ction1.2
description of the approach5

CENTRALESUPELEC
2nd year of cursus centralien
Department of Mechanics

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. 2 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. 3 is a detailed description of the approach. The different techniques studied throughout the project are presented and discussed. 4 explains our solution which is based on clustering and MST algorithms. 5 gives an insight into the more technical aspects of the project. Details on the software used can be found in this section. Finally, 6 presents the results obtained with the technique described in 4.

Chapter 2

State of the art

Nella seconda sezione si riporta lo stato dell'arte del settore, un inquadramento dell'area di ricerca orientato a portare il lettore all'interno della problematica affrontata. Bisogna dimostrare di conoscere le cose fatte fino ad ora in questo campo e il perché si sia reso necessario lo svolgimento di questo lavoro. Questa sezione deve essere grondante di citazioni bibliografiche [**?**].

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 n̈eed 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 ̈smarẗtechniques 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 5.1 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 5.2

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 5.3.

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 5.3. 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:

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

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:

$$y_k\,(t) = sgn(s_k(t-1))$$

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

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

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:

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

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 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.

```
# -*- 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
    #
```

```
    ---------------------------------------------------------------------------------------------

    def __init__(self, topo_file=None, building_file=None):

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

    #
    ---------------------------------------------------------------------------------------------
    # --- CLASS ATTRIBUTES
    #
    ---------------------------------------------------------------------------------------------

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

    def read_shp_bilding(self, file_name):

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

```
                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]
```

```
                    self.graph.add_node(shape.points[0], attributes)

            if shape.shapeType == 3:    # polylines
                attributes1 = dict(zip(fields, shapeRecs.record))
                attributes2 = dict(zip(fields, shapeRecs.record))
                for i in range(len(shape.points) - 1):
                    '''
                    attributes1["pos"] = shape.points[i]
                    n1 = self.add_node_unique(shape.points[i], attributes1)
                    attributes2["pos"] = shape.points[i + 1]
                    n2 = self.add_node_unique(shape.points[i + 1], attributes2)
                    attribute = {'dist': self.distance(n1, n2)}
                    print '{0}: {1}, {2}'.format(i, n1, n2)
                    self.graph.add_edge(n1, n2, attribute) '''
                    attributes1["pos"] = shape.points[i]
                    n1 = shape.points[i]
                    self.graph.add_node(n1, attributes1)
                    attributes2["pos"] = shape.points[i + 1]
                    n2 = shape.points[i + 1]
                    self.graph.add_node(n2, attributes2)
                    attribute = {'dist': self.distance(n1, n2)}
                    self.graph.add_edge(n1, n2, attribute)

            if shape.shapeType == 5:    # polygraph
                # chuncker: see commpresed row storage
                def chuncker(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)]

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

                for cell in chuncker(shape.points, shape.parts):
                    for n1, n2 in pairwise(cell):
                        attributes1 = dict(zip(fields, shapeRecs.record))
                        attributes1["pos"] = n1
                        attributes2 = dict(zip(fields, shapeRecs.record))
                        attributes2["pos"] = n2
                        # add nodes of the shape to the graph
                        n1 = self.add_node_unique(n1, attributes1)
                        n2 = self.add_node_unique(n2, attributes2)
                        attribute = {'dist': self.distance(n1, n2)}
                        # add edge
                        self.graph.add_edge(n1, n2, attribute)

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

```
                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])
```

```
        vtk = pyvtk.VtkData(pyvtk.UnstructuredGrid(points, line=line))
        vtk.tofile('example1', 'ascii')

    def add_node_unique(self, new_node, new_attributes):
        """
        grants that the node added is unique with respect to the pos
        attribute equality relationship
        """
        for node in self.graph.nodes(True):
            if node[1]["pos"] == new_attributes["pos"]:
                return node[0]
        self.graph.add_node(new_node, new_attributes)
        return new_node

    def read_vtk(self, file_name):
        import numpy as np
        try:
            from mesh import Mesh
        except ImportError:
            raise ImportError("read_vtk requires pymesh")

        # initialize the vtk reader
        reader = Mesh()

        # read the vtk
        reader.ReadFromFileVtk(file_name)

        # add nodes to the graph
        for index, node in enumerate(reader.node_coord):
            self.graph.add_node(index, pos=reader.node_coord[index])

        '''
        chuncker
        Principe bas  sur le stockage CSR ou CRS (Compressed Row Storage)
        dont voici une illustration :
        Soient six n uds num rotes de 0   5 et quatre  el  ements  form
    es  par
        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.
        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
        pour indiquer la position, dans ce tableau, ou‘ commence chacune de
        ces listes (table 2).
        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
        tableau elem2node. La derni‘ere valeur dans p elem2node correspond au
        nombre de cellules (la taille) du tableau elem2node.

        elem2node
        0 | 2 | 3 | 1 | 2 | 4 | 0 | 1 | 3 | 4 | 1 | 5
        1   2   3   4   5   6   7   8   9   10  11  12
        ^       ^         ^                   ^       ^
```

```
        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
```

```
    def distance3D(self, nodei, nodej):
        xi = self.graph.nodes(data=True)[nodei][1]['pos'][0]
        yi = self.graph.nodes(data=True)[nodei][1]['pos'][1]
        zi = self.graph.nodes(data=True)[nodei][1]['pos'][2]
        xj = self.graph.nodes(data=True)[nodej][1]['pos'][0]
        yj = self.graph.nodes(data=True)[nodej][1]['pos'][1]
        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))

    # returns 2D coordinates of the nodes of self.graph
    def coord2D(self, G):
        coord2D = {}
        for key, value in nx.get_node_attributes(G,
                                                  'pos').iteritems():
            coord2D[key] = [value[0], value[1]]
        return coord2D

    # display the mesh using networkx function
    def display_mesh(self):
        nodelist = []
        node_color = []
        for node in self.graph.nodes(data=1):
            node_type = node[1]['FID']
            if not node_type == '':
                nodelist.append(node[0])
                node_color.append('r' if node_type == 'sink' else 'b')
        try:
            nx.draw_networkx(self.graph, pos=self.coord2D(), nodelist=nodelist,
                             with_labels=0, node_color=node_color)
        except:
            pass

    # display the mesh with a path marked on it
    def display_path(self, path):
        nodelist = []
        node_color = []
        for node in self.graph.nodes(data=1):
            node_type = node[1]['FID']
            if not node_type == '':
                nodelist.append(node[0])
                node_color.append('r' if node_type == 'sink' else 'b')

        color = {edge: 'b' for edge in self.graph.edges()}
        # returns an array of pairs, the elements of seq two by two
        def pairwise(seq):
            return [seq[i:i+2] for i in range(len(seq)-2)]
        # colors the edges
        for u, v in pairwise(path):
            if (u, v) in color:
                color[(u, v)] = 'r'
            if (v, u) in color:
                color[(v, u)] = 'r'

        # makes an array of the dictionary, the order is importatant!
        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]
```

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

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

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

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

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

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

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

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

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

        # connect graph
        gabriel_graph = nx.Graph()
        for n1 in G.nodes():
            for n2 in G.nodes():
                if n1 != n2:
                    if neighbors(n1, n2):
                        gabriel_graph.add_edge(n1, n2)
        return gabriel_graph

    def graphToEdgeMatrix(self, G):
        node_dict = {node: index for index, node in enumerate(G)}

        # Initialize Edge Matrix
        edgeMat = [[0 for x in range(len(G))] for y in range(len(G))]

        # For loop to set 0 or 1 ( diagonal elements are set to 1)
        for i, node in enumerate(G):
            tempNeighList = G.neighbors(node)
            for neighbor in tempNeighList:
                edgeMat[i][node_dict[neighbor]] = 1
            edgeMat[i][i] = 1

        return edgeMat

    def clusters(self, G):
        '''
        Finds the clusters
        '''
        # imports from a machine learning package skit-learn
        from sklearn.cluster import MeanShift, estimate_bandwidth

        # creates a array with the 2D coordinats for each node
        X = [[node[0], node[1]] for node in G.nodes()]
        # extimates the dimensions of single clusters
        bandwidth = estimate_bandwidth(X, quantile=0.1,
                                       random_state=0, n_jobs=1)
        # find clustes
        ms = MeanShift(bandwidth=bandwidth)
        ms.fit(X)

        # labels is an array indicating, for each node, the cluster number
        labels = {node: ms.labels_[i] for i, node in enumerate(G.nodes())}

        # --- ADDUCTION ---
        '''
        # add cluster centers to the graph
        for node in cluster_centers:
```

```
            attribute = {'label': 'water tower', 'pos': node}
            G.add_node(node, attribute)
            attribute = {'type' : 'sink'}
            self.sinksource_graph.add_node(node, attribute)
        '''
        adduction = nx.Graph()
        cluster_centers = [(node[0], node[1]) for node in ms.cluster_centers_]
        for node in cluster_centers:
            adduction.add_node(node)
        self.complete_graph(adduction)
        adduction = self.mesh_graph(adduction, weight='dist')

        print(len(adduction.edges()))

        nx.draw_networkx(adduction)
        # coord = {elem[0]: [elem[0][0], elem[0][1]] for elem in adduction.nodes(data=True)}
        # nx.draw_networkx(adduction, pos=coord, label=False)
        self.write2shp(adduction, "adduction_network")
        self.acqueduct.add_edges_from(adduction.edges())

        # --- DISTRIBUTION ---
        # add label info to the graph
        nx.set_node_attributes(G, labels, 'label')
        # initialize distribution graphs
        distribution = [nx.Graph() for cluster in cluster_centers]
        for node in labels:
            cluster = labels[node]
            distribution[cluster].add_node(node)
        '''
        # connect each node with his the cluster center
        node_list = []
        for index, node in enumerate(G):
            node_list.append(node)
            labels = nx.get_node_attributes(G, 'label')
            label = labels[node]
            if label is not 'water tower':
                G.add_edge(node, cluster_centers[label])
        '''
        for dist_graph in distribution:
            self.complete_graph(dist_graph)
            dist_graph = nx.minimum_spanning_tree(dist_graph, weight='dist')
            self.acqueduct.add_edges_from(dist_graph.edges())

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

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

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

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

def render_vtk(file_name):

    import vtk

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

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

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

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

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

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

def tsp_example():
    return 0

def clustering_example():
    return 0

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

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

    optics_instance = optics(sample, eps, minpts, amount_clusters, ccore);
    (ticks, _) = timedcall(optics_instance.process);

    print("Sample: ", path_sample, "\t\tExecution time: ", ticks, "\n");

    if (visualize is True):
        clusters = optics_instance.get_clusters();
        noise = optics_instance.get_noise();

        visualizer = cluster_visualizer();
        visualizer.append_clusters(clusters, sample);
        visualizer.append_cluster(noise, sample, marker = 'x');
        visualizer.show();

        ordering = optics_instance.get_ordering();
        analyser = ordering_analyser(ordering);

        ordering_visualizer.show_ordering_diagram(analyser, amount_clusters);

def vesuvio_example():
    router = Router(topo_file="vtk/Vesuvio")
    router.route_vesuvio(32729, 31991)
    # write to vtk
    router.write2vtk(router.acqueduct)
    # render_vtk("vtk/Vesuvio")

def paesi_example():
    router = Router(building_file="geographycal_data/paesi_elev/paesi_elev")
    router.clusters(router.graph)
    router.write2shp(router.acqueduct, "acqueduct1")

def cluster_simple_example():
    import random;

    from pyclustering.cluster import cluster_visualizer;
    from pyclustering.cluster.optics import optics, ordering_analyser,
    ordering_visualizer;

    from pyclustering.utils import read_sample, timedcall;

    from pyclustering.samples.definitions import SIMPLE_SAMPLES, FCPS_SAMPLES;

    template_clustering(SIMPLE_SAMPLES.SAMPLE_SIMPLE1, 0.5, 3);

paesi_example()
# 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
# Railroads/Railroads
# Routesnaples/routesnaples
# "shapefiles/Domain", "shapefiles/pointspoly"
```

```
# "shapeline/shapeline", "shapeline/points"
# nx.draw_networkx(router.sinksource_graph, pos=router.coord2D(), with_labels=0)
# router.design_minimal_aqueduct()
# router.display_path(path)
# nx.draw_networkx_nodes(router.graph, pos=router.coord2D())
# router.display_mesh()
# print nx.clustering(router.graph)
# 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$_{n}ame$
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 4.4. 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.