Master in High Performance Computing

Characterization of the "Generali" customers as a network and profiling of its communities

Supervisors:
Dr. Stefano Cozzini,
Dr. Valerio Consorti

Candidate:
Alessia Andò

2nd edition
2015–2016

Contents

Chapter 1

Introduction

The work presented in this thesis has been performed within the collaboration among MHPC and the Analytic Solution Center (ASC), an office which is part of Assicurazioni Generali Head Office and is responsible of advanced analytic and big data projects. Generali proposed the overall topic: modelling of the customer database as a social network and community identification. The Analytic Solution Center took also care to identify such a large dataset within the company and its subsidaries. They finally identify the Genertel (a subsidiary company of Generali) customer-base archive as the dataset to work on. The data have been first anonymized by Genertel and then the anonymized dataset has been delivered for the analysis.
The goal of the project is to implement on HPC resources a software able to detect the communities from the dataset identified.

The work performed will be presented in this document as follows.
In the rest of this chapter we will give a short introduction to Cluster Analysis and the most important related algorithms.
In chapter 2 we will describe the hardware and software architecture we implemented, and we will give more details about the two clustering algorithms we adopted in this work.
In chapter 3 we will describe the implementation on HPC platforms of our software architecture, and the whole "optimization journey" we performed.
Finally, in chapter 4 we will analyse the performance of our implementations, and the results obtained in terms of network communities identified.

The whole activity was co-supervised by Generali Big data group (dr. Valerio Consorti), which suggested for the greatest part the way to proceed. We remark that some deliverables has been submitted to Genertel and intermediate results were periodically discussed during several meetings.

1.1 Introduction to Clustering

Clustering, or *Cluster Analysis* is a collection of methods aimed at partitioning a set of *objects* so that objects in the same subset are "more similar" to each other than to those in other subsets, according to some *similarity* or *dissimilarity* measure.
Throughout the document, given a set D of objects, a similarity will be a symmetric function from $D \times D$ to $[0, 1]$, while a dissimilarity or *distance* will be a symmetric function d from $D \times D$ to $[0, +\infty)$ such that

$$d(x, y) = 0 \Leftrightarrow x = y$$

for all x, y in D. If d also satisfies the triangle inequality, it will be a *metric* distance.

Cluster analysis is not a specific algorithm, but the general partitioning problem. In fact, there is not even a universal strict definition of what a cluster is. This leads to different cluster *models*, into which numerous clustering algorithms are grouped.
Amongst the main models we have, for example, *Hierarchical clustering*[1], which seeks to build a hierarchy of clusters, and is further divided into *Agglomerative clustering* (bottom-up approach) and *Divisive clustering* (top-down approach).
Alternatively, we have *k-means* and *k-medoids* methods, where clusters are defined as subsets of points which are closest to their corresponding cluster *center*, and the center of the clusters are found through minimization of an objective function (sum of the square distances).
The main drawback of this last class of clustering algorithms is that they are not able to detect clusters with an arbitrary shape. This is however achieved by algorithms based on *local density*, which is the number of points within a threshold distance ε, that must be given as a parameter. This means that for each point we are only interested in the points which are closest to him, therefore there is no problem, in principle, with using a distance function which is sometimes evaluated to ∞. Moreover, with both algorithms the number of clusters is determined automatically and does not need to be provided in advance (unlike in k-means).
Algorithms based on local density include DBSCAN[2] and Density Peak[3].

When D is a set of vertices in a network and the definition of the similarity/dissimilarity measure involves the network structure (i.e. the set of edges and their weights) we talk about *Community detection*. The general idea behind it is to form strongly linked subnetworks starting from the original network, and the weight of the edges between the different pairs of vertices defines a similarity measure.
Community detection can also by achieved by numerous classes of algorithms[4],

but few are efficient enough to be used on a very large network. Among these, we have *Label propagation* methods, where the main idea is that objects are intialized with some *labels* and then, iteratively, labels are somehow propagated according to the network structure. At the end, two vertices end up in the same community if and only if they share the same label.

1.2 Cluster analysis on the given dataset

The dataset we are working with is composed by Genertel customers, and each of the customers is provided with an n-tuple of *features*. Such features have different types, they can be represented by real numbers, integers, strings or lists. Therefore, we are not able to use a *standard* distance such as an L^p distance in $\mathbb{R}^n$.
In the following, we will refer sometimes to customers as *nodes* in the obtained network.
The idea is to assume that each feature f contributes *independently* with probability p_f (which is defined for each f individually) to the final link probability, and therefore that the final probability p that two given individuals are linked is

$$p = 1 - \prod_f (1 - p_f).$$

The probability above is then used as a similarity function.

Starting from the data referring to the last 6 months of the 20-years time period which is covered by the dataset, we proceed by analyzing bigger and bigger subsets of the dataset, investigating the evolution of the communities and of the patterns describing connections between customers/nodes. To this end, according to the similarity function above, we use two clustering algorithms. In order to use DBSCAN, we define a metric distance function which is compliant with the similarity function. In particular, the distance between two individuals is bigger when they are less similar according to the function p above.
One possibility is to assume that p_f is 0.99 (or some number which is very close to but less than 1) for each feature that indicates a “deterministic” link, and then define the distance d as $-\log p$. In this case, d satisfies the triangle inequality whenever, given individuals A, B, C we have

$$p_{A \leftrightarrow B} \cdot p_{B \leftrightarrow C} \leq p_{A \leftrightarrow C},$$

which is a reasonable constraint for a probability function as above.

The second algorithm is Label propagation[5], which requires the dataset to be first modeled as a network. We compare then the communities obtained by both algorithms for different subsets of the set of features.

Chapter 2

Methodologies and procedures

In this chapter we will describe the basic elements and tools we used to perform our work. We will briefly describe the hardware architecture. We will then discuss the data structure and some preliminary work done on them. After that we will present the software implementation we performed, discussing our technical choices.
We will finally illustrate in detail the clustering algorithms that we considered. We remark here that our implementation choices were mostly dictated by the company's needs.

2.1 The hardware facilities

We developed our code and mostly worked, whenever possible, on the ELCID cluster, which is property of CNR-IOM. Otherwise, we worked on the COSILT (Amaro, UD, Italy) cluster.

	compute nodes	cores per node	processor	RAM per node
ELCID	8	64	AMD 6376	128 GB
COSILT	10	24	E5-2697 v2 Intel processor	64 GB

The table above summarizes the technical specification of the two infrastructures.

2.2 The data

The dataset we have been working with consists of the Genertel customers (car segment) spawning more than 20 years, for a total of 7.6M customers, therefore 7.6M nodes in the network. The database, originally stored in an Oracle DB, has been exported, anonymized and given to us in the form of

11 csv files.

The data contained in such files allow us to determine the features associated to each customer. Some features are associated to *deterministic links*, i.e. similarity 0 or 1 between the pairs of customers. The features that are associated at first to a deterministic link are:

- credit card number,
- agent code,
- quotation number,
- plate number.

All the features above are actually represented as lists, since each individual may register more than one credit card, ask for more than one estimate and so forth. For all the above, we are interested in knowing whether they are the same for the two individuals, or better if the intersection of the corresponding lists associated to the individuals are non-empty.
The features that are associated to a probabilistic link are:

- e-mail address, which will be considered together with an estimated probability that the e-mail address is real
- phone number, same as above
- e-mail domain, which will be considered whenever it refers to a company, or a workplace in general
- address, which will be analyzed according to the distance between the two individuals, the population of their towns and their density
- last name, which may indicate a family link whenever the two individuals are physically close to each other, depending on how much the last name is spread in that particular area
- date of birth, also to be considered whenever the two individuals are physically close.

2.2.1 Address anonymization

With regard to the addresses, the company proceeded with the anonymization using geocoding, i.e. given an address, they retrieved the correspondent latitude and longitude, ad then translated all the coordinates by the same vector. Such procedure is compatible with our analysis, since we only need the distance between the pairs of customers, in order to compute their similarity.

In the preliminary phase, we were also asked to check and verify the use of Google's Geocoding API - a service which provides both geocoding and reverse geocoding - to anonymize the data. We therefore developed a Python script to fulfill such request.
The API can be accessed through an HTTP interface. [1]

Since the API needed to process an input file with several millions of addresses, we checked whether the process could be parallelized, for example by splitting the file in advance and then starting a different independent process for each chunk of the file at the same time.
The first attempts were made in a 4-core laptop, using an input file with 100 addresses, GNU parallel, and the following Python code:

```
import googlemaps
import time
import sys

gmaps = googlemaps.Client(key=MYKEY)

start = time.time()

with open(sys.argv[1]) as inputfile:
    for line in inputfile:
        geocode_result = gmaps.geocode(line)
        res = geocode_result[0][u'geometry'][u'location']
        print res[u'lat'],res[u'lng']

print("--- %s seconds ---" % (time.time() - start))
```

The program scaled very well up to this point: modulo fluctuations, it takes 32.5 seconds with one single process, 16.5 seconds with two processes and about 9 seconds with 4 processes.
The Python interface to the API requires the modules **googlemaps** and **requests**, both installed through **pip** in this case.
The program was later tested on ELCID and scaled just as well.

2.3 Software implementation

Our software implementation was performed using the Python language. There are two main reasons for our choice:

[1]To this end, one must first activate the API in their Google API Console[6], then obtain credentials. The only option for credential in the case of Standard (free) plan is a server API key[7], which is provided with 1000 free requests per day. Unless specified otherwise, the same key can be used from different machines.

- Having to deal with several deadlines, dictated by the company's needs, it was necessary to keep the developing time short enough
- the company needed a final version of the codes in Python, in order to export them to their Cloud system.

We identified three main parts in our software implementation.
In the first part we read and process the input files, save the data we need into some data structures, and then compute the similarities/distances between the pairs of objects. This part has quadratic complexity, and it is the most expensive part of the program, therefore it runs in parallel.
The second part consists in running the clustering algorithm on the dataset we obtained. Since the elapsed time is negligible with respect to the rest of the program, there was no need for parallelization.
The third part consists in creating two output files containing some information about the clusters/communities that have been output by the algorithm, in order to profile them. All the requested information assumes that the dataset has been modeled into a network, regardless of the clustering algorithm, which may not require a network structure.
The first output file contains, for each object that ended up in a cluster (with more than one object), the following:

- cluster ID
- degree within cluster (number of neighbors in the same cluster)
- norm degree (degree within cluster/number of points in the cluster)
- closeness centrality within cluster
- betweenness centrality within cluster
- eigenvector centrality within cluster
- PageRank centrality within cluster

The closeness centrality is the the sum of the length of the shortest paths between the node and all other nodes in the cluster (viewed as a subnetwork). The betweenness centrality is the number of shortest paths from all vertices in the cluster to all others that pass through that node. The eigenvector centrality is another indicator of the influence of a node in the cluster, in particular it assigns *scores* to the nodes with the assumption that connections to high-scoring nodes contribute more to the score of the node in question than equal connections to low-scoring nodes. Finally, the PageRank centrality is the result of one of the algorithm used by Google Search to rank websites in their search engine results.
Due to the computation of the centralities, the creation of this file has also

quadratic complexity.

The second output file contains, for each cluster (with more than one object), the following:

- size
- number of edges
- average degree
- maximum lenght of a shortest path between a pair of objects in the cluster
- degree centralization

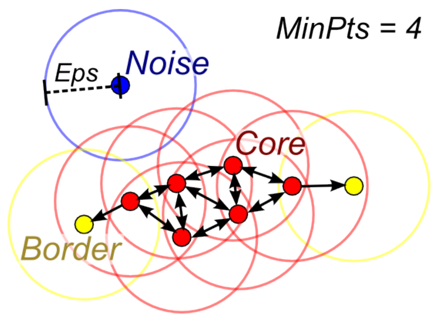
The degree centralization is a measure of how central is the most central vertex (with respect to the degree centrality, i.e. the number of ties a node has) compared to the other vertices.
Due to the last two pieces of information, the creation of this file has also quadratic complexity. That is why the creation of the two files runs in parallel, too.
Given that most of the measures asked are strictly graph-related, we chose to use a library to manipulate our graphss, igraph[9]. The documentation also provides a detailed decription of such measures.

2.3.1 Dealing with graphs

igraph is open source graph package, distributed under the terms of the GNU GPL. It is a collection of network analysis tools, whose main goals are to provide easy implementation of graph algorithms, and fast handling of large graphs (millions of vertices and edges). Therefore, it not only provides efficient data structures for representing graphs and related objects, but it also contains the implementation of many graph algorithms, including community detection algorithms such as Label propagation and optimized algorithms to compute several centrality measures.
It was originally written in C, but can be embedded into higher level languages. In fact, it has also been developed as a Python extension module, and this is the form in which we use it.
Being part of the Python Package Index, it was installed through the `pip` command.

2.4 The clustering algorithms

As we briefly anticipated previously, the goal is to compare a data clustering algorithm - therefore, not based on a network structure - and a community detection algorithm.



2.4.1 Data clustering - DBSCAN

As mentioned in the previous chapter, DBSCAN (Density-based spatial clustering of applications with noise) is based on the concept of local density.
The idea is the following: a *core* point p is a point with at least **minPts** points within ε (**minPts** is also to be given as parameter) and those points are said to be *directly reachable* from p. A point q is reachable from p if there is a path $p = p_0\,p_1 \cdots p_n = q$ such that p_{i+1} is directly reachable from p_i (so, in particular p_i is a core point) for each i. Points which are not reachable from any point are considered noise. A core point will end up in the same cluster as all points which are reachable from it.

Although DBSCAN may visit each point multiple times, its computational complexity is mostly determined by the number of REGIONQUERY calls (see algorithm 1), which are exactly one per point. A meaningful value for ε is such that on average $O(\log n)$ points are returned by each REGIONQUERY call, therefore the average complexity is in this case given by $O(n \log n)$.

minPts must also be chosen carefully, otherwise DBSCAN will not be able to detect smaller clusters, or merge some of them that are supposed to be separate. In general, it is best if ε and **minPts** are chosen by someone who knows the data very well.
There is no good choice for the parameters whenever the points in the dataset have large differences in density. Otherwise, DBSCAN is able to detect clusters of any shape.
The performance of DBSCAN obviously depends on the quality of the distance function, too.

Algorithm 1 DBSCAN

```
Require: dataset $D$, $\varepsilon > 0$, $m$ = minPts
 1: function regionQuery($p$,$\varepsilon$)
 2:     return all points in $p$'s $\varepsilon$-neighborhood
 3: function expandCluster($p$, $neighPts$, $C$, $\varepsilon$, $m$)
 4:     add $p$ to cluster $C$
 5:     for point $p'$ in $neighPts$ do
 6:         if $p'$ is not visited then
 7:             mark $p'$ as visited
 8:             $neighPts' \leftarrow$ regionQuery($p'$, $\varepsilon$)
 9:             if size of $neighPts' \geq m$ then
10:                 $neighPts \leftarrow neighPts \cup neighPts'$
11:         if $p'$ is not yet in any cluster then
12:             add $p'$ to cluster $C$
13: function DBSCAN($D$, $\varepsilon$, $m$)
14:     $C \leftarrow 0$
15:     for point $p$ in $D$ do
16:         if $p$ is visited then
17:             continue
18:         mark $p$ as visited
19:         $neighPts \leftarrow$ regionQuery($p$, $\varepsilon$)
20:         if size of $neighPts < m$ then
21:             mark $p$ as NOISE
22:         else
23:             $C \leftarrow nextCluster$
24:             expandCluster($p, neighPts, C, \varepsilon, m$)
```

2.4.2 Community detection - Label propagation

Label propagation is a community detection algorithm that also works with networks where edges are weighted (unlike other such algorithms).

The main idea is the following: suppose that each of the neighbors of some node carries a label denoting the community to which its belong to. Then the node determines its community based on the labels of its neighbors, in particular it joins the community to which the maximum number of its neighbors belong to, with ties broken uniformly randomly.
At first each node is initialized with a unique label, then labels propagate at each step, until each node's label is the one which appears most frequently among its neighbors (or one of them, in case of ties).
Whenever the edges have weights, each neighbor counts as much as the weight of the edge connecting the it to the original node.

The update process is *asynchronous*, meaning that if $C_x(t)$ represents the label given to node x at step t, $x_0, \ldots, x_k$ are x's neighbors and $x_0, \ldots, x_m$ (with $m \leq k$) are exactly the nodes that have already been visited at step t, then we have

$$C_x(t) = f(C_{x_0}(t), \ldots, C_{x_m}(t), C_{x_{m+1}}(t-1), \ldots, C_{x_k}(t-1)),$$

where f returns the label occurring with the highest (weighted) frequency among neighbors.

The variation with *synchronous* updates, i.e. such that

$$C_x(t) = f(C_{x_0}(t-1), \ldots, C_{x_k}(t-1)),$$

is closer to be discrete but has drawbacks, in particular it leads to oscillations in the presence of subnetworks which are bipartite or closer to be bipartite.

The initialization runs in $O(n)$ (linear) time. At each step, each node has to compute the maximum in a list of at most d labels, where d is its degree. Therefore, its label is updated in $O(d)$ time, for a total of $O(m)$ time at each step, where m is the number of edges. The algorithm is experimentally proven to converge in a few steps, and that the number of steps does not depend on n. Therefore, the overall time complexity is $O(m+n)$, near linear time.

Algorithm 2 Label propagation

```
Require: network $N$
1: for node $x$ in $N$ do
2:     $t \leftarrow 0$
3:     $C_x(t) \leftarrow x$
4: repeat
5:     $t \leftarrow t + 1$
6:     $X \leftarrow$ random rearrangement of the order of the nodes
7:     for $x$ in $X$ do
8:         update label asynchronously
9: until every node has a label that the maximum number of its neighbors
   (counted with weights) has
```

Chapter 3

Implementation

In this chapter we will describe all the steps that we performed in order to reach the final software implementation on HPC platforms.

The first step was to write the Python software to deal with the program according to section 2.3.
We developed two Python scripts: the first one takes part of the dataset - according to the links we need to compute - as input, and outputs all the necessary information to describe the resulting graph (i.e. the pairs of nodes and corresponding weights defining the links). We call this action the *Link generation*. This is indeed the most time-consuming part of the whole implementation. We will discuss in section 3.2 the parallel implementation of this algorithm. The second script takes the obtained list of edges as input, together with the list of vertices (which is information contained in the dataset), performs either DBSCAN or Label propagation, and creates the two tables described in section 2.3. We will discuss this in detail in section 3.3, *Generation and profiling of the communities.*
As soon as we were able to access the dataset, we moved everything to the Lustre file system on ELCID. Before starting to work on the actual program, we spent some time to give a closer look at the dataset and try to figure out how to address the encountered issues. This took longer than expected.

In the next section we will explain in a bit more detail the analysis we performed in order to better understand how to use the dataset.

3.1 Preliminary dataset analysis

After a brief discussion with Generali, we concluded that our first goal was to compute the so-called **Broker** links, perform a clustering algorithm on the obtained network, and profile the communities. The idea behind the Broker link is that two people are connected whenever they have, at some

point, interacted with the same agent in order to get an quotation for some policy.
The first thing we needed to understand was then how to determine when two quotations were given by the same agent, from the information contained in the dataset. In particular, one of the files defining the dataset gives information related to each quotation, including sometimes the agent code and/or the branch code.
We discussed possible ways to determine the probability that two customers were linked whenever the agent code information was missing for all the estimates asked by one of (or both) the customers. To this end we ran a few Python scripts to compute, for istance:

- how many estimates had both the agent code and the branch code information
- how many branch codes were never associated to an agent code
- how many agent codes were never associated to a branch code
- for each agent code, the list of branch codes that were at least once associated to it.

We established a first definition of the Broker (probabilistic) link which used the above information.

Meanwhile, we tried to address the problem of missing information in the file containing personal/anagraphic information on the customers. We needed to use the (anonymized) fiscal code as a node-identifier but most customer IDs did not have that information. In order to find a possible way to retrieve a new fiscal code ID for at least some of the customers, we ran a few Python scripts in order to compute, for istance:

- how many customers without fiscal code information had information on name, date of birth or other data which normally defines the fiscal code
- how many of them had a VAT number code (such customers would in fact be companies, and not physical persons)
- whether any tuple {name, date of birth, place of birth, gender} was associated to more than one customer.

Given the not ideal results, we decided to stick to the customers who were provided with fiscal code information.
After further consultation with Genertel, we also decided to only consider estimates which were provided with an agent code, in order to define the Broker link.

3.2 Link generation

In order to determine the list of edges according to any kind of link, we needed to read (part of the) file containing personal information on the customers. This is because the fiscal code ID had to be used as a node-identifier, so we needed to know which fiscal code ID each customer ID was mapped to. Moreover, within all our tests/attempts we selected a subset of the customers according to the time they were first registered in the database, and the file above also contains that kind of information.
The first results we were asked to provide Genertel concerned the network consisting of the customers who were first registered in the past six months (slightly less than 1 million customers), connected through the Broker links. Therefore, we did all our first attempts with the Broker links in mind.
In order to generate the Broker links we needed two more tables of the database: the first one maps each estimate to the people who were involved, and the second one maps each estimate to other information including the agent code (whenever present). The first part of the script is all about reading the input files and storing the needed data into well-suited dictionaries which were then used to define the `Point` class:

```
class Point(object):
    def __init__(self, name, rolesdict, entdict):
        self.name = name
        self.ent = rolesdict[self.name]
        self.ci = []
        for e in self.ent:
            try:
                if entdict[e] != '':
                    self.ci.append(entdict[e])
            except KeyError:
                pass
    def link(self, other):
        listci = [a for a in self.ci if a in other.ci]
        if len(listci):
            return 1
```

The `link` method of the class returns the weight of the link (whenever present) between two `Points`.

Once the set of points is defined, we need to compute the links between them. Given their amount, the computation must be parallelized.

3.2.1 First attempt of parallelization

Our first attempt of parallelization involved the pp (Parallel Python) library[8] . We only needed to compute, in fact, the upper-triangular part of the adjacency matrix, since the graph is not oriented, and the work was divided between the processes according to the Figure 3.1.

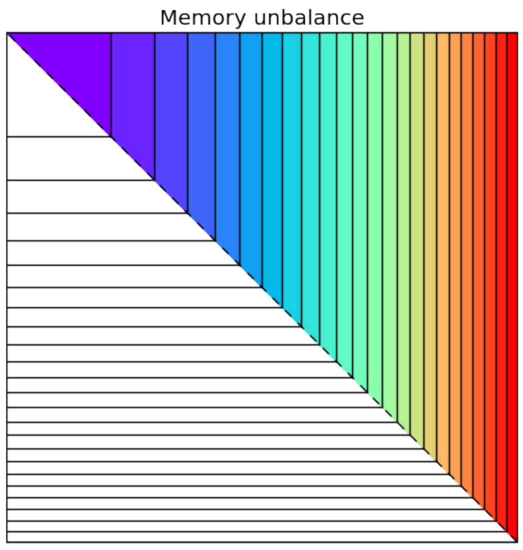


Figure 3.1: Case of 24 processors, as in one COSILT node. Each color corresponds to one thread

The width of the slices varies so that each thread performs the same amount of computations. Memory consumption is however not balanced: the last process (assuming processes are numbered from left to right) needs to know everything about all customers - therefore, it consumes a lot of memory - while the first one only needs to know the data concerning a small subset of the customers.
At the end of the computation, one single process collects all the results and does the actual writing to the output file.

We performed two different kinds of measure: we first measured scalability fixing the number of customers and increasing the number of processes, then fixing the number of processes to the maximum (24 when using COSILT, 64 for ELCID) and increasing the number of customers.

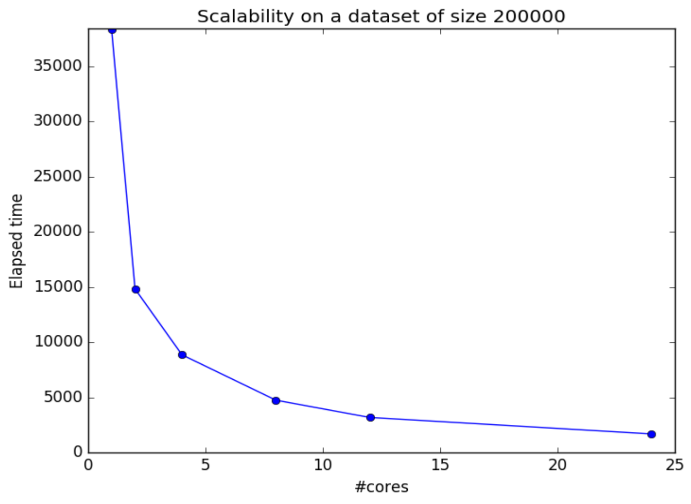


Figure 3.2: Scalability on a node of the COSILT cluster, elapsed time is expressed in seconds

The computation scales almost perfectly with the number of process, as Figure 3.2.

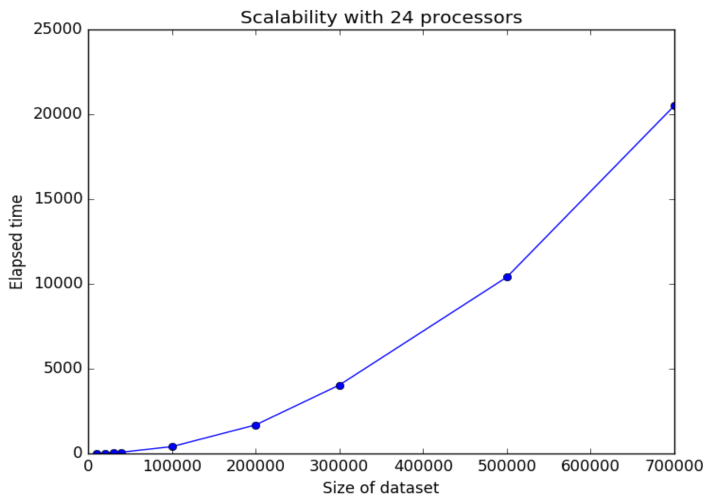


Figure 3.3: Scalability on a full node of the COSILT cluster, elapsed time is expressed in seconds

The scalability with respect the size of the network seems to be quadratic (as it should be). However, we were not able to complete the computation for 1 million customers within the expected amount of time according to

Figure 3.3.

Parallel Python copies everything which is needed by the different threads, so the memory occupation grows a lot with the number of threads. Therefore, the number of threads (together with the size of the network) is actually a bottleneck. Indeed, we were only able to analyse up to 500 thousand customers using one full ELCID node (64 cores) without filling up its RAM, less than the 700 thousands we were able to analyse on a COSILT node (24 cores).

3.2.2 Final shared memory parallelization

For the final implementation, we switched then to the `multiprocessing` library. Despite being normally a little less efficient than Parallel Python, `multiprocessing` solves the memory problem, being able to work with shared memory.

Switching to a different library is not the only thing that changed in our implementation. In fact, we also changed the distribution among processes:

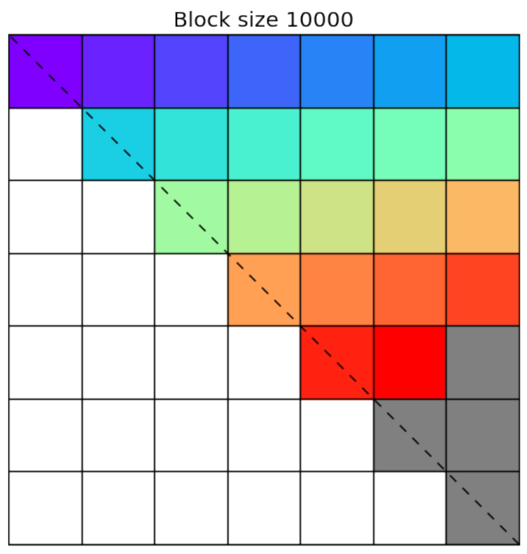


Figure 3.4: Case of 24 processors, as in one COSILT node. Each color corresponds to one thread

The adjacency matrix was divided into blocks of size around 10000 × 10000 (a bit less in some cases, depending on the remainder of the division {size of the matrix}/10000), and processes have to analyse the upper-triangular blocks. The number of cores is always an upper bound for the number of blocks analysed at the same time. Therefore, in the case of a COSILT node, the first 24 blocks are assigned to the 24 processes, as soon as a process completes its job it takes care of the next block that must be analysed, and so forth. After this change, processes are also balanced in memory.

The block size was chosen so that each process takes a "reasonable" amount of time (about 4 minutes in the case of Broker links), meaning that processes do not need to be reassigned too often, and at the same time there is no memory-related issue.

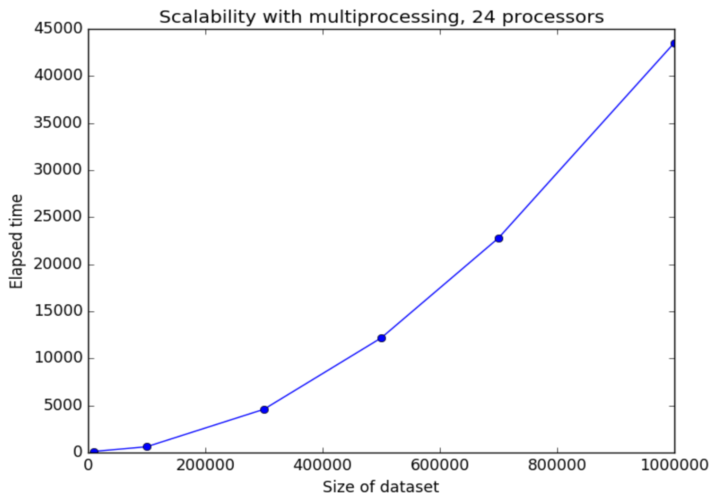


Figure 3.5: Scalability on a full node of the COSILT cluster, elapsed time is expressed in seconds

As Figure 3.5 shows, we were able to run the script for the entire network consisting of 1 million customers.

The elapsed time for this Python script is obviously dependent on the type of link we have to analyse.
Normally - at least in the deterministic case - in order to determine whether two customers are linked we have to compare lists and check whether they have any intersection, and the length of these lists is highly variable depending on what they represent.

Sometimes more than one type of link must be used in order to build the network. This is never a problem, since if the definition of the link becomes too complicated one can just run the script once for each type of link and then merge the output files, provided that the graph which is later obtained through the second Python script is simplified (i.e. multiple links are removed) - more about it in the next section.

After trying a few runs with the other deterministic links, given that the maximum time that can be asked for a run on a COSILT node is 48 hours, we concluded that 1 million is a good size for a network to be analysed by one single shared-memory node.

3.2.3 Inter-node parallelization

Due to the fact that the entire dataset consists however of over 7 million customers, it is clearly needed to perform a further parallelization level including more than one node.

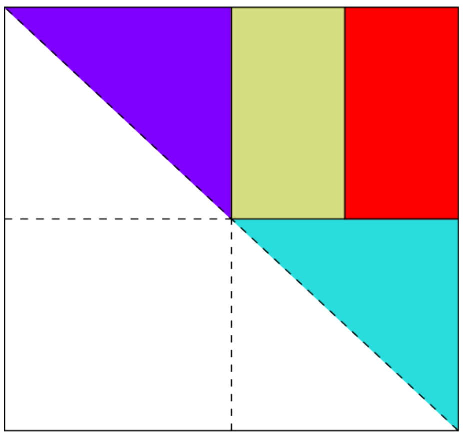


Figure 3.6: Example for 2 million customers

Our parallel approach consisted in splitting the adjacency matrix over multiple nodes. The matrix is divided into blocks of size around 1 million$\times$ 1 million, and each node takes care of one block in the diagonal just as we described so far in the single node case with 1 million customers. As for the other blocks in the upper-triangular part, they need to be split in two parts for memory reasons, and this way they perform the same amount of computations as the blocks in the diagonal. The number of nodes we need to complete the computation for n million customers is therefore

$$n + n(n-1) = n^2.$$

After each node is done with its block, the output files only need to be merged. This means that, if an arbitrarily high number of nodes were available, the time needed to compute all edges within a graph with over 1 million vertices would be the same as the 1 million case.

3.3 Generation and profiling of the communities

The second Python script makes large use of the igraph library. After reading the vertices file, it creates a graph with exactly those vertices, using the fiscal code as a name for each vertex. After that, it reads the edges file while storing the edges and corresponding weights in separate lists, with the purpose of adding all edges to the graph at the same time. This is simply because it is a lot more efficient with igraph to add all edges at once (whenever there are "many" of them).

Then, the clustering algorithm is applied. In the case of Label propagation, the algorithm is already implemented in the library, while DBSCAN was implemented manually through a class. The way DBSCAN is used in our program basically means that a core point is a point with at least **minPts** neighbors connected to it through edges with weight greater than a fixed $r > 0$, and those neighbors are precisely the points directly reachable from it. Therefore, in the DBSCAN case, we could simply only add the edges with weight greater than r and define the class as follows:

```
class DBSCAN(object):
    def __init__(self, minPts, g):
        self.minPts = minPts
        self.graph = g
        self.npoints = g.vcount()
        self.it = iter(range(self.npoints))
        self.clusters = []
    def createCluster(self, p):
        p["group"] = self.it.next()
        self.clusters.append([p])
    def expandCluster(self, c):
        for p in c:
            if p["checked"] != None:
                continue
            if len(self.graph.neighbors(p.index)) >= self.minPts:
                for j in self.graph.neighbors(p.index):
                    q = self.graph.vs()[j]
                    if q["group"] == None:
                        q["group"] = p["group"]
                    c.append(q)
        p["checked"] = 1
```

The `group` attribute of a vertex defines the cluster it belongs to, and it is set as soon as a point is visited by the algorithm.

The clustering algorithm is then followed by the production of the two output files described in the previous section. This last part was also parallelized. In both cases, we have to compute graph-related measures within the obtained clusters, therefore the overall computation consists of a loop over the clusters. We divided the computation so that each process takes care of the same amount of clusters, and measured the scalability in the Label propagation case, fixing the number of processes and varying the number of customers (using the Broker links):

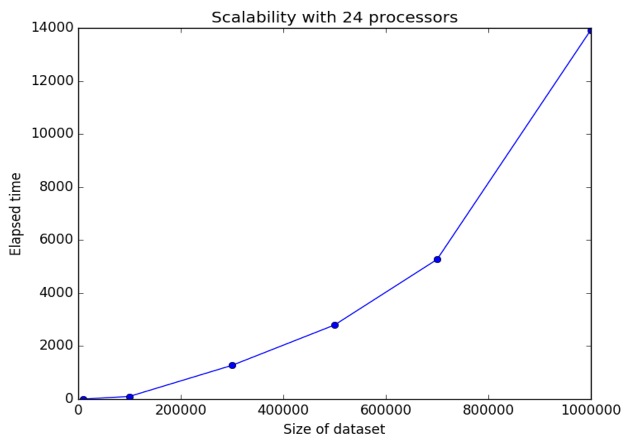


Figure 3.7: Scalability on a full node of the COSILT cluster for the first output file with Label propagation, elapsed time is expressed in seconds

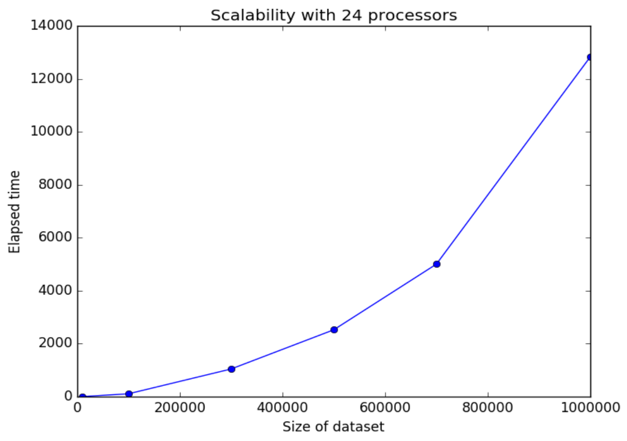


Figure 3.8: Scalability on a full node of the COSILT cluster for the second output file with Label propagation, elapsed time is expressed in seconds

Overall, we notice that the time taken by the generation and profiling of the communities is definitely not negligible with respect to the time taken by the generation of the links. However, it is less. Moreover, such part can trivially be split in two parts, each of which performs the same clustering algorithm independently and produces one of the two output files. This means that the overall computation can additionally benefit from this further trivial parallelization, and that the total time taken only depends on the creation of the slowest of the output tables.

According to Figures 3.7 and 3.8, scalability is quadratic on the number of customers. These preliminary scalability tests, nevertheless, do not explain the situation very accurately. In fact, given the computations involved, the elapsed time is highly dependent on the size of the obtained communities. Computational complexity is (in both cases) not $O(n^2)$, where n is the number of customers, but rather

$$\sum_{\text{communities}} |\text{size of community}|^2.$$

The bottleneck is then the size of the biggest community and the corresponding process takes inevitably longer than the others to complete its computations.
In the Broker case, customers are connected whenever thay have interacted with the same agent. In the entire dataset relatively few (tens of thousands) agent codes appear, and few people have interacted with more than one agent. Therefore, we expect clusters to be “grouped” by one single agent. It is also not strange that all (or most) agent codes appear within the 1-million - or less - subset of customers, given their number. Summing up everything,

it is probably the case that in the Broker case the ratio $n/\{$size of greatest community$\}$ does not vary too much with n, therefore the apparent $O(n^2)$ complexity. This is however highly dependent on the type of link.

Chapter 4

Results obtained with the clustering algorithms

In this chapter we will show some results obtained with our clustering algorithms concerning the following links:

- Broker links
- Credit Card links
- Plate links
- E-mail links

As previously mentioned, the first results we were asked to provide concerned the Broker link for the 1 million customers registered in the past six months. The main purpose was to test both clustering algorithms and compare them. For our first runs with DBSCAN, we set **minPts** to 15. As for ε there is no real choice in the case of deterministic links: we have to include all neighbors in the graph in order to compute the local density.

The results are rather different with the two algorithms:

	DBSCAN	Label propagation
communities	191	1373
communities with more than 15 nodes	191	494
nodes in communities	63020	67438
max size	55128	4207
min size	16	2

The huge community found by DBSCAN is definitely not represented by one single agent code. Indeed, in order to merge two of them it is sufficient that one customers has interacted with both and has at least 15 neighbors in total (which is true in most cases). Of course we could try to increase

the value of **minPts** heavily, however in that case there would be many more agents not forming any community. This is in fact an example of bad performance of DBSCAN in the case of high variations in local density.
Label propagation found a much higher number of communities because most of them are very small (2 or 3 customers). It is not theoretically impossible for DBSCAN to find clusters of size less than **minPts**, but it would be strange if it did in this case, given how the links are defined.

Figure 4.1 shows the distribution of the community size if we only consider cluster of size between 15 and 400. The main difference between the two outputs is due to the huge community found by DBSCAN, which is not considered here.

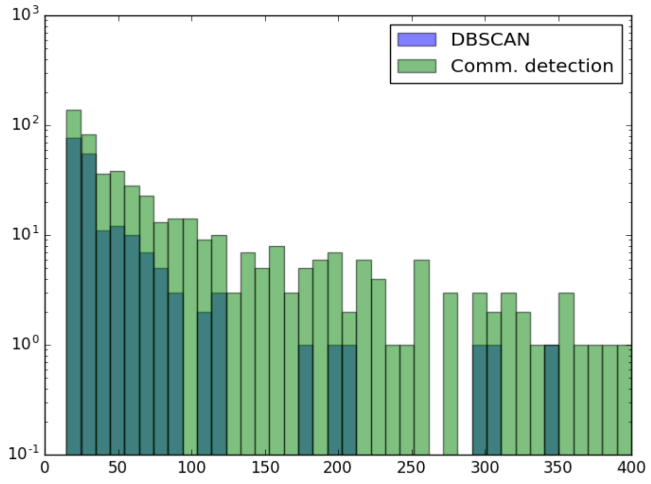


Figure 4.1: Distribution of community size among communities

The difference between the two outputs also results in more internally interconnected communities in the case of Label propagation, being the communities smaller on average and not subjected to “undesired merging”. This can be easily observed from the distribution of the norm degrees in the two cases, as shown in Figure 4.2.

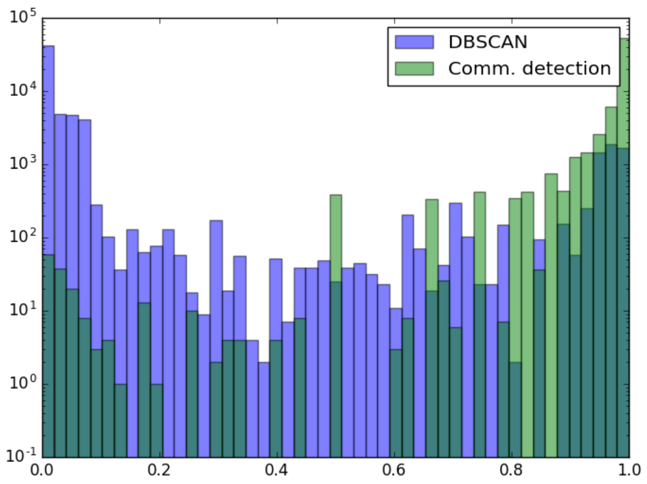


Figure 4.2: Distribution of norm degree within community among nodes

The second output file provides the degree centralization for each community, that is a measure of how central is the most central vertex (with respect to the degree centrality) compared to the other vertices. For both algorithms, in most communities this centralization is close to 0: all customers have about the same role within the community, as expected given the type of link.

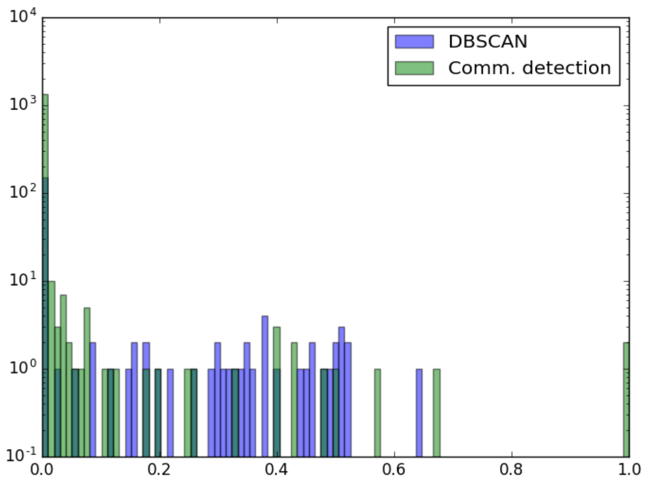


Figure 4.3: Distribution of centralization among communities

As another example, we will show now the results obtained using the Credit Card link, i.e. two people are connected if and only if they share a credit card number. As expected, a lot less customers are part of some cluster in this case, and communities are smaller on average.

	DBSCAN	Label propagation
communities	71	1080
communities with more than 15 nodes	71	96
nodes in communities	4169	7317
max size	443	278
min size	16	2

Extending the computation to all the customers registered in the past year (almost 2 million) the results are the following:

	DBSCAN	Label propagation
communities	119	2762
communities with more than 15 nodes	119	216
nodes in communities	11445	18311
max size	3713	598
min size	16	2

The behavior is similar when restricting to communities of size between 15 and 100:

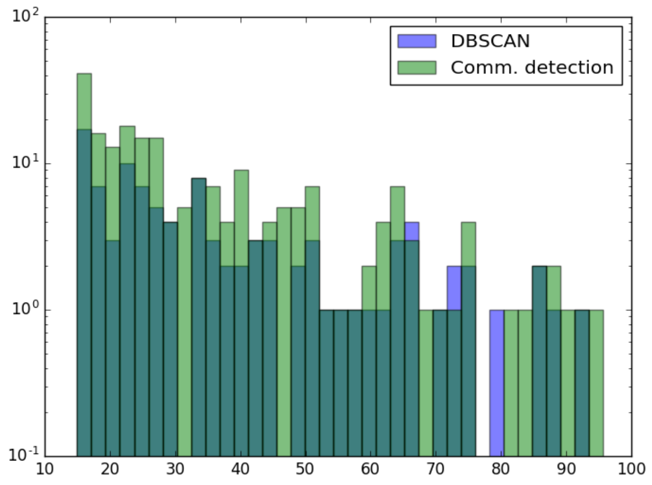


Figure 4.4: Distribution of community size among communities

In this example we can also notice that communities are smaller and more interconnected with Label propagation, but the difference between the two algorithms is smaller compared to the Broker example.

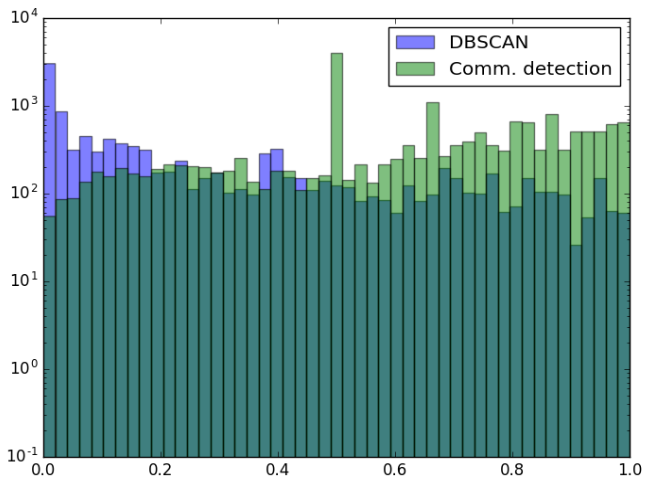


Figure 4.5: Distribution of norm degree within community among nodes

The peak at 0.5 is due to the high number of communities of size 2. Overall, we can see that DBSCAN has still a non-ideal behavior - after all the function we are using as a distance is not a metric - since it merges too much and finds very big clusters. However the situation is not as dramatic as the Broker case, also because the local densities of the points in this case are not as unbalanced.

The Plate links (two people are connected if and only if they share a plate number) led to a similar behavior for both algorithms after we filtered the presumably fake plate numbers, found in a list that we were given by Genertel.

Given the results so far, we decided together with Generali to stick to Label propagation for the following runs. In order to better understand the meaning of the Broker links and the way different links interact, we were asked to run the first script a few times using the Credit Card link as deterministic and the Broker link as probabilistic with some fixed probability $p < 1$. The results were the following:

	$p = 0.5$	$p = 0.1$	$p = 0.01$
communities	2112	2145	3963
nodes in communities	72002	72005	72003
max size	4207	4210	4283
min size	2	2	2

The number of customers who are part of some community is basically always the same. Given the results for the Broker and Credit Card links separately, we may conclude that almost all customers who have at least one neighbor in the network end up in some community (this is technically always true with Label propagation if we also count communities of size 1, but here we are not considering them).

The last link we were able to examine was the E-mail link (two people are connected if and only if they share an e-mail). Unlike the Plates case, it is hard to tell whether an e-mail is fake given the data, therefore we were not able to use a similar filter, and the links obtained were a lot more:

	E-mail	E-mail+Plates	E-mail+Plates+Credit Card
communities	41921	75892	75427
nodes in communities	243747	303770	298478
max size	18744	18753	18753
min size	2	2	2

Chapter 5

Conclusions

In this last chapter we briefly summarize the activities performed within this thesis, underlining the most important achievements we reached. The overall goal of the thesis described in the Introduction has been achieved. In particular, we implemented on HPC resources cluster algorithms able to process and evaluate communities on the dataset provided. The usage of HPC resources allowed us to process and analyse an amount of data several times larger than the amount of data Genertel is able to process internally.

The main results of the thesis concern two important aspects, the HPC implementation and the Cluster analysis.

From the HPC implementation point of view:

- We implemented an intra-node parallelization by means of a shared memory approach
- We implemented an inter-node parallelization
- The total amount of HPC resources used is more than 77000 core-hours. This means 855 jobs submitted so far
- We parallelized and measured speed-up of the tools developed within the node and on a multinode infrastructure

Our implementation is able in principle to deal with arbitrarily large datasets for all kinds of links. However, we did not extend all the computations to the entire dataset for the following reasons:

- Genertel was more interested in analysing different kinds of links on smaller subsets (mostly 1 or 2 million customers) than analysing one single link on the whole dataset

- The HPC resources available to us are shared with other users, which did not allow us to complete all the jobs in a sufficiently small amount of time

With respect to clustering algorithms:

- We generated communities for different kinds of links
- We compared two algorithms for the generation of such communities

Given the limited amount of time, we were only able to analyse only some of the links, mostly deterministic ones.
In order to complete the work, the following has to be done:

- Complete the analysis for all the probabilistic links independently
- Complete the analysis for all the links together
- Extend the analysis to the complete dataset consisting of 7.6 million customers

Bibliography

[1] Lior Rokach, Oded Maimon, *Clustering methods,* Data mining and knowledge discovery handbook, Springer US, pp. 321-352, 2005;

[2] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, *A density-based algorithm for discovering clusters in large spatial databases with noise,* Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96), AAAI Press, pp. 226–231, 1996;

[3] Alex Rodriguez, Alessandro Laio, *Clustering by fast search and find of density peaks,* Science, 344(6191), pp. 1492-1496, 2014;

[4] Santo Fortunato, *Community detection in graphs,* Physics reports 486(3), pp.75-174, 2010;

[5] Usha Nandini Raghavan, Reka Albert, Soundar Kumara, *Near linear time algorithm to detect community structures in large-scale networks,* Physical Review E, 76(3), 2007;

[6] https://console.developers.google.com/flows/enableapi?apiid=geocoding_backend&keyType=SERVER_SIDE&reusekey=true

[7] https://developers.google.com/maps/faq#using-google-maps-apis

[8] http://www.parallelpython.com/

[9] Gábor Csárdi, Tamás Nepusz, *The igraph software package for complex network research,* InterJournal Complex Systems, 1695, 2006;