CentraleSupélec

Computer aided design of a water supply network

AI & R Lab
Laboratorio del dioporco facile del Poliminchia

Relatore: **Filippo Gatti**
Correlatore: **Andrea Barbarulo**

Authors:
calvet-litzell Pere
Cognetti Giovanni
Marzorati Edoardo
Vaccarino Marcello

Academic year 2017-2018

In remembrance of Jean-Baptiste...

Abstract

Water supply remains a major issue in several countries. When designing a water supply network optimality is a priority. The aim of this project is to find optimal network structures using automation and machine learning. The development process is divided into several stages. During the first stage, network topology has been studied. A network has been designed using our software on real-world data.

Next stages will involve adding further parameters such as water velocity or pressure to the existing model. The project has a multidisciplinary nature. Using geographical data requires a certain level of acquaintance with different formats and software such as QGis. On the other hand, mastering a programming language like Python is required to implement the different algorithms and libraries.

Acknowledgements

Ringrazio ....

Chapter 1

Introduction

Water supply remains one of the major issues in developing countries. Through this project we seek to improve the design of water supply networks. Designing water supply networks is an optimization problem in which engineers have to find the best balance between cost, transport efficiency and resistance to failure.

1.1 General Context and Objectives

This project is centered on the design of water supply networks. More specifically it focuses on the design of new networks in developing countries with no existing infrastructures. The aim is to develop a software able to design a complete network from the map of a given area. In order to do so the team examined several techniques that could be suited to solve the problem. Among these were Root Architecture and Neural Networks. Eventually, clustering techniques and a Minimum Spanning Tree (MST) algorithm were chosen given their simplicity. They enabled to draw a water supply network from geographical data. This data was downloaded from Open Street Maps (OSM) and analysed with QGis. With regard to future developments, the project could be improved by focusing on water loss management, which is one of the major issues in water supply networks.

1.2 Brief description of the approach

The first step was to gather information on possible design techniques. Biologically-inspired models were examined. Prof Chloé Arson from GeorgiaTech provided very interesting information on root growth models and their applications to network design. Another path of research were Artificial Neural Networks. The idea was to explore the possibility of using this cutting-edge technology to draw a complete water supply network. However, it proved to be rather complex to master and not fully suited to the necessities. Finally, a more simple technique was retained. It uses a clustering and a Minimum Spanning Tree algorithm. The clustering algorithm allows to identify the villages on the map and the MST links them. The algorithms were implemented on Python using two libraries: "scikit-learn" and "NetworkX". The geographical data that was used as input of the algorithms was downloaded from Open Street Maps. Maps were processed using QGis, an open-source geographical information system. QGis allowed to select the most important data from the maps and save it as a

"shapefile", a format supported by the NetworkX library.

1.3   Report Structure

The report is organised as follows. **??** covers the state-of-the-art in water supply network design. It is an overview of the current techniques. It also provides information on the current issues in this domain. **??** is a detailed description of the approach. The different techniques studied throughout the project are presented and discussed. **??** explains our solution which is based on clustering and MST algorithms. **??** gives an insight into the more technical aspects of the project. Details on the software used can be found in this section. Finally, **??** presents the results obtained with the technique described in **??**.

Chapter 2

State of the art

A water supply system has two primary requirements: first, it needs to deliver adequate amounts of water to meet consumer consumption and fire requirements that must be available 24 hours a day, 365 days a year. Second the quality of the water must meet the regulatory requirements to guarantee a human consumption. Nowadays water distribution systems can be divided into four main components:

- Water sources and intake works: the common sources for the untreated or raw water are surface water sources such as rivers, lakes, springs, and man-made reservoirs and groundwater sources such as bores and wells.

- Treatment works and storage: The intake structures and pumping stations are constructed to extract water from the water sources. The raw water is transported to the treatment plants for processing through transmission mains and is stored in clear water reservoirs after treatment. The degree of treatment depends upon the raw water quality and finished water quality requirements. The clear water reservoir provides a buffer for water demand variation as treatment plants are generally designed for average daily demand.

- Transmission mains: water is carried over long distances through transmission mains. If the flow of water in a transmission main is maintained by creating a pressure head by pumping, it is called a pumping main. On the other hand, if the flow in a transmission main is maintained by gravitational potential available on account of elevation difference, it is called a gravity main

- Distribution network: There are no intermediate withdrawals in a water transmission main. Similar to transmission mains, the flow in water distribution networks is maintained either by pumping or by gravitational potential. Generally, in a flat terrain, the water pressure in a large water distribution network is maintained by pumping; however, in steep terrain, gravitational potential maintains a pressure head in the water distribution system. A distribution network delivers water to consumers through service connections. Such a distribution network may have different configurations depending upon the layout of the area. Generally, water distribution networks have a looped and branched configuration of pipelines, but sometimes either looped or branched configurations are also provided depending upon the general layout plan of the city roads and streets.

The design problem of a looped network is one of the most difficult problems of optimisation, the main objective is to estimate design variables by minimizing total system cost subject to safety and system constraints. The essential parameters are the projection of water demand, peak flow factors, minimum and maximum commercial pipe size, pipe material and reliability consideration. These nonlinear programming problems can only be solved numerically and not mathematically.

sectionWater supply networks equations The equations that characterise a water supply networks are the continuity equation and the equation of motion in most of the case for a study flow, that come down to the Bernulli's equation.

The continuity equation for steady flow in a circular pipe of diameter D is

$$Q = \frac{\pi}{4} D^2 V \tag{2.1}$$

V = average velocity of flow Q = volumetric rate of flow, called discharge.
The equation of motion for steady flow is:

$$z_1 + h_1 + \frac{V_1^2}{2g} = z_2 + h_2 + \frac{V_2^2}{2g} + h_L \tag{2.2}$$

z1 and z2 = elevations of the centerline of the pipe, h1 and h2 = pressure heads, V1 and V2 = average velocities at sections 1 and 2, g = gravitational acceleration hL = head loss between sections 1 and 2. The head loss hL is composed of two parts: hf is head loss on account of surface resistance (also called friction loss), and hm is head loss due to form resistance, which is the head loss on account of change in shape of the pipeline (also called minor loss).

$$h_L = h_f + h_m \tag{2.3}$$

The term z + h is called the piezometric head; and the line connecting the piezometric heads along the pipeline is called the hydraulic gradient line.
The head loss on account of surface resistance is given by the Darcy-Weisbach equation

$$h_f = \frac{fLV^2}{2gD} \tag{2.4}$$

where L = the pipe length f = friction factor. The friction factors f = f(Re,) is function of the Reynolds number and the roughness projection . The form-resistance losses are due to bends, elbows, valves, enlargers, reducers. A form loss develops at a pipe junction where many pipelines meet. Similarly, form loss is also created at the junction of pipeline and service connection. All these losses, when added together, may form a sizable part of overall head loss. Form loss is expressed in the following form:

$$h_m = k_f \frac{V^2}{2g} \tag{2.5}$$

kf is form-loss coefficient that depend from the nature of the special piece.
sectionWater Sources

Most of the water supply networks use more than one type of water source. The most common water sources are:

Impounding Reservoir: Reservoirs created by placing dams across rivers, streams, or at the neck of a valley to capture runoff water

Fresh-Water Lakes: This is generally considered a functionally, inexhaustible supply of water, provided the intake lines are below the ice formation level, in order to prevent jamming at the inlet ports of the water supply system.

Wells: Driven or bored wells are excavated to the level of the water table. of well require low-lift or high-lift pumps depending on the depth of the water or the distance of the water from the surface. Larger municipalities may use a series of well fields, and usually pump up to gravity tanks to provide the required flow and pressure to the distribution system. sectionWater system classification There are some basic water system classifications that have been established to provide a basic reference for examining the adequacy and reliability or water systems.

High-level reservoir system: This classification refers to a water source that is at least 30 meter in elevation above the treatment facility in order to provide sufficient head pressure so that nonpumping station is required. (Figure). This type of system gives the opportunity to develop a very reliable water supply system. If there is sufficient elevation difference between the water intake source on the reservoir system and the distribution piping in the community, it is possible to design a water system that does not require one or more pumping stations. The head pressure for supplying water to the treatment plant and from the treatment plant to the distribution system is sufficient to meet both consumer demand and needed fire flows. This is also a very economical system since there is no substantial power requirement to run the water system beyond normal lighting requirements and requirements for establishing the level of water quality needed for the community. A gravity flow system that does not require a pump interface is considered the most reliable type of water system, since there is no mechanical component to break down or fail when the power source goes down. For the same reasons, it is the most economical type of water supply system to operate but also the less common.

Low level reservoir systems: Low-level dams, usually up to about 6 meters high, are typically built on steam flow that is considered reliable year round. The reservoir basin may be bulldozed to form a small lake that will supply water to communities of populations up to about 25,000. A basic configuration for a low-level dam is depicted in Figure. Low-level retention dams typically require a pumping station to transport water to the treatment plant and, if the land area is relatively flat, a second pumping station to pump treated water directly to the distribution system or to elevated storage to provide the required pressure and volume to meet instantaneous flow demand. The elevated storage can be designed to minimize the direct pumping requirements.

Direct pumping systems: Figure illustrates how a direct pumping station feeds water to the treatment plant and then a second pumping system transports water to a storage holding

area, such as a clear well, to a standpipe storage tank that is maintained full as domestic consumption varies throughout a single day. This minimizes the time the pump or pumps actually have to run. The pumps also may be designed and arranged to pump the treated water directly into the distribution system when there is a high demand on the water system. This could occur when there is a major fire in the community.

Pumping station at well sites and gravity storage: In this type of supply system, one drilled well, or a field of wells, feed water to a ground-level pumping station. This concept is presented in Figure 4-5. The quality of the ground water in most of the case is so good that the only treatment necessary is chlorination through an injection method in the pipes that carry the non-potable water. In most cases, any other required water treatment generally is handled in a similar manner. The treated water either flows by gravity to the distribution system or is pumped to one or more elevated storage tanks. Potable water flows by gravity from the storage tank to the distribution system.

sectionMethod of water distribution

Water is dispersed throughout the distribution system in a number of different ways, depending on local conditions or on regulations and requirements that influence water system design. The common methods of water distribution to the pipe network are reviewed under the following titles.

Pumps and elevated storage: Through the use of pumps and elevated storage, the excess water pumped during periods of low consumption is stored in elevated tanks or reservoirs. During periods of high consumption, the stored water supplements the water that is being pumped. This method allows fairly uniform flow rates and pressures throughout the water system. Consequently, this method generally is economical because the pumps may be operated at their rated capacity. Since the stored water supplements the supply used for fires and system breakdowns, this method of operation is fairly reliable. However, it is necessary that fire department pumpers be available to boost the pressure from fire hydrant to delivery water through hose streams at the proper nozzle pressure to confine, control, and extinguish developing structural fires and other related fire events. A more complete understanding of these concepts is presented under **distribution storage** below.

Pumps without storage: When stationary pumps are used to distribute water, and no storage is provided on the distribution system, the pumps force water at the required volume and pressure directly into the mains. The outlet for the water is through domestic taps on the system or through fire hydrants. This is the least desirable type of distribution system because a power failure could interrupt the water supply. In addition, as consumption varies, the pressure in the water mains is most likely to fluctuate. To conform to varying rates, several pumps are made available to add water output when needed, a procedure requiring constant attention at the water plant. Another disadvantage is the fact that the peak power demand of the water plant is likely to occur during periods of high electric power consumption, thus increasing power costs to operate the water system. However, one advantage of direct pumping is that a large stationary fire pump may be used on demand for structure fires. This pump increases the residual pressure to any desired amount permitted by the construction of the water mains.

Water demand rates

Three historical or predicted water demand rates are involved in the discussion of water system demand and design flow rate criteria for both consumer consumption and needed fire flow. These are as follows: 1) Average daily consumption: This is the average of the total amount of water used each day during a 1-year period (usually expressed in million litres per day, MLD) 2) Maximum daily consumption: This is the maximum total amount of water used during any 24-hour period in a 3-year period. This number should consider and exclude any unusual and excessive identified used of water that would affect the calculation. Such abnormal uses would include a water main break, a large-scale fire, or an abnormal industrial demand. This is often referred to as the MDC rate. 3) Maximum hourly demand: This is the maximum amount of water used in any single hour, of any day, in a 3-year period. It is normally expressed in litres per day. It is determined in litres per day by multiplying the peak hours by 24.

Distribution System Appurtenances

Water systems typically have three classifications of pipe used to transport to demand points throughout a community. These are identified as follows:

1) Primary feeders: These are large pipes, usually with diameters ranging from 30 to 120 cm, based on the size of the population served. Primary feeders transport water form the water treatment plant to corporation line of the community and/or to major water storage locations within the community. 2) Secondary feeders: These are connected to the primary feeders to transport water along the major streets of the community. Secondary feeders need to be in place to supply all commercial property, public buildings, and private sector buildings that have a needed fire flow over 3500 litres. Secondary feeders typically are 20 to 40 cm in diameter. 3) Distributor mains: These are used to transport water from the secondary feeders to individual streets in the areas of the community that have small businesses like convenience stores and gas stations but, more importantly, along residential streets. The minimum pipe size should be 10 cm and, based on the system design, a possible dead- end pipe may need to be 20 or even 25 cm.

The sizes of pipe associated with the three classifications of pipe in a typical water system are approximations. The needed pipe size throughout the built-upon areas of a community is based on the hydraulic gradient of the community, consumer consumption profiles through the community, needed fire flow at representative locations throughout the community, and, quite importantly, the two methods for laying and connecting pipe throughout the community. The traditional pipe system design is referred to as a Single-Point Feed System. In this case, water moves from the treatment plant to the community corporation line with a single primary feeder. The primary feed, in turn, supplies the secondary feeders along the main streets of the community, and the distributor mains supply the block frontage along the residential streets. In the single- point feed system, the pipe sizes need to meet the maximum daily consumption demand plus the needed fire flow. This results in a larger pipe than is needed under normal daily usage, without any fires. The second major weakness of this type of system is that there may be a lot of dead-end mains in the residential areas and at the end of secondary mains. This leads to the stagnation of water which rapidly reduces the quality of water.

The more modern approach to water system design is to loop all the water mains, or cross-tie the mains, so that at any demand point the water is supplied from two directions. This

allows the designing engineer to develop a hydraulic model of the system and to determine mathematically the proper size of pipe according to flow paths to meet the consumer and needed fire flow demand points. It is important to note that many older water systems have been updated. By laying a primary feeder around the perimeter of the community to tie in all of the dead-end mains to improve both flow distribution and water pressures through the community.

Chapter 3

Approach

In this chapter we would like to show the approach we followed in the solution of the problem. We will give brief explanations of the techniques we investigated and justify the reasons why finally chose to use some of them an not others.

3.1 Root architecture

Biologically inspired models can provide interesting insights. Organisms that have gone through several rounds of evolutionary selection seem to be able to deliver efficient and nearly-optimal solutions. The use of such models seems to have produced satisfactory results for transport networks.

Reading Chloé Arson's presentation on bio-inspired geomechanics, we discovered the potential advantage of using root system architecture to design water lines. Prof. Arson conducted an experiment to compare the predictions of a root growth model with real water line networks. Root growth is a gene-controlled phenomenon. Therefore, different species may present different growth patterns. In addition, soil structure has also an influence on root structures. For example the presence of physical obstacles, such as boulders, alters geotropic growth. Prof. Arson also pointed out that a rocky soil would require a different model. Other characteristics like water and nutrient gradients or bacteria play a key role in root growth. Prof. Arson's experiment consisted in growing roots on a scale plastic model of the Georgia Tech campus. The results would allow to validate the accuracy of the mathematical model. Afterwards they could be compared to the existing water network and thus assess its efficiency. Prof. Arson also introduced leaf venation systems which bear certain resemblance to water line networks. Indeed, the growth of a leaf is governed by the presence of auxin (plant hormones) sources which can be seen as the nutrient sources of the root model.

We contacted Prof. Arson who gave us a very interesting bibliography on the subject of root growth models. Prof. Pierret's article stresses the complex relationship between soil structure and soil biological activity. Soil is a habitat for many organisms and is also responsible for the movement and transport of resources which are necessary for their survival. Through their roots, plants play a key role in many soil processes. Soil properties affect root growth which in turn affects resource acquisition and therefore the plant's impact on its environment (soil). Interest for root systems architecture comes from the necessity in agriculture of increasing

productivity and minimizing water and nutrient losses. A good understanding of soil processes seems necessary to achieve this end. Moreover, Pierret points out that whereas soil biological and chemical processes have been carefully studied, physical processes need more attention. The article examines main biological factors that influence soil processes. It underlines the complex interactions between physical and chemical-biological processes and the impossibility to treat them separately. According to Pierret, roots are essential to study this complexity. In the second part of the article, the huge diversity of root classes is examined. This implies the necessity of using specific models for each species. The last part of the article discusses how modelling can provide clearer insights on the interactions between roots and soil.

Lionel Dupuy's article describes the evolution of root growth models. The first models appeared in the early 1970s and focused mainly on root length. However since the 1990s new complex models have emerged thanks to the use of more powerful computers. phenomenon has been fostered by the ïeed for predictive technologiesät different scales. Dupuy suggests a new theoretical framework which takes into account individual root developmental parameters. He introduces "equations in discretized domains that deform as a result of growth". Simulations conducted by Dupuy have revealed some patterns in what seemed a complex and heterogeneous problem. More precisely, it seems that roots develop following travelling wave patterns of meristems.

V. M. Dunbabin also mentions the progress accomplished in the area of root growth modelling. The early models did not take into account the root growth in response to a heterogeneous soil environment. Nowadays, models must include soil properties and accurate descriptions of plant function. The aim of these simulations is again to provide a better understanding of the efficient acquisition of water and nutrients by plants. Resource availability has a clear impact on both the roots and the stem of the plan. For example, a low nutrient concentration diminishes shoot growth and therefore leaf and stem mass fractions as well. It has been observed that roots respond locally to soil properties. This characteristic allows the plant to forage with more precision and reduce metabolic cost. Three-dimensional models are able to seize the complexity of the problem. Previous models were rather simple and relied upon one-dimensional functions of rooting depth vs. time.

One of the most interesting articles is Atsushi Tero's "Rules for Biologically Inspired Adaptive Network Design". In order to solve the problem of transport networks efficiency, Tero created a mathematical model based on organisms that build biological networks. He explains that these biological networks have been honed by many rounds of evolutionary selection and that they can provide inspiration to design new networks. He praises their good balance between cost, transport efficiency and, above all, fault tolerance. One of such organisms is physarum polycephalum, a type of slime mold. Tero let physarum grow on a map of the Tokyo area where major cities were marked by food sources. A first network was obtained. In order to improve the results, the experiment was carried out a second time. However, illumination was used to introduce the real geographical constraints such as coastlines or mountains (illumination reduces physarum's growth). The results were very satisfactory and the biological network was very similar to the existing Tokyo transport network. Tero developed a mathematical model that tried to reproduce Physarum's behavior. The principle of the model is that tube thickness depends on the internal flow of nutrients. Thus a high

rate tends thickens a tube and a low rate leads to its decay. As shown by Prof. Arsons' paper "Bio-inspired fluid extraction model for reservoir rocks", slime mold growth can also be used to study the flow in a porous medium. The use of Root Architecture Models was abandoned in order to investigate the use of artificial intelligence techniques

3.2 Artificial intelligence

The growth of network usage and their increasing complexity, in particular for communication technologies application, drives towards the improvement of routing technique. One track of this research is the development of s̈martẗechniques for network design and management.

For our project we chose to follow this direction, combine sub-optimal AI algorithms to develop a possibly innovative solution. Lead by example, we will give an overview of the most edge braking applications in this field. This will allow us to introduce the main concepts and get down to the techniques we focused on.

AI is applied to many complex routing problems: one example is very large-scale integration (VLSI). The process of designing integrated circuits is hard due to the large number of often conflicting factors that affect the routing quality such as minimum area, wire length. Rostam Joobbani, a knowledge-based routing expert from Carnegie-Mellon University (1986), proved that an AI approach to the subject could dramatically improve performances.

A more recent example is the use of AI in Wireless Sensor Networks. WSNs are spatially distributed autonomous sensors to monitor physical or environmental conditions, such as temperature, sound, pressure, etc. and to cooperatively pass their data through the network to other locations. Management of those networks is particularly challenging because of the dynamic environmental conditions. J. Barbancho and al. (2007) wrote a review about the use of artificial intelligence techniques for WSNs for path discovery and other purposes. The study shows the potential of Artificial Neural Networks.

3.2.1 Artificial Neural Networks

ANNs learn to do tasks by considering examples, generally without task-specific programming. An ANN is based on a collection of connected units called artificial neurons. Each connection (synapse) between neurons can transmit a signal to another neuron. The receiving (postsynaptic) neuron can process the signal(s) and then signal downstream neurons connected to it.

N. Ahad, J. Quadir, N. Ahsan (2016) published a review focused on techniques and applications of artificial neural networks for wireless networks. The advantage of using ANN is that can make the network adaptive and able to predict user demand.

Concerning shortest path problems Michael Turcanik (2012) used a Hopfield neural network as a content-addressable memory for routing table look-up. A routing table is a database that keeps track of paths in a network. Whenever a node needs to send data to another node on a network, it must first know where to send it. If the node cannot directly connect to the destination node, it has to send it via other nodes along a proper route to the destination

node. Most nodes do not try to figure out which route(s) might work; instead, a node will send the message to a gateway in the local area network, which then decides how to route the p̈ackageöf data to the correct destination. Each gateway will need to keep track of which way to deliver various packages of data, and for this it uses a Routing Table. Turcanik replaced the table with an ANN. His study shows the performance of routing table look-up in terms of speed and adaptability.

This excursus gives an idea of the incredibly various applications of ANN in routing problems. We would like now to focus on the use of Hopfield Neural Network, which is the most classical solution for routing problems with ANN's. To finish we will explain the conclusions we came to regarding neural networks.

Hopfield (1984) proposed the use of his algorithm to give heuristic solutions to the travel salesman problem. TSP is a well known NP-hard minimization problem. As defined by Karl Menger the TSP is "the task to find, for finitely many points whose pairwise distances are known, the shortest route connecting the points". So having n cities, our travel salesman has to associate to each city X a position k in the tour so that:

$$\sum_{X} \sum_{Y \neq X} \sum_{j} d_{XY} y_{Xj} (y_{Y,j+1} + y_{Y,j-1})$$

is minimal, where dXY is the distance between city X and Y.

E. Wacholder and al. (1989) developed a more efficient implementation of the Hopfield NN for the travel sales-man problem. The algorithm was successfully tested on many problems with up to 30 cities and five salesmen. For comparison a non-optimized brute-force approach would take billions of billions of years to return but optimal algorithms with record breaking performance can solve problems with more than 3000 cities.

Mustafa K. Mehmet Ali and Faouzi Kamoun (1993) considered modelling shortest path problem with Hopfield Neural Network for the first time. The researchers asserted that HNN can find shortest path effectively and sometimes it would be better to use such a network instead of classic algorithms such as Dijkstra. Please find in section **??** a thecnical explanation on how the Hopfield Neural Networks works, with particular attention to the TSP application

After a wide research in the field of Artificial Neural Networks and besides all the encouraging proofs we collected we concluded that are not the best suited tool to solve routing problems. ANN alone are less efficient in solving routing problems than optimal approaches. In addition, it is not possible to know whether the ANN will deliver a solution that will converge and whether this solution is optimal. Thus ANNs as a poorly efficient way to solve the problem. Nonetheless, machine learning can still be useful. Indeed, the interpretation of geographical information files can be done through clustering algorithms.

3.2.2 Clustering

Chapter 4

Progetto logico della soluzione del problema

4.1   Problem staterment and modelisation

The overall idea of our software solution is to connect water sources and consumers to a network of pipes in the most efficent way possible. Indeed the definition of the best net is a key problem. Factors that makes an aqueduct the optimal one are not easily modeled. We can suppose that variables such length, height, water speed and pressure, viscosity ecc should be taken into account.

For our first approach, we decided to consider only the pipeline length so that we simplify acqueduct design to a classical routing problem. On this base, we will then be able to add complexity. We can now more formaly define our poblem. Being a topography a graph representing the meshed surface of a region, the problem of designing an acqueduct as the one of finding the recovering-graph on the topograpy graph connecting water consumers and sources. We will use the euclidian metric on a space with tree dimensions.

data

Graph
initialis-
taion

Topograpy graph

Aqueduc

Routing

Clustering

Water network

4.2 Data reading

In **??** is shown to procedure followed by our software. The input is composed of three geographical files: mesh represent the topology of the studied region, source-sinks contains the locations for each water source and consumer (called sink from now on). The last file describes the roads network in the area as the pipes should preferably run along roads for is cheaper to place them. For more information about the geographical data format, please refer to **??** Reading those input two graphs can be initialised: the topography and the aqueduct describing graphs. The first is a graph where nodes represent positions and edges the possible transitions between them. Transitions on roads will be preferable. The aqueduct graph is initialised with source and sinks as nodes and no edges. An insight of the data structure we used to represent graph is given in section **??**

4.3 Clustering

Running classical algorithms such a brute-force TSP or a minimum spanning tree to link the nodes in the sink-source graph would not be feasible for computational reasons. Thanks to the particular nature of our problem a simplification is possible: we divide the aqueduct system in two layers: adduction and distribution nets. The adduction layer brings water from the source to the inhabited areas whereas the distribution segment is in charge of the "last kilometre" distribution. This two layer solution is commonly used in aqueduct design and network design in general: internet is an example. To achieve this need to recognise group of buildings such as villages or neighbourhoods. Those sets are called sink clusters. This approach caries multiple advantages. From the computational point of view reduce the dimension of the sets on which we run the routing algorithms. On the other side, once the two layers are identified we can use different strategies to connect the nodes, as we will se in the next paragraph. Efficient implementations of clusterings algorithms are provided in scikit-learn. Scikit-learn is a well-known machine learning library for Python and it features various classification, regression and clustering algorithms. After this operation sinks are labeled as part of they respective cluster. A more detailed explanation can be found at **??**.

4.4 Routing

We now can design the water systems connecting the sinks. Let's first consider the distribution layer, which is to say the problem of connecting the sinks of a cluster. This operation is broken down in two tasks. First, find all the paths connecting sinks, than chose the smallest recovering graph, i.e. the smallest aqueduct satisfying the specifics. To find the path connecting the sinks an optimal approach is used on the topography graph, in our case the Dijstrak algorithm. The length of those path and the traversed nodes are saved as attributes of edges of the aqueduct graph. Please pay attention to the fact that edges on the aqueduct graph are paths on the topography graph. This operation creates a complete graph for each cluster. Note that the set of those graph is a partition of the aqueduct graph. The second stage consists in eliminating the redundant edges: to do this we run the Kruskal algorithm and calculate the minimum spanning tree. For more information read section **??**. An other approach, favoured in classical aqueducts

design would be to calculate a partially connected graph where certain nodes are connected to exactly one other node whereas others are connected to two or more other nodes. This makes possible to have some redundancy without the expense and complexity required for a connection between every node in the network. At this first stage of the project this part is left for further investigation. Considering now the adduction system a very similar approach can be used: the clusters should be interlinked and connected to a source. Each distribution network, as is a tree, has a root node. We can initialise the adduction graph with all the cluster's roots nodes and the water sources. Finally this graph is connected with the technique previously used.

Chapter 5

Technical aspects

5.1 Geographical Data

The overall idea is to take maps and automatically trace an aqueduct on it, in order to do that, we start from the map's shapefile. Shapefile is a popular geospatial vector data format for geographic information systems software. It spatially describes geometries: points, polylines and polygons. These, for example, could represent water wells, roads or buildings. As those primitive geometrical data types come without any attributes to specify what they represent, a table of records to store attributes is provided. Websites like osm2shp or Geofabrik provide an immense database of shapefiles available for download. Moreover desktop software like Qgis provides shapefile editing tools. This way we can both download real-world maps and create our own. Then through Qgis' meshing plug-in Gmsh we can mesh the surfaces of the map and export the result in vtk format as seen in Fig. 3.1 However, shapefiles seldom have information on the elevation (that is the Z coordinate) of the objects they represent. It is therefore necessary to use another format: the Digital Elevation Model (DEM). Digital Elevation Models provide this missing piece of information that can subsequently be added to the shapefile's attribute table. DEMs can be converted into meshes thanks to software such as SAGA. Meshes saved as vtk files can easily be used in Python. Vtk files are a simple and efficient way to describe mesh-like data structures. The vtk file boils down to those two elements: points and cells. Points have 3D coordinates while cells are surfaces, expressed by the points delimiting them. Point and cell data (scalar or vector) can also be assigned. We have therefore a file representing a graph, a classical mathematical model on which many operations can be performed: routing and clustering among others. We now come to our software. Python has been chosen as easy to use, widespread programming language, good for rapid prototyping and rich in package and libraries. The problem is divided in two main tasks: modelling the data structure that represents the graph and the algorithmic part, the aqueduct design.

5.2 Data Structure

To implement the data-structure we chose to use NetworkX. NetworkX is a Python package for the creation, manipulation, and study of complex networks. The package provides classes for graph objects, generators to create standard graphs, IO routines for reading in existing

datasets, algorithms to analyze the resulting networks and some drawing tools. The software takes as input two shape files: the first describes the topology, the second the source and sinks. The topology is either a mesh, representing the geography of the region or a polyline with just the road network of the region. The roads are particularly important because aqueducts are built along roads for logistical reasons. The second file is a polygon file containing the buildings that should be served by the aqueduct and the water sources. From these data, a first graph is obtained. The graph has as nodes the points described in topology file plus the buildings. The coordinates of binding-representing nodes are the average of the coordinates that also have the metadata associated. The edges are the edges described in the topology file plus the edges connecting the building to the nearest node of the network in order to obtain a connected graph.

5.3   Hopfield neural network

We will now explain what Hopfield Neural Networks are, with particular attention to the TSP application, although the definition we will give is general. It is a recurrent ANN, as opposed to feed forward NN, which means neurons interconnections forms a directed cycle, so neurons are both input and output. Hopfield nets are sets n2 nodes where X [1, n] and k [1, n] and the state is characterized by the binary activation values y = (yXj) of the nodes. A TSP problem with n cities can be modeled as an Hopfield net of dimension n2, where yXj is 1 if the city X is in the k-position of the tour.

The input sk(t+1) of the neuron k is:

$$s_k\left(t+1\right) = \sum_{j \neq k} y_i\left(t\right) w_{jk} + \theta_k$$

where wjk is the weight of the connection between j and k and is the bias The forward function is applied to the node input to obtain the new activation value at time t+1:

$$y_k\left(t\right) = sgn(s_k(t-1))$$

The energy function is as follow so that the optimal solution will minimize it:

$$E = \frac{A}{2}\sum_X\sum_j\sum_{k \neq j} y_{Xj}y_{Xk} + \frac{B}{2}\sum_j\sum_X\sum_{X \neq Y} y_{Xj}y_{Yj} + \frac{C}{2}(\sum_X\sum_j y_{Xj} - n)^2$$
$$+ \frac{D}{2}\sum_X\sum_Y\sum_j d_{XY}y_{Xj}(y_{Y,j+1} + y_{Y,j-1})$$

The first two terms are null if and only if the there is a maximum of one active neuron for each row and column respectively. The third term is null if and only if there are n active neurons. The last term takes in account the distance of the path, that should be minimized as well.

The Hebbian rule to update the weights is deduced from the energy function:

$$w_{Xj,Yk} = -A\delta_{XY}\left(1 - \delta_{jK}\right) - B\delta_{jk}\left(1 - \delta_{XY}\right) - C - Dd_{XY}(\delta_{k,j+1} + \delta_{k,j-1})$$

where kj = 1 if j = k and zero otherwise. As in the energy function the first term inhibits connection within each row, the second within columns, the third is the global inhibition and the last term takes into account the distance between the cities.

Under the hypothesis $w_{Xj,Yk} = w_{Yk,Xj}$ the method can be proved to have stable points. At each iteration the net updates his parameters according to the Hebbian rule and the evolution of the state can be proved to be monotonically nonincreasing with respect of the energy function. Performing then a gradient descent, after a certain number of repetition the state converge to a stable point that is a minima of the energy function.

5.4 Clustering

In this section we will explain the different routing techniques used.
Dijstrak

Dijkstra's algorithm is an algorithm for finding the shortest paths between two nodes in the graph. Here is the pseudo-code

function Dijkstra(Graph, source): 2 3 create vertex set Q 4 5 for each vertex v in Graph: // Initialization 6 dist[v] INFINITY // Unknown distance from source to v 7 prev[v] UNDEFINED // Previous node in optimal path from source 8 add v to Q // All nodes initially in Q (unvisited nodes) 9 10 dist[source] 0 // Distance from source to source 11 12 while Q is not empty: 13 u vertex in Q with min dist[u] // Node with the least distance 14 // will be selected first 15 remove u from Q 16 17 for each neighbor v of u: // where v is still in Q. 18 alt dist[u] + length(u, v) 19 if alt ¡ dist[v]: // A shorter path to v has been found 20 dist[v] alt 21 prev[v] u 22 23 return dist[], prev[]

5.5   Travelling salesman problem

Chapter 6

Presentation and validation of experimental results

The clustering algorithm and the MST were executed on a shapefile obtained from a map of Limitone, near Potenza in southern Italy. The network obtained is shown in Fig??. Several sub-networks can be distinguished. The main one is the adduction network which carries the water to every village. The other ones are distribution networks which distribute water within every village. This test showed that the clustering algorithm was able to identify villages from a shapefile that only displayed the buildings of the area. In addition, it also showed that it was possible to build a MST linking all the villages and buildings.
[ADD PHOTO]

Chapter 7

Conclusions

7.1 Summary of Thesis Achievements

Si mostrano le prospettive future di ricerca nell'area dove si è svolto il lavoro. Talvolta questa sezione può essere l'ultima sottosezione della precedente. Nelle conclusioni si deve richiamare l'area, lo scopo della tesi, cosa è stato fatto,come si valuta quello che si è fatto e si enfatizzano le prospettive future per mostrare come andare avanti nell'area di studio.

7.2 Applications

7.3 Future wok

Appendix A

Documentazione del progetto logico

Documentazione del progetto logico dove si documenta il progetto logico del sistema e se è il caso si mostra la progettazione in grande del SW e dell'HW. Quest'appendice mostra l'architettura logica implementativa (nella Sezione 4 c'era la descrizione, qui ci vanno gli schemi a blocchi e i diagrammi).

Appendix B

Documentazione della programmazione

Documentazione della programmazione in piccolo dove si mostra la struttura ed eventualmente l'albero di Jackson.

Appendix C

Listato

Il listato (o solo parti rilevanti di questo, se risulta particolarmente esteso) con l'autodocumentazione relativa.

```python
# -*- coding: utf-8 -*-

import networkx as nx
import math
import sys

# sys.setdefaultencoding('utf8')

"""
Created on Mon Dec  4 22:57:30 2017

@author: Conrad
"""


class Router(object):
    #_____

    # --- CLASS ATTRIBUTES
    #_____


    # Class description
    CLASS_NAME = "Router"
    CLASS_AUTHOR = "Marcello Vaccarino"

    # Attributes
    graph = nx.Graph()
    sinksource_graph = nx.Graph()
    acqueduct = nx.Graph()
    #_____

    # --- INITIALIZATION
    #
```

```python
33      def __init__(self, topo_file=None, building_file=None):

35          if topo_file != None and building_file != None:
                try:
37                  # [TODO] this function does not read building but single points
                    self.read_shp(topo_file, building_file)
39              except Exception as e:
                    raise e
41          elif building_file != None:
                try:
43                  self.read_shp_bilding(building_file)
                except Exception as e:
45                  raise e
            elif topo_file != None:
47              try:
                    self.read_vtk(topo_file)
49              except Exception as e:
                    raise e
51
    #

53  # -- CLASS ATTRIBUTES
    #

55      def avg(self, node_list):
            x = 0
57          y = 0
            for node in node_list:
59              x += node[0]
                y += node[1]
61          x /= len(node_list)
            y /= len(node_list)
63          return (x, y)

65      def read_shp_bilding(self, file_name):

67          try:
                import shapefile
69          except ImportError:
                raise ImportError("read_shp requires pyshp")
71
            sf = shapefile.Reader(file_name)
73          fields = [x[0] for x in sf.fields]

75          for shapeRecs in sf.iterShapeRecords():
                shape = shapeRecs.shape
77              for cell in self.row_chuncker(shape.points, shape.parts):
                    center_coord = self.avg(cell)
79                  attributes = dict(zip(fields, shapeRecs.record))
                    attributes['pos'] = center_coord
```

```python
                    self.graph.add_node(center_coord)

    # chuncker: see commpresed row storage
    def row_chuncker(self, array, p_array):
        p_array.append(len(array))
        return [array[p_array[i]:p_array[i+1]]
                for i in range(len(p_array)-1)]

    def read_shp(self, file_name, point_file=None):
        """Generates a networkx.DiGraph from shapefiles. Point geometries are
        translated into nodes, lines into edges. Coordinate tuples are used as
        keys. Attributes are preserved, line geometries are simplified into
        start and end coordinates. Accepts a single shapefile or directory of
        many shapefiles.

        "The Esri Shapefile or simply a shapefile is a popular geospatial
        vector data format for geographic information systems software."

        Parameters
        ----------
        path : file or string
            File, directory, or filename to read.

        simplify : bool
            If ``True``, simplify line geometries to start and end coordinates.
            If ``False``, and line feature geometry has multiple segments, the
            non-geometric attributes for that feature will be repeated for each
            edge comprising that feature.

        Returns
        -------
        G : NetworkX graph

        Examples
        --------
        >>> G=nx.read_shp('test.shp') # doctest: +SKIP

        References
        ----------
        .. [1] http://en.wikipedia.org/wiki/Shapefile
        """
        try:
            import shapefile
        except ImportError:
            raise ImportError("read_shp requires pyshp")

        sf = shapefile.Reader(file_name)
        fields = [x[0] for x in sf.fields]

        for shapeRecs in sf.iterShapeRecords():

            shape = shapeRecs.shape
            if shape.shapeType == 1:      # point
                attributes = dict(zip(fields, shapeRecs.record))
                attributes["pos"] = shape.points[0]
```

```python
                        self.graph.add_node(shape.points[0], attributes)
137
                if shape.shapeType == 3:      # polylines
139                 attributes1 = dict(zip(fields, shapeRecs.record))
                    attributes2 = dict(zip(fields, shapeRecs.record))
141                 for i in range(len(shape.points) - 1):
                        '''
143                     attributes1["pos"] = shape.points[i]
                        n1 = self.add_node_unique(shape.points[i], attributes1)
145                     attributes2["pos"] = shape.points[i + 1]
                        n2 = self.add_node_unique(shape.points[i + 1], attributes2)
147                     attribute = {'dist': self.distance(n1, n2)}
                        print '{0}: {1}, {2}'.format(i, n1, n2)
149                     self.graph.add_edge(n1, n2, attribute) '''
                        attributes1["pos"] = shape.points[i]
151                     n1 = shape.points[i]
                        self.graph.add_node(n1, attributes1)
153                     attributes2["pos"] = shape.points[i + 1]
                        n2 = shape.points[i + 1]
155                     self.graph.add_node(n2, attributes2)
                        attribute = {'dist': self.distance(n1, n2)}
157                     self.graph.add_edge(n1, n2, attribute)

159             if shape.shapeType == 5:      # polygraph
                    # chuncker: see commpresed row storage
161                 def chuncker(array, p_array):
                        p_array.append(len(array))
163                     return [array[p_array[i]:p_array[i+1]]
                                for i in range(len(p_array)-1)]
165
                    # given a cell returns the edges (node touple) implicitely
        defined in it
167                 def pairwise(seq):
                        return [seq[i:i+2] for i in range(len(seq)-1)]
169
                    for cell in chuncker(shape.points, shape.parts):
171                     for n1, n2 in pairwise(cell):
                            attributes1 = dict(zip(fields, shapeRecs.record))
173                         attributes1["pos"] = n1
                            attributes2 = dict(zip(fields, shapeRecs.record))
175                         attributes2["pos"] = n2
                            # add nodes of the shape to the graph
177                         n1 = self.add_node_unique(n1, attributes1)
                            n2 = self.add_node_unique(n2, attributes2)
179                         attribute = {'dist': self.distance(n1, n2)}
                            # add edge
181                         self.graph.add_edge(n1, n2, attribute)

183         if point_file != None:
                sf = shapefile.Reader(point_file)
185             new_fields = [x[0] for x in sf.fields]
                nodes2attributes = {node: data \
187                                 for node, data in self.graph.nodes(data=1)}
                for shapeRecs in sf.iterShapeRecords():
189                 shape = shapeRecs.shape
```

```python
                    if shape.shapeType != 1:     # point
                        raise ValueError("point_file must be of type 1: points")
                    new_attributes = dict(zip(new_fields, shapeRecs.record))
                    nodes = [tuple(point) for point in shape.points]
                    for node in nodes:
                        # print node in nodes2attributes, new_attributes
                        nodes2attributes[node].update(new_attributes)
            for node, data in self.graph.nodes(data=1):
                    data.update(nodes2attributes[node])
            # nx.set_node_attributes(self.graph, nodes2attributes)

    def write2shp(self, G, filename):
        try:
            import shapefile
        except ImportError:
            raise ImportError("read_shp requires pyshp")

        w = shapefile.Writer(shapeType=3)
        #w.field("DC_ID", "LENGHT", "NODE1", "NODE2", "DIAMETRE", "ROUGHNESS",
    "MINORLOSS", "STATUS", "C")

        w.fields = [("DeletionFlag", "C", 1, 0), ["DC_ID", "N", 9, 0],
            ["LENGHT", "N", 18, 5], ["NODE1", "N", 9, 1], ["NODE2", "N", 9, 0],
            ["DIAMETRE", "N", 18, 5], ["ROUGHNESS", "N", 18, 5], ["MINORLOSS",
    "N", 18, 5],
            ["STATUS", "C", 1, 0]]

        i = 0
        lenghts = nx.get_edge_attributes(G, 'dist')
        print(lenghts)
        for edge in lenghts:
            line = [edge[0], edge[1]]
            w.line(parts=[line])
            w.record(i, lenghts[(edge[0], edge[1])],
                        1, 2, 100, 0.1, 0, "1")
            i+=1
        w.save(filename)

    def write2vtk(self, G):

        # import sys
        # sys.path = ['..'] + sys.path
        import pyvtk

        points = [list(node) for node, data in G.nodes(data=True)]
        line = []
        for edge in G.edges():
            for i, node in enumerate(G.nodes()):
                if node == edge[0]:
                    n1 = i
            for i, node in enumerate(G.nodes()):
                if node == edge[1]:
                    n2 = i
            line.append([n1,n2])
```

```
243            vtk = pyvtk.VtkData(pyvtk.UnstructuredGrid(points, line=line))
               vtk.tofile('example1', 'ascii')
245
        def add_node_unique(self, new_node, new_attributes):
247            """
               grants that the node added is unique with respect to the pos
249            attribute equality relationship
               """
251            for node in self.graph.nodes(True):
                   if node[1]["pos"] == new_attributes["pos"]:
253                    return node[0]
               self.graph.add_node(new_node, new_attributes)
255            return new_node

257    def read_vtk(self, file_name):
               import numpy as np
259            try:
                   from mesh import Mesh
261            except ImportError:
                   raise ImportError("read_vtk requires pymesh")
263
               # initialize the vtk reader
265            reader = Mesh()

267            # read the vtk
               reader.ReadFromFileVtk(file_name)
269
               # add nodes to the graph
271            for index, node in enumerate(reader.node_coord):
                   self.graph.add_node(index, pos=reader.node_coord[index])
273
               '''
275            chuncker
               Principe bas  sur le stockage CSR ou CRS (Compressed Row Storage)
277            dont voici une illustration :
               Soient six n uds num rotes de 0    5 et quatre    el    ements    form
        es   par
279            les n uds (0, 2, 3) pour l' el ement 0, (1, 2, 4) pour l'element,
               (0, 1, 3, 4) pour  l     el    e − ment 2 et (1, 5) pour  l     el
        ement    3.
281            Deux tableaux sont utilis  es ,  l un   pour stocker de fa   con    contigu
         e
               les listes de n uds qui composent les    el    ements   (table 1),
        l  autre
283            pour indiquer la position , dans ce tableau , ou' commence chacune de
               ces listes (table 2).
285            Ainsi , le chiffre 6 en position 2 dans le tableau p elem2node indique
               que le premier n ud de l     el    ement  2 se trouve en position 6 du
287            tableau elem2node. La derni' ere valeur dans p elem2node correspond au
               nombre de cellules (la taille) du tableau elem2node.
289
               elem2node
291            0 | 2 | 3 | 1 | 2 | 4 | 0 | 1 | 3 | 4 | 1 | 5
               1   2   3   4   5   6   7   8   9   10  11  12
293            ^       ^           ^               ^       ^
```

```python
            p_elem2node
            0 | 3 | 6 | 10 | 12
            1   2   3    4    4
            '''

            def chuncker(array, p_array):
                return [array[p_array[i]:p_array[i+1]]
                        for i in range(len(p_array)-1)]

            # given a cell returns the edges implicitely defined in it
            def pairwise(seq):
                return [seq[i:i+2] for i in range(len(seq)-2)] + \
                       [[seq[0], seq[len(seq)-1]]]

            datas = np.asarray([data['pos']
                                for _, data in self.graph.nodes(data=True)])

            def distance3D(u, v, datas):
                xi = datas[u][0]
                yi = datas[u][1]
                zi = datas[u][2]
                xj = datas[v][0]
                yj = datas[v][1]
                zj = datas[v][2]
                return math.sqrt((xi-xj)*(xi-xj) +
                                 (yi-yj)*(yi-yj) + (zi-zj)*(zi-zj))

            # add edges to the graph
            for cell in chuncker(reader.elem2node, reader.p_elem2node):
                for u, v in pairwise(cell):
                    if u not in self.graph[v]:
                        self.graph.add_edge(u, v, weight=distance3D(u, v, datas))

        def distance(self, nodei, nodej):
            xi = nodei[0]
            yi = nodei[1]
            xj = nodej[0]
            yj = nodej[1]
            if len(nodei) == 3 and len(nodej) == 3:
                zi = nodei[2]
                zj = nodej[2]
                return math.sqrt((xi-xj)*(xi-xj) + (yi-yj)*(yi-yj) +
                                 (zi-zj)*(zi-zj))
            return math.sqrt((xi-xj)*(xi-xj)+(yi-yj)*(yi-yj))

        # cartesian norme in 2D
        def distance2D(self, nodei, nodej):
            xi = self.graph.nodes(data=True)[nodei][1]['pos'][0]
            yi = self.graph.nodes(data=True)[nodei][1]['pos'][1]
            xj = self.graph.nodes(data=True)[nodej][1]['pos'][0]
            yj = self.graph.nodes(data=True)[nodej][1]['pos'][1]
            return math.sqrt((xi-xj)*(xi-xj)+(yi-yj)*(yi-yj))

        # cartesian norme in 3D
```

```python
349      def distance3D(self, nodei, nodej):
             xi = self.graph.nodes(data=True)[nodei][1]['pos'][0]
351          yi = self.graph.nodes(data=True)[nodei][1]['pos'][1]
             zi = self.graph.nodes(data=True)[nodei][1]['pos'][2]
353          xj = self.graph.nodes(data=True)[nodej][1]['pos'][0]
             yj = self.graph.nodes(data=True)[nodej][1]['pos'][1]
355          zj = self.graph.nodes(data=True)[nodej][1]['pos'][2]
             return math.sqrt((xi-xj)*(xi-xj) + (yi-yj)*(yi-yj) + (zi-zj)*(zi-zj))
357
         # returns 2D coordinates of the nodes of self.graph
359      def coord2D(self, G):
             coord2D = {}
361          for key, value in nx.get_node_attributes(G,
                                                       'pos').iteritems():
363              coord2D[key] = [value[0], value[1]]
             return coord2D
365
         # display the mesh using networkx function
367      def display_mesh(self):
             nodelist = []
369          node_color = []
             for node in self.graph.nodes(data=1):
371              node_type = node[1]['FID']
                 if not node_type == '':
373                  nodelist.append(node[0])
                     node_color.append('r' if node_type == 'sink' else 'b')
375          try:
                 nx.draw_networkx(self.graph, pos=self.coord2D(), nodelist=nodelist,
377                              with_labels=0, node_color=node_color)
             except:
379              pass
381      # display the mesh with a path marked on it
         def display_path(self, path):
383          nodelist = []
             node_color = []
385          for node in self.graph.nodes(data=1):
                 node_type = node[1]['FID']
387              if not node_type == '':
                     nodelist.append(node[0])
389                  node_color.append('r' if node_type == 'sink' else 'b')

391          color = {edge: 'b' for edge in self.graph.edges()}
             # returns an array of pairs, the elements of seq two by two
393          def pairwise(seq):
                 return [seq[i:i+2] for i in range(len(seq)-2)]
395          # colors the edges
             for u, v in pairwise(path):
397              if (u, v) in color:
                     color[(u, v)] = 'r'
399              if (v, u) in color:
                     color[(v, u)] = 'r'
401
             # makes an array of the dictionary, the order is importatant!
403          array = []
```

```python
            for edge in self.graph.edges():
                array += color[edge]
            nx.draw_networkx(self.graph, pos=self.coord2D(),
                                nodelist=nodelist, with_labels=0,
                                node_color=node_color, edge_color=array)


    def shortest_path(self, node1, node2):
        """
        Calculates the shortest path on self.graph.
        Path is a sequence of traversed nodes
        """
        try:
            path = nx.shortest_path(self.graph, source=node1, target=node2,
                                weight="weight")
        except:
            pass
        return path


    def path_lenght(self, path):
        """
        given a path on the graph returns the lenght of the path in the
        unit the coordinats are expressed
        """
        if path is None:
            return float("inf")
        lenght = 0.0
        # given the path (list of node) returns the edges contained
        path_edges = [path[i:i+2] for i in range(len(path)-2)]
        # itereate to edges and calculate the weight
        for u, v in path_edges:
            lenght += self.distance(u, v)
        return lenght


    def TSP(self, cities):
        '''
        declaring the adjacency matrix
        T = numpy.empty(shape=(len(cities),len(cities)))
        for u, i in cities:
            for v, j in itertools(cities):
                uv_path = shortest_path(u, v)
                T[i][j] = path_lenght(shortest_path)

            start = timeit.default_timer() #start timer
            paths = []
            for combo in itertools.permutations(range(1,len(T[0]))):
                lenght = 0
                prev = 0
                path = []
                path += [0]
                for elem in combo:
                    lenght += T[prev][elem]
                    prev = elem
                    path += [elem]
                lenght += T[combo[len(combo)-1]][0]
                path += [0]
```

```
459            paths.append((path, lenght))
          stop = timeit.default_timer() # stop timer
461          time = stop - start
          return paths, time
463      '''

465    def is_sourcesink(self, node):
          '''given a node as in the networkx.Graph.nodes(data=1)
467      returns 1 if the node is a sink or a source, 0 elsewhere'''
          if not node[1]['FID'] == '':
469          return 1
          return 0

471
      def compute_source_matrix(self):
473      for node in self.graph.nodes(data=1):
              if self.is_sourcesink(node):
475              self.sinksource_graph.add_node(node[0], node[1])

477      for n1 in self.sinksource_graph.nodes():
              for n2 in self.sinksource_graph.nodes():
479              if n1 is not n2:
                  path = self.shortest_path(n1, n2)
481              if path is not None:
                      self.sinksource_graph.add_edge(n1, n2,
483                                          {'dist': self.path_lenght(path),
                                            'path': path})

485
      def design_minimal_aqueduct(self, G):
487      minimal = nx.minimum_spanning_tree(G, weight='dist')
          return minimal

489
      def display_recouvring_graph(self, G):
491      path = []
          for edge in G.edges(data=True):
493          path += edge[2]['path']
          self.display_path(path)

495
      def complete_graph(self, G):
497      for n1 in G.nodes():
              for n2 in G.nodes():
499              if n1 != n2:
                  # attributes = {'path': [n1, n2], 'dist': self.distance(n1,
    n2)}
501              G.add_edge(n1, n2)
                  G.edges[n1, n2]['dist'] = self.distance(n1,n2)
503              G.edges[n1, n2]['path'] = [n1, n2]

505    def mesh_graph(self, G, weight):
          """complexity (len(G.nodes))^3"""
507      distances = nx.get_edge_attributes(G, weight)
          # condition to create the gabriel relative neighbour graph
509      def neighbors(p, q):
              for r in G.nodes:
511              if r != q and r != p:
                  def dist(n1, n2):
```

```
513                            if (n1, n2) in distances:
                                   return distances[(n1,n2)]
515                            else:
                                   return distances[(n2,n1)]
517                      if max(dist(p,r), dist(q,r)) < dist(p,q):
                             return False
519              return True

521          # connect graph
             gabriel_graph = nx.Graph()
523          for n1 in G.nodes():
                 for n2 in G.nodes():
525                  if n1 != n2:
                         if neighbors(n1, n2):
527                          gabriel_graph.add_edge(n1, n2)
             return gabriel_graph
529
        def graphToEdgeMatrix(self, G):
531          node_dict = {node: index for index, node in enumerate(G)}

533          # Initialize Edge Matrix
             edgeMat = [[0 for x in range(len(G))] for y in range(len(G))]
535
             # For loop to set 0 or 1 ( diagonal elements are set to 1)
537          for i, node in enumerate(G):
                 tempNeighList = G.neighbors(node)
539              for neighbor in tempNeighList:
                     edgeMat[i][node_dict[neighbor]] = 1
541              edgeMat[i][i] = 1

543          return edgeMat

545      def clusters(self, G):
             '''
547          Finds the clusters
             '''
549          # imports from a machine learning package skit-learn
             from sklearn.cluster import MeanShift, estimate_bandwidth
551
             # creates a array with the 2D coordinats for each node
553          X = [[node[0], node[1]] for node in G.nodes()]
             # extimates the dimensions of single clusters
555          bandwidth = estimate_bandwidth(X, quantile=0.1,
                                            random_state=0, n_jobs=1)
557          # find clustes
             ms = MeanShift(bandwidth=bandwidth)
559          ms.fit(X)

561          # labels is an array indicating, for each node, the cluster number
             labels = {node: ms.labels_[i] for i, node in enumerate(G.nodes())}
563
             # --- ADDUCTION ---
565          '''
             # add cluster centers to the graph
567          for node in cluster_centers:
```

```
              attribute = {'label': 'water tower', 'pos': node}
569           G.add_node(node, attribute)
              attribute = {'type' : 'sink'}
571           self.sinksource_graph.add_node(node, attribute)
          '''
573       adduction = nx.Graph()
          cluster_centers = [(node[0], node[1]) for node in ms.cluster_centers_]
575       for node in cluster_centers:
              adduction.add_node(node)
577       self.complete_graph(adduction)
          adduction = self.mesh_graph(adduction, weight='dist')
579
          print(len(adduction.edges()))
581
          nx.draw_networkx(adduction)
583       # coord = {elem[0]: [elem[0][0], elem[0][1]] for elem in adduction.
      nodes(data=True)}
          # nx.draw_networkx(adduction, pos=coord, label=False)
585       self.write2shp(adduction, "adduction_network")
          self.acqueduct.add_edges_from(adduction.edges())
587
          # ——— DISTRIBUTION ———
589       # add label info to the graph
          nx.set_node_attributes(G, labels, 'label')
591       # initialize distribution graphs
          distribution = [nx.Graph() for cluster in cluster_centers]
593       for node in labels:
              cluster = labels[node]
595           distribution[cluster].add_node(node)
          '''
597       # connect each node with his the cluster center
          node_list = []
599       for index, node in enumerate(G):
              node_list.append(node)
601           labels = nx.get_node_attributes(G, 'label')
              label = labels[node]
603           if label is not 'water tower':
                  G.add_edge(node, cluster_centers[label])
605       '''
          for dist_graph in distribution:
607           self.complete_graph(dist_graph)
              dist_graph = nx.minimum_spanning_tree(dist_graph, weight='dist')
609           self.acqueduct.add_edges_from(dist_graph.edges())

611   def route_vesuvio(self, n1, n2):
          try:
613           import shapefile
          except ImportError:
615           raise ImportError("read_shp requires pyshp")
          # route
617       path = self.shortest_path(n1, n2)

619       # turn path into acqueduct graph
          datas = [data['pos'] for _, data in self.graph.nodes(data=True)]
621       path_coord = [tuple(datas[node]) for node in path]
```

```
              path_edges = [path_coord[i:i + 2] for i in range(len(path_coord) - 2)]
623           self.acqueduct.add_edges_from(path_edges)

625           # write shp
              def write2shape():
627               w = shapefile.Writer(shapeType=3)
                  w.field("name", "C")
629               line = path_edges
                  w.line(parts=line)
631               w.record('path')
                  w.save('path')

633
    def render_vtk(file_name):

635
        import vtk

637
        # Read the source file.
639     reader = vtk.vtkUnstructuredGridReader()
        reader.SetFileName(file_name)
641     reader.Update()  # Needed because of GetScalarRange
        output = reader.GetOutput()
643     scalar_range = output.GetScalarRange()

645     # Create the mapper that corresponds the objects of the vtk.vtk file
        # into graphics elements
647     mapper = vtk.vtkDataSetMapper()
        mapper.SetInputData(output)
649     mapper.SetScalarRange(scalar_range)

651     # Create the Actor
        actor = vtk.vtkActor()
653     actor.SetMapper(mapper)

655     # Create the Renderer
        renderer = vtk.vtkRenderer()
657     renderer.AddActor(actor)
        renderer.SetBackground(1, 1, 1)  # Set background to white

659
        # Create the RendererWindow
661     renderer_window = vtk.vtkRenderWindow()
        renderer_window.AddRenderer(renderer)

663
        # Create the RendererWindowInteractor and display the vtk_file
665     interactor = vtk.vtkRenderWindowInteractor()
        interactor.SetRenderWindow(renderer_window)
667     interactor.Initialize()
        interactor.Start()

669
    def tsp_example():
671     return 0


673 def clustering_example():
        return 0

675
    def template_clustering(path_sample, eps, minpts, amount_clusters=None,
```

```
         visualize=True , ccore=False ) :
677      sample = read_sample ( path_sample ) ;

679      optics_instance = optics (sample , eps , minpts , amount_clusters , ccore ) ;
         ( ticks , _ ) = timedcall ( optics_instance . process ) ;
681
         print ( "Sample: " , path_sample , "\t\tExecution time : " , ticks , "\n" ) ;
683
         if ( visualize is True ) :
685          clusters = optics_instance . get_clusters ( ) ;
             noise = optics_instance . get_noise ( ) ;
687
             visualizer = cluster_visualizer ( ) ;
689          visualizer . append_clusters ( clusters , sample ) ;
             visualizer . append_cluster ( noise , sample , marker = 'x' ) ;
691          visualizer . show ( ) ;

693          ordering = optics_instance . get_ordering ( ) ;
             analyser = ordering_analyser ( ordering ) ;
695
             ordering_visualizer . show_ordering_diagram ( analyser , amount_clusters ) ;
697
     def vesuvio_example ( ) :
699      router = Router ( topo_file=" vtk / Vesuvio " )
         router . route_vesuvio ( 32729 , 31991 )
701      # write to vtk
         router . write2vtk ( router . acqueduct )
703      # render_vtk ( " vtk / Vesuvio " )

705  def paesi_example ( ) :
         router = Router ( building_file=" geographycal_data / paesi_elev / paesi_elev " )
707      router . clusters ( router . graph )
         router . write2shp ( router . acqueduct , " acqueduct1 " )
709
     def cluster_simple_example ( ) :
711      import random ;

713      from pyclustering . cluster import cluster_visualizer ;
         from pyclustering . cluster . optics import optics , ordering_analyser ,
         ordering_visualizer ;
715
         from pyclustering . utils import read_sample , timedcall ;
717
         from pyclustering . samples . definitions import SIMPLE_SAMPLES, FCPS_SAMPLES;
719
         template_clustering ( SIMPLE_SAMPLES . SAMPLE_SIMPLE1 , 0 . 5 , 3 ) ;
721
     paesi_example ( )
723  # National__Hydrography__Dataset_NHD_Points_Medium_Resolution /
         National__Hydrography__Dataset_NHD_Points_Medium_Resolution
     # National__Hydrography__Dataset_NHD_Lines_Medium_Resolution /
         National__Hydrography__Dataset_NHD_Lines_Medium_Resolution
725  # Railroads / Railroads
     # Routesnaples / routesnaples
727  # " shapefiles / Domain " , " shapefiles / pointspoly "
```

```
   # "shapeline/shapeline", "shapeline/points"
729 # nx.draw_networkx(router.sinksource_graph, pos=router.coord2D(), with_labels
       =0)
   # router.design_minimal_aqueduct()
731 # router.display_path(path)
   # nx.draw_networkx_nodes(router.graph, pos=router.coord2D())
733 # router.display_mesh()
   # print nx.clustering(router.graph)
735 # print nx.floyd_warshall_numpy(router.graph, weight='dist')
   # runfile('/Users/Conrad/Documents/EC/Course deuxi mme ann e/Project Inno/
       Projet_P5C006/router.py', wdir='/Users/Conrad/Documents/EC/Course
       deuxi mme ann e/Project Inno/Projet_P5C006')
```

Appendix D

Il manuale utente

Manuale utente per l'utilizzo del sistema

D.1 Instructions for installation

The following instructions allow to install the environment to run our applications. First open
in terminal the folder in which you wish to create the dedicated environment: `cd /path/to/folder/folder_`
Then create a virtual environment using venv, python –version ¿= 3 is required python -m
venv my-virtualenv
Activate the environment source my-virtualenv/bin/activate
Finally install all the requested packages pip install -r requirements.txt

Appendix E

Use case scenario

In this section will be presented a couple of use case scenarios, from the easier to the more complex ones.

E.1   Simple 3D routing

In this example a simple 3D routing is computed and rendered. Calling the command `vesuvio_example()` in the python shell will execute the following instructions

```
router = Router(topo_file="vtk/Vesuvio")
    router.route_vesuvio(32729, 31991)
    router.write2vtk(router.acqueduct)
render_vtk("vtk/Vesuvio")
```

The topography of the Vesuvio and surrounding areas is loaded from a vtk file into a networkx graph data structure of the router python class. The shortest path is computed using the Dijstrak algorithm between two points. This path is then exported as a vtk file. Finally the two vtk, the topography and the path are rendered using the vtk library. The result is shown in figure

E.2 Clustering

E.3 Travel Salesman Problem

E.4 Minimum Spanning Tree

E.5 Automatic design

Executing the command `paesi_example()` will reproduce the case studied in **??**. The following instructions are executed. A shape file representing the buildings position

are loaded. Then clusters and paths are computed as above. The result is exported as shape file, named "aqueduct".

Appendix F

Datasheet

Eventuali Datasheet di riferimento.