Internship Report

Optimizing the transport network of a large metropolitan area to improve citizen accessibility

Participant:

- Zaika Vladyslav

Supervisor :

- Nicolas Nisse

Université Nice Sophia Antipolis

POLYTECH NICE-SOPHIA

Abstract

Today world is growing faster and faster every day. New cities, suburbs, streets appear everywhere. And the problem of building optimal and quick paths from one place to another becomes more and more complex. For this project we consider to study a large metropolitan area (Santiago, Chile) to improve transport system of this megapolis. First we examine what we have, our group has access to numerous data about the city : transportation system, education system, health system. We need to clean this data from errors and create good structures to work with. Also we will describe main metrics that we use for measuring the accessibility of basic services (health, school, police buildings) and predict the most suitable regions to build new ones. Then we will introduce some solutions that improve accessibility in this city by proposing a way for calculating best direction between two points and proposing good locations to place new important services like schools, police dept. and others. The sizes of data is huge, in this report we'll give a clean view how we solve a problem with processing big data amounts.

Table of Contents

1. Intro

In this project we will dive into work with big data amounts and huge graphs. For our work we consider to examine large metropolitan area of Santiago city. We have a connection with Adolfo Ibañez University in Santiago, so they provide all important data to work with.

Main part of the internship will cover cleaning the data from errors, choosing the correct data structures for faster work and formalizing the data. We'll show 3 main types of errors that we faced with in this project and the ways of handling them. After the first stage we'll get errorless, preprocessed graph (almost all the parts will be connected). We propose a special technique that helps to fix lost connections in graph that can't be fixed automatically by implementing special algorithm (combining automatic and manual fix).

At the end of section six we will errors in data and show the obtained results. After that we will integrate our graph into SAGEMath. This module will provide us clean representation of what we have and also provide clean proofs of error handling correctness.

Section eight will show the main ideas that was proposed for shortest path calculation on this graph, we will test them and compare with traditional solutions, but before doing that we examine related work for this project in section three.

Section nine will give ideas connected with optimization of transport system in examined metropolitan area. We will show our techniques that used for selecting the most suitable places for building important objects like hospitals, police dept., schools and others.

In the end we will give a conclusion what we have done, and what we are planning to do. Summary will cover the results achieved during the internship work, show how it helps and simplifies the work with ours graph (Santiago city map) and explain how to postpone the algorithms to similar graph.

2. Describe the problem

The topic of my internship is « Optimizing the transport network of a large metropolitan area to improve citizen accessibility ». When we have a first meeting we start discussing what exact problems exist in this topic? Why we need to optimize? For this project we will examine Santiago city. It is the capital of Chile and the biggest city in that area. It has long suburbs and very complex transport network. The population of this megapolis grows up and a mission of getting work every day and back in reasonable time becomes more and more complicated. Also there is a big problem of accessing existing and locating new basic services in this city like schools, hospitals, police offices. We have a co-working team that located in Santiago. They provide as a big dumps of row data that describe Santiago city roads and infrastructure.

We have a big problem that hides in data. All the records in databases were inputted manually by group of workers during a couple of years. It's mean that it has a lot of data repetitions, bugs and inaccuracies.

In this work, we deal with the following problems:

- Normalizing the data grid.
- Fixing errors in data.
- Proposing the algorithm for shortest path computation.
- Introducing the algorithm for placing basic services.

3. Related work

Studying the related work on this topic was very important step. It shows main techniques how other researches solve problems of shortest path calculation, traffic jams and transport system optimization.

First article[1] faces with problem of congestion control of public transport in Santiago city. They show the strategy of bus route optimization that gives us optimal holding time that minimizes user-waiting times on bus stop. The results of this work will help us to design new bus lines and reduce bus bunching via mathematical programming models. This article gives me understanding of basic principles of public transport route optimization and idea how to implement them in our project.

Next we study document[2] that solve the transportation problem of Berlin city the results give incredible increase in revenues by 22 %. From the article authors show how changing of frequency in main lines and suburbs improves the total utilization and travel times of passengers. We will use techniques from this article for our algorithm of bus route planning to improve passenger's utilization.

The third paper [3] gives us a clean view how researchers work on problem of urban area transport optimization. From the document we can see the techniques for evaluating of public transport networks and then how to use heuristic algorithms for improvements in transportation network. It give us ideas for algorithm optimization in future work.

Research study[4] work with problem of bus network optimization, authors transform network into pure grid and show that optimal bus routes is sensitive to demand distribution in area. The results of this work will be useful for designing new public transportation networks and also giving evaluation to existing ones.

4. Describe the data

Let's introduce basic definitions that will be used in our work. *Graph* is an ordered pair G = (V, E) comprising a set V of nodes or points together with a set E of edges or lines, which are 2-element subsets of V. We consider *Nodes* as featureless and indivisible objects and represent a unique point on map (i.e. crossroad point). Pair of *nodes* creates an *edge* that is represented by a street fragment. In our project we will consider only directed *edges* (ordered pairs of *nodes*), the direction of particular *edge* is set by *nodes* position: from start *node* to the end *node*. *Record* is a row from our dump that represents and object of *node* or *edge* on graph.

We receive 3 different files that gives us full information about transportation network in Santiago:

1. EJES – this file represents the edges in our graph
2. ALLPOINTS – in this file we have representation of all nodes in our graph
3. ENDPOINTS – in last file we have also nodes that create a backbone, it is mean that ENDPOINTS is a reduced version of ALLPOINTS, we keep only start and end node of each street and crossroad nodes for each street.

Let's consider each file more precisely and examine what we have inside.

EJES file represents connections between each two nodes in our graph. It has 209 thousands of records. From the file we can get following information about the edge from graph:

- OBJECTID – unique identifier for each edge
- FENAMEID – many-to many relation, that connects to nodes and represents unique street on map. Each street can consist of many edges.
- DIR – stores information about street direction, can have 3 different options :
    1. 0 – bi-directional street
    2. 1 – direction from start to end of the street.
    3. -1 – direction from end to start of the street.
- Hierarchy – this metric describes the level of the road. The value could be from 1 to 5, where 1 is a highway and 5 is a country road. We should say, if we have an intersection of two edges with hierarchy of one and different, they will not create a crossroad (it is impossible to cross a highway by suburb street). This parameter is very important and will be widely used in future computations.
- Id_eje – primary key of this table
- Shape_length – the length of the current street.

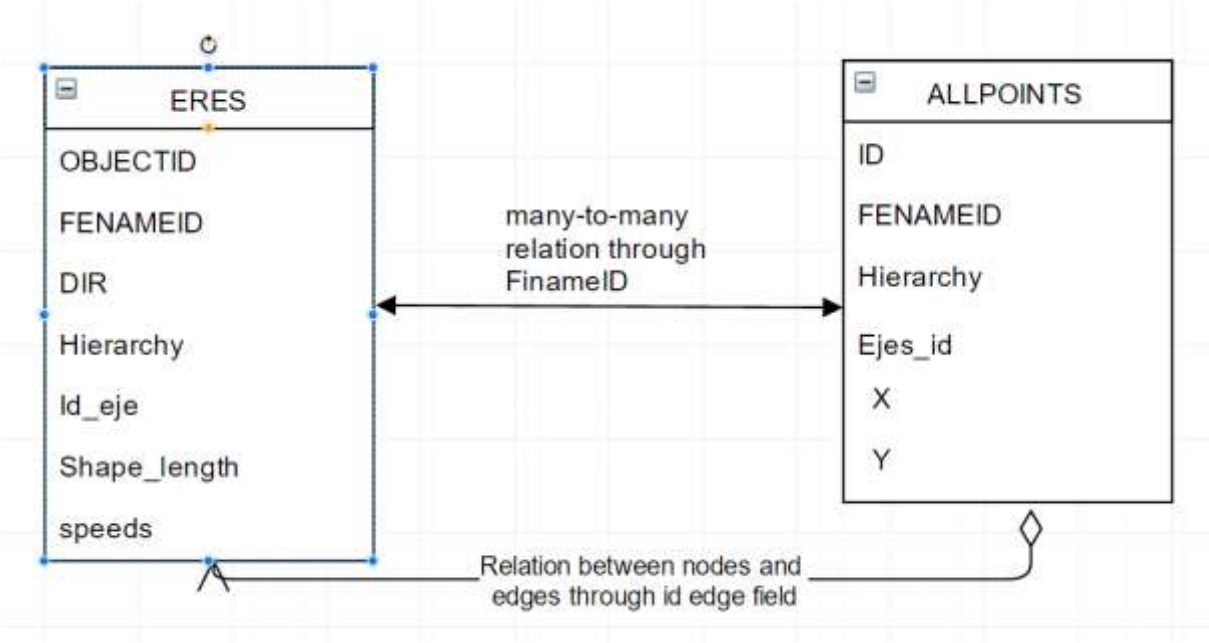
(Next attributes will describe speed limits for edges, they are constant for all the edges.)

- V_libre – maximum allowed speed limit for this road
- V_peak – speed on traffic jam.
- V_valle – normal speed during the day

- V_cam – walking speed of pedestrians.

Next let us examine the ALLPOINTS and ENDPOINTS files. Both of them have the identical structure, but ENDPOINTS has 419 thousands of records, ALLPOINTS has more than 600 thousands of records. From these files we retrieve information about:

- Id – primary key for this table.
- FENAME_ID – many-to-many relation that connects with EJES table and represents a unique street on map.
- Hierarchy – represents hierarchy of this street (the same from ERES).
- Ejes_id – foreign key that connects particular node with edge.
- X – represents x coordinate of node.
- Y – represents y coordinate of node.



Img 1. Relation diagram of initial data

We get this not very pretty database with redundant relations between records. But exploring deeply the data, we find even more fundamental problems in our data that makes impossible to continue work with this data:

1. Fename_Id relation actually relate to nowhere. After some tests we make a decision that this parameter in some streets gives totally opposite values that it should be.
2. Repetitive data. The databases are totally not normalized, so we have a lot of data repetitions that just slow down the computations.

3. Unique ids of edges stop being unique after some point.

The main reason why we have all this kind of problems is because of the process of initial data collecting. It was done in this way: worker just drives through the Santiago streets and manually input all the data and EVEN primary keys were inputted manually.

From the other hand we have more serious problems like data repetitions or lost connections between nodes (we don't have relations between nodes where they should be).

So, we made a decision to develop new structures for our data.

5. New data formats

In this chapter we'll introduce new data structures and start moving to fix errors in data. We are trying to make our data clean and simple. We should get rid of useless parameters, structure and normalize our datasets. From previous chapter we have two separate tables that represent edges and nodes in our data. Both of them are used to represent the graph.

Our proposition is to reduce data amounts to work with. First let us introduce a representation of a single node in our graph by instance of class *Point*, it has fields:

- Id – primary key.
- X – X coordinate of node.
- Y – Y coordinate of node.
- Edge_id – foreign key to the edge corresponding to this node
- Cell – is a number that represent a sector on map, where our node is situated. We will describe them in this chapter later.

For comparing of two distinct nodes we use hash functions, which helps us to determine if the two nodes are actually one. For doing that we apply hash function to their x and y coordinates:

$$hash(node_1.x, node_1.y)? = hash(node_2.x, node_2.y)$$

We use this idea to refuse the idea if ids and compare two node objects for equality we just use coordinates of particular node

For representation of edges we introduce *Segments*, each segment gives us a relation between just two different nodes, it has very simple structure:

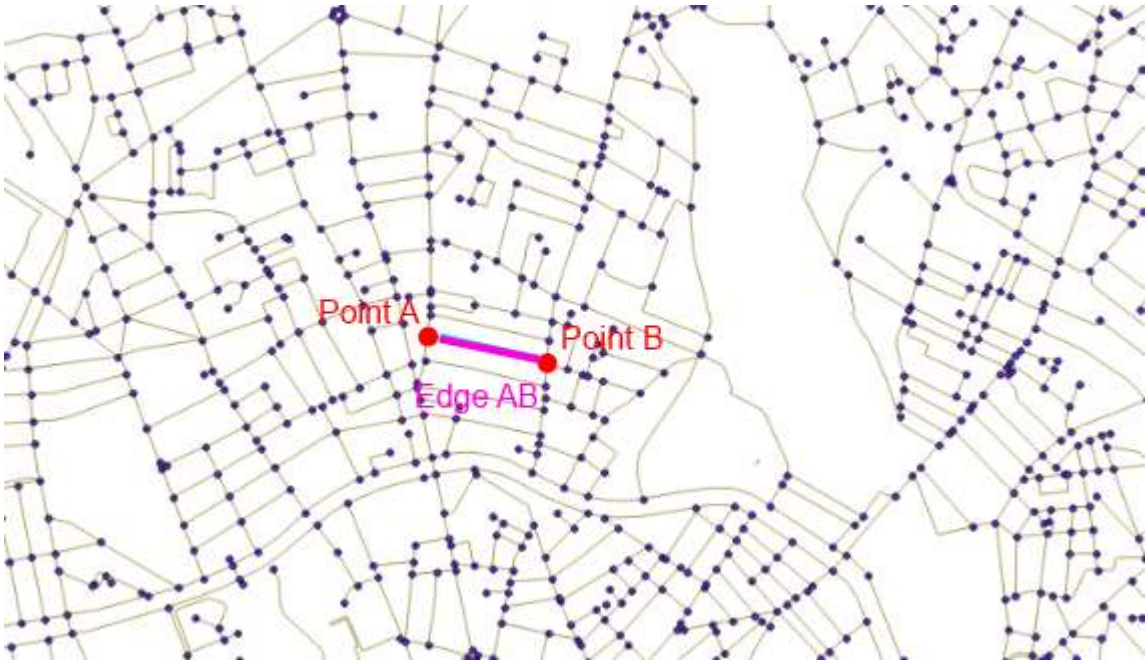
- From – Point that start this segment
- To – Point that end this segment
- Length – length of this segment, for the clearance of experiments, we recalculate all the segments distances.

All the segments are unidirectional and goes from start point to end point (from and to parameters). After creating the segments, we create a special dictionary that stores hash of each node as a key and a list of all possible segments connected with this node as a value. This gives us rapidly fast data access (because of the usage of dictionaries) and clear data structures.

Also we keep Edges table, because it gives us explicit information about each edge related to nodes. But we made normalization of this table by separation of information about speed limits and edges data into different tables. The structure is following:

- Id – as a primary key.
- Hierarchy - to represent edge hierarchy
- Direction – the direction of current road
- Street_id – unique street identifier

Edges are also represented as a dictionary with a key as id of each edge. To create a relation in bidirectional street, we just create two different segments. One for one direction and another for the opposite.



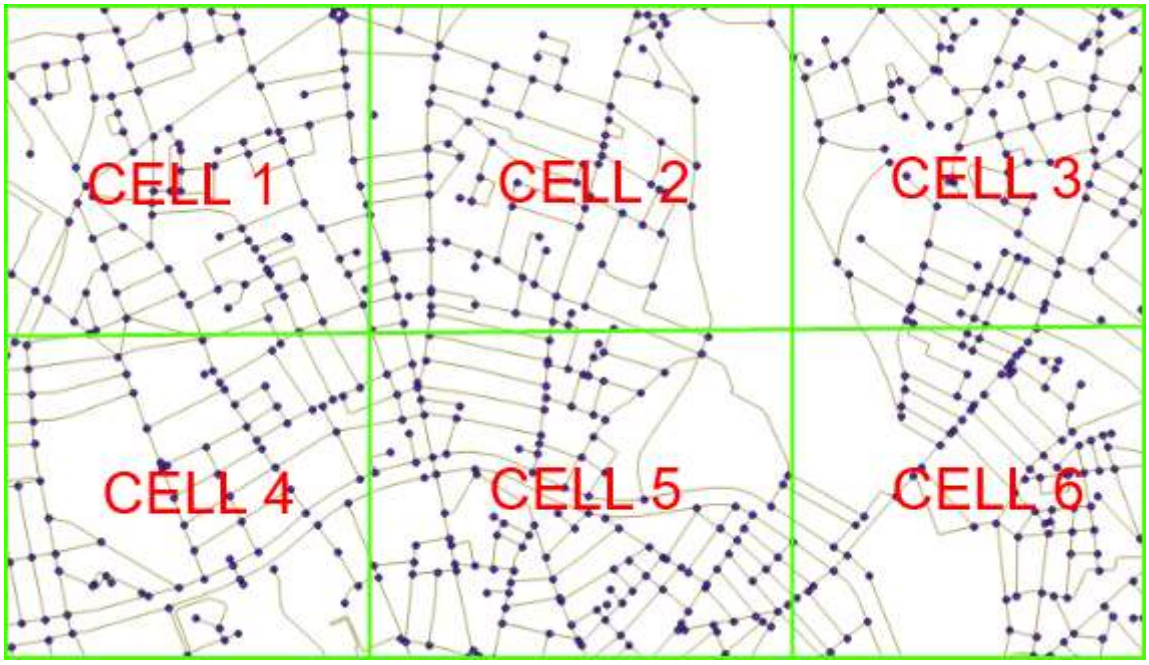
Img 2. Part of Santiago graph

Let's look to the Img. 2. On this image we can see a part from our map with an example of new data structures. So each violet point represents a unique node on map and each line an edge. Edges link only to two points from both sides.

Now let me explain the first difficulty we face with. As you remember we have a quite big graph. It starts with 630k of nodes and 210k of edges in raw data. From our side its mean:

- Very big amount of data
- Difficult to proceed
- Hard to update

To deal with it let me introduce algorithm that will allow us to deal with existing problems. We call this method map segmentation. To make the process more clear let us first look into Img. 3.



Img. 3. Segmentation of Santiago graph

The idea is simple. First we calculate the size of our graph using Euclidean distance algorithm. Then we separate our graph into thousands of cells. The size of the cell was selected by practical tests. We come to decision that 100K cells is the best metric for our graph in terms of correctness of work end speed. Each node from our graph referenced to particular cell by cell_id attribute. This separation helps us to divide big graph into many subgraphs, now we can proceed each of them separately and also use in particular algorithms for routing in our transportation network.

6. Error classes

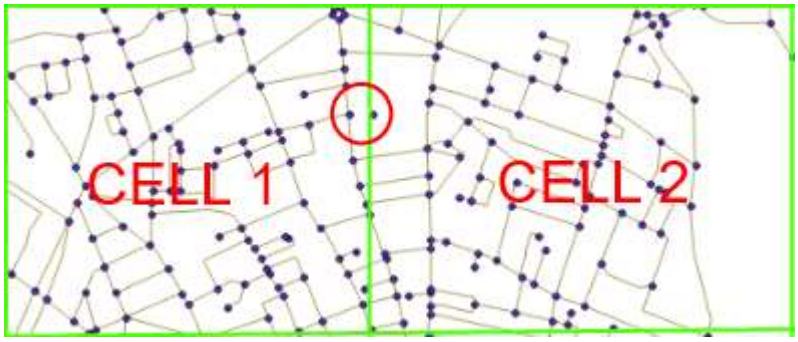
Finally we structure and formalize our initial data. But we did not solve other problem: Data errors. After examining the graph on map and computational tests of streets and edges connections we come to idea that not all relations between nodes are set correctly and not all nodes have proper coordinates. We determine three main error types in data:

1. Near Points errors.
2. Intersection errors.
3. Near segments errors.

For each error type we develop specific algorithm that will be described in this chapter. But first we want to introduce general idea of handling the error situations. Each error case represents failure when inputting the data by worker, it's mean that one node can be repeated several times, or there is no node in edges intersection or other. If we process the initial graph, it will take a lot of time (around 15m just from initial loading), resources and still be slow. So we'll use segments from previous chapter to process the grid. The general algorithm will consist from the next steps:

1. Iterate through each cell from graph.
2. Check the cell for error situations.
3. Fix error case.

The idea was good. But after practical tests we find out that we did not take into account all frontier cases.



Img. 4. Segmentation problem

Take a look at image 4. We have 2 different cells and in red circle you see two different points that should have a relation but actually they don't. Naturally will be to detect this error case and to create the connection. But because they are in different cells it was impossible.

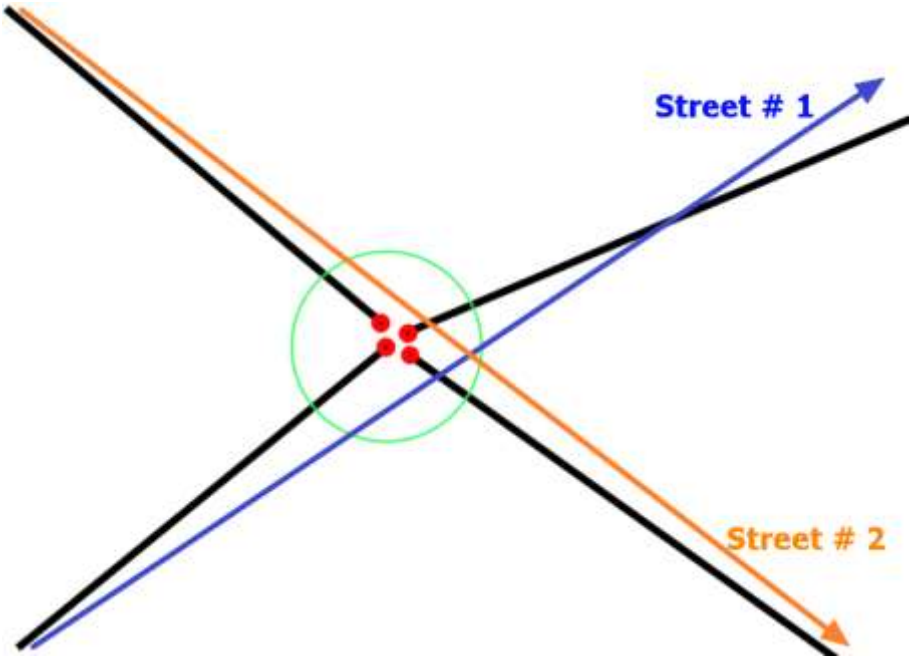
The idea how to deal with it is to proceed not only the central square but also with neighbors. To illustrate this take a look at image 5.

Img. 5. Map processing algorithm

Here we want to examine red cell. But to prevent loosing errors on edges we will also consider all his neighbors (cells from 1 to 8). And create a graph that consists of all nine cells. And do this algorithm for each cell. In this case we can be sure that all error situations will be determined and handled.

6.1 Near Points Error

Now we move to error classes handling. We will start form near points error. This is the most common error type (almost 400 thousand error cases). It can happen everywhere: in one street, in crossroad, in highway or in parking. And the problem that we have multiple nodes in area, where it should be only one. Let's examine an image 6 to get clear view how it's happening.



Img. 6. Near points error

We see two different streets 1 and 2. And they create a crossroad, but actually not. Here we have 4 different edges with 4 different nodes and no connections between them. Possible answers why we get this error type is:

- GPS failure.
- Human factor. (Don't forget that all the data are inputted manually)

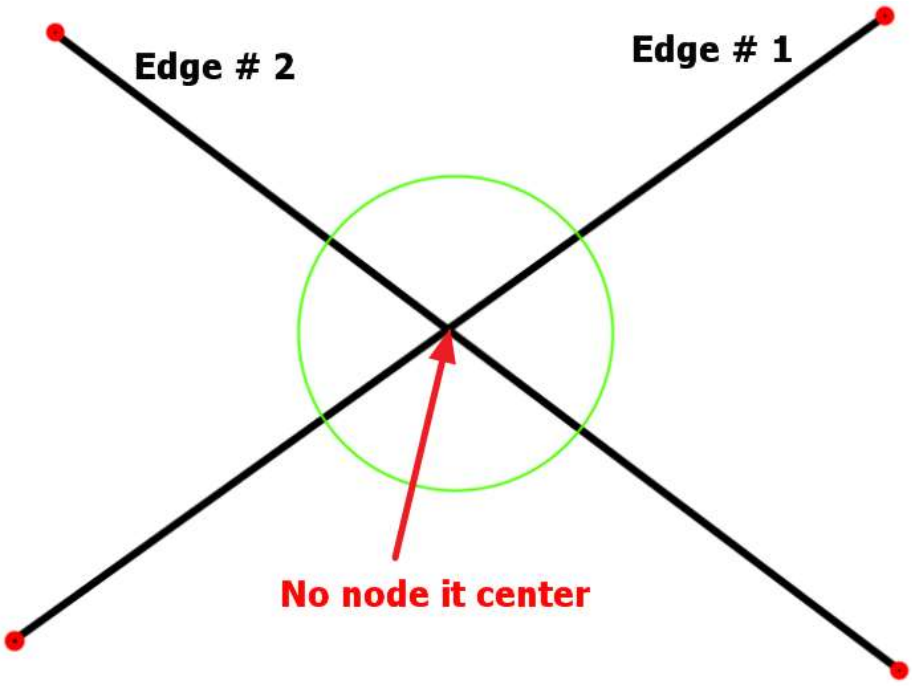
To fix near points error we develop *near point fix algorithm*:

- Get the list of neighbor cells and create a subgraph based on nodes from current cell and neighbors.
- Create all possible combinations of nodes in this subgraph.
- For each pair of nodes check the relation. They should be from the same street and have the same hierarchy. It's very important tip. Because we don't want to merge two nodes from different street that just pass by. Also we should check nodes for the hierarchy, because we don't want to merge for example highway node with parking under that highway.
- After that we calculate the Euclidian distance between two nodes, and if it's less than *merge radius* attribute we start merge them. *Merge radius* is a special attribute that determine the reasonable radius in which we can merge two nodes. On image 6 it is represented by green circle. The 'golden' merge radius was selected by practical experiments and is 1 meter. If we select a bigger radius, it will merge some small edges into one point, but if we select smaller radius we will omit a lot of error situations.
- For merging the points we use introduced in section four hash sums. Because comparison of two distinct nodes is based on calculation of X and Y coordinates hash sums, to merge two nodes we just need to set the equal coordinates. This coordinates are calculated as an average from X and Y coordinates of each point respectively.

After the near points error handling we test the graph and we get 211 thousands of fixed error situations. It helps us to reduce total number of nodes to 417 thousands and resume map connections.

6.2 Intersection errors

Next error type is intersection errors. It happens on street crossroad and in most cases connected with human factor. Workers forgot to place nodes in their places and we can get situation like this:



Img. 7. Intersection error.

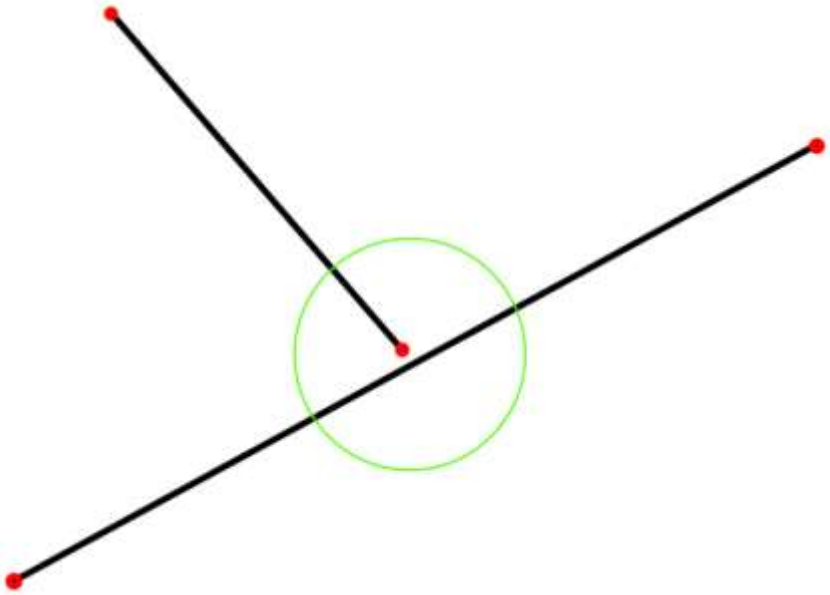
We have two different edges 1 and 2. They have the same hierarchy and should cross. For crossroad they should have a node in intersection point to split them. But there is not. To make data free from intersection errors we develop *intersection fix algorithm:*

- For current cell get all the neighbors and create a subgraph.
- Inside subgraph distinguish all the segments (unique edges).
- Create all combination of edges that is part of different streets and has the same hierarchy.
- Calculate the presence of intersection.
- If we have an intersection, create a new *Point* and split the edges 1 and 2 into 4 different edges.

Correction of intersection error classes give us near 2 thousand of fixed situation and make our graph more accessible by adding missing crossroads.

6.3 Near Segment Error

The last error class is near segment errors. This error case happens on street that are connected to another by one edge, but don't cross the street (like T-type crossroad). In the illustration we can see exact situation how it happen:



Img. 8. Near Segment Error.

From the image we can see that we have two different streets that crates T-type crossroad. The relation between node from street 1 and street 2 is not established, so we don't have direct access street 2 from street 1. To fix this we implement *near segments fix algorithm*:

- For each cell get all direct neighbors and build a subgraph on them.
- Create sets based on all edges and nodes from distinct streets.
- Create all possible combinations on prepared sets and check them for the same hierarchy.
- Calculate the distance from node to edge, if its closer then *merging radius* separate edge into two distinct ones and associate corresponding endings of them with node.

This fix gives us just 30 error situations, but this help us to finalize error handling and represents clearance of the final data set.

6.4 Finalize the data cleaning

Now we finish cleaning and normalizing the data. Small summary of achieved results. From the beginning we receive a huge dataset with a lot of repetitions, data collisions and errors. Applying our algorithms give us 200 thousand near points error fixies, around two thousands of added intersections and corrected segments errors. Finally we get a clean graph on 415,321 nodes that has:

- Clear and quick data structures with two-way access (Nodes -> Edges, Edges -> Nodes).
- Proper connections between objects.
- 213 thousands of fixed errors
- Easy to update and get.
- Normalization of dataset

Now we finish working with data and move to more interesting part. Making tests on our dataset and checking the algorithms for finding shortest paths and predicting the places to put important buildings.

7. Introduction in SAGE

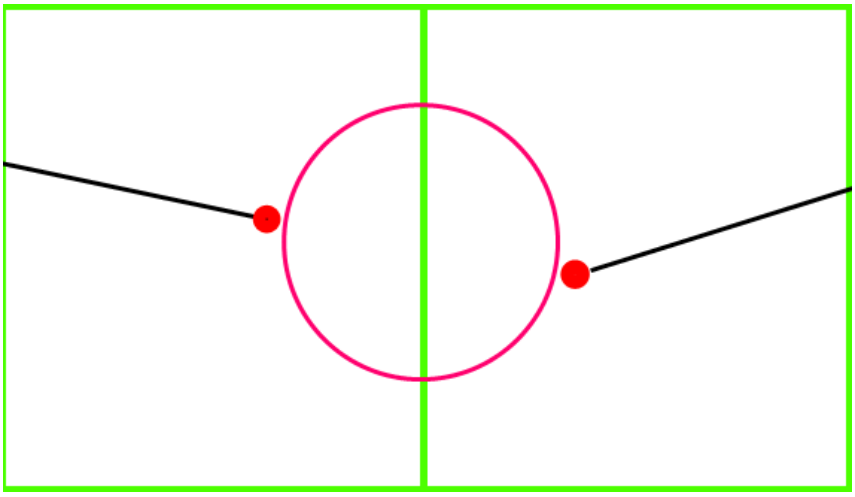
In next chapters we'll start presenting algorithms that was proposed by us to deal with main problems of this work. But before that we want to create a big graph with all the nodes and relations between them and collect different metrics that can be useful for us in future computations.

As you remember we have quite a big graph. And processing of big graphs means big problems for execution time and correctness check. Also we wanted to find a good tool for calculating different graph parameters and metrics. My supervisor propose to use *SageMath*[5] software. So the idea is to import *SageMath* package into our project and to convert our graph into *SageMath* graph. *SageMath* is mathematical software that was created by scientists from university of Washington and covers many aspects of mathematics including graph theory. *SageMath* give us full access to variety of graph algorithms and metrics that we can use. We determine the most important *SageMath* functions and attributes that we will use for our tests:

- Vertices – number of nodes in graph
- Edges – number of edges in graph
- adjacency_matrix – gives us the adjacency matrix for our graph
- average_degree – returns the average degree of the graph
- average_distance – returns the average distance of the graph
- blocks_and_cut_vertices – computes the blocks and cut vertices of our graph
- bridges – calculate total number of bridges in graph
- centrality – there are two options betweenness and closeness centrality for graph.
- is_connected – check if graph is connected.
- Connected_components – gives explicit information about connected components in graph
- Density – returns number of edges divided by total number of possible edges.
- Diameter – returns the maximum distance between two nodes

We import our graph into *SageMath* and start running the test functions. And almost all of them fail. But the most interesting parameter here is *connected_components_number* it gives us 307 connected components. Then we check the sizes of this connected components and what we get: [410693, 302, 297, 208, 108, 97, …].

So as you see we have one big subgraph that is on 410 thousand nodes and near 300 hundred of subgraphs smaller sizes.



Img. 9. Segments gap.

The most reasonable explanation why it so is this is also Near Points Error class problem but the distance between points is bigger than predicted by our merging parameter and because of that we did not connect these subgraphs together. On image 9 you can see particular case found by me on map how the gap between segments happens.

Generally the easiest way to continue the computation is to respect only the biggest component and continue all the computations on it (as we did first), but then we come to very interesting idea how to merge all this subgraphs into big one.

Now let us explain the main idea of subgraph merger algorithm:

1. We get a list with all border cells (remember the cell is a map sector with nodes and edges inside) of both subgraphs.
2. Search for the direct (touch each other from one side) border cells with neighbor subgraphs and if they are present mark them as a candidate for connection between subgraphs.
3. For each selected cell get list of neighbor cells considering only cells from the subgraph it belongs to and create a new subgraph on them.
4. Because *merge radius* parameter is useless (the distance to node is bigger) in this case we'll do a trick. In both cell subgraphs check edges for the same street ids. And if we find a match, its mean that we actually should have an edge in this place.
5. Find the endings of detected street in both cell subgraphs and create a new edge between them.

This algorithm help us to automatize the process of merging disconnected components and makes our graph connected.

8. Shortest path finding algorithm

Next big goal in our work is to calculate shortest paths between two nodes in our graph. In small graph you can just apply Dijkstra's algorithm and get the result, but our graph is built on more than 400 thousand nodes, so we need to find a more clever way to perform this.

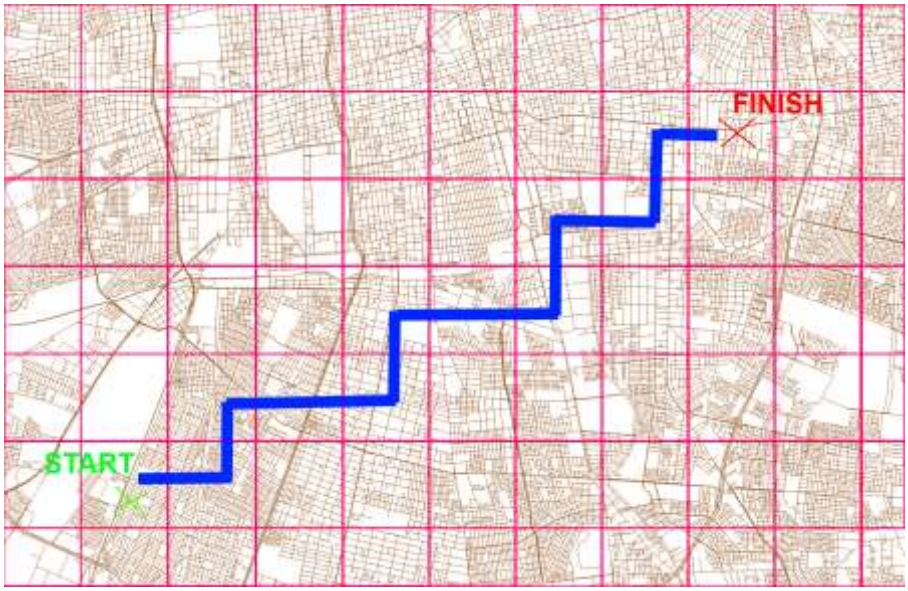
First algorithm we want to propose lets name as *fast cell path finding.* The main idea of this algorithm is to use segments for our graph and processing them independently.

Fast cell path finding algorithm starts from preprocessing phase. In this phase we determine each cell as independent objects and create something like a blackbox from each of them. The preprocessing algorithm has next steps:

- For each cell in graph we create a subgraph based on nodes
- Then we get a list of all 'boundary' nodes (its mean all the nodes that has connections to another cells).
- And calculate the distance matrix for boundary nodes in the subgraph.

We should also mention about weights in our graph. You remember that after creating a new data structures we recalculate the distances for each edge. Naturally will be to use this parameter as a weight of one edge in subgraph. But it's not the best metric. From initial data for each street we also store information about speed limits. Time that you spend to move through the edge is: *traveling_time = edge_length / speed_limit[10].* And now we propose to use traveling time on edge as a measure of weight in graph. Also it's good for dynamic path updates during the day, because of speed changings during the morning and afternoon traffic jams (it will update automatically, because we have different speed limits for each street depending on part of the day).

The main part of algorithm. We select two points (nodes) where to start and to finish. Next we look at our map as a matrix. Each element of matrix is one cell from map. In first version we decide to use A*[6] algorithm with Euclidean heuristics but with the rule don't cross the corners of the cells to find the shortest distance. And we build the path from the cell where we have the start point to the cell with finishing point.



Img. 10. *Fast cell path finding* algorithm.

From the image you can see exact work of the algorithm. We create a path *start -> finish* represented by blue line. Than we are going to second phase: *Cell path routing*.

As you can see from the image, for easiness of computation we do not consider corner crossings. First let's take a look to the cells where start and finish points are situated. Because this points are not laying on the edge of the cell, we cannot use our blackboxies from preprocessing phase. And it is mean that we should proceed this cells manually. We will use A* with the same heuristics algorithm for any shortest path finding needs.

We consider the start cell of our path. To begin the route building we determine the position of next cell to our position and then calculate the path to nearest edge node of the cell.

Each next cell of the route we will consider as a blackbox with node-connectors on edges. The algorithm of work on intermediate nodes looks like this:

1. We determine the node on which we stop the route computation from previous step and mark it as *a current start node*.
2. Consider current cell on map to detect the position of next cell and border to which we should move.
3. Take calculated routes data from preprocessing phase.
4. Build path from *current start node* to the edge node with the next cell.
5. Extend existing shortest path to finish node by path from current cell and go to the next step.

When you come to the cell with finish point you should repeat the calculation as with start point. In runtime (during the main program execution) calculate the path from border node to finish node. After that finalize the computation and return the founded path.

It can happen that during the algorithm execution there is no direct relation between two cells, that we create cell extension algorithm to fix this issue. If there is no direct border edge between two cells we start considering neighbors of current cell to find the shortest path. We do it recursively, until the time we find the match. It can happen that we will emit the direct next cell and build the path through neighbor cell directly to the next cell.

Now we are in blackbox computation phase. After finishing the computations we will start our tests on selecting best heuristics and compering achieved results.

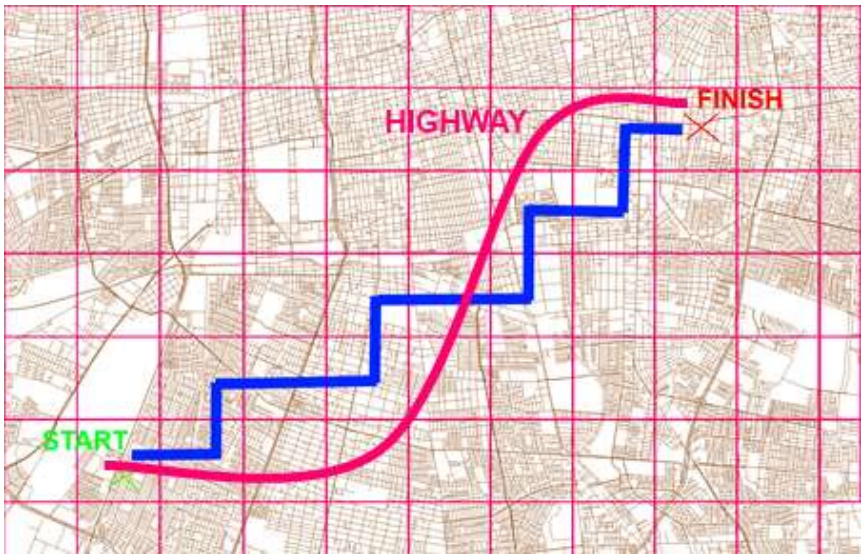
8.1 Highway Shortcuts

As an improving for *fast cell path finding* algorithm we propose to test nodes hierarchy separation technique. From the raw data for each edge we have special *hierarchy* attribute that plays important role in our graph. This attribute represents the level of the road (starting from highway until byroad).

The proposition is to split our graph into 4 different ones by hierarchy levels[7], then collect and examine metrics and propose improvement for existing algorithm.

The results give us the next proportion of nodes comparing to each other*: 1:2:2:4 (*this is the ratio of nodes number in each subgraph, built by different hierarchy, See Appendix 1*).* The smallest group is for hierarchy 1 who creates a graph of highways on our map it is only based on 25 thousand nodes. Than we have fast routes and big streets and more than 100 thousand nodes create a graph for small streets on suburbs.

The idea behind this segmentation is to provide alternative routes to main algorithm. Sometimes it can happen that shortest path on map is not optimal route (in terms of spent time). It happens because we respect only cells that lay on shortest path.



Img. 11. Highway shortcut

On image 11 we schematically draw a possible highway that lay from start to the end square but it is not laying on optimal path, so this way will be emitted.

For the algorithm we will consider 4 different graphs and made preprocessing phase as in initial algorithm. Then we check the approximate distance between start and finish nodes (Euclidian formula). If it's more than 5 km, it will be a good idea to use highway instead of regular roads.

First we consider the cell with start node. Depending on distance to finish point we will consider number of neighbors to proceed. We add one more circle of neighbors per each 10 km of way. Then we create a subgraph based on selected cells and start searching for the entrance to highway (nodes with hierarchy 1). And do the same with finish node. After that we use our subgraph based on nodes with hierarchy 1 to calculate the shortest path between two points. If path exist, we will propose it as alternative one.

Considering the highways we keep in mind that it's a problem sometimes to use them (they can be far, payed enter, few exits). Because of that we search path through highway only if the distance between start and finish points is big.

For the graphs of hierarchy 2 and 3 the algorithm will be different. If we can't build the path through highway, we'll try to build it only through edges with hierarchy 2 and 3. But these streets are big enough and can exceed the cells on the shortest rode, to fix it for each cell on path we will consider neighbors to expand the observed area and create a subgraph on selected cells based on graph with hierarchy 2 and 3. If the path exists, it will be also proposed as alternative one.

8.2 Evolutionary path optimization

Considering basic or even improved shortest path finding algorithms give us a good guess what can be a perfect route from initial point to destination. But in fact it happens very often that proposed path is not optimal or totally not acceptable for particular user.

We propose to create a special tool that will allow us to get feedback from user based on his experience on proposed route. It will be a mark that shows the correctness, simplicity and fastness of created path. Also we want to collect additional data from every trip that will help us to update the data and verify shortest paths.

To do that we should update our data structures. For each edge from our graph add an attribute called *passing time,* its's an automatically calculated attribute. It will represent the difference between predicted traveling times and actual time on edge (we will retrieve this data from users feedback). So in future path finding techniques we will take into account this value to correlate the optimal path.

After each trip user can give a feedback about the trip if it was good and well planned. He will put a mark from zero to ten. This mark will be recorded for each edge that was used in path building. For doing that we add new attribute called *customer mark.*

After 10 trips per edge we can make a new suggestion about it and recalculate the metrics of particular edge. Based on our observation, we can change the time that needed to pass the edge. And if the mark of particular edge is low (minimum than 5) its mean that this street is not handling the amounts of cars and we will set the new hierarchy of current edge to lower value. It will show us that the actually the maximum allowed speed cannot be reached and that we should avoid this edge because of traffic.

Still this work is related to future steps. For now we have a view how it should be implemented and pseudocode of algorithms for developing.

9. Placing administrative building on map

Calculating the paths in large metropolitan area is a big issue we faced with. The other one is creating the algorithm for placing important buildings like hospitals, schools, police departments and others. The algorithms for doing this was not developed yet, but I want give you general idea how we plan to deal with this problem. For each type of service you should calculate all the parameters individually, taking into account the specialization and costs. In my particular example I will describe the algorithm for placing new schools in area.

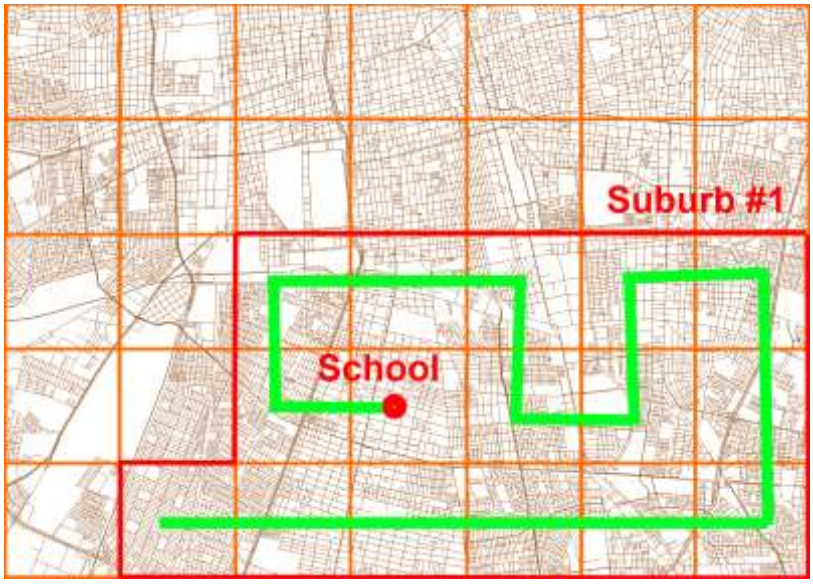
We will consider the cell of initial graph as a minimum unit for performing the calculation. When designing the algorithm for placing schools we should consider that the frequency of school placing is directly related to population in area. We will create 3 conditional zones on map that will represent city center, districts and suburbs.

For making the placement we should study the demand of particular area. As we don't have information about population in each region, we can create it based on number of nodes in area.

We will create cluster of cells for calculation. The minimum size is 5 square kilometers; this is the minimum area on which we can place schools one by one. The predicted parameter is to place a school in region that has more than 2 thousand of nodes. To place a particular school we should be aware from highways and roads with hierarchy 2 and 3. If it's not possible than go from bottom (search the place near road of hierarchy 3 and up).

Also we should consider already built schools. We have a dump with data structures that represent information about school system in Santiago. We will create a key-value storage with keys as a cell id's. Then after each calculated area for placing a new school we will check if we already have any school there and make a decision for placing new ones.

The frequency of placing schools in city center will be high. But for the suburbs to meet the predicted nodes number value we will cover very huge area. So it will be difficult for children to get to the school and back. The idea is to create a system of school buses for suburb area that will be save and functional.



Img. 12. Bus path.

Let's consider the particular suburb from image 12. To get all the children bus should drive through all the cells. To build the bus path we try to calculate the shortest circuit for driving through each cell exactly one time (or minimum number of times). The other point is that for calculating the bus path we consider only graph with hierarchy level 4 and 3 (for safety reasons).

As a conclusion we can say that this algorithm will help us to place near 200 schools. From that number almost 100 schools will be in suburbs so we will create a strong system of schools buses.

10. Future steps

Our project is not finished yet. We worked on formalizing and normalizing the datasets, fixing errors, researching and checking the algorithms that will be the most efficient for our problem this work was fundamental and should be done qualitatively. We start calculating shortest paths on our graph and blackbox processing.

For the future steps we can put next jobs that will effect on our data and paths by correlating fastest directions and updating blackboxes:

1. Finalize blackbox calculation for all the cells, now we still have errors in data, because of that some blackboxes are calculated not properly or completely corrupted.
2. Test the average time of shortest path calculation on map.
3. Study the cases when finding optimal path in graph is NP hard problem, find particular examples on our map and propose the way how to avoid them.
4. Develop online application that will help us to connect particular customers and our system. Start gathering feedbacks and data form them and test if evolutionary path optimization is an officiant way of updating the data.
5. Start work connected with social layer of grid (information about schools, police dept., hospitals). Apply algorithm of placing important objects on map to existing data and check if prediction of new places work properly.

11. Conclusion

As a conclusion of this internship report I can say that I like it. The problem we were working on relay to real people demands. And giving the solution of this problem will help many citizens to improve quality of their life.

We start the work by structuration of the initial data. Then we determine and correct three main error types: near point errors, intersection errors and near segment errors. After that we do a research and propose our own ideas of the most suitable shortest path finding algorithm for our graph. The graph was big so we separate into many subgraphs by hierarchy levels and sectors it give us many independent subgraph that can be used for building shortest paths. The next step was calculating the blackboxies (paths between all border edges for each cell) and testing initial shortest path algorithm.

The results that we get during graph processing and shortest path calculation give us clean ideas of correctness of our work. Each step of data processing gives us more clean and correct graph, each calculated shortest path help us to so improve and fix existing errors in algorithm.

The next idea is to develop a real time application that every citizen could use in their everyday life and collect the data from shortest paths that customer builds. This information will give us the most relevant data about particular edge or street.

Also we should mention that this algorithm is universal and can be applied to any city graph using the special intermediate layer that put the data into readable for the application data structures.

Acknowledgment. I want to give thanks to my supervisor Mr. Nicolas Nisse, he helps me a lot during this project for finding the related work, examining the data and the algorithms and to taste the researcher's work.

12. Bibliography

1. Bus Control Strategy Application: Case Study of Santiago Transit System. SEIT-2014. ANT-2014 (Pedro Lizana, Juan Carlos Muñoz, Ricardo Giesen)

2. More Passengers and Reduced Costs—The Optimization of the Berlin Public Transport Network. Journal of Public Transportation, Vol. 11, No. 3, 2008 (Tom Reinhold, A. T. Kearney GmbH)

3. Evaluation and optimization of urban public transportation networks. European Journal of Operational Research. Vol. 5, Issue 6 (Christoph E. Mandl, 2003)

4. Optimization of Bus Route Planning in Urban Commuter Networks. Journal of Public Transportation, Vol. 6, No. 1, 2003 (Steven I-Jy Chien, Branislav V. Dimitrijevic)

5. SageMath software (www.sagemath.org, William Stein, the University of Washington, 2005-now)

6. Engineering and Augmenting Route Planning Algorithms (Dissertation, Universit¨at Fridericiana zu Karlsruhe, Daniel Delling, 2009)

7. Using Multi-level Graphs for Timetable Information in Railway Systems. ALENEX 2002, LNCS 2409, pp. 43–59, 2002. (Frank Schulz, Dorothea Wagner, Christos Zaroliagis)

8. Route Planning in Transportation Networks. Updated report from MSR-TR-2014-4 (Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Muller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, Renato F. Werneck 2015)

9. Highway Dimension and Provably Efficient Shortest Paths Algorithms. Microsoft Research (SODA '10 Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms, Pages 782-793, Andrew V. Goldberg, 2010)

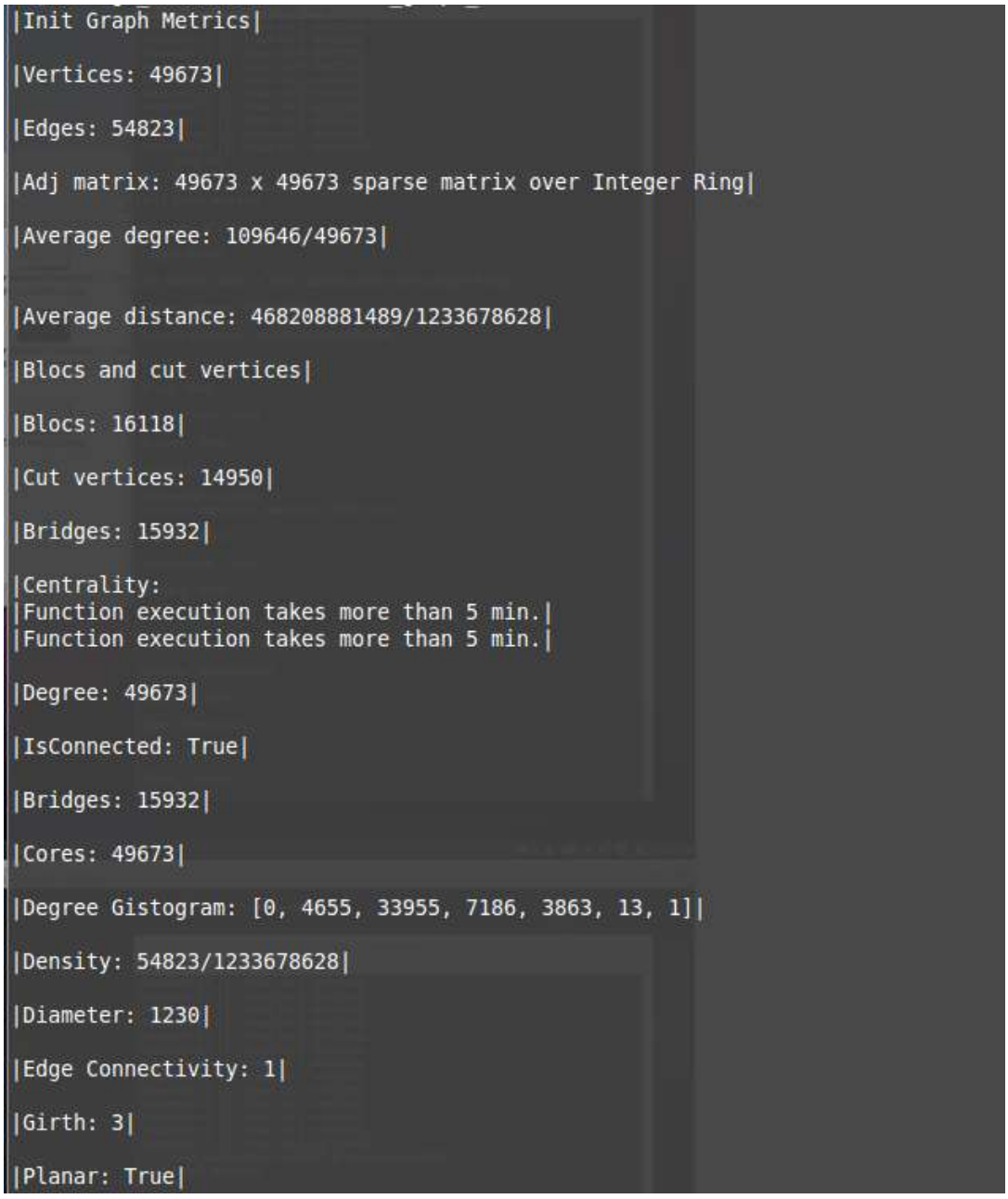
10. Shortest Paths in Graphs (tr. Кратчайшие пути в графах) (Н.Н. Писарук, Yandex, Minsk, 2014)

13. Appendix 1. Sage Tests Results.

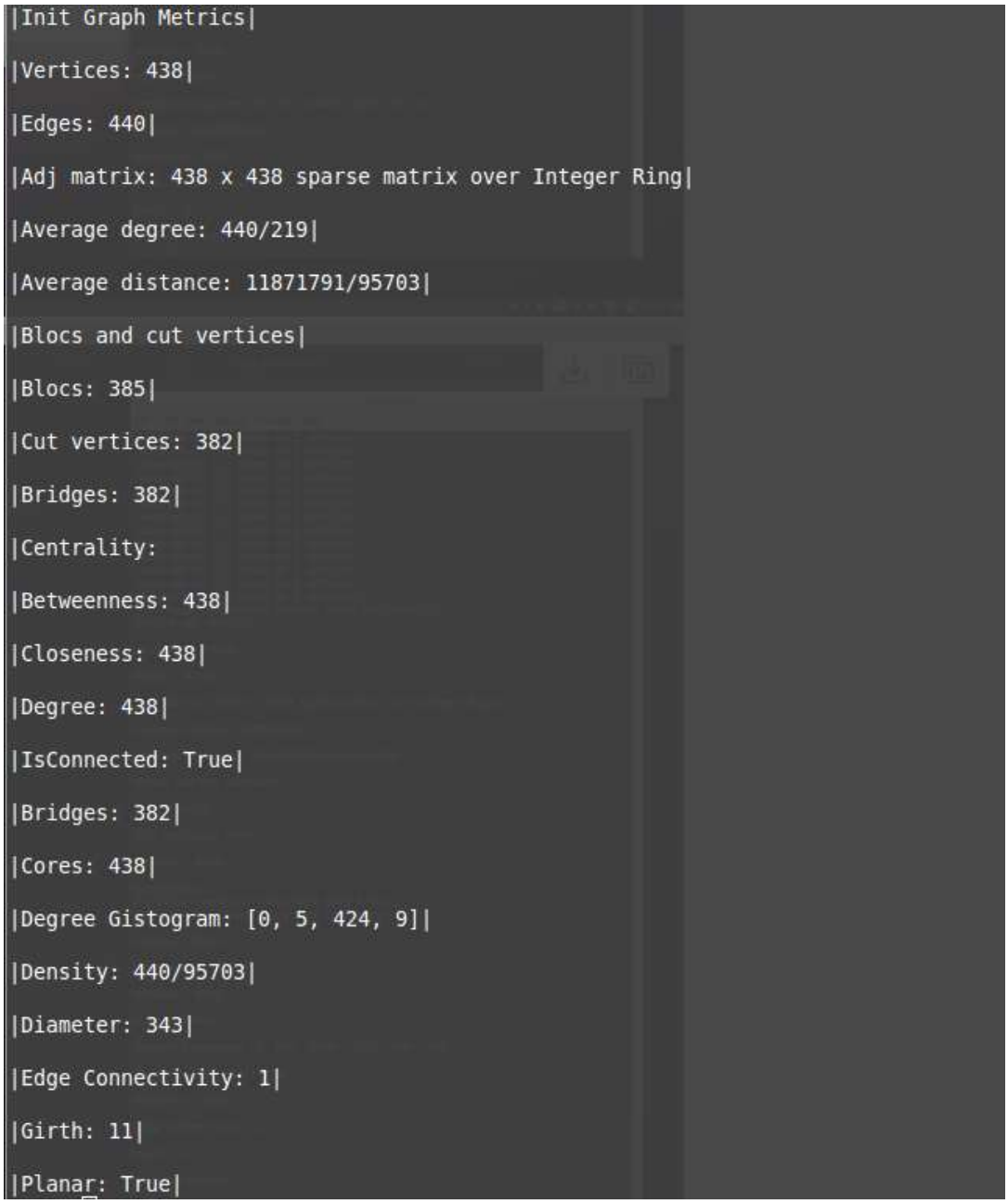
Img. 1. Subgraph with nodes of hierarchy 1 test.

```
sage: sage_calculation.collect_graph_metrics(t[0])
|Init Graph Metrics|

|Vertices: 47943|

|Edges: 50701|

|Adj matrix: 47943 x 47943 sparse matrix over Integer Ring|

|Average degree: 14486/6849|

|Average distance: 381470799086/1149241653|

|Blocs and cut vertices|

|Blocs: 9623|

|Cut vertices: 9524|

|Bridges: 9573|

|Centrality:
|Function execution takes more than 5 min.|
|Function execution takes more than 5 min.|

|Degree: 47943|

|IsConnected: True|

|Bridges: 9573|

|Cores: 47943|

|Degree Gistogram: [0, 634, 42695, 3127, 1438, 49]|

|Density: 7243/164177379|

|Diameter: 1933|

|Edge Connectivity: 1|

|Girth: 3|

|Planar: False|
```

Img. 2. Subgraph with nodes of hierarchy 2 test.



Img. 3. Subgraph with nodes of hierarchy 3 test.



Img. 3. Subgraph with nodes of hierarchy 4 and more test.