

CENTRALESUPELEC
2nd year of cursus centralien
Department of Mechanics



CentraleSupélec file=LogoCS.png,width=3.5cm

Computer aided design of a water supply network

AI & R Lab
Laboratorio del dioparco facile del Poliminchia

Relatore: Filippo Gatti
Correlatore: Andrea Barbarulo

Auth
calvet-litzell F
Cognetti Giova
Marzorati Edoa
Vaccarino Marc

Academic year 2017-2018

In remembrance of Jean-Bapti

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

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 given an area. In order to do so the team examined several techniques that could be suited to solve the problem. Among them were Root Architecture and Neural Networks. Eventually, clustering techniques and a Minimum Spanning Tree (MST) algorithm were chosen for 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 Motivation and Objectives

1.3 Brief description of the approach

La seconda parte deve essere una esplosione della prima e deve quindi mostrare in maniera più esplicita l'area dove si svolge il lavoro, le bibliografiche più importanti su cui si fonda il lavoro in maniera sintetica (una pagina) evidenziando i lavori in letteratura che presentano attinenza con il lavoro affrontato in modo da mostrare da dove e perché è sorta la tematica di studio. Poi si mostrano esplicitamente le realizzazioni, le direzioni future di ricerca, quali sono i problemi aperti e quali quelli affrontati, si ripete lo scopo della tesi. Questa parte deve essere piena (ma non grondante) come la sezione due) di citazioni bibliografiche e deve essere lunga circa tre facciate.

1.4 Structure of the report

La terza parte contiene la descrizione della struttura della tesi ed è organizzata nel modo seguente. “La tesi è strutturata nel modo seguente.

Nella sezione due si mostra ...

Nella sez. tre si illustra ...

Nella sez. quattro si descrive ...

Nelle conclusioni si riassumono gli scopi, le valutazioni di questi e le prospettive future ...

Nell'appendice A si riporta ... (Dopo ogni sezione o appendice ci vuole un punto).”

I titoli delle sezioni da 2 a M-1 sono indicativi, ma bisogna cercare di mantenerne un significato equipollente nel caso si vogliano cambiare. Queste sezioni possono contenere eventuali sottosezioni.

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 finora 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

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 covered the potential advantage of using root system architecture to deliver 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 produce 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 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 in the root model.

plex 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 system architecture comes from the necessity in agriculture of increasing productivity and minimizing water and nutrient losses. A good understanding of these 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 the 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 models 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. This phenomenon has been fostered by the need for predictive technologies at different scales. Dupuy suggests a new theoretical framework which takes into account individual root development parameters. He introduces "equations in discretized domains that describe growth 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 from 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 effect of 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

to soil properties. This characteristic allows the plant to forage with 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 the 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. At a high rate it tends to thicken a tube and at a low rate it leads to its decay. As shown by Prof. Arson's paper "Bio-inspired fluid extraction model for resource-rich 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 Machine Learning, more specifically, Artificial Neural Networks.

3.2 Artificial neural networks

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

For our project we chose to follow this direction, combine sub-optimal algorithms to develop a possibly innovative solution. Lead by example

AI is applied to many complex routing problems: one example is very large scale integration (VLSI). The process of designing integrated circuits is difficult 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 performance.

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 can 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.

Artificial Neural Networks 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 it 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 lookups. 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 might work; instead, a node will send the message to a gateway in the local area network, which then decides how to route the package of data to the correct destination. Each gateway will need to keep track of which

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.

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 d_{XY} 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 salesman problem. The algorithm was successfully tested on many problems with up to 30 cities and five salesmen while a non-optimized brute-force approach would take billions of billions of years to return. In all test cases, the algorithm always converged to optimal solutions.

Mustafa K. Mehmet Ali and Faouzi Kamoun (1993) considered modeling the shortest path problem with Hopfield Neural Network for the first time. These 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.

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 of n^2 nodes where $X \in [1, n]$ and $j \in [1, n]$ and the state is characterized by the binary activation values $y = (y_1, \dots, y_n)$ of the nodes. A TSP problem with n cities can be modeled as an Hopfield net of dimension n^2 , where y_{Xj} is 1 if the city X is in the k -position of the tour.

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

where w_{jk} is the weight of the connection between j and k and θ_k is the bias. The sigmoid forward function is applied to the node input to obtain the new activation value at time $t+1$:

$$y_k(t) = \text{sgn}(s_k(t-1))$$

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

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

The first two terms are null if and only if 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 into 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 $\delta_{jk} = 1$ if $j = k$ and zero otherwise. As in the energy function, the first term inhibits connection within each row, the second within column, 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 converge to 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 a gradient descent, after a certain number of repetition the state converges to a stable point that is a minima of the energy function.

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.

Chapter 4

Progetto logico della soluzione del problema

4.1 Problem statement and modelisation

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

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

Read
data

Graph
initialis-
taion

Topog

Routing

Wate

4.2 Data reading

In ?? is shown to the flow chart of 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 put 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 sources and sinks as nodes and no edges. An insight of the data structure we use 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 has 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 see in the next paragraph. Efficient implementations of clustering algorithms are provided in scikit-learn. Scikit-learn is a well-known machine learning library for Python and it features various

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 sinks of a cluster. This operation is broken down in two tasks. First, find all the paths connecting sinks, then choose the smallest recovering graph. To find 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 Dijkstra algorithm. The length of those paths 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 graphs is a partition of the aqueduct graph. The second step 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 initialize the adduction graph with all the cluster's root nodes and the water source. 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 them. 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 geometries, for example, could represent water wells, roads or buildings. As those primary 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 processing 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 surface 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 through 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 structures. The vtk file boils down to those two elements: points and cells. Points have 3D coordinates while cells are surfaces, expressed by the point indices delimiting them. Point and cell data (scalar or vector) can also be assigned to them.

to use, widespread programming language, good for rapid prototyping, 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 describes 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. 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 this data, a first graph is obtained. The graph has as nodes the points described in topology file plus the buildings. The coordinates of building-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 edges connecting the building to the nearest node of the network in order to obtain a connected graph.

5.3 Clustering

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

TSP

Chapter 6

Presentation and validation of experimental results

Si mostra il progetto dal punto di vista sperimentale, le cose materialmente realizzate. In questa sezione si mostrano le attività sperimentali svolte, si illustra il funzionamento del sistema (a grandi linee) e si spiegano i risultati ottenuti con la loro valutazione critica. Bisogna introdurre dati sulla complessità degli algoritmi e valutare l'efficienza del sistema.

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 work

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'hardware. 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. listings

Appendix C

Listato

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

```
1  # -*- coding: utf-8 -*-
3  import networkx as nx
   import math
5  import sys
7  # sys.setdefaultencoding('utf8 ')
9  """
   Created on Mon Dec  4 22:57:30 2017
11  @author: Conrad
13  """
15
16  class Router(object):
17      #
18
19      # -- CLASS ATTRIBUTES
20
21      # Class description
22      CLASS_NAME = "Router"
23      CLASS_AUTHOR = "Marcello Vaccarino"
```

```

29     acqueduct = nx.Graph()
    #


---


31     # -- INITIALIZATION
    #


---


33     def __init__(self, topo_file=None, building_file=None):
34
35         if topo_file != None and building_file != None:
36             try:
37                 # [TODO] this function does not read building
but single points
38                 self.read_shp(topo_file, building_file)
39             except Exception as e:
40                 raise e
41         elif building_file != None:
42             try:
43                 self.read_shp_building(building_file)
44             except Exception as e:
45                 raise e
46         elif topo_file != None:
47             try:
48                 self.read_vtk(topo_file)
49             except Exception as e:
50                 raise e
51
52     #


---


53     # -- CLASS ATTRIBUTES
    #


---


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

```

```

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[1:]))
        attributes['pos'] = center_coord
        self.graph.add_node(center_coord)

# chuncker: see compressed 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
    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

```

```

    repeated for each
        edge comprising that feature.

109
    Returns
    -----
111
    G : NetworkX graph
113
    Examples
    -----
115
>>> G=nx.read_shp('test.shp') # doctest: +SKIP
117
    References
    -----
119
    .. [1] http://en.wikipedia.org/wiki/Shapefile
121
    """
    try:
123         import shapefile
    except ImportError:
125         raise ImportError("read_shp requires pyshp")

127 sf = shapefile.Reader(file_name)
    fields = [x[0] for x in sf.fields]
129
    for shapeRecs in sf.iterShapeRecords():
131
        shape = shapeRecs.shape
133         if shape.shapeType == 1: # point
            attributes = dict(zip(fields, shapeRecs.record.attributes))
135             attributes["pos"] = shape.points[0]
            self.graph.add_node(shape.points[0], attributes)
137
        if shape.shapeType == 3: # polylines
139             attributes1 = dict(zip(fields, shapeRecs.record.attributes))
            attributes2 = dict(zip(fields, shapeRecs.record.attributes))
            for i in range(len(shape.points) - 1):
141                 attributes1["pos"] = shape.points[i]
143                 n1 = self.add_node_unique(shape.points[i], attributes1)
145                 attributes2["pos"] = shape.points[i + 1]
147                 n2 = self.add_node_unique(shape.points[i + 1], attributes2)
                attribute = {'dist': self.distance(n1, n2)}
                print '{0}\t{1}\t{2}'.format(i, n1, n2)

```

```

153         self.graph.add_node(n1, attributes1)
154         attributes2["pos"] = shape.points[i + 1]
155         n2 = shape.points[i + 1]
156         self.graph.add_node(n2, attributes2)
157         attribute = {'dist': self.distance(n1, n2)}
158         self.graph.add_edge(n1, n2, attribute)
159
160     if shape.shapeType == 5:    # polygraph
161         # chunker: see compressed row storage
162         def chunker(array, p_array):
163             p_array.append(len(array))
164             return [array[p_array[i]:p_array[i+1]]
165                     for i in range(len(p_array)-1)]
166
167         # given a cell returns the edges (node tuple
168         # implicitly defined in it
169         def pairwise(seq):
170             return [seq[i:i+2] for i in range(len(seq)
171 -1)]
172
173         for cell in chunker(shape.points, shape.partition
174         for n1, n2 in pairwise(cell):
175             attributes1 = dict(zip(fields, shape.partition
176             .record))
177             attributes1["pos"] = n1
178             attributes2 = dict(zip(fields, shape.partition
179             .record))
180             attributes2["pos"] = n2
181             # add nodes of the shape to the graph
182             n1 = self.add_node_unique(n1,
183             attributes1)
184             n2 = self.add_node_unique(n2,
185             attributes2)
186             attribute = {'dist': self.distance(n1,
187             n2)}
188             # add edge
189             self.graph.add_edge(n1, n2, attribute)
190
191     if point_file != None:
192         sf = shapefile.Reader(point_file)
193         new_fields = [x[0] for x in sf.fields]
194         nodes2attributes = {node: data \
195                             for node, data in self.graph.
196         nodes(data=1)}
197         for shapeRecs in sf.iterShapeRecords():
198             shape = shapeRecs.shape

```

```

new_attributes = dict(zip(new_fields, shapeRe
record))
193     nodes = [tuple(point) for point in shape.point
195             for node in nodes:
196                 # print node in nodes2attributes,
197                 new_attributes
198                 nodes2attributes[node].update(new_attribu
199             )
200             for node, data in self.graph.nodes(data=1):
201                 data.update(nodes2attributes[node])
202                 # nx.set_node_attributes(self.graph,
203                 nodes2attributes)
204
205     def write2shp(self, G, filename):
206         try:
207             import shapefile
208             except ImportError:
209                 raise ImportError("read_shp requires pyshp")
210
211         w = shapefile.Writer(shapeType=3)
212         #w.field("DC_ID", "LENGHT", "NODE1", "NODE2", "DIAMETRE",
213         "ROUGHNESS", "MINORLOSS", "STATUS", "C")
214
215         w.fields = [("DeletionFlag", "C", 1, 0), ["DC_ID", "N", 9, 0],
216         ["LENGHT", "N", 18, 5], ["NODE1", "N", 9, 1], ["
217         NODE2", "N", 9, 0],
218         ["DIAMETRE", "N", 18, 5], ["ROUGHNESS", "N", 18,
219         ["MINORLOSS", "N", 18, 5],
220         ["STATUS", "C", 1, 0]]
221
222         i = 0
223         lenghts = nx.get_edge_attributes(G, 'dist')
224         print(lenghts)
225         for edge in lenghts:
226             line = [edge[0], edge[1]]
227             w.line(parts=[line])
228             print (edge[0], edge[1])
229             w.record(i, lenghts[(edge[0], edge[1])],
230                     1, 2, 100, 0.1, 0, "1")
231             i+=1
232         w.save(filename)
233
234     def write2vtk(self, G):
235         # import sys

```

```

233         points = [list(node) for node, data in G.nodes(data=True)]
234     ]
235     line = []
236     for edge in G.edges():
237         for i, node in enumerate(G.nodes()):
238             if node == edge[0]:
239                 n1 = i
240             for i, node in enumerate(G.nodes()):
241                 if node == edge[1]:
242                     n2 = i
243                 line.append([n1, n2])
244
245     vtk = pyvtk.VtkData(pyvtk.UnstructuredGrid(points, line))
246     vtk.tofile('example1', 'ascii')
247
248     def add_node_unique(self, new_node, new_attributes):
249         """
250         grants that the node added is unique with respect to
251         pos
252         attribute equality relationship
253         """
254         for node in self.graph.nodes(True):
255             if node[1]["pos"] == new_attributes["pos"]:
256                 return node[0]
257         self.graph.add_node(new_node, new_attributes)
258         return new_node
259
260     def read_vtk(self, file_name):
261         import numpy as np
262         try:
263             from mesh import Mesh
264         except ImportError:
265             raise ImportError("read_vtk requires pymesh")
266
267         # initialize the vtk reader
268         reader = Mesh()
269
270         # read the vtk
271         reader.ReadFromFileVtk(file_name)
272
273         # add nodes to the graph
274         for index, node in enumerate(reader.node_coord):
275             self.graph.add_node(index, pos=reader.node_coord[
276 index])

```

```

Row Storage)
    dont voici une illustration :
279     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) po
l'element,
281     (0, 1, 3, 4) pour l      el      e - ment 2 et (1, 5) pou
        l      el      ement 3.
        Deux tableaux sont utilis es , l un pour stocker d
fa con contigu e
283     les listes de n uds qui composent les      el      ements
        table 1), l autre
        pour indiquer la position , dans ce tableau , ou' comm
chacune de
285     ces listes (table 2).
        Ainsi, le chiffre 6 en position 2 dans le tableau p
elem2node indique
287     que le premier n ud de l      el      ement 2 se trouve
        position 6 du
        tableau elem2node. La derni'ere valeur dans p elem2no
correspond au
289     nombre de cellules (la taille) du tableau elem2node.

elem2node
291     0 | 2 | 3 | 1 | 2 | 4 | 0 | 1 | 3 | 4 | 1 | 5
293     1  2  3  4  5  6  7  8  9  10 11 12
        ^      ^      ^      ^      ^
295
p_elem2node
297     0 | 3 | 6 | 10 | 12
        1  2  3  4  4
299     '''

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

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

309
        datas = np.asarray([data['pos']
311                             for _ , data in self.graph.nodes(d
-True)])

```

```

315         yi = datas[u][1]
316         zi = datas[u][2]
317         xj = datas[v][0]
318         yj = datas[v][1]
319         zj = datas[v][2]
320         return math.sqrt((xi-xj)*(xi-xj) +
321                           (yi-yj)*(yi-yj) + (zi-zj)*(zi-zj))
322
323     # add edges to the graph
324     for cell in chunker(reader.elem2node, reader.
325 p_elem2node):
326         for u, v in pairwise(cell):
327             if u not in self.graph[v]:
328                 self.graph.add_edge(u, v, weight=distance(
329 u, v, datas))
330
331     def distance(self, nodei, nodej):
332         xi = nodei[0]
333         yi = nodei[1]
334         xj = nodej[0]
335         yj = nodej[1]
336         if len(nodei) == 3 and len(nodej) == 3:
337             zi = nodei[2]
338             zj = nodej[2]
339             return math.sqrt((xi-xj)*(xi-xj) + (yi-yj)*(yi-yj)
340                               + (zi-zj)*(zi-zj))
341         return math.sqrt((xi-xj)*(xi-xj)+(yi-yj)*(yi-yj))
342
343     # cartesian norme in 2D
344     def distance2D(self, nodei, nodej):
345         xi = self.graph.nodes(data=True)[nodei][1]['pos'][0]
346         yi = self.graph.nodes(data=True)[nodei][1]['pos'][1]
347         xj = self.graph.nodes(data=True)[nodej][1]['pos'][0]
348         yj = self.graph.nodes(data=True)[nodej][1]['pos'][1]
349         return math.sqrt((xi-xj)*(xi-xj)+(yi-yj)*(yi-yj))
350
351     # cartesian norme in 3D
352     def distance3D(self, nodei, nodej):
353         xi = self.graph.nodes(data=True)[nodei][1]['pos'][0]
354         yi = self.graph.nodes(data=True)[nodei][1]['pos'][1]
355         zi = self.graph.nodes(data=True)[nodei][1]['pos'][2]
356         xj = self.graph.nodes(data=True)[nodej][1]['pos'][0]
357         yj = self.graph.nodes(data=True)[nodej][1]['pos'][1]
358         zj = self.graph.nodes(data=True)[nodej][1]['pos'][2]
359         return math.sqrt((xi-xj)*(xi-xj) + (yi-yj)*(yi-yj) +
360                           -zi)*(zi-zj))

```

```

361         coord2D = {}
362         for key, value in nx.get_node_attributes(G,
363                                                    'pos'):
364             iteritems():
365                 coord2D[key] = [value[0], value[1]]
366         return coord2D
367
368     # display the mesh using networkx function
369     def display_mesh(self):
370         nodelist = []
371         node_color = []
372         for node in self.graph.nodes(data=1):
373             node_type = node[1]['FID']
374             if not node_type == '':
375                 nodelist.append(node[0])
376                 node_color.append('r' if node_type == 'sink'
377                                  else 'b')
378             try:
379                 nx.draw_networkx(self.graph, pos=self.coord2D(),
380                                 nodelist=nodelist,
381                                 with_labels=0, node_color=
382                                 node_color)
383             except:
384                 pass
385
386     # display the mesh with a path marked on it
387     def display_path(self, path):
388         nodelist = []
389         node_color = []
390         for node in self.graph.nodes(data=1):
391             node_type = node[1]['FID']
392             if not node_type == '':
393                 nodelist.append(node[0])
394                 node_color.append('r' if node_type == 'sink'
395                                  else 'b')
396
397         color = {edge: 'b' for edge in self.graph.edges()}
398         # returns an array of pairs, the elements of seq two
399         two
400         def pairwise(seq):
401             return [seq[i:i+2] for i in range(len(seq)-2)]
402         # colors the edges
403         for u, v in pairwise(path):
404             if (u, v) in color:
405                 color[(u, v)] = 'r'
406             if (v, u) in color:

```

```

importatant!
    array = []
405     for edge in self.graph.edges():
        array += color[edge]
407     nx.draw_networkx(self.graph, pos=self.coord2D(),
                        nodelist=nodelist, with_labels=True,
409                        node_color=node_color, edge_color=
=array)

411     def shortest_path(self, node1, node2):
        """
413         Calculates the shortest path on self.graph.
        Path is a sequence of traversed nodes
415         """
        try:
417             path = nx.shortest_path(self.graph, source=node1,
            target=node2,
                                weight="weight")
419         except:
            pass
421         return path

423     def path_lenght(self, path):
        """
425         given a path on the graph returns the lenght of the path
        in the
        unit the coordinats are expressed
427         """
        if path is None:
429             return float("inf")
        lenght = 0.0
431         # given the path (list of node) returns the edges
        contained
        path_edges = [path[i:i+2] for i in range(len(path)-2)]
433         # iterate to edges and calculate the weight
        for u, v in path_edges:
435             lenght += self.distance(u, v)
        return lenght

437     def TSP(self, cities):
        """
439         declaring the adjacency matrix
        T = numpy.empty(shape=(len(cities),len(cities)))
441         for u, i in cities:
            for v, j in itertools(cities):
443                 uv_path = shortest_path(u, v)

```

```

449         paths = []
450         for combo in itertools.permutations(range(1, len(T
[0])))):
451             lenght = 0
452             prev = 0
453             path = []
454             path += [0]
455             for elem in combo:
456                 lenght += T[prev][elem]
457                 prev = elem
458                 path += [elem]
459             lenght += T[combo[len(combo) - 1]][0]
460             path += [0]
461             paths.append((path, lenght))
462             stop = timeit.default_timer() # stop timer
463             time = stop - start
464             return paths, time
465         '''
466
467     def is_sourcesink(self, node):
468         '''given a node as in the networkx.Graph.nodes(data=1)
469         returns 1 if the node is a sink or a source, 0 elsewh
470         '''
471         if not node[1]['FID'] == '':
472             return 1
473         return 0
474
475     def compute_source_matrix(self):
476         for node in self.graph.nodes(data=1):
477             if self.is_sourcesink(node):
478                 self.sinksources_graph.add_node(node[0], node[1])
479
480         for n1 in self.sinksources_graph.nodes():
481             for n2 in self.sinksources_graph.nodes():
482                 if n1 is not n2:
483                     path = self.shortest_path(n1, n2)
484                     if path is not None:
485                         self.sinksources_graph.add_edge(n1, n2,
486                                                             {'dist': self.
487                                                             path_lenght(path),
488                                                             'path': path})
489
490     def design_minimal_aqueduct(self, G):
491         minimal = nx.minimum_spanning_tree(G, weight='dist')
492         return minimal

```

```

495         path += edge[2][ 'path' ]
496     self.display_path(path)

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

505     def mesh_graph(self , G, weight):
506         """complexity (len(G.nodes))^3"""
507         distances = nx.get_edge_attributes(G, weight)
508         # condition to create the gabriel relative neighbour
graph
509         def neighbors(p, q):
510             for r in G.nodes:
511                 if r != q and r != p:
512                     def dist(n1, n2):
513                         if (n1, n2) in distances:
514                             return distances[(n1,n2)]
515                         else:
516                             return distances[(n2,n1)]
517                     if max(dist(p,r), dist(q,r)) < dist(p,q):
518                         return True
519                     return False
520             return True

521         # connect graph
522         gabriel_graph = nx.Graph()
523         for n1 in G.nodes():
524             for n2 in G.nodes():
525                 if n1 != n2:
526                     if neighbors(n1, n2):
527                         gabriel_graph.add_edge(n1, n2)
528         return gabriel_graph

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

531
532
533         # Initialize Edge Matrix
534         edgeMat = [[0 for x in range(len(G))] for y in range

```

```

1)
    for i, node in enumerate(G):
539         tempNeighList = G.neighbors(node)
        for neighbor in tempNeighList:
541             edgeMat[i][node_dict[neighbor]] = 1
            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
551     from sklearn.cluster import MeanShift,
    estimate_bandwidth

553     # creates a array with the 2D coordinats for each node
    X = [[node[0], node[1]] for node in G.nodes()]
555     # estimates the dimensions of single clusters
    bandwidth = estimate_bandwidth(X, quantile=0.1,
557                                random_state=0, n_jobs=1)

    # find clustes
559     ms = MeanShift(bandwidth=bandwidth)
    ms.fit(X)
561

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

565     # — ADDUCTION —
    '''
567     # add cluster centers to the graph
    for node in cluster_centers:
569         attribute = {'label': 'water tower', 'pos': node}
        G.add_node(node, attribute)
571         attribute = {'type': 'sink'}
        self.sinksources_graph.add_node(node, attribute)
573     '''

    adduction = nx.Graph()
575     cluster_centers = [(node[0], node[1]) for node in ms.
    cluster_centers_]
    for node in cluster_centers:
577         adduction.add_node(node)
        self.complete_graph(adduction)

```

```

adduction.nodes(data=True)}
    # nx.draw_networkx(adduction, pos=coord, label=False)
583     self.write2shp(adduction, "adduction_network")
    self.acqueduct.add_edges_from(adduction.edges())
585
    # — DISTRIBUTION —
587     # add label info to the graph
    nx.set_node_attributes(G, labels, 'label')
589     # initialize distribution graphs
    distribution = [nx.Graph() for cluster in
cluster_centers]
591     for node in labels:
        cluster = labels[node]
593         distribution[cluster].add_node(node)
        ,,,
595     # connect each node with his the cluster center
    node_list = []
597     for index, node in enumerate(G):
        node_list.append(node)
599         labels = nx.get_node_attributes(G, 'label')
        label = labels[node]
601         if label is not 'water tower':
            G.add_edge(node, cluster_centers[label])
603         ,,,
    for dist_graph in distribution:
605         self.complete_graph(dist_graph)
        dist_graph = nx.minimum_spanning_tree(dist_graph,
weight='dist')
607         self.acqueduct.add_edges_from(dist_graph.edges())

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

617     # turn path into acqueduct graph
    datas = [data['pos'] for _, data in self.graph.nodes(
data=True)]
619     path_coord = [tuple(datas[node]) for node in path]
    path_edges = [path_coord[i:i + 2] for i in range(len(
path_coord) - 2)]
621     self.acqueduct.add_edges_from(path_edges)

```

```

        w.field("name", "C")
627         line = path_edges
        w.line(parts=line)
629         w.record('path')
        w.save('path')
631
def render_vtk(file_name):
633
    import vtk
635
    # Read the source file.
637     reader = vtk.vtkUnstructuredGridReader()
    reader.SetFileName(file_name)
639     reader.Update() # Needed because of GetScalarRange
    output = reader.GetOutput()
641     scalar_range = output.GetScalarRange()
643
    # Create the mapper that corresponds the objects of the v
    vtk file
    # into graphics elements
645     mapper = vtk.vtkDataSetMapper()
    mapper.SetInputData(output)
647     mapper.SetScalarRange(scalar_range)
649
    # Create the Actor
    actor = vtk.vtkActor()
651     actor.SetMapper(mapper)
653
    # Create the Renderer
    renderer = vtk.vtkRenderer()
655     renderer.AddActor(actor)
    renderer.SetBackground(1, 1, 1) # Set background to whit
657
    # Create the RendererWindow
659     renderer_window = vtk.vtkRenderWindow()
    renderer_window.AddRenderer(renderer)
661
    # Create the RendererWindowInteractor and display the
    vtk_file
663     interactor = vtk.vtkRenderWindowInteractor()
    interactor.SetRenderWindow(renderer_window)
665     interactor.Initialize()
    interactor.Start()
667
def tsp_example():
669     return 0

```

```

673 def template_clustering(path_sample, eps, minpts,
675     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);

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

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

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

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

693         ordering_visualizer.show_ordering_diagram(analyser,
            amount_clusters);

695 def vesuvio_example():
697     router = Router(topo_file="vtk/Vesuvio")
        router.route-vesuvio(32729, 31991)
699     # write to vtk
        router.write2vtk(router.acqueduct)
701     # render_vtk("vtk/Vesuvio")

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

707 def cluster_simple_example():
709     import random;

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

```

    from pyclustering.samples.definitions import SIMPLE_SAMPLES,
    FCPS_SAMPLES;

717
    template_clustering(SIMPLE_SAMPLES.SAMPLE_SIMPLE1, 0.5, 3)
719
    paesi_example()
721 # 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
723 # Railroads/Railroads
    # Routesnaples/routesnaples
725 # "shapefiles/Domain", "shapefiles/pointspoly"
    # "shapeline/shapeline", "shapeline/points"
727 # nx.draw_networkx(router.sinksources_graph, pos=router.coord2D
    , with_labels=0)
    # router.design_minimal_aqueduct()
729 # router.display_path(path)
    # nx.draw_networkx_nodes(router.graph, pos=router.coord2D())
731 # router.display_mesh()
    # print nx.clustering(router.graph)
733 # print nx.floyd_warshall_numpy(router.graph, weight='dist')
    # runfile('/Users/Conrad/Documents/EC/Course deuxieme ann
    Project Inno/Projet_P5C006/router.py', wdir='/Users/Conrad
    Documents/EC/Course deuxieme ann e/Project Inno/
    Projet_P5C006')

```

Appendix D

Il manuale utente

Manuale utente per l'utilizzo del sistema

Appendix E

Use case scenario

In this section will be presented a couple of use case scenarios, from 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 code:

```
router = Router(topo_file = "vtk/Vesuvio")
router.route_vesuvio(32729, 32729)
```

The topography of the Vesuvio and surrounding areas is loaded from a file into a networkx graph data structure of the router python class. The shortest path is computed using the Dijkstra 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.1.

E.2 Clustering

E.3 Travel Salesman Problem

E.4 Minimum Spanning Tree

E.5 Automatic design

Executing the command `paesi $\scriptstyle example()$` will reproduce the case studied in

E.5. Automatic design

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

Appendix F

Datasheet

Eventuali Datasheet di riferimento.