tile server [30] which is written in Node.js and based on Tilelive interface (developed by Mapbox). Moreover, Tessera is really easy to install using npm package manager. The configuration file should be written in JSON format and can be easily generated from our data. The only needed parameters are source path to the data and the URL on which particular vector tiles will be available.

6 INTEGRATION

Development of the software is very related to the problems of the distribution and setting up development environment. In this section the methods which were used for building client code will be described. Another section will be devoted to the deploying process.

6.1 Development Environment

The tools on the client side are developed with enormous speed. If previously the

6.2 Building a Project

6.3 Deploying a Project

7 CONCLUSION

REFERENCES

[1] GRASS GIS: General Overview. https://grass.osgeo.org/documentation/general-overview/. [Online; accessed 01-May-2016].

[2] Google Maps JavaScript API. https://developers.google.com/maps/documentation/javascript/. [Online; accessed 28-April-2016].

[3] Leaflet – an open-source JavaScript library for mobile-friendly interactive maps. http://leafletjs.com/. [Online; accessed 28-April-2016].

[4] Maps API – Yandex Technologies. https://tech.yandex.ru/maps/. [Online; accessed 28-April-2016].

[5] Render Mapbox styles in the browser using JavaScript and WebGL. https://github.com/mapbox/mapbox-gl-js. [Online; accessed 26-April-2016].

[6] 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].

[7] The gzip home page. http://www.gzip.org/. [Online; accessed 01-May-2016].

[8] Protocol Buffers. https://developers.google.com/protocol-buffers/. [Online; accessed 01-May-2016].

[9] Geobuf. https://github.com/mapbox/geobuf. [Online; accessed 01-May-2016].

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

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

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

[13] Google Maps JavaScript API: Browser Support. https://developers.google.com/maps/documentation/javascript/browsersupport, 2016. [Online; accessed 20-April-2016].

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

[15] Google Maps Directions API. https://developers.google.com/maps/documentation/directions/, 2016. [Online; accessed 20-April-2016].

[16] World Geodetic System. https://en.wikipedia.org/wiki/World_Geodetic_System, 2016. [Online; accessed 20-April-2016].

[17] Google Maps Directions API: Encoded Polyline Algorithm Format. https://developers.google.com/maps/documentation/utilities/polylinealgorithm, 2016. [Online; accessed 20-April-2016].

[18] Geohash. https://en.wikipedia.org/wiki/Geohash, 2016. [Online; accessed 21-April-2016].

[19] python-geohash 0.8.5. https://pypi.python.org/pypi/python-geohash/0.8.5. [Online; accessed 21-April-2016].

[20] 12.1. pickle — Python object serialization. https://docs.python.org/3/library/pickle.html. [Online; accessed 25-April-2016].

[21] Python Data Analysis Library. http://pandas.pydata.org/. [Online; accessed 21-April-2016].

[22] 12.3. shelve — Python object persistence. https://docs.python.org/3/library/shelve.html. [Online; accessed 25-April-2016].

[23] Build vector tilesets from large collections of GeoJSON features. https://github.com/mapbox/tippecanoe. [Online; accessed 25-April-2016].

[24] Backbone.js. http://backbonejs.org/. [Online; accessed 25-April-2016].

[25] Client-side MVC with Backbone.js. http://www.slideshare.net/iloveigloo/clientside-mvc-with-backbonejs. [Online; accessed 25-April-2016].

[26] React: A Javascript Library for Building User Interfaces. https://facebook.github.io/react/index.html. [Online; accessed 25-April-2016].

[27] Flux: Application Architecture for Building User Interfaces. https://facebook.github.io/flux/. [Online; accessed 25-April-2016].

[28] Predictable state container for JavaScript apps. https://github.com/reactjs/redux. [Online; accessed 25-April-2016].

[29] Dan Abramov. Presentational and Container Components. https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0#.2vyf9ikcr, 2015. [Online; accessed 26-April-2016].

[30] Tessera: A tilelive-based tile server. https://github.com/mojodna/tessera. [Online; accessed 26-April-2016].