University of Nice Sophia Antipolis
Department of Computer Science

Master of Science in
Ubiquitous Networking and Computing

Research Internship Report

Bicycle-oriented itinerary computation for smart cities

Participant
Marko Oleksiyenko
Supervisors
David Coudert
Nicolas Nisse

31/08/2017

Abstract

The concept of smart cities is aiming at combining of information and communication technology with Internet of Things solutions to provide a full control over city assets, which include everything from schools to industries. Considering this, mobility is a very important aspect of this idea as we should connect them not only in the network but also in real life. In this paper we are dealing with the problem of mobility using the real-world example of cities like Nice and Cagnes-sur-Mer. In this context, we start with extracting the map of Nice and Cagnes-sur-Mer into the road graph format that can be understood by the programming language. Further, we survey existing shortest path algorithms and select the best one for real life dynamic road picture. In the scope of dynamic road situation, we then show the approach on how to combine this algorithm in order to accommodate it to continuously updating information, by dividing the region into sufficiently small zones with the help of linear programming. With the results obtained on both artificial and real-world data we show that our approach has a big potential and can be used for creating the application for mobility in cities.

Table of Contents

1. General Project Description

1.1 Framework/Context

Smart cities are being thoroughly studied nowadays. The ability to estimate constraints required to pursue a route from any place in the city to another one has become very important. These constraints can be anything: from time to number of stops. Minimizing them could have a significant impact on the outside world: the less stops driver (or cyclist) makes – less effort he or she puts, which itself produces less exhaust. The project is aiming at studying already existing solutions of shortest path discovery and finding the most efficient one for the bicycle transport system. By using real-world data extracted from resources like OpenStreetMap, the algorithms can be tested and the best ones can be found. The bicycle study can then be combined with whatever existing means of transport by introducing changes into the transport network graph and algorithms so that they receive the required information for computation. The difficulties are mainly concerning the constraints that affect the itinerary planning and are specific to each way of traveling, and the nature of the means of transport (whether it is scheduled or roams randomly).

Having the possibility to build paths from one point to another in the static environment (i.e. without any changes in time, itineraries, etc.) is good. However, in the real world we can observe that situation changes continuously and preserving the same route as it was built from the beginning can drastically delay the actual arrival time with regards to estimated one. Based on these assumptions in this work we try to approach the dynamic scheme by providing both basic and sophisticated methods to deal with the posed problem.

Nowadays when someone wants to go to some destination in time he or she plans the route and, with the hope of correctness of the estimated time, goes on a trip. With big transport companies and means of transport like airplanes, the estimation is usually correct with an error varying in range of several minutes (i.e. a few % of total travel time). With public transport like buses, error may increase due to the traffic jams and time that passengers spend with boarding or getting of the vehicle. However, with the popular means of travel like rental bicycle (e.g. vélo bleu in Nice) there is a very big amount of factors influencing the correct estimation and building the itinerary, such as bicycle availability at the starting station, possibility to leave a bicycle at the destination, safeness of itinerary, number of traffic lights needed to cross and many more.

1.2 Motivations

The rising popularity of such public transport systems as bicycle is playing a major role for searching solution of efficient itinerary planning mechanisms. For example, there are two biggest online route-planning systems like Google Maps and OpenStreetMap; however, the correctness of produced estimation and scalability of the algorithms needs some improvements. An error even of 5 minutes can sometimes be crucial so that the preciseness of the solution must be maximized as much as possible. This error influences the outcome of the client's trip and decreases his or her satisfaction of the system, which

can also result in the loss of the target audience. Moreover after working with the actual real-world data during the PFE, we have verified that the size of it is very huge (e.g. for the map of Nice and Cagnes-sur-Mer the road graph contains around 82 thousands of nodes and 88 thousands of edges). This urges the desire to discover ways of reducing the working data range to increase the speed of the computation. Along with the aforementioned reasons we would like to model the realistic picture of the city which is dynamic. Route planning systems as Google Maps and OpenStreetMap have this feature, however, it is limited and we want to achieve the level of the dynamicity where even malfunctioning traffic lights, road accidents or road works will instantly update the route search process. Moreover, if the algorithm is being thoroughly studied and implemented, it can be used to build not only efficient itineraries for bicycles, but with some adjustments (as changing metrics and modifying graph) it can model routes for various transport systems as well as ordinary pedestrian walk planning.

1.3 Challenges

Itinerary planning of the static system is similar to the search of the shortest path from source to the destination in directed graph where all the weights (depending on the metrics system implemented in the graph) are known. Even in this case we need to consider the problem of choosing the metrics because some users would like to travel as fast as possible, some of them could care about safety (less road crossings for children) and other preferences. First, it is hard to get the clear static picture of city transport network that can be represented as suitable for a graph algorithm as it can be of a very big size and even powerful computation machines will take a lot of time to process it. After having the data to work with we had to read through many shortest path algorithms to pick and implement a fast and efficient one to fit our goals. The main difficulty of this step was achieving the exact similarity between the theoretical description of the algorithm in the paper and real implementation of it. Then the challenges mainly start arising in the dynamic systems such as realistic representation of the city's global traffic picture. In this case weights of the graph as well as arcs can be changed accordingly to the traffic jams, repairing works, broken traffic lights and even accidents. This means that now, we cannot just compute the route once and provide it to the user all the time. We need to continuously look at the values of the weights and arc availability to search for the most appealing itinerary from the user's point of view. Here we cannot permit the updating of the full map of the world because it will take a lot of time and resources. So we have applied the notion of map partition to reduce the number of the changes that need to be forced onto the working data. Very important challenge appears when we try to implement the customization of the itinerary planning mechanism: the algorithms process linearly expressed constraints normally but when they become non-linear, the resolution might be much more difficult. The key is to choose constraints wisely and make them feasible for computation. Last but not least, an efficient itinerary planner should possess a good trade-off between memory consumed and CPU work. Every system has different hardware, however, usually the platforms (e.g. smartphones or tablets) that use itinerary planners are not big and have lower computational power than experimental machines.

1.4 Goals

The main goal of the internship is to implement service offering efficient itinerary planning. This requires having a shortest path algorithm capable of providing us with exact route from stated source of the trip and its destination as well as being fast and flexible in case of weight changes in the graph. The algorithm may not seem flexible for the task, however, we can find the way of adapting it to the dynamic case and possibly even improve the algorithm for better performance. Other important objective is to identify different metrics that will create possibility of user customization of the mechanism which will give the system the notion of novelty in comparison to existing solutions for route planning. Additional goal is to learn how to extract and operate the graph representation of city's map on example of Nice using the OpenStreetMap project. Having this graph representation, we should be able to run some shortest path algorithms on it and try to adapt them to the dynamic scheme, when weights of arcs constantly change. Of course, the algorithm can be tested on the manually created graphs but having real-world data will prove not only the theoretical but also the practical value of the system.

2. State of the Art

There has been done a lot of work in the field of finding the shortest path in the recent years due to the growing number of cars on the streets and needs of efficient route planning in personal or business purpose. In order to produce fresh results one needs to survey the existing literature, correct inaccuracies and add new features.

2.1) In the article "Cycling the Green Wave" [1] the problem of creating shortest paths for bicycle users is being tackled. The authors are trying to apply the adaptive cruise control to bicycles so that we could achieve the same efficiency that the car gets using it, for example, spending less energy for starting the movement by reducing number of stops required on the path. The approach of the work is based on the usage of classic Dijkstra path finding algorithm on the graph, representing the traffic network, which was specifically modified for the problem. The modifications are introduced by the notion of bypasses, which are the additional arcs that are placed on the traffic light nodes and standing for the paths for crossing the traffic lights in each of available direction. The weight of the bypass is computed from the time of arrival to the traffic light and shows the estimated waiting time at the red light. This means that the graph that is created for the Dijkstra algorithm already has a dynamic element, which is also considered in our work. So the bypass notion is a very important result which can help us finding the most efficient path of all. The paper also shows modifications of Dijkstra algorithm specifically for presented task. The difference between the original algorithm is that whenever we consider the bypass road we need to ensure that the arcs, which were bypasses will never be reviewed in subsequent search. This helps to evade the wrong path (bypass can have a longer estimated time than the basic route, as the waiting at the red light is added) arising as a shortest. This modification shows an example of possible ways to adjust various algorithms to a particular task.

2.2) The article "Route planning in Transportation Networks" [2] similarly to our goal surveys current solutions while trying to find a most optimal one to deal with itinerary planning task.

Based on it, there are three main algorithms for finding the shortest path in directed graph with weighted arcs: Dijkstra algorithm, Bellman-Ford algorithm and Floyd-Warshall algorithm. These algorithms are used to find the shortest distance between the source and the destination in the graph.

From the results that were found we can now know that Dijkstra algorithm has the time complexity in $O\big((|V| + |A|) * log|V|\big)$ using binary heaps. However, if we use Fibonacci heaps the time complexity drops to $O(|A| + |V| * log|V|)$, where V is the set of vertices of the graph and A the set of arcs. On the other hand, the Bellman-Ford algorithm worse case time complexity in $O(|V| * |A|)$, but in some situations, it can compete with the Dijkstra solution. As for the Floyd-Warshall algorithm, it has time complexity in $\Theta(|V|^3)$, while computing distances between all pairs of vertices and for sufficiently dense graph is faster than running |V| times the Dijkstra algorithm. This gives us the understanding that depending on the graph of the transport network some algorithms will work better but for the completely different scenario they will perform poorly.

The idea of using the bidirectional search for reducing the search space looks very appealing from the computation time point of view: we simultaneously start the forward search from the source and the backward search from the destination; it may stop whenever the intersection is found, which means that the shortest path is a sum of two of them. However, it still needs to be checked whether this solution is good for a dynamic system, as we could find the shortest path based on time that has already passed and user will stand in a traffic jam that has not been taken into account.

The article [2] goes on to present many different techniques of finding the shortest path: goal-directed (e.g. A* search, geometric containers, arc flags, precomputed cluster distances, compressed path databases) which reduce number of vertex scanning by ignoring the ones that do not go in destination direction. The road networks usually cannot be represented as planar graphs, however, it is stated that there are some separators of small sizes in them. So some separator-based techniques are shown further: vertex separators, arc separators. This can help us find the smaller overlay graph that will allow faster computation [3]. There are also hierarchical techniques that do not guarantee the discovery of the shortest path but are based on heuristics that long short paths eventually have some small arterial network of important roads, such as highways [2]: contraction hierarchies (exploit the hierarchy by using shortcuts, order vertices in increasing importance and contract them to find a shortcut), reach. Bounded-hop techniques (using "virtual shortcuts") are of great interest because they are dealing with precomputed distances between pairs of vertices but not the whole graph which allows to have less initial computation: labeling algorithms, transit node routing, pruned highway labeling. This overall description gives us the set of algorithms that can possibly be chosen in our work.

In the article "Route Planning in Road Networks" [4] similar techniques (hierarchical and bounded-hop) are presented which shows us the popularity of those methods.

The article [2] gives some possible approaches for dealing with dynamicity in the system, which are of great importance for our research. The approaches divide into two groups: one-step solutions and two-step solutions. One step-solutions mainly consider picking up the algorithms that can tolerate the arc-weight changes such as ALT (landmarks and triangle inequality algorithm that uses small set of landmarks and distances to them to compute a valid lower bound on the distance to every vertex). However, two-step solution proposes an idea of dividing the task into metric-independent stages and metric-dependent. During the first phase, we could compute basic route because the graph in this case contains only the network topology (no weights) and can be considered more or less static. The second phase is designed to compute the actual cost of the itinerary based on arc weights. This approach can save a lot of time; however, it still has to be tested and proved correct and efficient. Furthermore, it requires to store a significant amount of data.

Another interesting feature of an itinerary planning is time-dependence, which is also briefly mentioned in the paper [2]. It plays a major role in our study because depending on the time there could or could not be bicycles available at the station, traffic jams could be longer or shorter based on the morning, mid-day or evening time cycles and so on.

The paper [2] also provides some experimental results, which show the performance of various algorithms that can be used to pick the initial set of techniques faster.

2.3) Among all the shortest path algorithms in our project we decided to focus the attention at the Hub-based labeling algorithm by Adrian Kosowski and Laurent Viennot [5].

The main idea of the algorithm is to pick sufficiently small set of edges (which are called hubs) for each vertex in the road network graph. The authors give that following definition of the hub set: for a network $G = (V, E)$, where V – set of vertices and E – set of edges; we assign a small subset $S(u) \subseteq V$ to each node $u \in V$, in such way that for any pair of nodes u, v, the intersection of hub sets $S(u) \cap S(v)$ contains a node of the shortest uv-path.

The principle bases on the way these hubs are picked: for each pair of nodes in the graph an edge on the shortest path between them is picked. This approach uses precomputing to prepare the data for very fast shortest path query response. As to provide the answer for the query the only thing that we need is to look at the intersection of hub sets of target nodes: the hub that is common between the two of the nodes and has the smallest distance to each vertex lies on the shortest path in between them.

The algorithm uses several novel parameters and notations as well as already known ones. First authors introduce Tree skeleton. Taking the shortest path tree of the road graph G rooted at node $u \in V$ we apply the notion $\mathrm{Reach}_T(v) = \max_{x \in V(T)} d_T(u, x)$ to create a subtree that is pruned at $2/3$ of the distances. In this case we get smaller tree consisting of the paths by which you can get to every node in the graph.

The paper uses the notion of geometric realization of the graph, which is a "continuous" representation of the graph where each edge is presented as infinite number of vertices of degree two. Now, applying this notion to the tree we can define the width of a tree T: the maximum number of nodes in it at a given distance from its root.

$$Width(T) = max_{r>0}|Cut_r(T)|$$

The width of the tree was introduced to present another important parameter called Skeleton dimension. It is defined as the maximum width of the skeleton of the shortest path tree.

All these parameters and notations are provided to show the efficiency of the algorithm. The authors show that their method allows us to find in polynomial time hub sets with size $O(k \log D)$ on average and $O(k \log\log k \log D)$ in maximum case, where k is a skeleton dimension and D is a diameter of the graph.

In addition to hub set size authors present the comparison of algorithm's speed to other shortest path algorithms on the road networks of 20 million nodes, which is shown in Table 1.

The speed of the algorithm and research interest in implementation of novel approach were the reasons that made us pick it for the project. In chapter 4.3.6 we will discuss the algorithm step by step and provide results that have been acquired during the internship.

Dijkstra (1959)	1 min
Bidirectional Dijkstra	10 sec
Bidirectional A* (1968)	1 sec
Reach-Pruning, Contaction Hierarchies (2005)	10 ms
Hub-labeling (2010-13)	10 μsec

Table 1. Comparison of algorithm shortest path query response time.

2.4) All the documents on the topic that we have read consider finding the shortest path uniformly for every user. The novelty of this project is also to introduce different metrics that depend on the user. This means that based on the user choice, the mechanism will compute the weights of the arcs and only then solve the shortest path problem. For example, parents would like to choose the safest path for their children and this means less road crossing; delivery guy needs the fastest route to get a better tip, so we go with a least trip time; some people would like to have a nice view while riding the bike and this will need another metrics based on user reviews. This makes the algorithm more user-friendly and appealing for use.

2.5) OpenStreetMap is a collaborative project that allows people from all over the world to create and modify the map of the globe. This resource can provide us with the required basis such as a graph representation of the city of Nice including the bicycle stations and other important landmarks for our study (e.g., known cycling roads).

One of the main goals is to find a way to get the data from OpenStreetMap and convert it to the form that can be recognized by the algorithm. To perform this task, we can use existing Python libraries such as NetworkX.

The next step is to summarize the techniques that have been already discovered and pick the most relevant for bicycle network. Then we should obtain the metrics considering various users' preferences.

3. Contributions

There are two approaches for computing the itinerary: considering static and dynamic picture of the traffic. In this work we are studying the first one to discover a good way of approaching the latter, presenting the results and implementation of required methods as well as the prospective solutions to improve them.

The experimental study of the theoretical developments is based on a real-world road network of Nice and Cagnes-sur-Mer with the number of nodes reaching up to 88 thousands. As the outcome of these experiments, we provide the visual representations of the basic and sophisticated partitions of the map into regions, which are cropped parts of the full road network graph. We provide the average and total CPU time to compute the shortest paths between all the node pairs within these zones and compare the time required to build such a partition. The obtained results show that using naïve partition is very time consuming and inefficient.

In addition to performed tests we created a sophisticated division of the map by using the linear programming. Implementation and experiments using this approach has shown us that there is a better decomposition of the graph into small regions in terms of size and time to compute shortest paths.

We have spent an immense part of the internship on examining the hub-labeling shortest path algorithm and implementing it with Python programming language. As a result we have the procedures that can be used to run this algorithm on graphs of different sizes and structures.

All the implementations of the hub-labeling shortest algorithm as well as the partition procedure including the disposal of overlapping can be used in future work and be modified for an increase of performance.

4. Approach followed in the work

4.1 Tools used

In this section we are describing the tools and their specific choices considering the scope of the task and the possibilities that they can provide us with.

In the task of creating the itineraries, we supposed to have sufficient and up-to-date data showing the road picture of inspected area. We used OpenStreetMap project to get the map of Nice and Cagnes-sur-Mer. To extract data we used QGIS, which is a cross-platform open-source desktop Geographic Information System. As every GIS system, QGIS is created to capture, store, manipulate, analyze, manage and present spatial or geographic data; however, unlike ArcGIS it is a free software. The map provides us with a detailed picture of streets, traffic lights and many more features that we might need.

Unfortunately, the only thing that is important for us – VeloBleu stations' location is not present on the downloaded map. In order to find them and combine with already collected data we found the website Open Data Nice Cote d'Azur. It has all information about Velobleu stations including their coordinates, names, short description and other. Moreover, it is stored in convenient for QGIS format – json. By importing this file into QGIS we got two layers in our application. The crucial thing at this point was to connect the obtained points of VeloBleu with the map itself, because otherwise the algorithm would not be able to build path between them. In order to perform the procedure we used GRASS GIS embedded into QGIS. GRASS GIS is a Geographic Resources Analysis Support System software suite to allow the geospatial data and analysis image processing, producing graphs and maps, spatial and temporal modeling, and visualizing. This packet has operation v.net.connect, which creates nodes from a vector points file (in our case it is json file downloaded from Open Data website) and adds these nodes to an existing vector network of arcs (the extracted map of Nice and Cagnes-sur-Mer from OpenStreetMap). By performing this, we obtain the map with VeloBleu stations on it, and these stations are connected to the map with a shortest distance. By the shortest distance, we presume the perpendicular one, which is the shortest interval between a point and a fixed line in Euclidian geometry.

As a main programming language, we have chosen Python. It is a scripting language, which allows writing programs very quickly and has enormously large number of open-source libraries, which can be easily downloaded and even modified in order to fit better into the scope of a problem (which was exactly the point in our research).

Having the desired data, we had to convert it into the file format that could easily be understood by Python in order to run algorithms on it. We chose the Shapefile that is a geospatial vector data format for geographic information system (GIS) software. The shapefile format can spatially describe vector features: points, lines, and polygons, representing, for example, VeloBleu stations. Each item usually has attributes that describe it, in our case we can transfer bicycle station information, like geographical coordinates, name, number of bicycles available directly into python code.

Among all available Python libraries we chose NetworkX to work with graphs. NetworkX is a Python language software package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. In particular, it provides a built-in Shapefile reader with the help of which we can convert data into convenient representation of directed graph.

In addition to this, we used several Python packages in order to perform visualization (Matplotlib), generate the cell-grid of the map (Numpy), search the outgoing edges of each cell (Shapely) and measure time taken (Time). To plot the results of the work we used MS Office Excel.

For the part with partitioning the map into regions we had to deal with linear programming. To solve the program that we created we used SageMath which is a free open-source mathematics software system licensed under the GPL. Combining this system with two different solvers such as GLPK and CPLEX allowed us to obtain the solution of the problem. As we have predicted CPLEX has shown itself to be more efficient and fast than GLPK, so all the final results are produced by running programs under CPLEX solver.

4.2 Difficulties and modifications

This part contains the details of the obstacles we have stumbled upon and what were the solution that we implemented in order to overcome them and achieve posed goal.

The native NetworkX method read_shp decomposes the Shapefile into points and linestrings (specific class in Shapely Python package for representing lines) in order to add them to DiGraph as nodes and edges respectively. However, for our task it is not enough. In order to visualize the solution we need to add specific attributes like “position” with geographical coordinates explicitly. This will allow us to plot the results using MatPlotLib package of Python.

Moreover, as each linesting consists of the list of points we need to add every one of them to DiGraph in order to avoid lost paths. In the figure below (fig. 1) we can observe the “lost path” situation. Let us take the point in the top as a VeloBleu station; after using the v.net.connect method in QGIS we combined map with bicycle stops, however, as was noticed sometimes it does not divide already formed linestrings on the line intersections. This leads shortest path functions to ignore the newly added path as one that does not exist. These methods work vertex-by-vertex, so if there is no node in between edge start and end points, it will assume that there is no path attached to this edge.

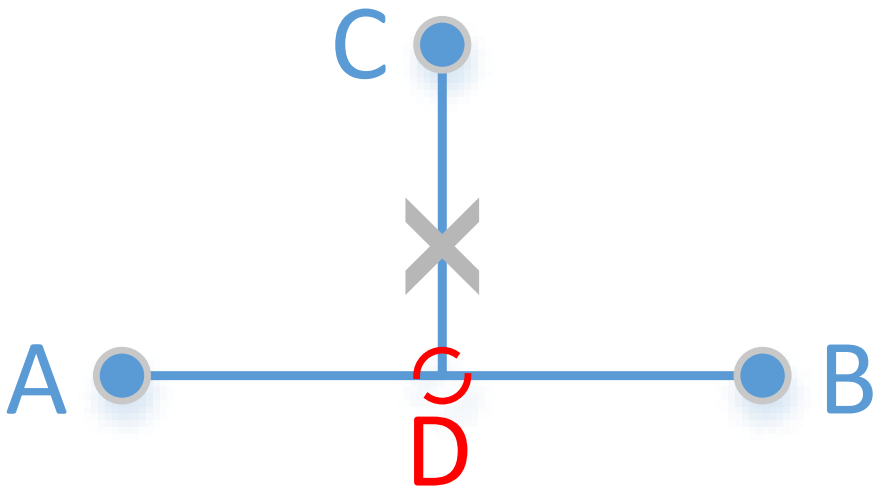


Figure 1. Representation of "lost path" problem. In case of junction absence on the lower line (point D) algorithms for shortest paths, which start working from point A, will ignore point C and interval DC and go straight to point B. The cross represents the skipped path.

Next chapters include difficulties that arose during the shortest path algorithm implementation and linear program solving. We will explain them further in the context of the problem.

4.3 Approach to the dynamic picture of roads

4.3.1 Motivation

Having the possibility to build paths from one point to another in the static environment (i.e. without any changes in time, itineraries, etc.) is good. However, in the real world we can observe that situation changes continuously and preserving the same route as it was built from the beginning can drastically delay the actual arrival time with regards to estimated one.

Many factors influence the length of the path; in our case length can be seen not only as a physical distance that a person should travel but also it can be time, number of traffic lights on the way, etc. The dynamic assumption tells us that in case when something changes along the path we need to consider it and recalculate the way to be optimal.

4.3.2 Basic idea

We used the idea of local areas used in Highway Hierarchies [4] to create our partition of the map. In the work mentioned these areas are defined by introducing the neighborhood radius for each node u. Thus, neighborhood of u (local area around u) consists of all nodes whose shortest path distance from u does not exceed the neighborhood radius. This is used to perform a search in local zone around the source and the target and after that

switch to another level of highway network that is much thinner than the initial graph in terms of number of edges included on this level.

However, we just take the notion of local areas and not the multiple-level scheme. The primitive realization of this approach can be splitting the map into cells of different size.

The core idea, which we want to implement, is having our map divided into small regions (see appendix A). For each pair of the nodes of graph that are located in the region we want to have the shortest paths between them. This will allow us instantly (without computing the paths) have the list of routes between the nodes located on the end of edges that cross the boundary of the computed cells. This precomputing is much faster than computing the shortest paths as we go, however, it is taking more memory.

One possibility to reduce the memory consumption is to store shortest paths between the outgoing nodes (i. e. the nodes that are in the beginning of the arcs, the end-node of which is located outside the region). In this case, on the entrance to the region we will see the possibilities to exit the area, which would be helpful if the node is not in the region, otherwise we should compute the path inside. This incentive urges us to create local regions of small size to optimize the number of nodes inside the area while having a low quantity of outgoing arcs from it.

We will also get significant benefits of this partition in case of any changes in the road picture of the city. If anything happens on the map, we will just update the zone where we had change and then recompute the shortest paths within the region instead of doing it for the whole map. This will lead to enormous time savings and get us closer to the dynamic view of the city.

4.3.3 Experiments with map of Nice and Cagnes-sur-Mer

In order to create an efficient algorithm of itinerary computation within the dynamic constrains we need to perform the experiments to find the optimal (in terms of number of nodes and edges in the zone) set of regions, which will allow us to get closer to the best possible performance.

Using the NetworkX library we have succeeded in converting the Shapefile into the directed graph. As a result, we have a graph with 82895 nodes and 88263 edges. Naturally, the task of computing the shortest paths between all the pairs in this graph is very heavy for CPU and memory. The testing machine with 6 GB Ram and CPU Intel ® Core™ i3-2350M, 2.30 GHz could not perform all_pair_shortest_paths function due to memory limitations. The most important reason for this is that method all_pair_shortest_path of NetworkX library returns the dictionary of shortest paths, keyed by source and target, which is very inefficient.

We performed three experiments with different cell sizes of 1, 5, 7 kilometers respectively. For each cell size we have measured the average and total time spent for computing shortest paths inside the regions, average time to extract the regions from the global map that we have. Each experiment was performed 10 times to get the mean approximation of time elapsed among these runs.

The time that we used was counted with the help of the method time.clock() from Python library time. This method returns the wall-clock time expressed in seconds elapsed since the first call to this function, so we can estimate the time that it takes for CPU to start and finish certain functions.

In the table 2 the average number of nodes and edges is shown per zone of each size. In this output, we do not consider the empty zones (i.e. without any edges or nodes in it). The outgoing edge is an arc for which one side of it lies inside of the region and other does not.

Zone sizes (km)	1	5	7
Total number of zones	307	25	13
Number of nodes	295	3767	6376
Number of edges	304	3985	6752
Number of outgoing edges	10	26	36

Table 2. Representation of average number of nodes, inside edges and outgoing edges in each zone of different partitions.

We can see that the number of nodes in one-kilometer zone is very low in comparison to 5 km one, which explains why the average time of shortest paths computation is lower (fig. 3). Another interesting observation is that we have a very small quantity of outgoing edges in comparison to number of nodes, which means that in case when we store shortest paths only between the outgoing nodes the memory savings will be massive.

The following graph will show the total CPU time elapsed for computation of all the paths in all the regions.

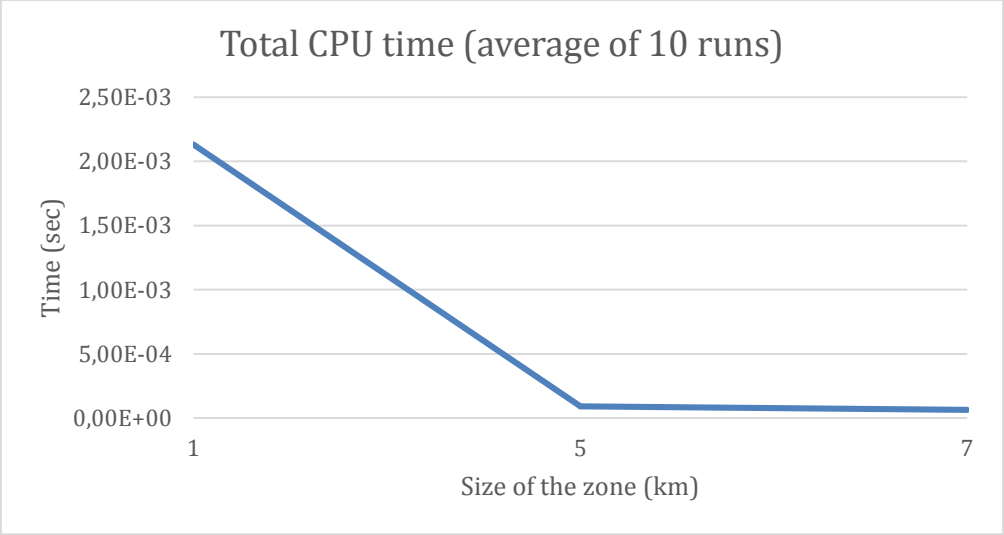


Figure 2. Total CPU time that takes to compute all pair shortest paths in all the regions (1,5,7 km zones)

The following figure (fig. 3) shows the average time spent for computing shortest paths between all the pairs of nodes in the region. The search of shortest paths was performed by using the all_pairs_shortest_path(graph) method from NetworkX library.

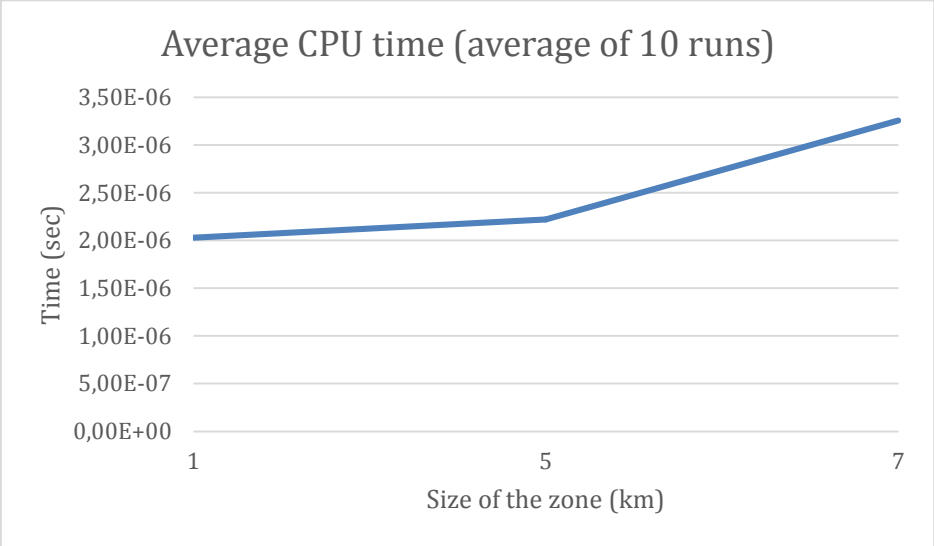


Figure 3. Average CPU time spent for all pair shortest path computation in each region of the partitions (1, 5, 7 km zones)

From the fig. 3, we can see that the computation time inside the regions grows with the size of the area. It is obvious because the more nodes we have the more paths we should evaluate.

We see that despite of low computation time inside the region we observe massive aggregated sum, as number of regions in the 1-kilometer partition is much higher than in 5 km zone (307 against 25). The more detailed explanation of this outcome will be presented further.

In addition to these diagrams, we have the visual representation of the total time spent to extract all regions of the partitioning for further shortest paths computation (fig. 4).

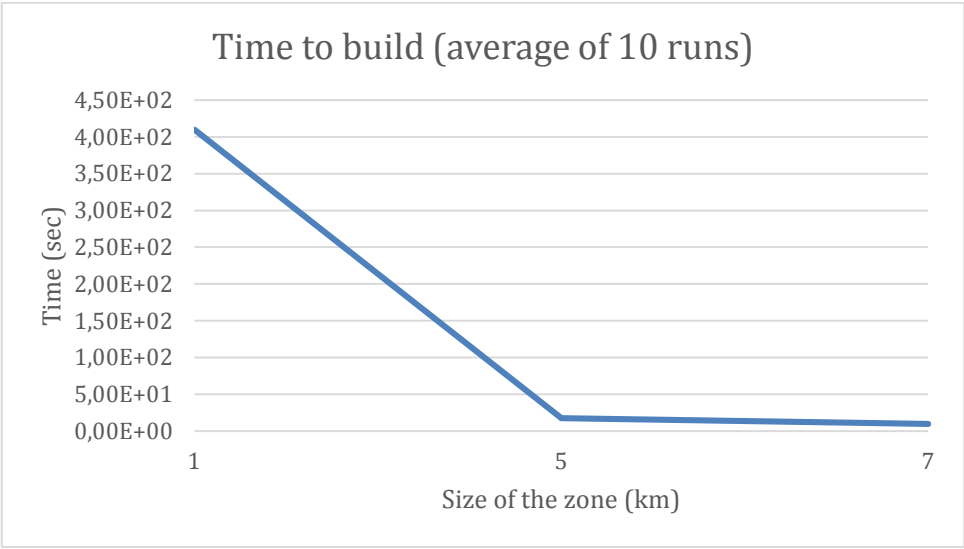


Figure 4. Time taken from the start to the end of partitioning process (1,5,7 km zones)

As shown above (fig. 4) the difference between 1 km and 5 km zones is enormous: average time for 5 km zone to be partitioned is 17.7 seconds, however, for 1 km zone it is 409.7 seconds, which is almost 23 times more.

One of the reasons of such colossal gap is that the algorithm for partitioning is not optimized enough. In order to select either node or edge to the region we walk through the whole initial graph. The approach could be possibly improved by sorting the nodes and edges. However, we have no guaranties that it will work properly. The problem hides in the representation of DiGraph. When we were trying to sort the nodes and edges in order to access specific intervals for each zone, some isolated nodes (edges) were observed, meaning that they were outside the presumed interval. This also can happen in case of edges that cross several zones; in this case the order also can be disrupted.

The table 3 shows the results for all three regions. As we can observe the difference between 5 and 7 km regions is almost 2 times.

Zone sizes (km)	1	5	7
Time to build (sec)	409,7	17,7	9,6

Table 3. Time taken to build the partitions of sizes 1, 5, 7 km.

4.3.4 Idea of partition improvement

The crude method of splitting the map into cells shows its potential, however, we want to have more sophisticated approach of zone computation, which can possibly decrease the number of nodes in the area to perform even faster shortest path computation inside the region. We have decided to use a linear program to find a good division of map into the regions.

Let us introduce some variables for the problem:

- Region with the center in node u:

$$R_u = \begin{cases} 1, \text{if region centered in u is selected} \\ 0, \text{otherwise} \end{cases}$$

- Ball(u)
  Let Ball(u) will be a set of vertices at distance at most k from u
- Boundary(u)
  Let Boundary(u) be a set of edges (x, y) such that x ∈ Ball(u) and y ∉ Ball(u)

We present the complete linear program below:

$$min \sum_u R_u * |Boundary(u)|$$

$$s.t. \forall u \sum_{v \in Ball(u)} R_v \geq 1$$

The constraint ensures that we will cover all the nodes in the regions.

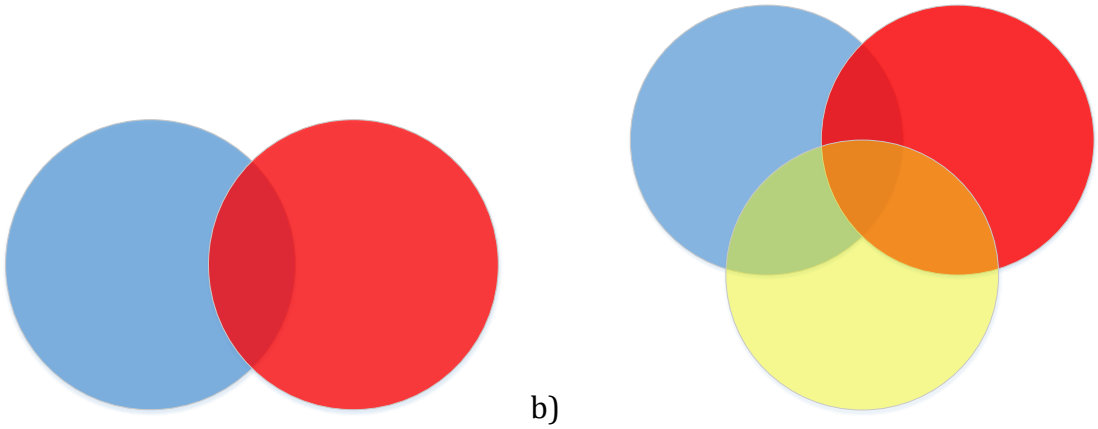
There is also a possibility to combine multiple sizes of the regions. To allow this we simply need to add the parameter k to all the variables, ensuring that for each node there can be only one radius of the ball:

$$min \sum_{k} \sum_{v \in Ball(u,k)} R_{u,k} * \left|Boundary\left(Ball(u,k)\right)\right|$$

$$s.t. \forall u \sum_{k} \sum_{v \in Ball(u,k)} R_{v,k} \geq 1$$

$$\sum_{k} R_{u,k} \leq 1$$

In perspective, the solution of these linear programs can give us good partition of the map. However, there is a problem of nodes in multiple regions. There can be at least two possibilities:



a) b)

Figure 6. The problems that can arise after running the linear program. a) The selected node belongs to two regions at once. b) The selected node lays in the intersection of three different zones.

The problem creates a set of tasks. At first we want to ensure that a vertex belongs to a unique region. At the same time we have to minimize the number of edges at the boundary as we described in the linear program.

The first one (a) can be solved as the minimum cut problem; however, the second one (b) may require manual correction of the deviating nodes. The main caveat is to avoid disconnected nodes after finishing the process. Because the presence of such nodes will not allow to perform shortest path search in the region.

4.3.5. An improved partition idea development during the internship

We have extended the solution we used during the PFE by adding R_u:

$$min \sum_{u} (R_u * |Boundary(u)| + R_u)$$

This allows to pick the same balls every time we run the program.

4.3.5.1. Challenges faced during the test runs

We met several problems while conducting the experiments. However, there were two main ones: first with the covering of the vertices and second with overlapping.

4.3.5.1.1. Covering problem

Considering the fact that our linear program is a simple covering problem, the result of it should contain the set of balls, the union of which must include all the nodes in the graph. However, in our case we had some vertices missing from the covering.

After double-checking the linear program we could not find a problem in it, performing the experiments of the covering on various artificial grid and non-grid graphs we had full covering. This fact made us look into the representation of the road graph of Nice and Cagnes-sur-Mer.

On the figure 7 we can observe the situation when two black nodes in the circle are not covered by selected balls because they lie in between the intersection of the three picked regions. The problem is in the form of the road graph, these nodes are outside the purple (zone 1) and red (zone 2) regions because of the radius constraints (distance to the center is more than 500 meters); however, it would have been covered by the pink (zone 3) ball if only the constraint of connectivity between all the nodes was satisfied.

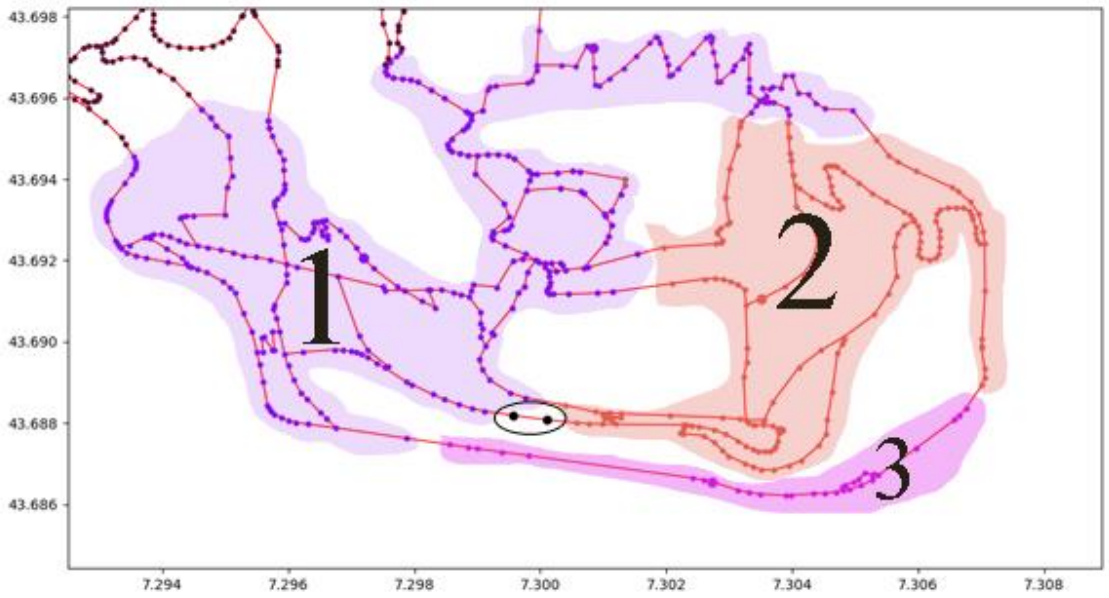


Figure 7. Black nodes omitted by the linear program after the first run.

To deal with this problem we decided to run the linear program twice. First run is over all the nodes in the graph. Second one is over the set of uncovered nodes that were left by the first round. Using this approach, we achieve full covering. We tested this assumption by using regions of zones with radii 250, 500, 750 meters.

4.3.5.1.2. Overlapping problem

Another important problem is the overlapping of the regions. When it occurs, we have to store the superfluous information for the repeating nodes. When the number of duplicated vertices is not big, the problem can be tolerated. However, in our case, over the half of nodes in the graph were residing in multiple regions, which could lead to crucial memory loss.

In the previous chapter about the ideas of partition, we have mentioned two cases of overlapping, but in practice we got a third one, when node was covered by four regions simultaneously. As we also mentioned before the two-zone overlapping can be dealt by solving a minimum cut problem; however, we decided to create an approach, which could cope with all cases of overlapping.

The algorithm that we created consists of two steps and can be seen on the figure below.

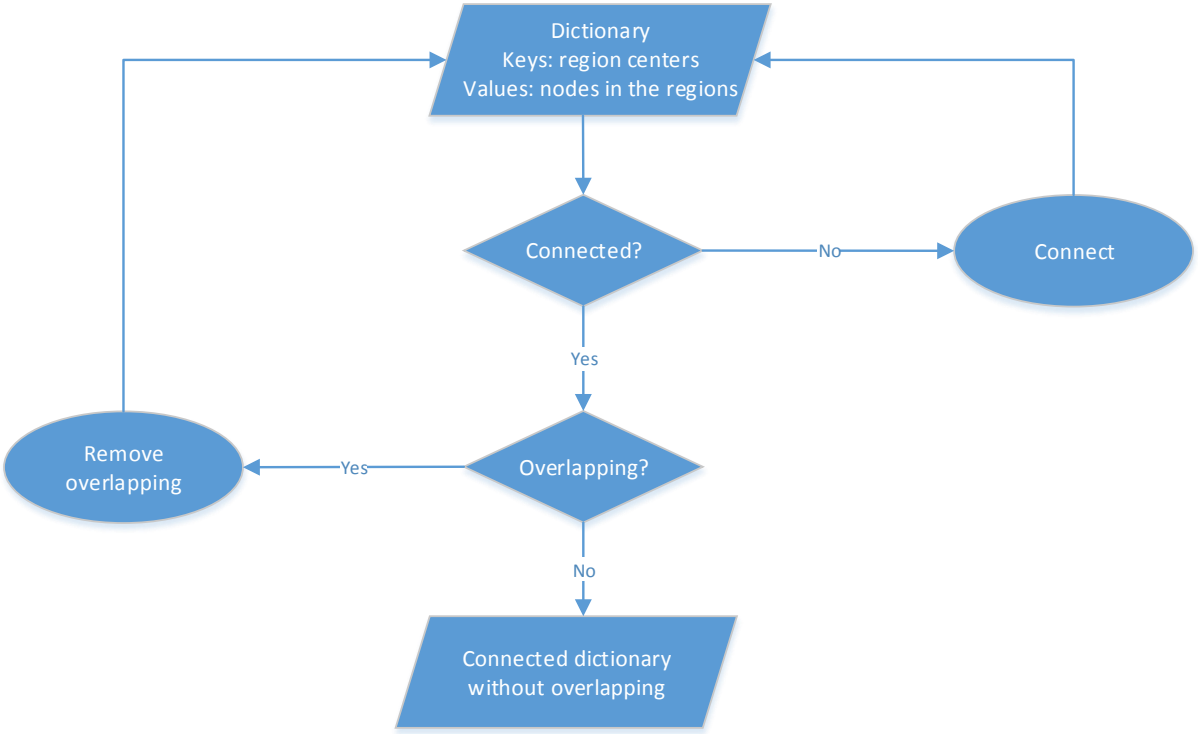


Figure 8. Algorithm for dealing with overlapping and node connectivity in the region

We pursue the idea of having small regions nodes of which are connected and no zones have the same vertices inside. Before the procedure we have a dictionary with region centers as keys and the set of nodes in those regions as values.

We start with checking the connectivity inside each region. If there are any disconnected vertices from the central node of the zone we rearrange them to avoid it. After doing several test runs we obtained the best way to remove disconnected nodes. The procedure takes first detached node and looks at its neighbors within the full graph picture. Initially, we connected it to the first neighbor we saw, however, this leads to zones overflowing with vertices. Then we decided to look though all the neighboring zones and pick the one

with the smallest number of nodes inside. In this case we balance the size of the regions and make them more or less equal.

Moving forward, we go back to the initial idea of removing the overlapping. Having the function that returns the node and all the regions it is in, we simply check the distances from the overlapping node to the overlapping zone centers and remove the vertex from every overlapping region except for the closest one. Doing this we remove the overlapping, however, some vertices can become isolated in the region. That is why the algorithm goes back to verifying whether regions are connected inside. The whole procedure finishes only when all the zones are connected and there is no overlapping.

The application of this procedure can be widened as there are cases when the graph is being modified (e.g. station has no available bicycles or the road is closed). In this case the node is removed from the graph and the regions may become disconnected, so we start the procedure and it removes any detached nodes that might have appeared. The same thing is when new parts of the graph are being added. In this case we run the linear program to create regions, then start the overlapping procedure which then gives us the connected zones without any overlapping. The main feature of the algorithm is that all the changes will be done locally with a very small probability of major zone changing during the rearrangements.

4.4. Enhanced shortest path algorithm used in project

4.4.1 Motivation

After conducting shortest path search experiments on artificial and real-world data using implemented functions from NetworkX library, which used Dijkstra or Bidirectional Dijkstra depending on the parameters provided to the function, we had an urge to discover and implement an algorithm that is fast and exact, improving the time computation seen before.

As it was already stated in the State of Art section, we had decided to pick the hub-labeling shortest path algorithm. Its authors Adrian Kosowski and Laurent Viennot have presented outstanding results: sufficiently small size of hub sets and incredible response time to shortest path query. One of the biggest parts of the internship was devoted to delving into the mechanisms of the algorithm and then implementing it to test its efficiency and whether it can be fit into the frames of the project. In this section we will discuss the algorithm step by step, providing the results we have got on the way.

4.4.2 General principle and algorithm structure

The key idea of the algorithm is to use preprocessing to be able to respond to a shortest path query in a very short period of time. Preprocessing in this case means the computation of the hub sets for each vertex in the graph and storing the distances to them. In addition to having just hubs and distances we can also store the shortest path, which then can be returned as a query response.

In the paper [5] the hub set is defined as follows : for a network $G = (V, E)$, where V – set of vertices and E – set of edges; we assign a small subset $S(u) \subseteq V$ to each node $u \in V$, in such way that for any pair of nodes u, v, the intersection of hub sets $S(u) \cap S(v)$ contains a node of the shortest uv-path.

This means that algorithm by itself consists of two main parts. Hub computation and then query answering by picking relevant hubs.

4.4.3. Hub computation

Using the paper [5] definition hub is an edge of the graph on the shortest path between two target nodes that satisfies certain conditions in order to decrease average and maximum hub set size.

4.4.3.1. Skeleton trees

However, before we introduce these conditions we should start with one of the most important features of the algorithm – skeleton trees. As we have already mentioned before, skeleton tree is a pruned shortest path tree of the graph.

The notion of skeleton trees significantly reduces the search space for finding shortest paths between the target nodes and thus picking the hubs into sets. One important detail about the algorithm is that all the shortest paths in the graph have to be unique. This is necessary because otherwise the intersection of two skeleton trees can be disjoint resulting in absence of the path between the nodes while it exists in full graph.

Having this figured, we ensure that all the paths are unique, for example, in grid graphs it can be done by adding insignificant value to the length of the non-unique path in sense that it will not have any influence onto the distance between the nodes except from differentiating each and every shortest path.

In order for algorithm to work we need to build shortest path trees starting from each and every node in the graph. Using NetworkX library in Python this task is very easy and takes only moments for each tree to build.

When we have all the shortest path trees we can proceed to the next step of pruning them and leaving only nodes that are within the 2/3 of the initial distance to the leaves. Using the depth-first search principle we move from the root of the tree and check if the distance to the leaf of the node is less than half the distance to the root we mark it as a false node. After labeling all the redundant for the skeleton tree nodes we call a function to remove them.

We have discovered a possibility of having smaller skeleton trees than needed. The described situation can be observed on the Fig. 9. This happens if the graph contains a number of edges that greatly exceeds the average length of edges. In this case, the furthest node from the root of this edge will not be considered as a part of the skeleton and enormous part of the tree will be lost (the longest edge on the first picture). We can deal with this problem by checking the edge that can be potentially removed from the divided

perspective (second picture). This means that we subdivide the edge by parts of length 1/12 and check whether more than a "parameter" (in our case half) of it can be considered as a part of the skeleton. If so, then the whole edge is included in the output of the skeleton function (third picture).

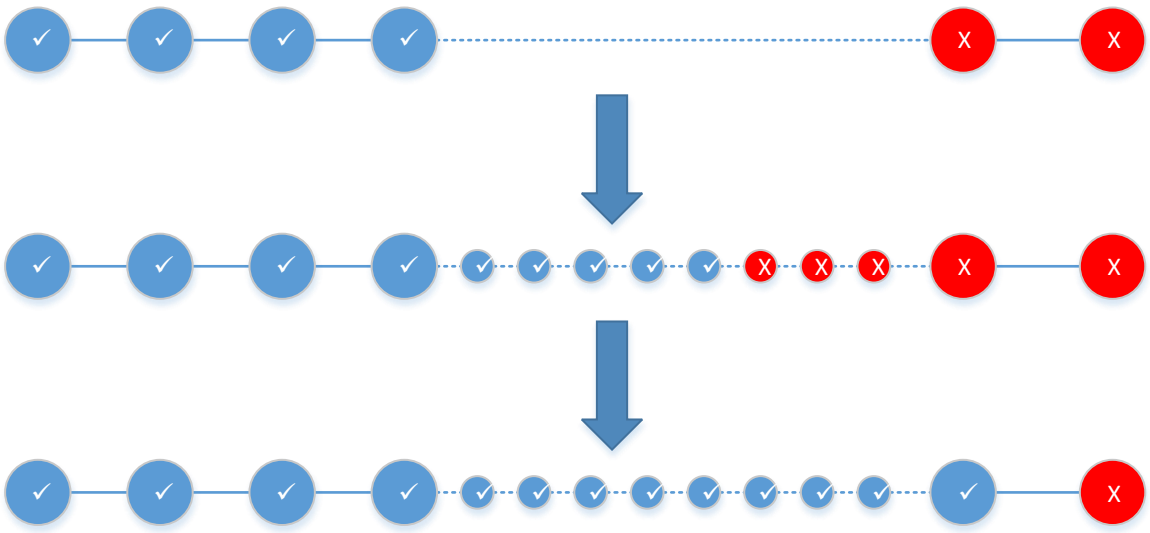


Figure 9. Longer skeleton tree situation. Considering the fact that over half (tuned parameter) of nodes fit the skeleton tree we include whole edge into the result. Blue (ticked) – nodes accepted nodes, red ones (crossed) – superfluous.

4.4.3.2. Hub selection stage

Another key aspect of the algorithm is how to reduce the hub size in order to use less memory on storing data. For these purposes the authors of the paper suggest preliminary edge labeling with random real values in range [0,1] which are uniform, independent and considered distinct.

After labeling the edges and constructing skeleton trees, follows the phase where we pick the hubs for each vertex in the graph, storing the distance to them.

Based on the fact that shortest paths in the graph are unique we can easily walk through every pair of nodes in the graph combining their skeleton trees obtaining the exact shortest path that we search for.

In order to minimize the size of the hub sets we consider the central subpath of shortest path between two vertices. It is defined as a set of middle edges of the whole path which lie between the nodes on this path bounded by $\frac{5}{12} *$ length of the path and $\frac{7}{12} *$ length of the path. It also reduces the number of edges we should consider as potential hubs.

After restricting the search area we pick the edge with the least random value which we have assigned to each edge on the preliminary stage. This allows us not only to fulfill the nature of hub, by picking it directly on the shortest path but also gives a high possibility

that during hub selection for the other vertex shortest path of which coincides with this middle subpath the same hub will be picked and the overall hub set size will not increase.

4.4.3.3. How to use the hub set to give the shortest path query response

After computing the hub sets for each vertex in the graph the first stage is over. Now we have the array which contains the list of hubs for each node with distances and exact shortest paths to them.

Remembering the part of hub set definition from the paper [5]: "the intersection of hub sets $S(u) \cap S(v)$ contains a node of the shortest uv-path" we look only at the arrays for the two target nodes.

The idea is to walk through those arrays in parallel (as hubs are sorted by the name) until we stumble upon the same hub name in both of them. The combined sum of the distances from each vertex to this hub is considered the shortest one between them. After finding the first mutual hub we look further to find another one with lower distance. We consider the path and its length through the hub that is picked after looking though both arrays as the shortest one.

4.4.4. Implementation testing

After careful reading of the paper and implementing the algorithm comes the phase of testing. The main purpose of it was to check whether the program is doing the exact task for which it was created and in fact gives back the hub set of the region that we provide on the input. During the implementation we conducted experiments on many graphs ranging from small to big as well as artificial or real life.

For the purposes of being brief we decided to put two major test cases into the paper. The first one is an 11 by 11 grid graph and another one is a real-world map of Nice and Cagnes-sur-Mer.

4.4.4.1. 11x11 grid graph results

Before running the test over the real-world map of Nice and Cagnes-sur-Mer we had to ensure that the implementation was correct. Testing each stage of the algorithm we eliminated the errors which were occurring and for final test before applying the program we decided to pick an 11x11 grid graph.

This graph contains 121 nodes and 220 edges. In the beginning all the distances were equal, however, after adding sufficiently small random value to the length of each edge we ensured that all shortest paths are unique. For the reasons of algorithm quality assurance we also need to know the diameter - 20 and the skeleton dimension of the graph – 4, we remind that this parameter is the maximum width (the maximum number of nodes in it at a given distance from its root) over all the skeletons of shortest path trees starting from each node in the graph.

By using the formulas shown in the paper [5]:

$O(k \log D)$, average

$O(k \log \log k \log D)$, max number of hubs

we can see that in our case we achieved the numbers of 30 on average and 40 at maximum. These numbers are in the same order of magnitude, so we can consider our implementation sufficient.

However, what do these numbers mean in real life? In a trivial case whenever we want to compute the shortest path between two nodes we can store shortest paths to each vertex from the target one, which needs N-1 memory units. Using hub-labeling algorithm we dramatically drop this number as many shortest paths go through the same initial routes. Looking at the 11x11 grid example, we can observe that instead of storing 120 shortest paths we need to save only from 30 to 40 distances to hubs.

4.4.4.2. Experiments on the map of Nice and Cagnes-sur-Mer

Having our implemented algorithms successfully applied on the small grid example, we decided to use it on the real-world map that we have extracted from the OpenStreetMap project.

On the Figures 10 and 11 the comparison of the full map and the skeleton tree from one (black) node can be seen. This skeleton tree representation shows that all the shortest paths from the black node start from a relatively small number of edges.

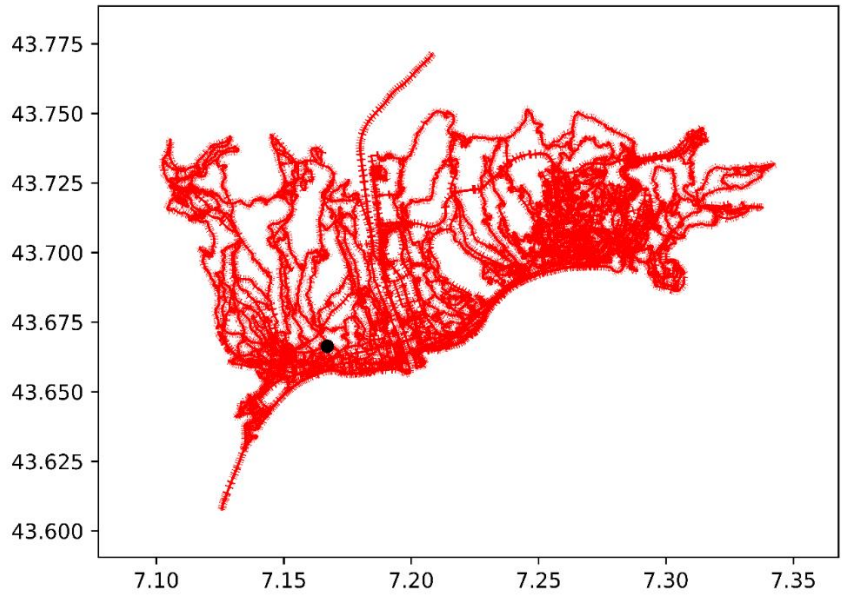


Figure 10. Shortest path tree of Nice and Cagnes-sur-Mer starting from the black node

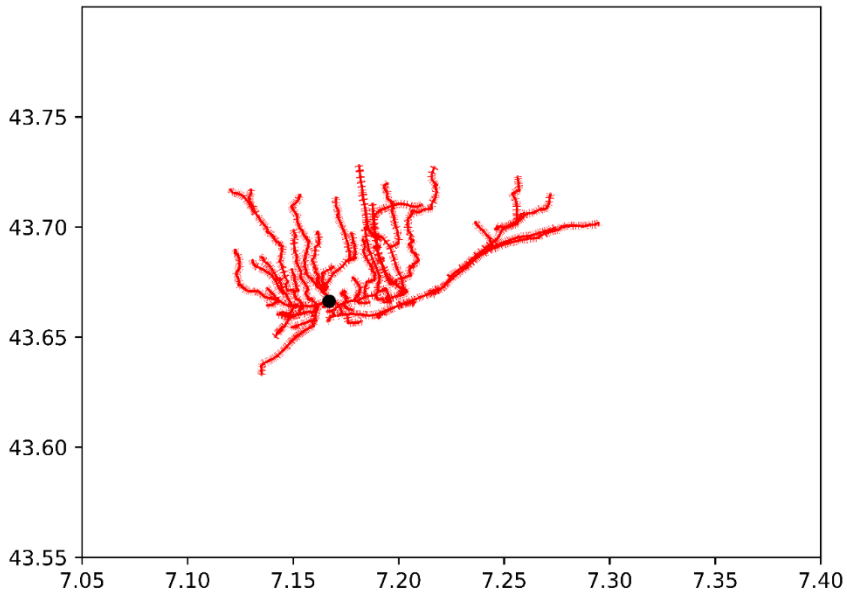


Figure 11. Skeleton tree from the same node as on Fig 10.

Creation of skeleton trees for the region has taken a very small amount of time, however, computation of hubs for the whole region occurred to be much more time consuming than we have predicted. After taking the idea of dynamicity into take account (where hub recomputation is inevitable), we have reached the conclusion that we need to use the partition of the region into smaller zones in order for the algorithm to be effective.

By using our sophisticated partition method we have broken the region into 127 zones of approximate radius of 500 meters having number of nodes inside ranging from 106 to 799. In the table 4 we provide sample of 10 run regions of different size showing their number of edges and nodes along with the average and maximum hub set size.

Zone #	# of nodes	# of edges	Average hub size	Maximum hub size
1	328	366	47	79
2	445	457	49	76
3	487	508	44	63
4	412	448	48	76
5	249	309	37	55
6	293	295	35	58
7	246	249	39	54
8	245	268	41	68
9	188	232	35	55
10	296	310	40	66

Table 4. 10 random runs of our partitioning method.

Observing the results we can summarize that in average number of hubs is 5.3 times smaller than number of edges in maximum case and in average it is 8.3 times smaller. This justifies that the usage of the algorithm will decrease memory consumption and at the same time will give us the ability to give a fast response on a shortest path query.

4.4.5. Summary and future algorithm improvements

Applying the hub-labeling approach for shortest path computation provided us with good results and has shown that it can improve the overall response time. However, we should underline that all experiments were conducted on the undirected graph of roads of Nice and Cagnes-sur-Mer. This means that in a directed case we will need to generate two different skeleton trees for each node (outgoing and ingoing shortest path trees) and in hub computation stage just merge appropriate trees to obtain a real shortest path. This should not be a problem regarding the fact that every major step of the algorithm (shortest path tree generation, skeleton tree pruning, hub picking) can be done in parallel which dramaticallyincreases the performance.

5. Discussion

5.1 Conclusion

While there exists a lot of ways to compute shortest paths considering static traffic picture, such as Dijkstra, A* search, geometric containers, arc flags, precomputed cluster distances, compressed path databases we have tried to develop the method to work with the dynamic, continuously changing scenario.

As the result of the performed work we obtained theoretical and practical outcomes that confirmed our assumptions and encouraged us to pursue the topic. The effectiveness of initial graph partitioning proved that dynamic scenario can possibly be solved this way with efficiency. The fact that we used the real-world data (map of Nice and Cagnes-sur-Mer) to perform practical tests is of significant importance, because it helps investigating the problem closer to the reality and shows its potential.

Using the already created approaches as Highway hierarchies, we were able to create our own way to find a partition with the help of linear programing. The minimization of region number combined with low quantity of outgoing edges can reduce memory consumption of the algorithm and bring us the good trade-off between the preprocessing and on-the-flight computing to achieve the high performance.

We have completed most of the tasks that were stated in the description of work and planned for the internship period. During this work period, the literature considering new approaches was surveyed as well as the documentation of the tools we used to perform experiments. We were able to extract the data and build the graph representation of it, which allowed us to understand the effectiveness of the solution by running specific algorithms that we developed.

Implementation of the sophisticated partition has given us the opportunity to compare it to the straightforward cell partitioning. By doing this, we proved that it provides us with better performance and reduces memory consumption.

Dealing with the provisioned problems of overlapping allowed us to think of situation of graph modification (i.e. inclusion of new nodes into the graph or removal of already existing ones).

5.2 Future work

The performed work has given promising results, which shows that there is a possibility of effective itinerary computation mechanisms that consider the changing traffic scheme. There are several tasks that are planned for further work and solving them will advance the research of the topic.

First, we need to optimize our implementation of hub-labeling shortest path algorithm and possibly pick more efficient programming language than Python. Having the algorithm implemented in Python allowed us to test its usefulness and looking at its performance in Python urged us to think of finding various approaches to increase its speed.

Next task is to apply the modifications to the hub-labeling algorithm, which were mentioned in the last chapter. This will allow us to deal with more advanced case of real-world (considering the directions of the roads).

Having done this, we need to proceed with extensive testing using already merged map of Nice and Cagnes-sur-Mer with VeloBleu bicycle stations. This will give the better view of the approach performance and show its practical appliance.

Finally yet importantly, one of the main points we wanted to implement, however, had no time, is introduction of new metrics into the system. We have been thinking about the possible metrics, such as “Safety of the road”, which means that the route has the least number of road crossings on the path from source to destination; or “Lightness of the route” that implies the itinerary with the least inclines.

6. Bibliography

[1] Matthias Prandtstetter, Ulrike Ritzinger, and Markus Straub. Cycling the Green Wave, preprint.
[2] Bast, H., Delling, D., Goldberg, A., Müller-Hannemann, M., Pajor, T., Sanders, P. & Werneck, R. F. (2015). Route planning in transportation networks. arXiv preprint arXiv:1504.05140.
[3] Richard J. Lipton and Robert E. Tarjan. A separator theorem for planar graphs. SIAM Journal on Applied Mathematics, 36(2):177–189, April 1979.
[4] Dominik Schultes, "Route Planning in Road Networks." Ausgezeichnete Informatikdissertationen. 2008.
[5] Adrian Kosowski, Laurent Viennot: Beyond Highway Dimension: Small Distance Labels Using Tree Skeletons. SODA 2017: 1462-1478
[6] NetworkX documentation (2017) https://networkx.github.io/
[7] SageMath documentation (2017) http://www.sagemath.org/
[8] Matplotlib documentation (2017) https://matplotlib.org/

Appendix

A.1 Three zone partitions

The first figure shows the 1 km cell side sized partitioning grid with map of Nice and Cagnes-sur-Mer.

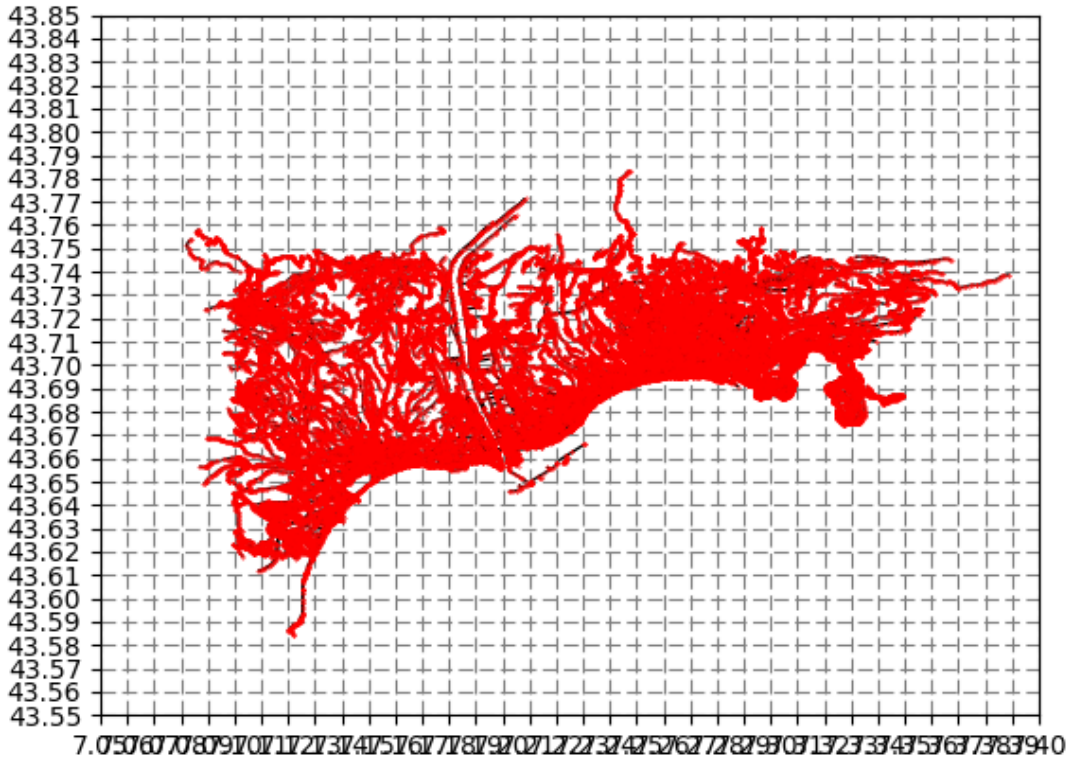


Figure 1. Each cell has a size of 1 km

The figures 2 and 3 depict 5 and 7 km zones respectively.

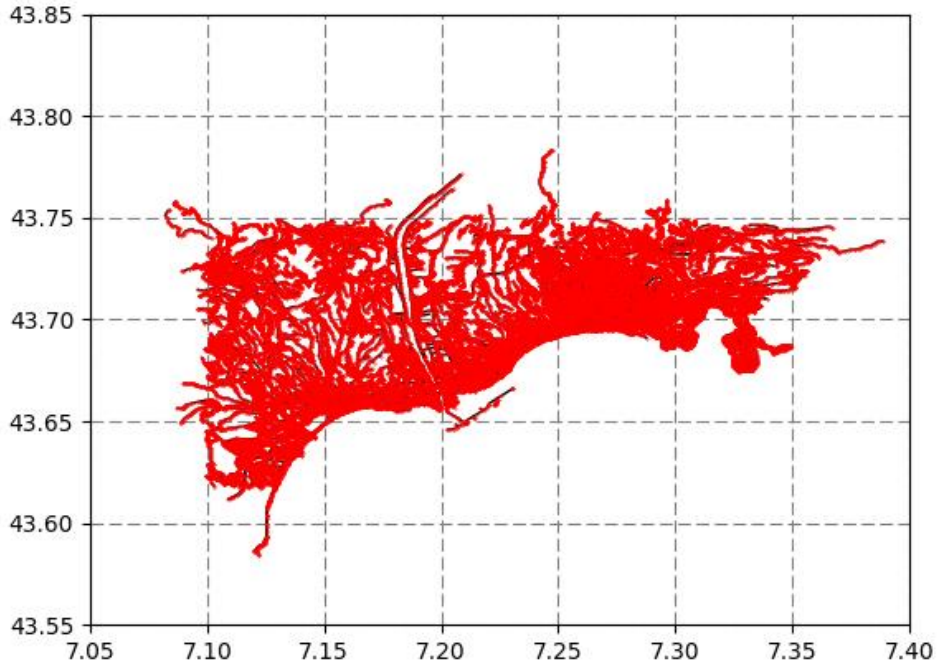


Figure 2. Each cell has a size of 5 km

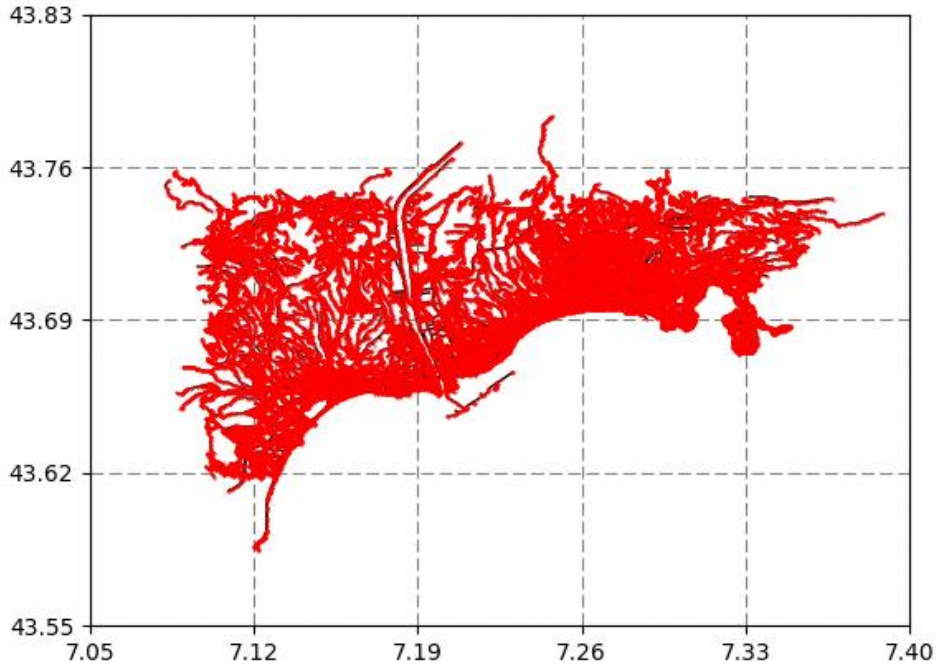


Figure 3. Each cell has a size of 7 km

A.2 Partition examples

After creating regions based on the 5 km cell sizes we obtain 35 zones. On the figures (fig. 4 and fig. 5) below, we provide two examples of formed regions.

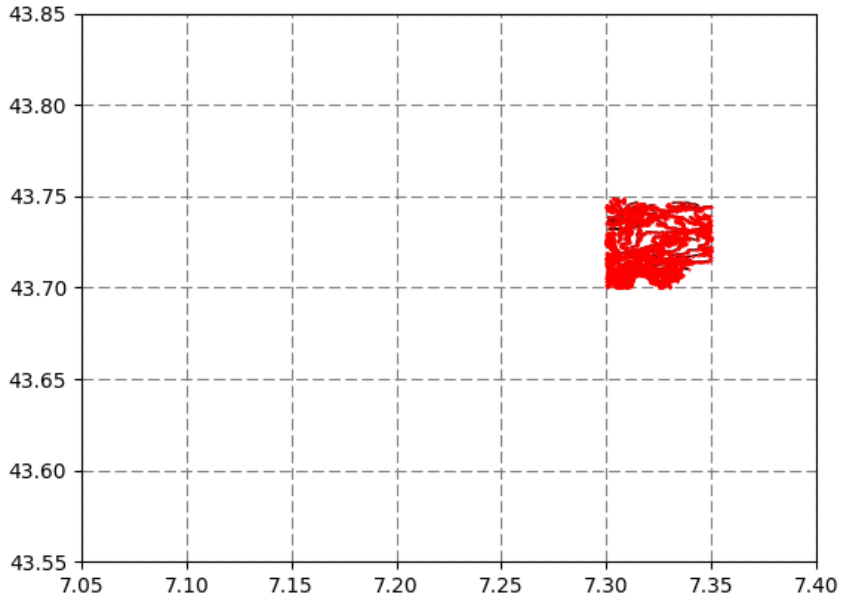


Figure 4. Region formed with 5 km sized partition.

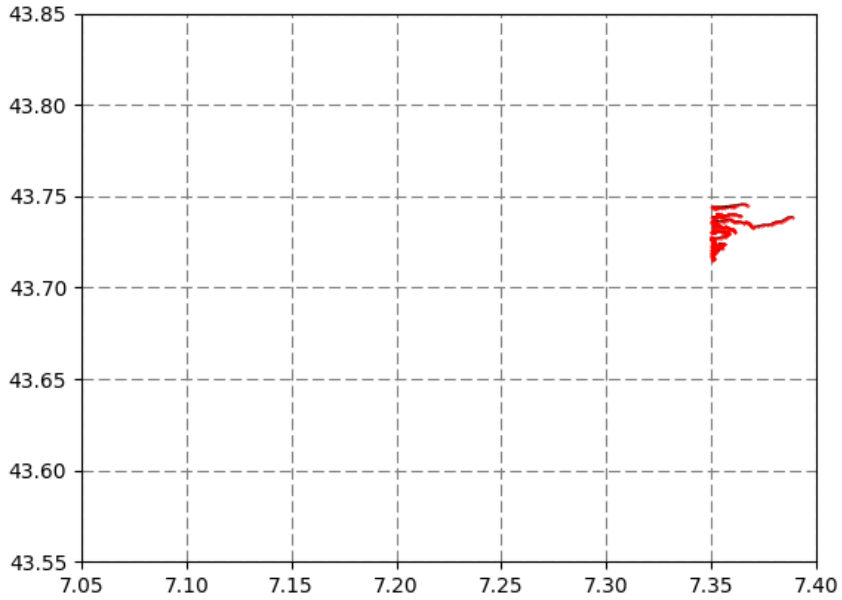


Figure 5. Region neighboring the one on the fig. 4

A.3 Closer look at the data

From the distance it may look that the plot is a solid region, however, the actual form of the plot is a graph. If we zoom the picture, we will obtain the real view. As the example we take the fig. 4.

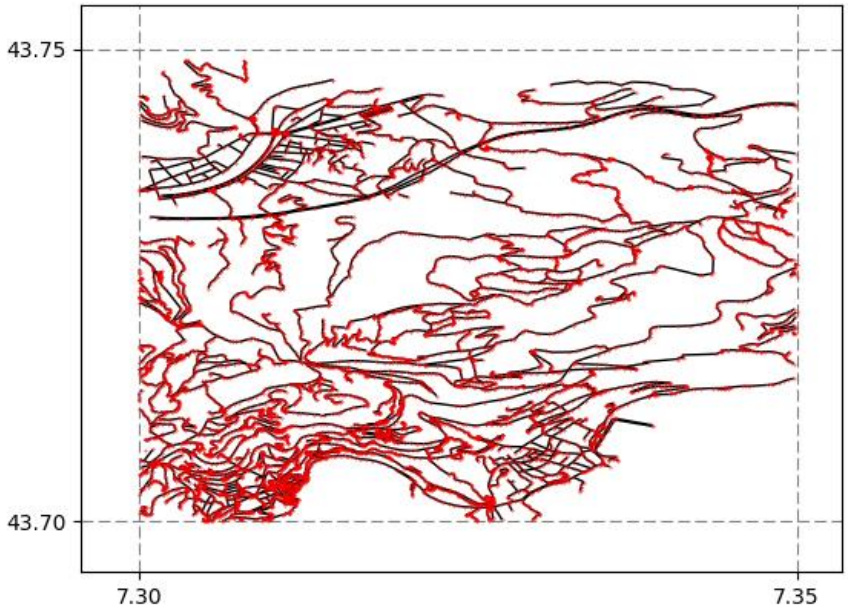


Figure 6. Zoomed fig.4

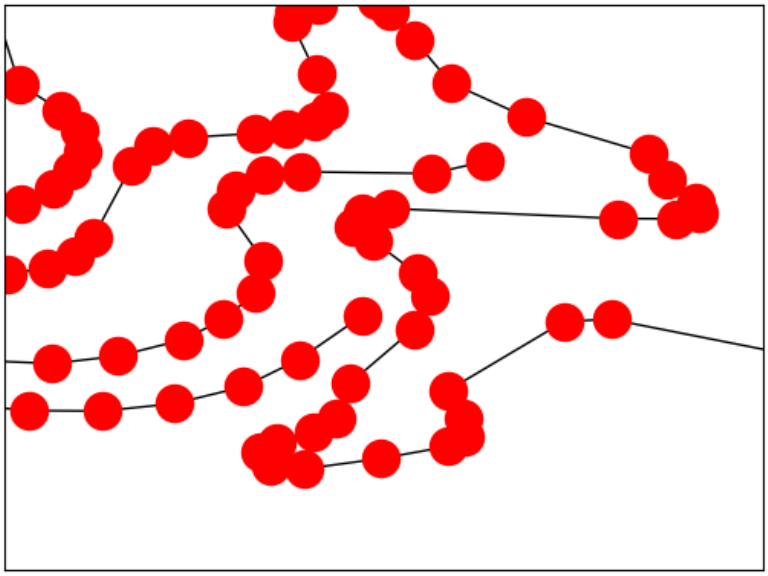


Figure 7. The real view of the graph.

A.4 Sophisticated partition

Results of our partition method applied to the map of Nice and Cagnes-sur-Mer are depicted on figure 8.

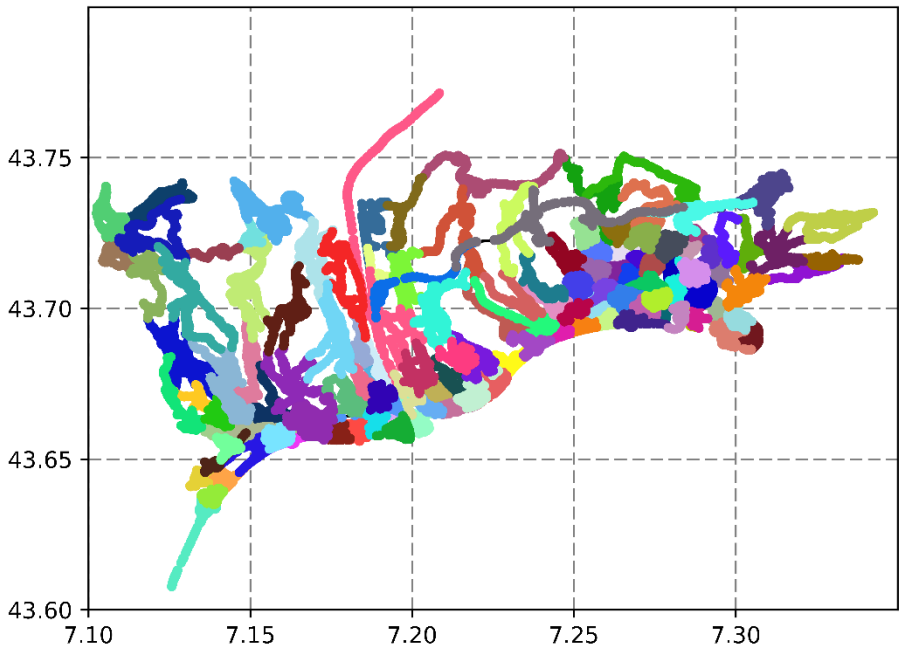


Figure 9. Results of sophisticated f partition

Closer look at the partition is shown on the figure 10.

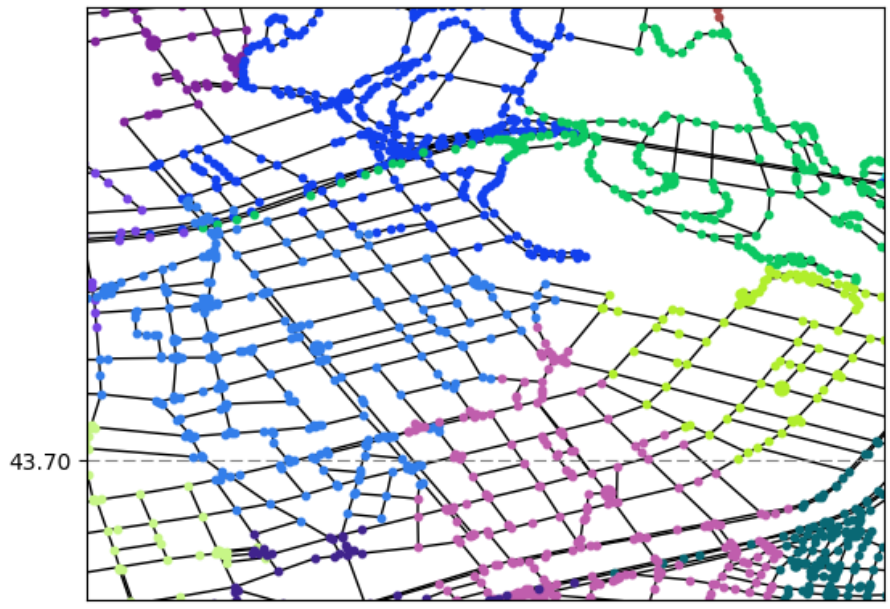


Figure 10. Zoomed results of partition.