NTNU – Trondheim
Norwegian University of
Science and Technology

CampusGuiden: Indoor Positioning, Data Analysis and Novel Insights

Santiago Diez Martinez

Master of Science in Communication Technology
Submission date: February 2013
Supervisor: Harald Øverby, ITEM
Co-supervisor: Thomas Jelle, Trådløse Trondheim
Gergely Biczóck, ITEM

Norwegian University of Science and Technology
Department of Telematics

NTNU – Trondheim
Norwegian University of
Science and Technology

CampusGuiden: Indoor Positioning, Data Analysis and Novel Insights

Santiago Díez Martínez

Submission date:	February 2013
Responsible professor:	Harald Øverby (ITEM)
Supervisor:	Gergely Biczók (ITEM), Thomas Jelle (Trådløse T.)

Norwegian University of Science and Technology
Department of Telematics

Title:	CampusGuiden: Indoor Positioning, Data Analysis and Novel Insights
Student:	Santiago Díez Martínez

Problem description

A positioning system enables a mobile device to determine its position, and makes the position of the device available for position-based services such as navigation, tracking or monitoring. The fact that GPS satellite-based positioning systems cannot be deployed for indoor use, the mobility of people and the multi-path effects of the building geometry raise new challenges for indoor positioning systems (IPS). Nowadays public areas such as hospitals, train stations or universities have WLAN technology deployed. WLAN-based positioning systems reuse the existing WLAN infrastructures in indoor environments, cutting down the cost of the services. These systems can also be more easily and quickly set-up if the objects to locate are equipped with WLAN technology. CampusGuiden is a multi-platform application for indoor campus positioning that works integrated with an existing infrastructure. It uses both techniques to guide the user from his actual position to anywhere else inside the Gløshaugen campus, GPS for outdoor and the Wi-Fi network for indoor. CampusGuiden is an umbrella term that includes a central server and a Java application that the end users download and run from their mobile devices. This application has been developed by Wireless Trondheim, and provides a framework for coarsely tracking the indoor location of Gløshaugen campus students and employees. The student will start understanding the data at his disposal. After that, he will split the dump file according to the different kind of requests it contains. And with the location and guide demands, he will make the properly requests to the server obtaining the paths that were followed. The student then will draw a graph of all the possible paths between the points of interest, assigning a weight to each path according to the times it has been used. In this graph he will analyze the data in search for patterns such as of user mobility, traffic bottlenecks and popular paths. In addition to extracting these patterns, the student will also create an expressive visualization of the results. Depending on the insights derived, the student will look into possible business opportunities arising from this knowledge.

Responsible professor:	Harald Øverby (ITEM)
Supervisor:	Gergely Biczók (ITEM), Thomas Jelle (Trådløse T.)

Abstract

A positioning system enables a mobile device to determine its own position, and makes the devicеś position available for other position-based services. Navigation, tracking or monitoring are examples of these kind of services. However, GPS satellite-based positioning systems cannot be deployed for indoor use. People mobility and multipath-effects because of the buildings geometry raise also new challenges for Indoor Positioning Systems (IPS).

CampusGuiden is a multi-platform application for indoor positioning inside the campus. This application guides the user from his current position to any other point of interest in Gløshaugen. Wireless Trondheim has developed this tool, which combines both, GPS and wiFi network to achieve more accuracy. Campusguiden also provides a great framework for coarsely tracking the Gløshaugen campus' students and employees location.

Such location information can be a valuable asset. The student will analyse the data already collected looking for patterns. The goal is finding users' mobility patterns, traffic bottlenecks and popular paths. After that, the student will create an expressive visualization of the results. Depending on the insights derived, the student will look into possible business opportunities arising from this knowledge. Most of the outcomes could be predicted looking the campus distribution, but know we have data that corroborate them.

Keywords: Indoor Positioning Systems, Campusguiden, mobility patterns

Preface

This master's thesis is the result of my exchange program during the last semester of my degree in the Department of Telematics at the Norwegian University of Science and Technology. First of all, I would like to thank my family for allowing this amazing experience and their support received during these months. I also thank my supervisors Gergely Biczók, Harald Øverby and Thomas Jelle for accepting me, and offering me the opportunity of writing my master thesis under their supervision. I want to thank, especially, Gergely for his patience, advice and guidance.
At last but not least, I would like to also thank the friends I have made here (both international and Norwegian) for making my stay here easier. I have learnt a lot, and it has been a period of my life that I will not ever forget.
Thank you everybody, tusen takk Norge.

Contents

List of Figures

List of Tables

Chapter 1
Introduction

The utilization of geographic positional systems has experienced a huge rise since Bill Clinton decided to eliminate the accuracy constraints that prohibited a civil receiver to have a precision better than 20 meters. Nowadays, all kinds of vehicles use these signals for navigation, there are plenty of handy devices for personal guidance, and even the smartphones are equipped with a small receiver. Global Positioning Systems (GPS) could guide us to every outdoor place all over the world, but inside buildings, signals are not strong enough to provide a positioning service. However, in recent years, many technologies and techniques have emerged for indoor positioning. Tracking personnel and expensive equipment in hospitals, managing products in a warehouse or establishing security systems are employing some of these indoor positioning techniques. However, predictions expect the positioning systems linked to personal networks to be the source of the next killer application. The quick spreading of smartphones among young and not so young people (mobile devices equipped with GPS receivers, Internet connection and 3G/4G) encourages companies to develop new applications each day, expecting huge revenues. Although this market is on the rise, bringing new promising opportunities, there are still some open questions and uncertainties about where all this is leading to, which areas are more profitable, or how can the offered services can be improved to succeed in a strongly competitive market.

Cities are becoming bigger, and their public spaces are growing with them: airports, universities and shopping malls are places, where anybody can get lost easily. Thus, an indoor positioning system guiding the users from their current position to a target point within the desired building can reach a high number of potential customers. We asked ourselves whether collecting people's tracks could give an insight to the mobility patterns within the premises the system is deployed at. Our aim was to find significant patterns in those traces which could be used to improve the system: a type of measure-analyze-improve feedback cycle. Our approach consists of, first, analyzing the traces coming from an indoor positioning system deployed in a large university campus. Second, we visualize the routes of the users in a graph, and finally describe our insights. This new analysis can open new working branches. Discover the mobility patterns of the people can bring new services an strategies to attach to the restless partnerships that offer together the personal networks and indoor positioning systems.

1.1 Methodology

The steps to identify and exploit significant patterns from the recorded routes by the campusguiden users across the campus are the following:

- process the dump of requests in order to obtain the routes;
- convert those routes to the kind of data that can be represented as a graph;
- calculate basic metrics and apply network analysis tools to the graph;
- visualize the graph and identify relevant patterns: the most influential nodes/edges functioning as junctions between the different points of interest requested, the contextual clusters and path communities, and the main quantitative properties of the graph as a whole and of the most influential nodes;
- briefly discuss location-based business opportunities in light of the derived insights.

1.2 Outline

Chapter 2 gives the reader a background on the topics we are going to deal with. In addition to place the reader in the proper context, this chapter helps to understand deeper technical concepts that are going to be used afterwards. It starts with indoor positioning systems, focusing the WiFi-based solutions, like the one we work with: Campusguiden. It also describes how to work with geographic information, and ends explaining concepts and metrics of graph theory.

Chapter 3 describes the process we have followed working with the provided data, the methodology. It starts describing the data we have had at our disposal, and then explains in detail the main program code snippets I wrote in order to build a graph from the traces. Once the graph is constructed, we implement a program to infer basic metrics before analyzing it with a visual software.

Chapter 4 gathers the results and interpretations obtained from the analysis. First, it starts with the raw metrics of the graph itself, and, after checking the consistency of the graph, it continues with more complex insights. We interpret the data first at a general campus level, while we give an in-depth analysis at the building level afterwards.

Chapter 5 deals with the economic issues and business potential of location based services. It starts with a look at the state-of-the-art of mobile market, focusing on location-based systems and services. Then it describes some business models to get revenues from mobile applications. Later, it correlates those models with Campusguiden, making preliminary proposals that might help monetizing.

Finally, Chapter 6 summarizes all the efforts of this thesis, and proposes some future lines of work.

Chapter 2 Background

2.1 Positioning systems

The Global Positioning System is the most used outdoor positioning system. Most devices can enable it just adding a simple card for receiving and processing the different GPS signals from the satellites. However, these systems cannot be used for indoor positioning because a line-of-sight in transmission between transmitter and receiver can't be kept. Also walls, building geometry, people mobility or electromagnetic interference from other devices make indoor environments even more complex. The biggest challenges the designers face are multipath and environmental effects, so they have to trade-off between the performance and the complexity of these systems according to the final requirements.

The position of a device could be used for a lot of position-based services, starting from navigation to other more personalized applications. There are several scenarios like the "fitness centre" where position data can bring for a user a wide of personalized services based in his current location through a Personal Network. On this example, deploying an Indoor Positioning System in a fitness center could estimate user's position inside the building. And network-connected fitness machines could use that information to provide a more personal training.

Indoor positioning systems

An Indoor Positioning System continuously and in real-time can determine the position of something or someone in a physical space such as in a hospital, a gymnasium, a school, etc.[VWG+03] An Indoor Positioning System has to estimate a target's position before a determinate time delay in the whole area required to cover. Those relative or absolute updated positions could be also displayed respect to a map of the area, depending on the application.

Location-based Applications
Software-Location Abstractions
Location Sensing Systems

Table 2.1: Location-Aware System Architecture

Architecture

The architecture of any location-aware computing system is shown in Table 2.1. It consists of three main layers. At the location sensing system layer we have to choose the sensing technology to locate the devices of the users. The second layer converts to any required presentation the data provided by the sensor system. Finally the highest layer implements a functionality using the location information measured and calculated by the lower layers.

Localization techniques

Many wireless technologies have been developed for indoor location sensing. These technologies may use IR, ultra-sound, RFID, WLAN, Bluetooth, UWB, etc. Each one has its unique advantages in performance and some limitations at the same time depending on the properties it explodes. An Indoor Positioning System equipped with more than one location technology can improve cost-effectively its performance. Several techniques can be used to locate objects and offer absolute, relative and/or proximity location. The main four are: triangulation, fingerprint, proximity and vision analysis.

Proximity positioning technique can only offer proximity information. In the area where we want the targets to be located a number of detectors have to be fixed at already known positions. We will know that a tracked target is in the proximity area of the sensor that has just detected it. Although this technique cannot give an absolute position, it can specify whether a target is in a room or not.

Fingerprinting positioning technique uses pre-measured location related data. Through two phases, it first creates a fingerprint map to calculate later the current location estimation. In the off-line training phase, we collect location measures in the positioning area. And in the on-line position determination phase, the system compares the just pre-measured data with a database to find a similar case in order to estimate the location.

Triangulation positioning technique is based on the geometric properties of the triangles. If we know the position of three references and the distances or angle from a target to all of them, we can easily calculate that target's absolute position. To calculate the distance between the target and the reference elements in this technique we have to measure either the received signal strength

(RSS), the angle of arrival (AOA) or time of arrival (TOA). It is important to remember that this technique needs to know the position of two (AOA) or three (RSS,TOA) reference elements at least. [LDBL07]

Vision analysis technique estimates a location from the images received by one or multiple check points. We don't need to carry any extra tracking device, just fixing a camera to cover the area. This technique tracks the targets from the real-time received images.

The algorithms are specifically designed by the designers for each technique. They calculate the position of a target object with the data provided by the sensors. The accuracy of the position estimation relies on how correct are the measured data, whether they contain errors or not -and in if it is the case on how big they are.

Characteristics

We can classify the Indoor Positioning Systems according to several criteria, like for example the network infrastructure employed. If the system takes advantage of an existing wireless network infrastructure we have a network-based approach and we don't need any additional hardware infrastructure. This approach is cheaper, but the designers may prefer the higher accuracy or the more freedom of physical specifications that the non-network-based approach offers.
Other criteria is taking a look to the architecture to find where the system actually calculates the position. If the targets themselves calculate their position taking advantage of an existing infrastructure we have a *self-positioning infrastructure*, more private and secure for the users. However, if the infrastructure estimates the targets' position and can automatically track the devices within the covered area we have an *infrastructure positioning architecture.* In these two kind of architectures the positioning measurements start when the targets send requests to the system, and then the targets obtain the location information from the system. Finally, in the *self-oriented infrastructure-assisted architecture* the system needs the the devices allowance to track them, otherwise no positioning activities could be carried out.
The existing systems can be also divided in commercially available and research oriented. The last ones have their specifications detailed in a free way to the common knowledge, in order to achieve future improvements. The first ones keep their working principles secret because of their competition commercial availability.

Some other criteria for classifying these systems could be the physical medium used: radio frequency, ultrasound waves, IR signals, electromagnetic waves or vision-based. [GLN09]

Requirements and specifications

The indoor positioning systems are designed depending always on the client specifications. Like on every technical system there is no a perfect solution, so designers have to trade-off between prize and performance deciding among the existing techniques choosing those that suit most to the desired application.

The users' **security and privacy** can decide whether the current location is estimated in their own devices, allowing or not the system to have access to that information and/or to the history of past activities. Other critical requirement is the **cost** of the system, it can be split in several parts: the cost of the infrastructure components, the prize of each positioning device and the cost of the system installation and maintenance. GPS technique has a large expensive and complex infrastructure to support, meanwhile other systems that reuse WLAN existing infrastructures are more cost-effective. Devices with self-positioning calculation ability offer more privacy, but that improvement is reflected in the cost. They are more expensive and their battery life duration is decreased by the larger and more complex calculations that take place in the device itself. And we have to consider also the **space and time costs**, how much time is going to require the system installation and how much space the infrastructure needs. The devices size is fundamental if the users have to carry them on their daily lives.

To judge the **performance** we evaluate two parameters: accuracy and precision. The accuracy means the average error distance between the estimated position and the real one. The precision is the probability of a successful position estimation respect a pre-defined accuracy. The total delay since the users make a request and they get a response is also a performance measurement. It adds the delay of measuring, position calculations and forwarding the estimation results to the request. The delay can swing if the tracked target moves quickly or the indoor environment changes dynamically.

An Indoor Positioning System **robustness and fault tolerance** relies on its ability to keep on operation even if some components stop working properly or unexpected situations difficult normal behaviour. For example, if the mobile device runs out of battery energy or sensors in a public area are stolen. The system's **complexity** measures the human efforts during the deployment and maintenance of the system itself. For example, some applications may require a fast set-up and an easy software platform for the users. The system's complexity also indicates the computing time it is going to take the device estimating its position. Lower calculations complexity (CPU processing) increases battery power devices' life.

We cannot forget the **user preferences**, because the personal networks are thought to solve their needs. The users' comfort require wireless, small, light-weight and low-power consumer devices, they want also rapid, accurate and real-time positioning services with an easy and friendly interface.

Finally, we should remember that each system although having its own valuable improvements also has some **limitations**. Mostly because of the medium used in the position sensing. For example, WLAN technology reuses an existing infrastructure saving money, but the multipath reflection effects also rise its error range. Other systems may only cover a short range and they are not scalable for larger areas.

When evaluating a positioning system, in addition to find the potential limitations, we have to consider that these systems can influence the performance of other already existing wireless systems in the area.

WiFi-based indoor positioning

Wireless Local Area Network (WLAN) standard allows creating mid-range networks that operate in the industrial, scientific and medical band -the range of the frequency spectrum between 2.4 and 5 GHz. This band can be used for free by everybody so it is a really noisy band, and the most common standard to use here is IEEE 802.11 [Bre97][IEE11].

Nowadays public areas like hospitals, train stations or universities have WLAN networks already deployed for their internal and users behalf. Even shopping malls and coffee shops are starting to offer these free services. WLAN-based positioning systems reuse the existing WLAN infrastructures in indoor environments, cutting down the cost of the services. These systems can be more easily and quickly set-up; in case the objects to locate are equipped with WLAN transceivers we would only have to add a location server. However, movement and orientation of human body, walls, doors, nearby tracked mobile devices and Application Point's overlapping weak the WLAN signal strength, reducing the accuracy of the system. We have to consider also privacy issues because a device with a WLAN interface may be tracked even if the user don't want to.

Novel studies are working on techniques that do not even require any explicit pre-deployment effort in places where almost the whole infrastructure was already set up. For example, configuration-free location scheme like *EZ Localization* algorithm [CPIP10] does not need to create any detailed Radio Frequency map or propagation model based previously on the environment. The only requirements are having enough WiFi Application Points and tracking devices with GPS signal access like common smartphones or netbooks. This technique learns by collecting data from users' mobile devices while they are moving along the area of interest.

2.2 CampusGuiden

CampusGuiden is the multi-platform indoor positioning application we are going to utilize at our case of study Figure 2.1. Wireless Trondheim is a young technology com-

Figure 2.1: CampusGuiden user interface

pany in a rapidly growing industry (http://tradlosetrondheim.no/). CampusGuiden is the result of a research and development project between Wireless Trondheim AS and NTNU and it has provisional status as a beta. This initial version of Campus-Guiden covers only the Gløshaugen campus, but that's just a start. Using a PC, a tablet or a smart phone, the user can search for any room or resource at NTNU campus Gløshaugen and the application provides a navigation tool: CampusGuiden shows the users the way from their current position to any point of interest, such as an auditorium, reading room, bus stop, toilet or the nearest restroom. The routes contain details on buildings, floors and rooms inside the building. This is the first application in the world for indoor location inside an university.

Operation Gløshaugen

Each year about 20000 students come to NTNU. Gløshaugen Figure 2.2 is the main science and technology campus, consisting on 60 buildings with about 13,000 rooms. It covers a 350,000 m2 area, the size of a small town, where both students and visitors often get lost. New students can spend one month or two before they start feeling comfortable with the new area. NTNU student's lectures sometimes take place in auditoriums in different buildings. First days, finding these places can be really confusing, and even afterwards, if suddenly you have to go to a different place than usually. Some students may would like to buy a piece of fruit or a coffee during the break. Others could be really short of time to have lunch or just wondering in what

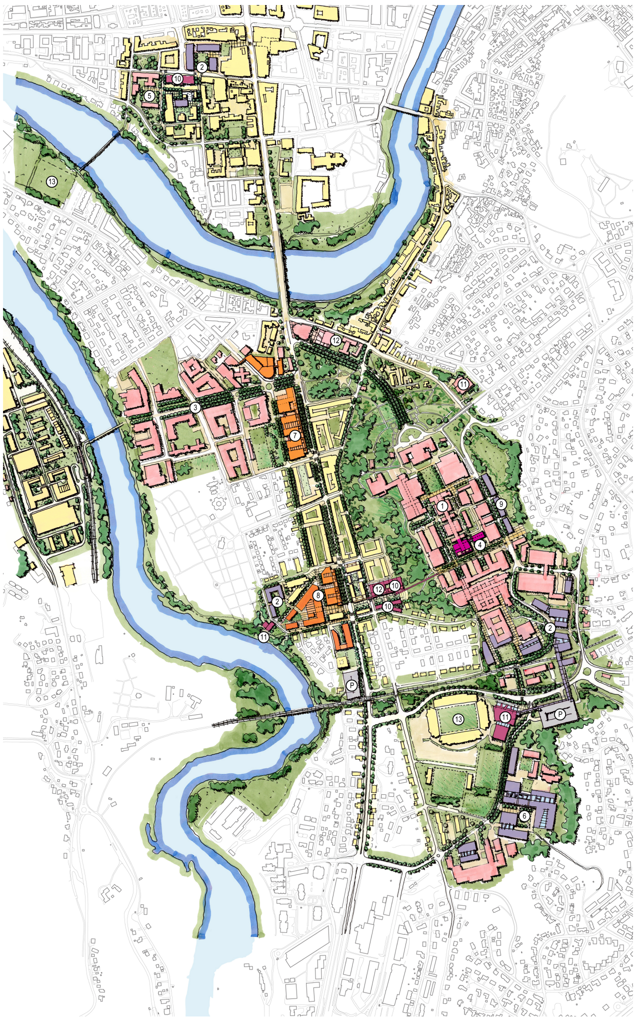


Figure 2.2: Gløshaugen campus overview

cafeteria it is offered their favourite meal. All of them will appreciate an application that not only tells them where the nearest store, cafeteria, auditorium or printing room is; but also leads them to the exactly place through the different buildings if that is the case.

Starting with this application, further information could be given in a future. For example availability of the rooms, gym timetables or cafeteria menus could increase the application popularity. The more students use the application, the bigger data could be collected about their routines and as I will face in this paper: those collected dumps can turn to other interesting results related to people's behaviour offering new services or rethinking the existing ones.

Server-client architecture

This positioning system employs the WiFi network for indoor environments -with an accuracy of up to 5-10 meters- and for outdoor, it uses GPS signals providing roughly the same precision.

CampusGuiden is an umbrella term that includes a central unit (1) and an application client (2) that the end users download and run from their mobile devices Figure 2.3. The central server contains map basis, sketches and all content as names of places or lists of rooms; and it also keeps track of users. On the other side, the user's application list different information: allowing the users to display on their devices just their current position on the digital map, or the route to their target location.

CampusGuiden is implemented as a Java application, and it works integrated with an existing infrastructure. The application server uses the access points in the campus wireless network to determine the current location of the mobile device. CampusGuiden denotes the service as a whole with the two main sections mentioned above Figure 2.3, not a location technology by itself.
CampusGuiden server manages tools and solutions to enable location-based services through the campus network components such as routers and switches. Manufacturers like Cisco supply complete infrastructure for wireless networks including location services. The NTNU campus Gløhaugen has Cisco boxes across the board, and therefore the location server is Cisco equipment [Hal11].
CampusGuiden client is a platform-independent application. The same version can be downloaded and run on a web browser, iPhone or Android. The client proceeds the following location steps:

1. The client alerts the server that it is on-line.

2. The client asks its position.

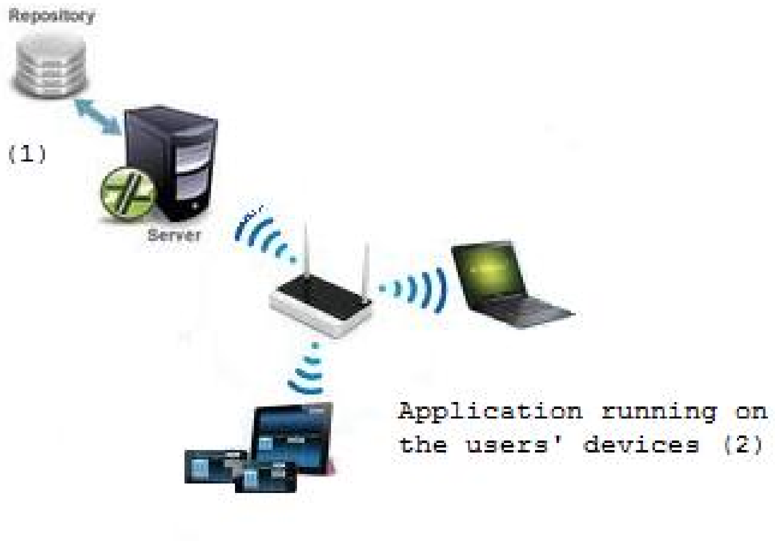


Figure 2.3: CampusGuiden structure: client-server

3. The client allows the server to use the IP address for identification.
4. The server sends a map image to the server. That image is the position that the server has already computed.
5. The client can request more map data to the server, for example when the user uses the zoom.

2.3 Geographic Information Systems

Just as we use a word processor to write and read documents on a computer, we can use a Geographic Information System (GIS) application system to deal with spatial information. These systems are designed to store, manipulate, analyze and present all types of geographical data. And they merge cartography, statistical analysis and database technologies concepts. We can find the origins in the maps drawn of the cholera cases distribution in London: visualizing the cases on the places they emerged led to the source of the disease in 1854, a contaminated water pump.
A GIS consists of digital data, computer software and computer hardware. With a GIS Application you can open digital maps on your computer, create new spatial information to add in them and perform spatial analysis. Generally, organizations use expensive custom-designed GIS, but there are also free open-source packages and distributions like QuantumQGIS.

Data

GIS allow the user to associate non-geographical information with geographical data-places. We can easily change the appearance of a map based on the non-geographical data associated with those places. GIS is a great visualization tool that can show you how your data are related in the space [SDS09]. GIS also work with different types of data:

Vector data

Vector data are stored in series of coordinate pairs. The geographical data are the features, and their related information are the attributes of the features. The features represent discrete real world's features in the GIS environment. We can represent points, lines and areas with these features. A *point* consists of only a single vertex represented in space using an x, y and optionally z axis. When a feature's geometry consist of more than one point, and the first and the last are not the same we have a *polyline* feature. If the feature's first and last point are the same we have a *polygone* feature. The X and Y values will depend on the Coordinate Reference System that is being used. We can use vector data for spatial analysis because the attributes are the same for all the points that shape each single feature.

Raster data

Raster data are stored as a grid of values. The raster images are made up of a matrix of pixels or cells. These pixels contain a value that represents the conditions for the covered area by them. We can represent continuous information across an area that we cannot easily divide into vector features. Satellite images are usually this kind of data. But raster data are also good for representing more abstract ideas, like temperature or fire-risk maps. If we want raster images with higher resolution, we will need larger space to storage them.

Coordinate reference system

People need any representing approach of the earth if they don't want to carry a globe in their pockets, but a flat paper can not perfectly wrap a round object. So a map projection is a way to represent the Earth's curved surface on a flat paper or computer screen. Each coordinate reference system *CRS* defines how the two-dimensional projected map is related to the real places on the earth with the help of coordinates. The choice of each map projection and coordinate reference system depends on the area and the kind of analysis we want to carry on.

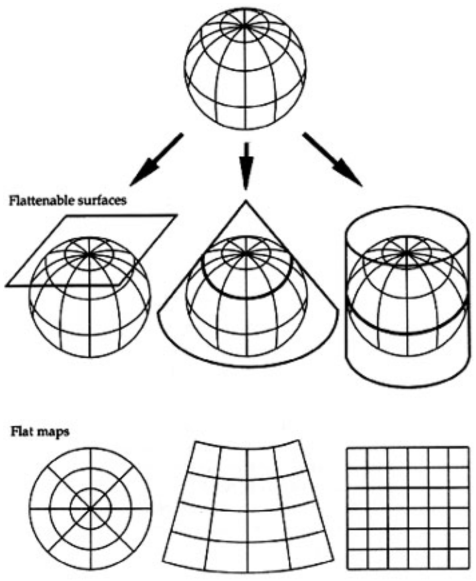


Figure 2.4: The three families of map projections: planar, cylindrical and conical

Map projections

Cartographers have developed a set of techniques called map projections. These map projections show the spherical earth in two-dimensions with reasonable accuracy. At close range, the earth appears to be relatively flat but this changes when we move towards the space. Maps represent the reality, so each map projection has advantages and disadvantages at the same time. Some projections are good for small areas, and others are good for areas with a large either East-West or North-South extension. We can surround the globe with a **cylindrical** shape, a **cone** shape, or even a **flat** surface. Each projection family Figure 2.4 is based on one of the three previous method.

Every map shows angular, distance and area distortions. A map projection can trade-off among these distortions, preserving one specific relationship but deteriorating the others. There are hundreds of different projections. For example, conical projections are commonly used to represent wide areas of the earth like continents or oceans, transversal cylindrical projections are the only ones that represent the equator without distortion. And conforming projections preserve the shape of the projected figure, while equivalent projections preserve the area.

We can specify any place on the earth by a set of three numbers. These three numbers are called coordinates, and their specifications lie on coordinate reference systems. We can split the coordinate reference systems into projected (also called

Cartesian or rectangular) and geographic.

Geographic coordinate systems

These coordinate systems employ latitude and longitude degrees, an height value can be also used. They can describe a location on the earth's surface without any other reference point. The *equator* is the reference line for latitude and the *Greenwich, England Meridian* is the reference line for longitude. The most popular geographic coordinate system is **WGS 84**, used for example by GPS.

Projected coordinate systems

These systems define a two-dimensional plane by two axes at right angles to each other. This plane is called the XY, because we normally label the horizontal axis X and the vertical one Y. An height value can be also used in this reference system, giving the third dimension Z. A popular projected CRS is **UTM**, used in South Africa. This system divides the world in 60 equal zones to avoid too much distortion.

EPSG

The European Petroleum Survey Group identifier include in the own code the two geographic references needed: the map projection and the coordinate reference system or datum. We are going to use the following ones across our study:

EPSG:4326 Datum WGS84 without projection. It describes latitude/longitude coordinates in degrees.

EPSG:900913 Spherical Mercator projection. It describes x/y coordinates in meters. Projection used by Google Maps, Microsoft Virtual Earth, Yahoo Maps, and other commercial API providers.

EPSG:32633 Datum WGS84 with Transversal Mercator projection-UTM zone 33N.

Geographic markup languages

We can easily distinguish what a mark up languages are by their syntactical form. They combine, at the same time, the text with extra information about that text itself. These extra words are called tags, and they contain information about the document structure and presentation. These tags are mixed across the primary text and instruct the reader software to carry out appropriate actions when displaying the document. Mark up languages differ from programming ones not having variables nor arithmetic functionalities.

We have to follow some semantic and structure rules when using these mark up languages, obtaining a richer document. Examples of these language are XML, HTML or LaTeX. But there are also particular specifications of these languages for publishing geographic data.

XML

An eXtensible Markup Language is a markup language that defines a set of rules for encoding documents in a format that is both: human and machine-readable. These languages are also thought to encode a document for data transport and storage. The design goals of XML emphasize simplicity, generality, and usability over the Internet. Most of the other mark up languages like GML or HTML are just derived -XML specifications- thought for concrete applications (geographic:GML or web pages:HTML for the given examples).

GML

Geography Markup Language (GML) is the Open Geospatial Consortium XML standard grammar for expressing spatial feature information. GML serves as a modeling language for geographic systems. This language allows users and developers to describe generic geographic data sets that contain point, lines and polygons. These features are defined by their coordinates and may also have attributes.

As with most XML based grammars, there are two parts to the grammar: the schema that describes the document *(fileName.xsd)* and the instance document that contains the actual data *(fileName.gml)*. A GML document is described using a GML Schema (this schema contain, for example, information about which are the attributes or the coordinate reference system)[ISO07].

KML

Keyhole Markup Language is the spatial XML format used by Google Earth. Google Earth was originally written by a company named “Keyhole”, hence the reference in the name. KML complements GML - whereas GML is a geographic content encoder language for any application, KML is a geographic information visualization language tailored for Google Earth. KML instances can be transformed losslessly to GML, but roughly 90% of GML’s structures cannot be transformed to KML.

GeoJSON

The GEOgraphic JavaScript Object Notation specifies a spatial text format that is very fast to parse in Javascript virtual machines. GeoJSON is another format for encoding geographic data structures. A GeoJSON object may represent a geometry, a feature, or a collection of features. The features contain a geometry object and additional properties, while a feature collection represents a list of features.

CSV

A Comma-Separated Values file stores data in plain-text form (plain text means that any sequence of characters can be interpreted). A CSV file consists of any number of entries typically separated by line breaks, those records usually have the identical sequence of fields. The first line in the CSV file must contain attribute names and single entries the rest ones. The attribute fields of those entries in the CSV file must be separated by a comma, decimal values by a decimal point and text values should be quoted.

If we attach to a CSV an OVF file (OGR Virtual Format file is an XML control file) we can turn both files into any other GIS-datasource language using the tool http://howto.mygeodata.eu. In the OVF file we have to follow a fixed structure to define which attributes -columns- contain the spatial information, which one contains the coordinate reference system or any other attribute we find interesting.

2.4 Graph theory

Nowadays we can have access to any kind of data, comprehending the complex systems around us could lead nowhere without the proper computational tools and models. Large data sets can make sense only by visualizing the information they content; graph drawing allows us to visualize the relationships captured between the objects by a simple graph. Even some elements of graph theory are not concerned with the cartographic characteristics (like length or shape of the edges), other graph theoretic descriptions (like connectivity) are topological invariant of a network and capture the underlying relationships well. We are going to describe several concepts that are highly detailed in the book [EK10].

Terminology

A graph is a non empty finite set of vertices V along with a set E of two-element subsets of V. The elements of V are called vertices and the elements of E are called edges. A graph can be also undirected if each edge is bidirectional between the two nodes or directed if it is not (each edge follows a singular direction -then instead of edges they are called arcs).
The graphs are just points and lines connecting on those points. However, graphs make problems easier to understand, they allow graphic visualization of their content (edges and vertices could represent whatever you want: since villages and roads to neuronal or social networks). Also some graph characteristic measures give more information about the represented data behaviour; nodes or edges where more connections gather or whose disappearance could isolate other components.
Cardinality. The cardinality of a graph represents the number of vertices it has. In the given example Figure 2.5 the cardinality of the graph is six.

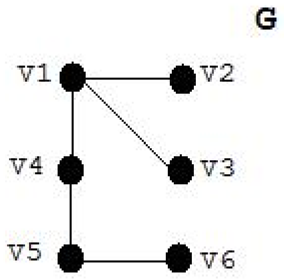


Figure 2.5: An example of a simple graph

. This graph ***G*** has the following vertices V={V1,V2,V3,V4,V5,V6} and edges E={{V1,V2},{V1,V3},{V1,V4},{V4,V5},{V5,V6}}.

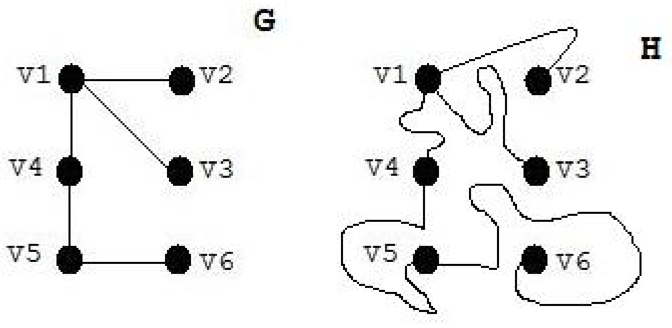


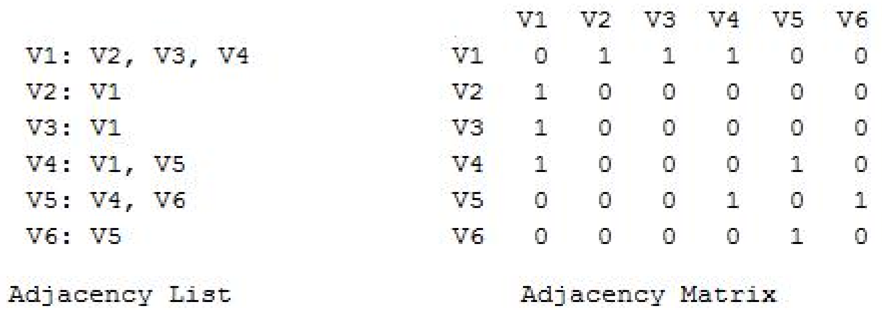
Figure 2.6: An example of two isomorphic graphs.

They both have the edges E={{V1,V2},{V1,V3},{V1,V4},{V4,V5},{V5,V6}} connecting the same nodes, no matter the shape of the edge.

Vertex degree. In undirected graphs like the example Figure 2.5 the degree of a vertex is the number of edges leaving or coming -here it is the same- that particular node, for example deg(V1)=3. In case of having directed graphs, we make a difference between in-degree (edges coming to the interesting node) and out-degree (edges leaving the interesting node). We have to give a look to the arrow of each edge to know whether is coming or leaving.

Isomorphism. Two graphs are isomorphic if they have the same connections between every node, no matter the shape of the edges. For example in the figure Figure 2.6 the graphs G and H are isomorphic. It does not matter the way you draw a graph, what really matter is the information it contains about those connections.

Simple graph. It is a graph where any pair of nodes can not be joined by more



```
V1: V2, V3, V4
V2: V1
V3: V1
V4: V1, V5
V5: V4, V6
V6: V5
```

Adjacency List

	V1	V2	V3	V4	V5	V6
V1	0	1	1	1	0	0
V2	1	0	0	0	0	0
V3	1	0	0	0	0	0
V4	1	0	0	0	1	0
V5	0	0	0	1	0	1
V6	0	0	0	0	1	0

Adjacency Matrix

Figure 2.7: An example of an Adjacency List and an Adjacency Matrix.

than one edge (more than one edge in the same direction for directed graphs). In case of multiple edges between nodes we have a multiple graph.

Path. The path from one source vertex A is the list of distinct vertices and nodes you have to go through to get to another target vertex B. Distinct vertices means that you don't go twice through any of the vertices composing the route. A high number of paths between a pair of nodes can indicate the robustness degree of the system, it is a measure of redundancy.

Circuit or cycle. If the path starts and ends at the same vertex we called that path a circuit.

Loop. A loop is a path leading from one vertex to itself.

Giant component. A component of a graph is a big subset of vertices strongly connected and isolated from the rest of the graph, the giant component is the biggest one. We can easily distinguish them by visualization. A connected graph has only one component, every node can be reach from another one through some path (no matter how long).

Adjacency. The adjacency of a vertex show which are its neighbours, which vertices it is connected to by edges. Adjacency can be written in a list (adjacency list) or in a matrix (adjacency matrix). These are alternative ways of summarize the information content in a graph. For the graph example Figure 2.5 we have been using, the adjacency list and matrix are the followings Figure 2.7. Whether it is a matrix or list, they both have the same edges E={{V1,V2},{V1,V3},{V1,V4},{V4,V5},{V5,V6}} connecting the same nodes.

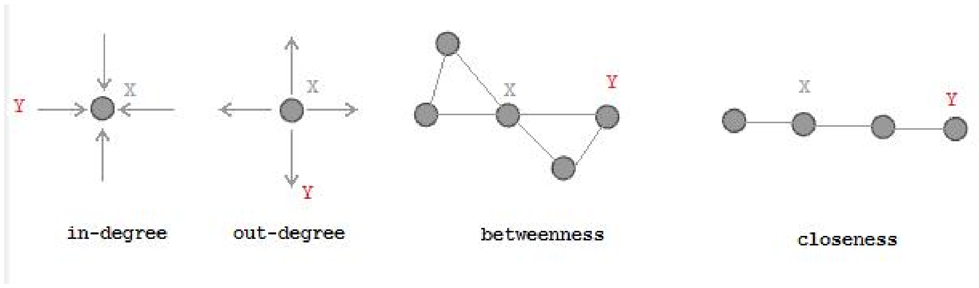


Figure 2.8: Different notions of centrality.

In all the examples node X has more centrality than Y according to the corresponding notion.

Centrality

We can easily see that not all the nodes carry out the same functionality; there are some nodes that removing them would significantly impede the functioning -critical nodes. How can we measure the importance of a node? Which node is more central? Is counting the edges coming or leaving enough? A node can have many edges but still not be in the centre of the picture.

We have different notions of centrality. We can show in the Figure 2.8 some examples where node X will have always more centrality than node Y depending on a different notion each time.

Undirected graphs

Degree. This notion measures the number of edges coming and leaving a node; it gives a natural intuition about how many relations keeps a node, how easily can spread some information or a disease and at the same time how easily can be exposed for example. It can be seen as the opportunity of influence or be influenced directly. Normally you don't expect large hubs in the network: nodes with a lot of edges and others with really a few). However, degree for example does not capture brokerage, what advantages could get a node in better situated despite having less connections?

Betweenness. This notion captures brokerage. A node that occupies such roles is more likely to have advantages: it can set constraints to the other nodes only connected through it. It can be interpreted as a control power over the network, a flow constrictor that keeps different components together. Betweenness can be measured counting how many pairs of vertices would have to go through you in order to reach between them in the minimum number of hopes -in how many shortest paths between pairs of nodes you are.

Closeness. Nevertheless, having many connections or being between many nodes might be not so important. We may be interested in the nodes that are in the "middle" of the others, not too far from the center. Closeness measures how far away the rest of the network is from you, if you have easy access to the whole network.

Eigenvector: Degree measures the number of connections, and closeness the length of shortest path to the rest of the nodes; but your centrality can also be measured depending on how central your neighbours are. This is a recursive definition: you are as important as your neighbours, it is a "popularity" measure. We can measure it with Bonacich's algorithm.

Directed graphs or digraphs

The concepts are almost the same as undirected graphs. Directed graphs characterize for having directed edges, not reciprocal ones. Besides considering directed paths we will have to change the normalization factor (because it is more difficult to get connected all the graph).

The directed closeness measure will split into in-closeness and out-closeness; and so the connection degree (in-degree and out-degree), both for all the nodes shaping the directed graph. A high in-degree indicates that a vertex (the item represented) is influenced by a large number of vertices (other items of the system). While a high out-degree indicates how much influence has that vertex along the others it is connected to.

The eigenvector measure now will be called PageRank thank to the algorithm that avoids the "drunk problem" in directed graphs -this algorithm allows some random 'teleportation' when you get trapped into circles (since you have to follow directions when choosing the edges, it is possible to get trapped into circles; so compute the eigenvector will result impossible if you are not able to escape). Betweenness remains the same.

Centrality applied to spatial networks

The measures developed for network analysis are more focused on social or communications rather than spatial networks. However, they are still consistent with urban theory. A degree-based centrality concept can be used when the number of interactions with other points is more important than their characteristics. A betweenness-based conception adjudges a greater influence mediating in the transactions, seen as a control power. Closeness-based centrality looks for scope interaction, seen as access to more resources. And competitive distance conception gives a higher value to a vertex with access to more vertices with few other connections, seen as a mixture between closeness and betweenness -avoiding competitors. We can find more detailed descriptions and examples in [IH92]

Community structure

Graphs can depict networks whit some regions where the nodes are interacting within that structure more than they are with the rest of the network. Obtaining a clear image from the hairball the edges shape provides information about what is actually going on in the network. We can guess the structure of collaboration among the nodes if we highlight the group of nodes that tend to communicate more between each other.

In our case, detecting the different communities can picture accurately the current user movement patterns across the campus; showing the points that bridge those communities. We can use a social network analysis idea: each node tends to adopt the majority opinion of its neighbours. So if those bridge-points exist, they will play a key role in the community structure and opinion formation -they are the 'door' to spread some information easily in that community.

Finding communities

The first criteria we can follow to decide what makes a community is looking for completely connected subgraphs. Every node is connected to everyone else within a community by an edge: *clique*. However, one missing link can disable a clique, and we want get a densely connected core and more peripheral nodes.

A similar criteria but less stringent is looking for nodes that at least are connected with k others within the group: *k-core*. But still remains the same problem, leaving nodes from the natural community outside because they don't rise the k links. This problem can trouble the natural communities identification.

The other criteria we can follow is based on closeness, diameter or reachability of subgroup members. If we are more interested on the information flow in the network -like our case- we can determinate how many hops would take any node to reach everyone else, the maximal distance. We look for communities where information can potentially flow, even if the all pair of nodes are not connected between each other.

But still we can have the stringent problem, leaving out some nodes with bigger diameter. To solve that we can split the network into clusters where the nodes have at least a proportion p (number between zero and one) of neighbours inside the cluster.

With directed and weighted networks, like the one we will work with, we can also use other kind of criteria. We can use the weight to filter members of the same community, because they are more likely to have stronger connections between each other (higher weight). Without any algorithm, just threshold the weights we can at list draw the strongest links and community structures.

Some approaches like hierarchical clustering, betweenness clustering help to discover community structures in large networks in an automated way [CNM04].

Other metrics

Reachability and connectedness. Reachability means whether a path exist between two selected vertices, it is common to reflect in a matrix the reachability between all the vertices: the reachability matrix. With that matrix we can calculate the connectedness, a concept that gives us an idea from how strong/weak are the vertices connected in the graph. We can figure by connectedness how easy some information, a subject or a disease can spread across a graph (indeed the real feature that could be representing as a country or a social network).

Distance. We can store in a matrix the minimum length in number of jumps between every pair of vertices. If there is no path between a pair of numbers that distance is zero. This distance does not make reference to any real metric spaces (the length of the paths, just the number of jumps). The distance matrix can give us an idea of "directness", which pairs of elements of the graph can interact between themselves.

Average shortest path length. The mean value among all the distances between all vertex pairs.

Eccentricity. The eccentricity of a vertex is the maximum of all its distances from itself to the other reachable vertices.

Radius. The radius of a graph is the minimum eccentricity of all its vertices.

Diameter. The diameter of a graph is the maximum eccentricity of all its vertices.

Cluster index. It indicates how many connections are maintained between a vertex’s neighbours. For digraphs, neighbours are all the connected vertex (no matter whether incoming or outgoing). This indicator shows how the neighbours of a vertex also are connected between each other, it captures the local connectivity.

Disjoint paths. Two paths between the same end points are disjoint as long as they don’t share any vertex or edge. It doesn’t matter if they haven’t got the same length.

Cut-vertex and bridges: It is a vertex that increases the number of graph's components if it is removed. The removal of that cut-vertex disconnects subsets of vertices. If it happens with an edge instead of vertex, that edge is called a bridge.

Chapter 3

Dataset and Processing

3.1 Traces

The main goal we pursue is getting a weighted-graph with the data at our disposal. The edges of this graph will be small areas of the Gløshaugen campus: the corridors, stairs and rooms that the users of CampusGuiden have been following during the past months. These edges have to be as much small as possible (that way we rise the resolution, allowing us to get more precise information of the real popular areas). The weight of the edges will be assigned according to the frequency they have been used.

3.1.1 Data files

We were provided with a file called *collection.zip* containing stored the data we needed. The zip file contained several files, including a dump of the CampusGuiden traces -the base of our study- where we could see the current users' position at the time the requests were made and their desired destinations across the campus. Among the different files, we found useful the following ones:

openstreetmap.gml: A Gløshaugen and surroundings map in GML format.

trackposition-31072012.csv: A recent dump done on 31st of July,2012. This file contain the traces of the requests the users made and it is the most important file for our study case.

Paths: *path_dump.gml, dump.xsd*
These files contain the segments that shape the paths that can be followed across the campus between each places.

Points of Interest: *poi_dump.gml, poi_dump.xsd*
These points of interest are the nodes the users are going to ask for (their ids are the ones requested by the users in the searches contained in trackposition-31072012.csv).

We remind the reader the information already given in Section 2.3 about geographic markup language: GML files can be loaded in most geographic information systems (obtaining a great visualization of the data), and XSD files are just schema files for the GML (they contain the attribute descriptors).

3.1.2 Data verification

trackposition-31072012.csv

This file contains a dump of the requests the users made to the system. Lets take a view of the three kind of data we are going to find in this file.

"time", "lon", "lat", "z", "q", "tonodeid", "qtype"
"2012-07-30 15:38:53", "10.402316717046666", "63.41781227348396", "1.0", "10A Vestre Gløshaugen (Internasjonalt hus)", "39488", "search"
"2012-07-30 15:38:12", "10.402316717046904", "63.41781227348449", "1.0", ""x":51.721512000000004,"y":20.022312,"longitude":7032908.0,"latitude": 570009.0,"geoLongitude":10.402316717046904,"geoLatitude":63.4178122734 8449,"changedOn":"1343655492386","floorId":94.0,"z":1.0, "confidencefactor": 9.7536, "elem":"Gloshaugen|Elektroblokk E|1. etasje|", "error":""","","geopos"
"2012-07-23 01:10:24", "10.403781431671758", "63.41726976443231", "2.0", "Toalett", "null", "objectsearch"

Table 3.1: The three type of requests that can be made to the system: position, search or objectSearch. The seven fields of each request are separated by commas, the last field represents the kind of request.

The first four attributes of each row Table 3.1 follow the same structure for each kind of request: *time*-time stamp, *lon*-longitude, *lat*-latitude and *z*-floor. The longitude and the latitude coordinates are in the EPSG:4326/WGS84 spatial reference Section 2.3. And the *z* field indicates the floor (1,1.5,4) inside the building, saving the '0' value for outdoor locations.

The last three attributes depend on the kind of request the system receives, we can easily identify the kind of request looking the last field of each request: *qtype*. The differences between these attributes are explained in Table 3.2.

The routing requests are the ones we are going to work with. They contain the destinations and the places where the users made the requests: the starting and ending points of the presumed followed routes. These are the routes we will try to split into edges to create the weighted graph. First, to obtain each route we need to access to a web server, so we will have to create a program to automatically do that

Type of Request	**qtype**	**q**	**tonodeid**
Position request	geopos	This field contains a raw dump of position information: the *x, y* are the coordinates within the floor; the *longitude, latitude* are coordinates in EPSG:32633; the *geoLongitude, geoLatitude* are coordinates in EPSG:4326; *floorId* is internal to the positioning server; *z* is the floor identification; the *confidencefactor* is the 95% certainty box for the position and the *elem* field is the floor name internally in the positioning system.	empty
Object search	objectsearch	This field carries now the object the user wanted to find: Lesesal, Datasal, Toalett, Parkeringsplass, Bussholdeplass, etc.	empty
Routing request	search	Destination name	Point of interest id of that destination

Table 3.2: Different fields of the three type of requests that can be made to the system.

for every request. In the next Section 3.2 we will face this task among other further ones.

path_dump.gml

All the possible paths that can be followed to get anywhere inside the campus are drawn in this file. These paths are split in small GML segments (shaped each of these lines only two points). This division between the paths into couples of points is the smallest we can get, so if we are trying to draw the most accurate graph the result should look like this file. As Figure 3.1 shows, these small lines together shape perfectly the campus pattern. The spatial reference system used for these GML file is EPSG:4326, this information is relevant because if we want to draw the same figure with data in other reference system, both figures won't perfectly match: they will just kind of look like. We will just look for a similar weighted graph: small segments with their periodicity as an attribute no matter the coordinate reference system.

3.2 Data processing

When we inspected the dump of files we used QuantumGIS to visualize the geographic files (a free distribution software for geographic information systems). This program comes with a plug-in called GRASS (Geographic Resources Analysis Support System) that allows the user to load raster and vector data, containing also a lot of powerful tools to manage and analyse those geographic data. However, it is quite complicated,

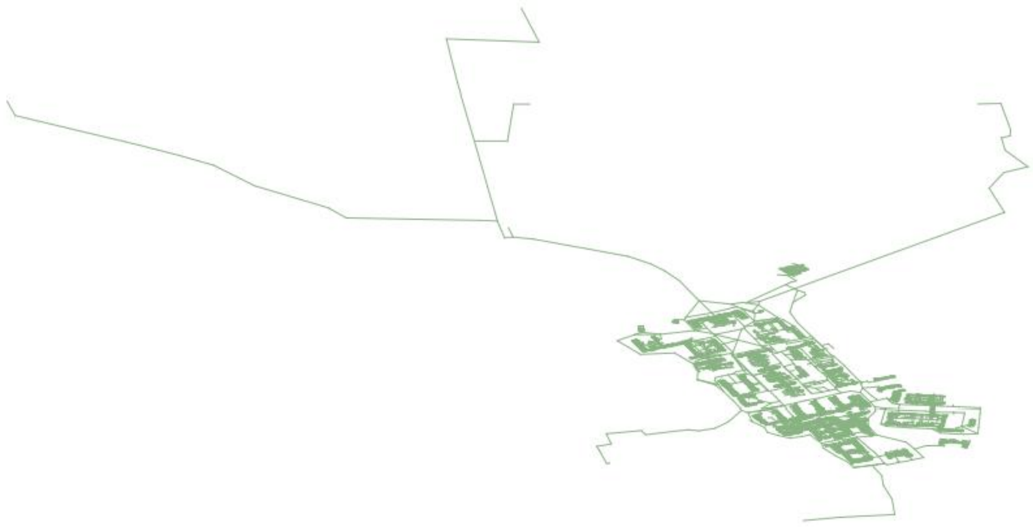


Figure 3.1: Possible paths to follow in Gløshaugen Campus (GML features with EPSG:4326).

a lot of documentation exists; but still it would have taken us really a lot of time to control it. That is why after reading about it we chose learning python in order to write our own applications. It was a good choice, because we were able to design our own python programs just for the exactly tasks we had to face.

Python is a great free-distribution object-oriented, interpreted, and interactive programming language. It is a popular high level language (independent of the machine you are running the program on) easy to learn and use (you don't need to have a deep knowledge about it to just make simple programs with concrete features, and you can always write a small piece of code and see that happens -without needing a debugger or a compiler). Python combines remarkable power with very clear syntax. It has modules, libraries, classes, exceptions and dynamic typing that can be easily documented in the internet. We can find all the libraries we need in this website *http://www.lfd.uci.edu/ gohlke/pythonlibs/.*

The main purpose of this section is obtaining a weighted graph from the file *trackposition-310720012.csv.* The edges of the graph should be the smallest paths we could split the campus into, and their weight will be be the number of times they have been used according to the requests made to the system. These segments

-graph edges- will be obtained from the actual paths that the application tells the people to follow.

We will have then to write the proper queries and send them to the web server to obtain the routes that were followed, extracting the fields to fill the queries from the trackposition file. Once we have those routes we will have to store and split them into the smallest paths we can. And finally, after creating those edges, we will count the number of times each edge is repeated in the file and an we will attach that number as an attribute: its weight.

We will follow these steps along several python programmes we wrote for these tasks, preferring to process the data in several modules (instead of just one) based on debugging reasons and to also facility future further work.

3.2.1 readTrackposition.py

To execute this program we have to type *$ python readTrackposition.py filename*, where filename will be in our case trackposition-31072012.csv. This file we want to read is the one that contains all the system requests. But each row of this file contains a lot of symbols separating each field (" ' , ;), so first we will edit the file making it simpler with the help of a free distributed editor like *notepad++*. The goal is getting each field of every row separated only by a semicolon like this *time;lon;lat;z;q;tonodeid;qtype*. These are the format changes to make:
- Find and delete: *","error":""* and *"floorId":*.
- Find *","* and replace it by *;*.
- Find and delete: *" { }*.
- Find and delete the regular expression: *[A-Za-z]+:*
- Insert an empty newline after the last item.

With these transformations we don't loose information and the file is more friendly to read by python: we can store in a python list every row separating their fields by a semicolon. We need to create that list where we could reach easily each item's attributes, because we will need access to them later for creating the web server queries.
This program also prints in three different files each kind of request: *geopos.txt; search.txt and objectsearch.txt*, so we will be able to select which one we want to keep working with. In addition we will count the times every target is requested, including that information at the end of the file. We will also store in a different file the number of times each word within every target's name is repeated for a posterior analysis of the popularity of the rooms and buildings.

*https://app.campusguiden.no/routing?type=shortestpath&**source__lon**=10.4061 &**source__lat**=63.4148&**source__z**=1.0&**source__proj__type**=ll4326&**target __poi**=36396&**guid**=test*

Table 3.3: URL grammar for accessing to the web server in order to obtain a route between two points.

3.2.2 writeQuerys.py

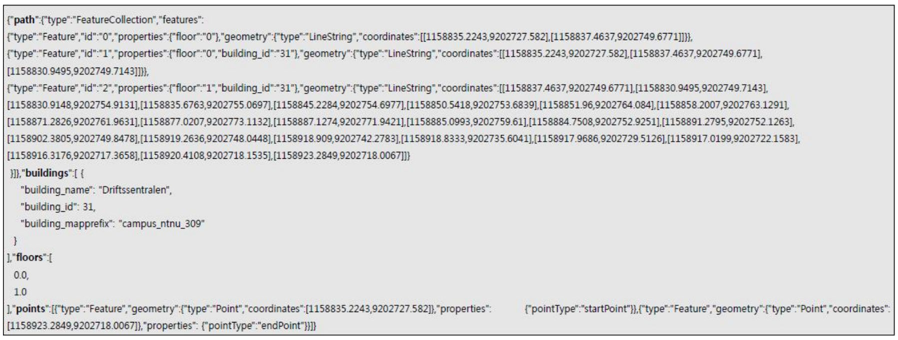
To execute this program we have to type *$ python writeQuerys.py filename*, where filename will be in our case *search.txt*. This file is the one we want to extract the requests from, but before executing the program we have to delete from the *search.txt* file the following symbols: *' [ ]*. We can use again the free distribution program *Notepad++* to do that.

With the writeQuerys.py python program we will create the grammatically proper queries to send them to the web server and saving the resultant paths (based on the search requests). The program starts reading the search requests from an input file and then it creates the queries to the web server following the proper grammar. Finally it connects to the server sending each query and it saves the resulting *geoJSON* data. These resulting data are geographic JSON Section 2.3 strings that contain among other fields the route the users should have followed to get to the desired points of interest from their past current positions.

The program also stores the queries in a file named *requests.txt* and the resultant data received from the web server in a file named *pathsGeoJSON.geojson*.

Query grammar

A route between two points can be calculated by accessing an url shown in Table 3.3. The starting and ending points can be defined either by their poi_id (point of interest identifier) or by their coordinates (longitude, latitude and floor). The parameters to define the endpoints in the queries Table 3.3 will be extracted from the search requests we have, and they are:
Source: source_poi or source_lat, source_lon, z and source_proj_type.
When we define the start point by its coordinates (like our case), we have to write also the reference system used: either EPSG:900913 or EPSG:4326. The *source_proj_ype* parameter will take the value ll900913 for EPSG:900913 or ll4326 for EPSG:4326. We are going to use the second system -EPSG:4326- to express the start point longitude and latitude because that is the reference system utilized in the search requests to write the users' past location.
Target: target_poi or target_lat, target_lon, z and target_proj_type.
When we define the target point by its coordinates we have to write also the reference



```
{"path":{"type":"FeatureCollection","features":
{"type":"Feature","id":"0","properties":{"floor":"0"},"geometry":{"type":"LineString","coordinates":[[1158835.2243,9202727.582],[1158837.4637,9202749.6771]]}},
{"type":"Feature","id":"1","properties":{"floor":"0","building_id":"31"},"geometry":{"type":"LineString","coordinates":[[1158835.2243,9202727.582],[1158837.4637,9202749.6771],
[1158830.9495,9202749.7143]]}},
{"type":"Feature","id":"2","properties":{"floor":"1","building_id":"31"},"geometry":{"type":"LineString","coordinates":[[1158837.4637,9202749.6771],[1158830.9495,9202749.7143],
[1158830.9148,9202754.9131],[1158835.6763,9202755.0697],[1158845.2284,9202754.6977],[1158850.5418,9202753.6839],[1158851.96,9202764.084],[1158858.2007,9202763.1291],
[1158871.2826,9202761.9631],[1158877.0207,9202773.1132],[1158887.1274,9202771.9421],[1158885.0993,9202759.61],[1158884.7508,9202752.9251],[1158891.2795,9202752.1263],
[1158902.3805,9202749.8478],[1158919.2636,9202748.0448],[1158918.909,9202742.2783],[1158918.8333,9202735.6041],[1158917.9686,9202729.5126],[1158917.0199,9202722.1583],
[1158916.3176,9202717.3658],[1158920.4108,9202718.1535],[1158923.2849,9202718.0067]]}
 }]},"buildings":[ {
   "building_name": "Driftssentralen",
   "building_id": 31,
   "building_mapprefix": "campus_ntnu_309"
  }
],"floors":[
  0.0,
  1.0
],"points":[{"type":"Feature","geometry":{"type":"Point","coordinates":[1158835.2243,9202727.582]},"properties":      {"pointType":"startPoint"}},{"type":"Feature","geometry":{"type":"Point","coordinates":
[1158923.2849,9202718.0067]},"properties": {"pointType":"endPoint"}}]}
```

Figure 3.2: Returned JSON string after accessing the web server by the proper query.

system (same procedure just explained for the source point). In our case we will define the target by its own point of interest identifier, because that is the way they are defined in the search requests we have.
The *guid* field is an unique identifier that might come back in the returned data. Keeping it consistent can help to filter easily the routing requests, but we are not using it right now.

Returned data

The returned data is a Geographic JavaScript Object Notation string containing:

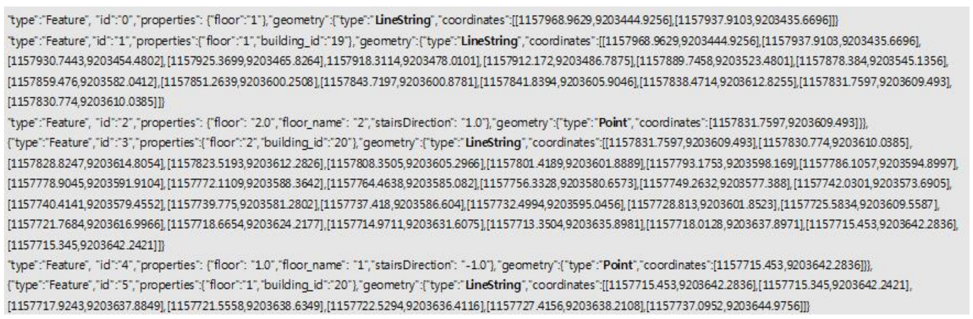
Path: GeoJSON feature set describing the route. The segments are grouped by floors, and each segment has the floor it belongs as a property.

Buildings: List of the buildings visited on the route.

Floors: List of the floors visited on the route.

Points: Start and end point of the route.

We will focus on the **path** strings from these data: they contain the routes that were followed by the users in a geographic JavaScript format. These features contain the segments that will become the edges of our weighted graph, so in the next program we will face how to isolate these segments from the resultant *pathsGeoJSON.geojson* file (this file contains every returned string from the web server after sending all the search requests).



```
"type":"Feature", "id":"0","properties": {"floor":"1"},"geometry":{"type":"LineString","coordinates":[[1157968.9629,9203444.9256],[1157937.9103,9203435.6696]]}
"type":"Feature","id":"1","properties":{"floor":"1","building_id":"19"},"geometry":{"type":"LineString","coordinates":[[1157968.9629,9203444.9256],[1157937.9103,9203435.6696],
[1157930.7443,9203454.4802],[1157925.3699,9203465.8264],1157918.3114,9203478.0101],[1157912.172,9203486.7875],[1157889.7458,9203523.4801],[1157878.384,9203545.1356],
[1157859.476,9203582.0412],[1157851.2639,9203600.2508],[1157843.7197,9203600.8781],[1157841.8394,9203605.9046],[1157838.4714,9203612.8255],[1157831.7597,9203609.493],
[1157830.774,9203610.0385]]}
"type":"Feature", "id":"2","properties": {"floor": "2.0","floor_name": "2","stairsDirection": "1.0"},"geometry":{"type":"Point","coordinates":[1157831.7597,9203609.493]}},
{"type":"Feature","id":"3","properties":{"floor":"2","building_id":"20"},"geometry":{"type":"LineString","coordinates":[[1157831.7597,9203609.493],[1157830.774,9203610.0385],
[1157828.8247,9203614.8054],[1157823.5193,9203612.2826],[1157808.3505,9203605.2966],[1157801.4189,9203601.8889],[1157793.1753,9203598.169],[1157786.1057,9203594.8997],
[1157778.9045,9203591.9104],[1157772.1109,9203588.3642],[1157764.4638,9203585.082],[1157756.3328,9203580.6573],[1157749.2632,9203577.388],[1157742.0301,9203573.6905],
[1157740.4141,9203579.4552],[1157739.775,9203581.2802],[1157737.418,9203586.604],[1157732.4994,9203595.0456],[1157728.813,9203601.8523],[1157725.5834,9203609.5587],
[1157721.7684,9203616.9966],[1157718.6654,9203624.2177],[1157714.9711,9203631.6075],[1157713.3504,9203635.8981],[1157718.0128,9203637.8971],[1157715.453,9203642.2836],
[1157715.345,9203642.2421]]}
"type":"Feature", "id":"4","properties": {"floor": "1.0","floor_name": "1","stairsDirection": "-1.0"},"geometry":{"type":"Point","coordinates":[1157715.453,9203642.2836]}},
{"type":"Feature","id":"5","properties":{"floor":"1","building_id":"20"},"geometry":{"type":"LineString","coordinates":[[1157715.453,9203642.2836],[1157715.345,9203642.2421],
[1157717.9243,9203637.8849],[1157721.5558,9203638.6349],[1157722.5294,9203636.4116],[1157727.4156,9203638.2108],[1157737.0952,9203644.9756]]}
```

Figure 3.3: Example of the features composing a route.

We highlight the distribution between lines and points (points represent the floor level change between lines).

3.2.3 createGraphDictionary.py

To execute this program we have to type *$ python createGraphDictionary.py paths-GeoJSON.geojson*. The last item is the name of the file we want to extract the edges from. That file contains the returned data after sending the queries to the web server with the previous program. The goal now is obtaining a list of segments -couples of points represented by their coordinates- from the paths the application told the users to follow in their search requests. In order to achieve this, we will split the given paths in the routes into the smallest units we can: couples of points.

We can find two types of features among the routes: lines and points. Each line means the path to be followed across the same floor, and it is represented by a variable amount of points ([longitude latitude] couples) that have in common some attributes like the floor and building id. Each point means a change of level (a kind of stair or elevator) and it is represented just by its coordinates, the number of the current floor and an attribute called *stairsDirection*. That last attribute shows the action taken to get to the current floor: for example '-1' means we had to descend one floor to get to the current floor (we came from one higher floor level), or '2' means we have gone up two floors to get to the current floor. There also half floors in some buildings.

When we have to change the floor between two routes, the first route will have the start point of the stairs as the ending point of its path, and the second one will have the ending point of the stairs as the starting point of its path. Besides the path, the second route will also start having the point feature that shows the change of floor. The next Figure 3.3 show an example of these data we are going to work with.

We first extract these path features" from all the JavaScript returned data. Right

now we are only interested on the path features because they shape the routes to follow. We will process these selected information line by line because each line contains either one path(line feature) or one path(line feature) plus one stair (point feature).
Within each line we start creating the small segments that would conform our graph: we split each line feature into couples of points (every line string feature has at least two points), and with these points we create a segments with the following format: *('x1','y1','z1','x2','y2','z2','buildingId').* All the points inside the same linestring feature share the floor and the building identifier.

The special case comes with the stairs. When we have a point feature before a line in the same line, it means that the route in that floor starts at that point coming from another floor. So we will create new segments -keeping the format- from these singular points to save the stairs. This kind of segments will have as starting point the last point we would have at that moment (belonging to the previous floor route, where the stairs started) and the ending point will be the stairs coordinates (where the route starts in the current floor). We can easily identify these stairs representing segments because their edge points have different floor number.

To be more accurate in our future study we also split these stair-segments into the smallest ones we can with the data we have. For doing that we first identify the buildings that have half floors and we separate the stairs in two different lists based on whether the building has half floors or not. Then we try to split each stair segment that has a difference between the two floors shaping the stair higher than one, or a half in case of the buildings with half floors. We first try to split a stair floor by floor, if we don't success then we try to split it using any intermediate floor, but if we don't have information enough about the point coordinates of the intermediate floors we have to keep the segments as they are without splitting them.

Finally with all the segments we have created (from the line routes and the stairs) we create a python dictionary and we store there every segment that is not still inside, adding them the counting with the times they appear. These segments with the number of repetitions will be the edges of our weighted graph, and we store them in a file called *edgesWeighted.txt.* We have to remember that the graph will be a directed graph, the segment between A and B is different that the one between B and A, so we will refer to the edges as arcs since now.

We can print more files in the program if we want, for example with all the segments without their weight, a list of the stairs or just the ones we were not able to split. The reader could find these extra files interesting for a further analysis or just debugging.

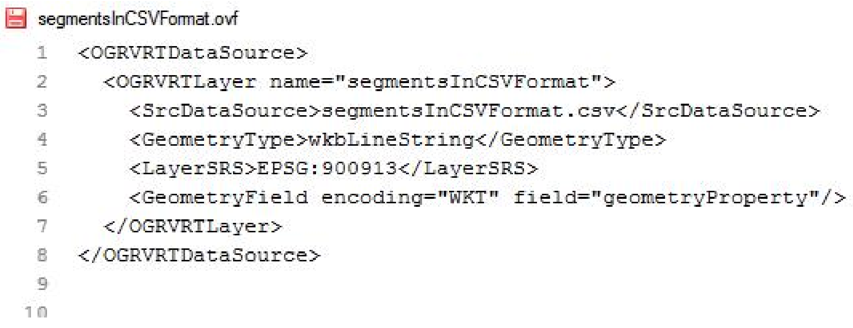
3.2.4 createCSVFile.py

To execute this program we have to type *$ python createCSVFile.py edgesWeighted.txt*, where the last item is the name of the file we want to re-write in a comma separated value format (CSV-files might be easily converted to any other geographic language or reference system as we saw in Section 2.3). But before executing this program, we have to delete all the brackets within the file with the help of *notepad++* to make it easier to read by python.

We have already the segments obtained from the routes the users requested to CampusGuiden. These segments are just couples of coordinates, with some attributes like their weight or the identifier of the building they belong to. With this program we will write these data in a comma separated value in order to convert them afterwards to GML format. The aim of having these segments written in GML format is visualizing them with QuantumGIS, because if the segments we have extracted from the routes shape a figure similar to the one we draw from the paths_dump.gml it would mean that our work has been well done. Then our segments -the future arcs of our graph- will be actually corresponding to the existing pathways, and our information about how much they have been used will be right.

3.2.5 Visual inspection

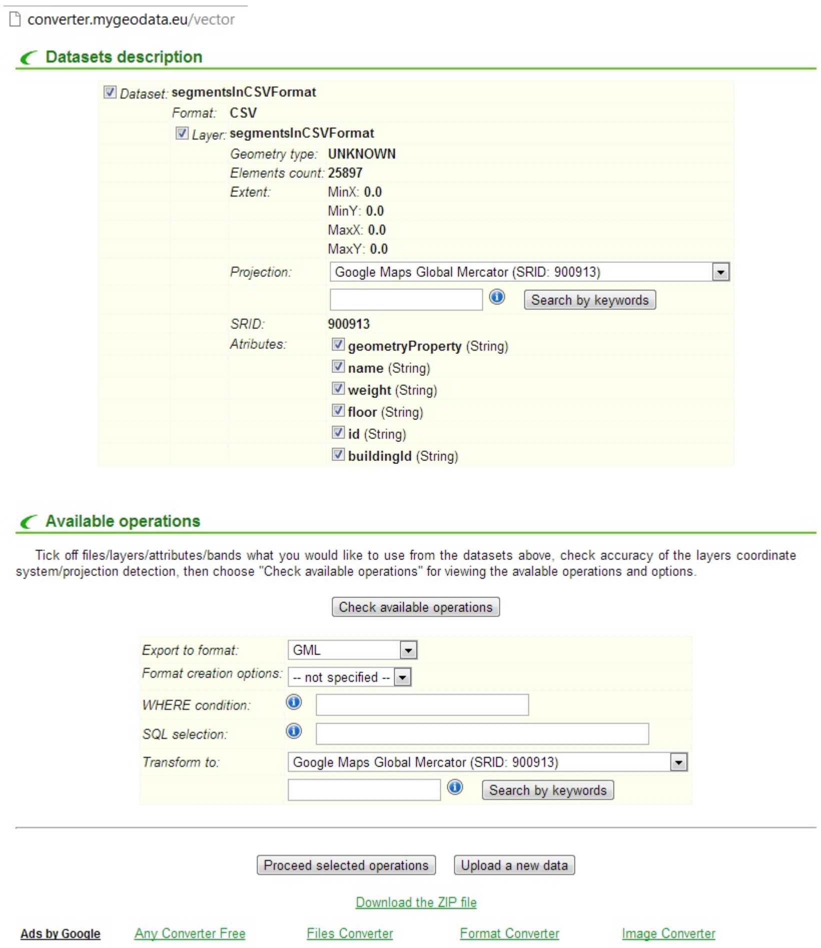
We can check if our work is being well conducted by visualizing the segments according to their coordinates, and as we saw in the previous paragraph, we can use GML format for that. First, after executing the createCSVFile.py program, we have to create the corresponding ovf file manually (an XML control file for the CSV file we just got) with the following text in Figure 3.4: Then we have to save both


segmentsInCSVFormat.ovf

```
<OGRVRTDataSource>
  <OGRVRTLayer name="segmentsInCSVFormat">
    <SrcDataSource>segmentsInCSVFormat.csv</SrcDataSource>
    <GeometryType>wkbLineString</GeometryType>
    <LayerSRS>EPSG:900913</LayerSRS>
    <GeometryField encoding="WKT" field="geometryProperty"/>
  </OGRVRTLayer>
</OGRVRTDataSource>
```

Figure 3.4: Text to write in the .ovf file Section 2.3

files with the same filename (the ones with .csv and .ovf extensions). Next step is comprising in a new zip file both files and we send the zip file to this web page


converter.mygeodata.eu/vector

Datasets description

Dataset:	segmentsInCSVFormat
Format:	CSV
Layer:	segmentsInCSVFormat
Geometry type:	UNKNOWN
Elements count:	25897
Extent:	MinX: 0.0
	MinY: 0.0
	MaxX: 0.0
	MaxY: 0.0
Projection:	Google Maps Global Mercator (SRID: 900913)
SRID:	900913
Atributes:	✓ geometryProperty (String)
	✓ name (String)
	✓ weight (String)
	✓ floor (String)
	✓ id (String)
	✓ buildingId (String)

Available operations

Tick off files/layers/attributes/bands what you would like to use from the datasets above, check accuracy of the layers coordinate system/projection detection, then choose "Check available operations" for viewing the avalable operations and options.

Export to format:	GML
Format creation options:	-- not specified --
WHERE condition:	
SQL selection:	
Transform to:	Google Maps Global Mercator (SRID: 900913)

Figure 3.5: Web server where we can turn a csv file into any other GIS format

http://converter.mygeodata.eu/ Figure 3.5. This website allow us to choose which GIS/CAD format we want to convert our data, and we will choose GML. Then we can download a comprised file from the web that contains a GML and XSD files with our data in the chosen format (GML in this case).

However, we have still to edit a small detail in the GML file after downloading it from http://converter.mygeodata.eu/. With the help of notepad++ we have to find and replace within the file the following strings: this *<geometryProperty>* by this *<geometryProperty><gml:LineString><gml:coordinates>*, and this *</geometryProperty>* by this other one *</gml:coordinates></gml:LineString></geometryProperty>*. Now we can open the GML file with QuantumGIS, obtaining the image shown in Figure 3.6. We can see the results were the expected ones. The differences between this image and the one we obtained before from the dump (path_dump.gml: Figure 3.1) are debt to the reference systems used (details already given in Section 2.3).

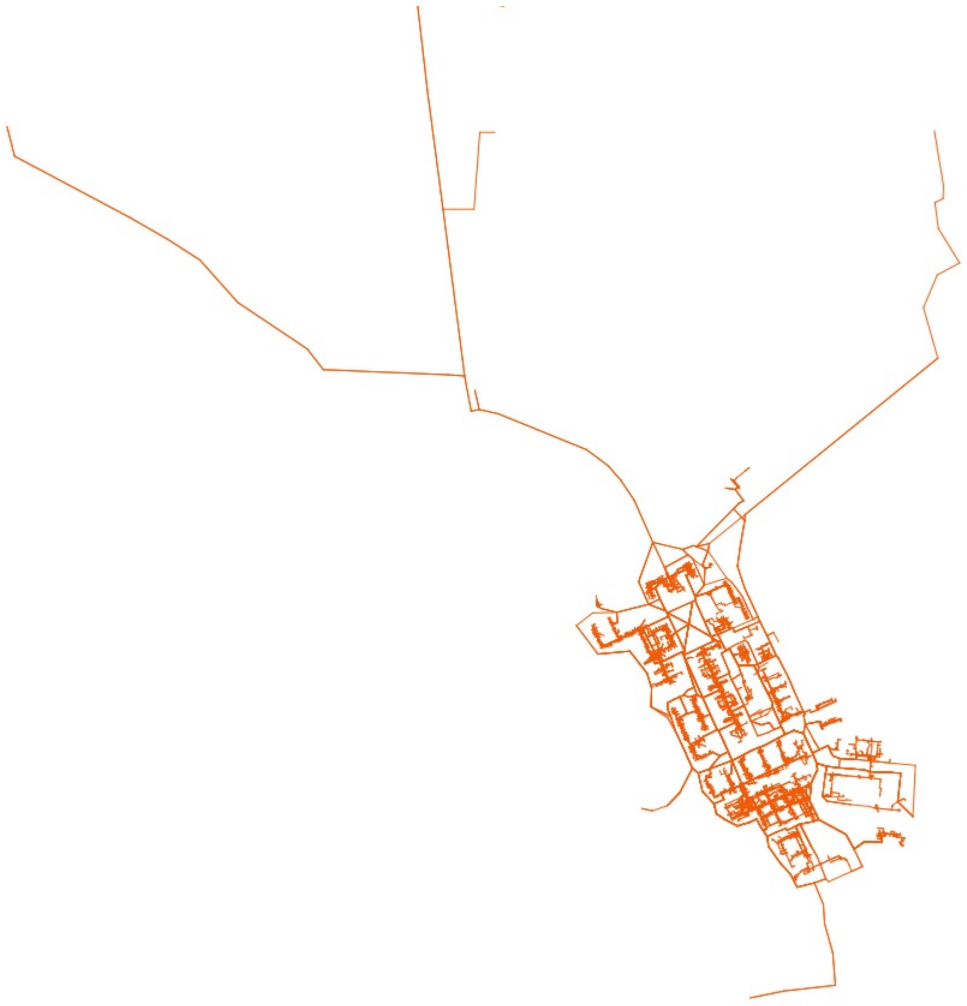


Figure 3.6: Possible paths to follow in Gløshaugen Campus(GML features with EPSG:900913)

3.3 Data Analysis

The main goal we pursued was creating a directed weighted graph from the given traces. We have obtained the segments splitting the requested routes into couples of points defined by their coordinates (x,y,z), adding as another attribute -weight- the number of times we have found them. Those segments that shape that graph cover the minimum possible area, trying to bring the highest resolution for our study (the smaller the segments are, the better we can know which are exactly the most popular ones).

We can assure we have a correct graph because the GML visualization of the edges matches with the available paths provided with the dump. We will corroborate this idea after visualizing with a geographic layout help a quite connected graph still shaping those available paths. Now we are ready to analyse some metrics of the graph and to play with its visualizations, but first we will also take a look to the requests to see what statistics can we find.

3.3.1 Request statistics

Before facing the graph, we will look again to the requests dump trying to find some statistics about the targets. With our program readTrackposition.py (Section 3.2.1), after separating the different type of requests, we are also collecting every target descriptor used. And we store at the end of the resulting file (after the requests) every distinct target descriptor ordered by the number of times it appeared. Our aim is to have an approximation to the targets popularity.

We have in Section A the target names requested at least ten times. The targets are described by the name of the room or an identifier, so if we want to have a higher-level idea of the differences between building solicitations we have to gather the weight of all the targets belonging to the same building. Many targets have the name of the building within and none of the words composing the name of the target is repeated in the own target name. So we can start creating a list counting the repetitions of every distinct word that appear in the target field among all the requests.

With the program readTrackposition.py we put in a list each target name that appears in the dump file. From every item of that list, we take every single word separated by an empty space. Then we store in another file (buildingsSearched.txt) those words ordered by the number of repetitions. Now we can find a building's name and have an approach to the number of times a room from that building was inquired, but not all of the targets had the building name on themselves.

To have a more accurate approximation, we will search manually for those destinations that don't contain the building name in the file with all the targets (Section A). Once we find one, we search for it in the CampusGuiden web application (to know in which building is located) and then we add the weight of that target to the weight of the building it belongs to.

3.3.2 Graph visualization

We already have the graph arcs, but we can not find features and look for insights in such a large graph structure without the proper tools. We need a software that allows us to play with the graph, to visualize the graph according to any edge/node attribute or graph metric. It would be also interesting being able to watch in real-time those changes in the graph, highlighting each time any any property of the graph.

Gephi

Gephi [BHJ09] is an open source graph exploration and manipulation software. We can display here large graphs in real time, allowing dynamic visualization. We are going to use this software because it is really user-friendly, and it has several developed modules for filtering, manipulating and analysing the graphs according to their properties [Con12].

The *ranking* module lets us configuring the color and size of the nodes and edges, creating also a table with the labels of the classification values. The *metrics* module calculates the betweenness, closeness and degree centrality for example, and it can even detect communities (Gephi implements the Louvian Method [BGLL08] to detect communities). And the *filter* module allows us to hide nodes or edges on the graph according to several parameters.

But still the main reason why we chose Gephi are the layout algorithms we are going to use to visualize the graph: *Yifan Hu*, *Force-Atlas 2* and *GeoLayout*. We can easily export those visualizations to several formats like pdf, svg or jpeg.

Layouts

A graph is an abstract mathematical object, it has no information about how representing its objects in two or three dimensions. So when we want to visualize a graph, we need a way to map every vertex to coordinates in the space, hopefully in a way makes the graph more eye-friendly. A graph theory branch -graph drawing-tries to solve this problem through graph layout algorithms.

Our graph is comprised of a large number of nodes and links between them, where the visual clutter hides high-level edge patterns. The next Figure 3.7 shows the graph nodes aligned randomly in Gephi where we cannot get a clear idea about the network structure, that visualization result useless. We need a more readable visualization.

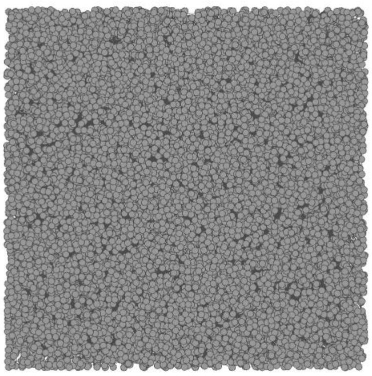


Figure 3.7: Visualization in Gephi of our graph before applying a layout.

A node-link diagram can be an intuitive representation about the amount of traffic between locations, but we need a more clean an uncluttered visualization than the one we have in Figure 3.7. The nodes depict location in traffic networks like ours, so instead of modifying the node positions to reduce the visual clutter we will focus on the edges representation.

The layout algorithms combine different edge bundling techniques for a better graph visualization like drawing edges between clusters of nodes instead of individual edges; allowing groups of edges to be merged and drawn together or bundling edges together according to a previous created hierarchy. The resulting bundled graphs show a clutter cut down, being high-level patterns more visible [HVW09]. We will be also interested in the layout algorithms that can also determinate the nodes position by geographic coordinates when representing geographic information.

The layout algorithms are designed trading off the previous techniques focusing on highlighting different aspects of the graph. The layout algorithm sets the graph shape, so it is an essential choice. They also have some properties we can adjust to get a more pleasing graph visualization.

The *Force-based* algorithm family have an easy principle: linked nodes attract each other and non-linked are pushed apart; they emphasise the complementarity feature of the graph. From this family we will use **Force Atlas**, **Force Atlas 2** and **Yifan Hu**. The first one is focused on quality to allow a good readability, being useful to explore real data. Its properties define how slow the nodes move, how strong they attract/reject, how dispersed we want the disconnected components and an option to push hubs (high out-degree nodes) at the periphery and push the authorities (high in-degree) more central. The second is a version of the first one to handle large networks. Its properties can make the clusters tighter, the graph sparser and define how the edges weight influence on the force calculations. The third layout reduces the complexity making itself likely to large graphs. Its properties define how far apart we want the nodes, how accurate we want the calculations and an extra ratio between quality and speed.

The other different kind of layout is **GeoLayout**; it uses latitude and longitude coordinates to set nodes position. Between all the available projections for representing those coordinates we can choose *Google Mercator*, the one employed by our nodes (Section 2.3). So with this layout we can place in their actual Gløshaugen location the hubs, authorities, communities, weighted edges and other graph key metrics.

3.3.3 Graph analysis

In this section we are going to take a deeper look into the graph comprised by the weighted edges we obtained from the CampusGuiden traces. We are going to calculate first with python some key metrics to check the consistence of the graph. And then we will manipulate the graph with Gephi according to more complex graph properties. With the metrics and the visualizations we will have the quantitative insight enough on the graph structural properties to be able to detect the key concepts, clusters and most used pathways [Par]. Then we will be able to discuss outcome practical applications.

Python

Python has a wide free distribution libraries already developed at everybody's disposal, and we chose the Networkx package for creation and manipulation of complex networks (documentation available at [Dev12]). We have to install the library in our computer first if we want to use the program we wrote: *loadWeightedGraph.py*. To execute it we follow a similar syntax than the previous programs: *$ python loadWeightedGraph.py arcsForPython.txt*, where the last file contains the edges in the following format x1,y1,z1;x2,y2,z1;buildingId;weight. As we can see in the Section 3.4 it is really important to keep this format without empty spaces or other symbols.
Our python program stores the nodes and edges in a CSV format that could be well read by Gephi. The nodes are going to be stored with their attributes as identifier, but adding also their coordinates and building identifier as attributes because that way Gephi will be able to play with them. The edges are going to be stored using for the source and target nodes fields the same node-identifier we used in the nodes list, adding as edge attributes the weight and building identifier. The nodes and edges files can be loaded on Gephi, and we need both files with all the attributes to be able to represent the edges with the geoLayout (we need the coordinates of the nodes as node attributes).

With this python program we will also to calculate some graph metrics, but we have to consider that some algorithms work only for directed graphs while others are not well defined for directed graphs. So after checking the bi-directionality of the edges and analysing the in&out degree, we will have to convert the directed graph to undirected to be able to calculate other measurements (we can easily convert the graph using undirectedGraph=directedGraph.to_undirected()). We can see the resulting metrics in the Figure 4.1 for the directed graph and in Table 4.2 for the undirected.

Gephi

There are some properties like the centrality that can be hard to measure with python, and they are more eye-friendly with Gephi visualizations. So we start loading on Gephi the CSV files we have created with python with the nodes and the edges. And then we are ready to play with Gephi in order to obtain overview visualizations of the graph's communities and centrality properties (degree, betweenness, closeness and eccentricity). With the visualization techniques, community detection mechanisms and quantitative metrics we will get a better insight into the graph structure.

Communities. Our network has a natural community structure based on the buildings. Before validating that idea with the graph tools [GKH09], we can presume that people requests within the same buildings would keep some kind of relation between them. We can think that the routes inside a building will share some common segments like stairs or main corridors, and that new users are more likely to ask for more than one room in the same building. Usually the courses belonging to the same department take place in the same building, so new students should have made more requests about their corresponding buildings. But it could also happened in the other way, a student can be familiar with his own building, so he will use the CampusGuiden service if he suddenly has to find a place he is not used to go.

Looking the community structure we will x-ray the structure to discover the natural community boundaries -the different buildings- and how are they related to each other. We are going to see how the nodes are distributed across the buildings, but if we want to have a clearer idea of the traffic between communities (how the buildings were linked) we need to get a more abstract view. So we are going to collapse the different communities into metanodes, and we will see how the metanodes are related to each other. That way we can see the mobility patterns of the users between buildings and we detect the walk highways. We are going to group the nodes by their building identifier using the partition module of Gephi.

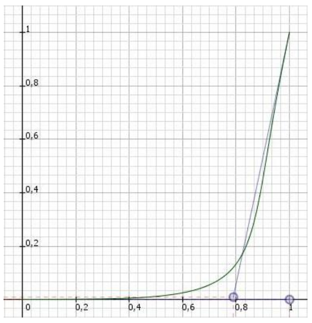


Figure 3.8: Function used to size the nodes according any property of them.

It can be changed any time, but we will keep it along all the centrality measures to highlight the highest values in the graph.

Centrality. In order to provide more meaningful images, we will range the nodes size according to the different centrality conceptions Section 2.4. That way we will be able to picture the interesting points on their current locations using the GeoLayout.

As a comment for these visualizations, we will keep the color-distinction between communities to be able to locate more easily the interesting places. The nodes are sized according to the corresponding centrality definition we are facing at each section, and the range goes between 1 and 60 utilizing the function shown in Figure 3.8. We selected that function to cut the clutter and highlight just the highest values, but for a deeper analysis a more linear function would be wiser (Gephi let us change the spline option).

Degree. This centrality notion measures the number of nodes either coming or leaving any node. It gives a natural intuition about how many relations a node keeps, how well connected it is. Points that bridge different segments -high degree values- should correspond to main corridors or stairs locations, since they are the natural boundaries among the rooms, floors and buildings. From these bridge-pints information could be spread across the community they belong to (its building). So they could be the perfect ones to settle advertisement campaigns to radiate some information across the desired building. We have a directed graph, so we are analysing distinctly those relations based on the direction people followed.

In Degree. The in degree of each node shows us how many segments come to it, how many paths lead to that point. In our directed network means from how many different places we can reach that node, they can be seen as route-gathering points.

Out Degree. The out degree of each node shows us how many segments leave from it, how many different paths can be chosen at that point. In our directed network means that from those divergent points we have several places to go, people spread across the buildings from those points.

Betweenness. This centrality measure counts how many times a node belongs to a shortest paths between a pair of nodes, how many times a shortest path goes through that node. The higher the betweenness centrality is, the more influential is the node, because it is a junction for communication within the network.

The difference with the in/out degree is having now a higher level of centrality conception (but lower than the one we achieved with the communities). Now we highlight the nodes where more routes cross within the whole network, not just in a local way like a building and its nearby (what we did with the degree). These notion captures brokerage.

Closeness. Closeness centrality measures how far away the rest of the network is from a node, how easily that node can access to the whole network. Instead of setting constraints (betweenness) or spreading information (degree), this notion captures scope interaction, seen as access to more resources. Gephi calculates closeness as the average number of hopes -distance- between each node and the rest of the network.

Eccentricity. Gephi measures the eccentricity of a vertex as the distances from itself to the most distant reachable vertex. While closeness measured the average distance to the rest of the network, eccentricity measures the longest distance from a node to the rest of the network. Giving us an idea of the most peripheral places in the campus.

3.4 Issues

3.4.1 First approach: too sparse

When we obtained the weighted arcs for the first time we faced an unexpected contradiction. The gml graph representation seemed to be correct, as long as the result shaped so well the actual paths that could be followed in the campus. But when we tried to represent the graph either in Gephi or in the python library for graphs Networkx: in both cases we obtained a highly unconnected graph with much more nodes than edges, like the represented in Figure 3.9. It called our attention by some reasons: we had a really good looking highly connected core, but at the same time so many unconnected segments didn’t make sense. The segments we had created should shape continuous routes, some of them thicker, owed to the weight, than others. So the core made sense, but not the peripheral unconnected segments. So we decided to look into the followed process again.

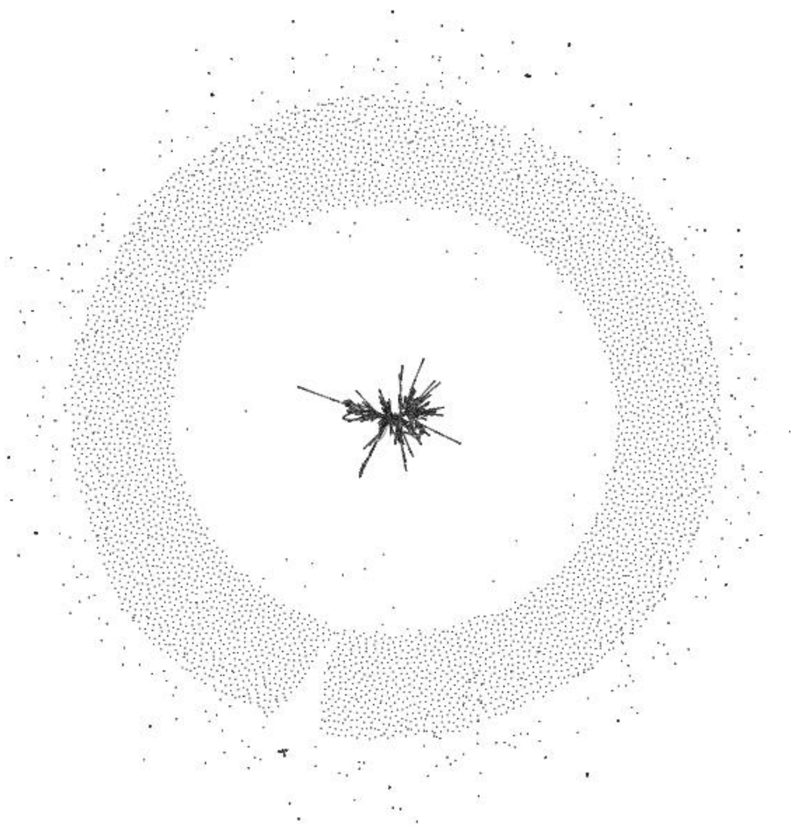
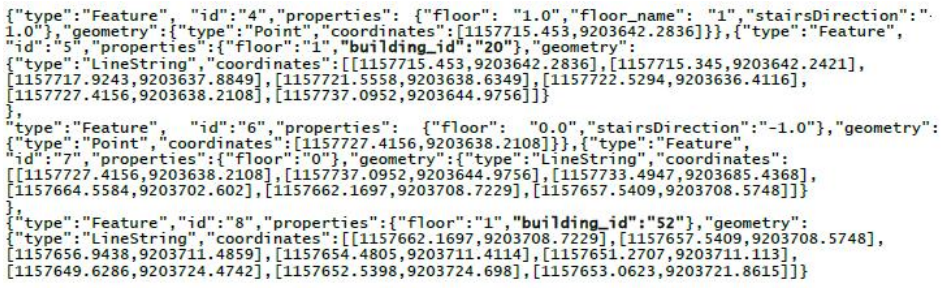


Figure 3.9: Visualization in Gephi of our first obtained graph applying the layout Force Atlas 2.

3.4.2 Problems fixed

We started assuming that the general followed methodology and the data we obtained from the server would be correct, and the gml representation supported that assumption. Anyway, after starting checking from the beginning the more likely point to be making something wrong was the segments creation. Somehow some segments seem to be unconnected in the same route, and that shouldn't happen. We fixed the following issues:

BuildingId. The paths that took place outdoors (outside any building), since they lacked of an own building identifier they were using the same as the previous path (example in Figure 3.10. So we had several segments with coordinates corresponding to outdoors repeated with different buildings identifier (since the segments had different building identifier they were not completely equal, and our program considered them as different even having the same coordinates). Assigning by defect the building identifier zero -being replaced if any other is found by itself- we solved this problem. Now the outdoors segments had the zero building identifier, we cut down the number of segments but the main problem persisted. We still had too many unconnected arcs.



```
{"type":"Feature", "id":"4","properties": {"floor": "1.0","floor_name": "1","stairsDirection":"·
1.0"},"geometry":{"type":"Point","coordinates":[1157715.453,9203642.2836]}},{"type":"Feature",
"id":"5","properties":{"floor":"1","building_id":"20"},"geometry":
{"type":"LineString","coordinates":[[1157715.453,9203642.2836],[1157715.345,9203642.2421],
[1157717.9243,9203637.8849],[1157721.5558,9203638.6349],[1157722.5294,9203636.4116],
[1157727.4156,9203638.2108],[1157737.0952,9203644.9756]]}
},
"type":"Feature",  "id":"6","properties":  {"floor":  "0.0","stairsDirection":"-1.0"},"geometry":
{"type":"Point","coordinates":[1157727.4156,9203638.2108]}},{"type":"Feature",
"id":"7","properties":{"floor":"0"},"geometry":{"type":"LineString","coordinates":
[[1157727.4156,9203638.2108],[1157737.0952,9203644.9756],[1157733.4947,9203685.4368],
[1157664.5584,9203702.602],[1157662.1697,9203708.7229],[1157657.5409,9203708.5748]]}
},
{"type":"Feature","id":"8","properties":{"floor":"1","building_id":"52"},"geometry":
{"type":"LineString","coordinates":[[1157662.1697,9203708.7229],[1157657.5409,9203708.5748],
[1157656.9438,9203711.4859],[1157654.4805,9203711.4114],[1157651.2707,9203711.113],
[1157649.6286,9203724.4742],[1157652.5398,9203724.698],[1157653.0623,9203721.8615]]}
```

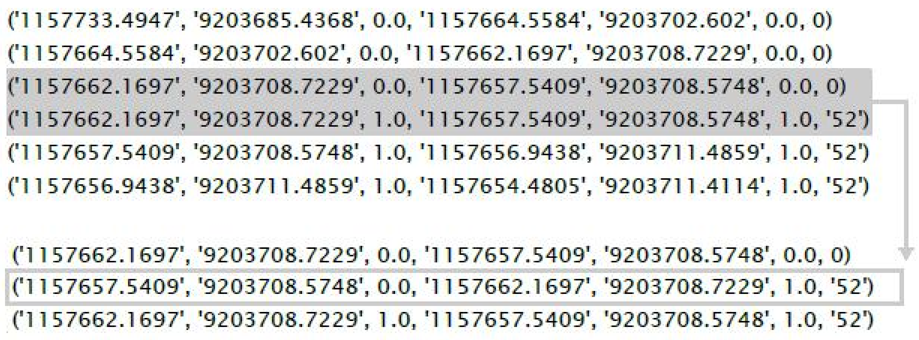
Figure 3.10: Example of three paths within the same route

The segments created from the middle one -before fixing the problem- were using the building identifier from the previous path: *20.* All these segments created from features without building identifier -outdoors- will have the *0* as identifier now.

Node identifiers. We still had too many unconnected arcs, but the problem this time was different. When we were creating the graphs, the coordinates of the end points were taking as node identifiers. The problem was that we had some blank spaces between the coordinates, and the software sometimes matched the points and other times it didn't. Deleting those empty spaces and separating the coordinates with a comma we reduced significantly those unconnected arcs.

Unlinked segments. Checking again the whole process we found some interesting situations. Sometimes, for example when getting inside a building from outside, we had a break in the route. At that points, we had two segments with the same latitude and longitude in both end points but different 'z'. The given route, to this situation (for example entering a building with different height from outside) answered duplicating the segment with different 'z' and building identifier instead of creating a stair. In result we had a split route (the last point of the first segment didn't match the beginning of the next one) when both parts actually were parts of the same one.

We decided to create an 'artificial stair' to link both parts of the same route when these situations would take place. So each time we see that two segments in a row shared the latitude and longitude of the start and end point -in the same position- but not the 'z', we were going to create and artificial segment linking them. We can



```
('1157733.4947', '9203685.4368', 0.0, '1157664.5584', '9203702.602', 0.0, 0)
('1157664.5584', '9203702.602', 0.0, '1157662.1697', '9203708.7229', 0.0, 0)
('1157662.1697', '9203708.7229', 0.0, '1157657.5409', '9203708.5748', 0.0, 0)
('1157662.1697', '9203708.7229', 1.0, '1157657.5409', '9203708.5748', 1.0, '52')
('1157657.5409', '9203708.5748', 1.0, '1157656.9438', '9203711.4859', 1.0, '52')
('1157656.9438', '9203711.4859', 1.0, '1157654.4805', '9203711.4114', 1.0, '52')

('1157662.1697', '9203708.7229', 0.0, '1157657.5409', '9203708.5748', 0.0, 0)
('1157657.5409', '9203708.5748', 0.0, '1157662.1697', '9203708.7229', 1.0, '52')
('1157662.1697', '9203708.7229', 1.0, '1157657.5409', '9203708.5748', 1.0, '52')
```

Figure 3.11: Two segments belonging to the same route are unlinked when changing the building.
They have the same latitude and longitude in their start and end point, but different 'z' (this will be the way to locate this situations) so we create an artificial segment.

understand this more easily with the example in Figure 3.11.

Finally we had the edges of the graph we were looking for. We couldn't appreciate these problems in the GML figure because there the segments were represented by their coordinates: so the node identifiers problem didn't happen, the edges were in their location despite their building identifier and the paths were overlapped even being unlinked. Thanks to the graph visualization (without the coordinates reference) we could fixed these underlying problems and obtain a more connected graph with the followed routes.

Chapter 4
Results

We already explained in Section 3 the process we were going to follow in order to obtain a graph from the traces through several python programs. Then we also described the tools we employed to analyze the metrics of the graph, as well as the properties we wanted to highlight in order to reveal some insights about the graph structure through different visualizations.
In this chapter we show the results we collected after analyzing the CampusGuiden traces file and the weighted graph shaped by the requested routes. We will present them reporting the outcomes those data reveal to our case of interest. We remember the reader that our aims are finding the most influential nodes/edges that function as junctions between the different requested points of interest and the main quantitative properties of the graph.

4.1 Raw results

This section introduces a statistical view on the results from the main metrics and visualizations of the graph.

4.1.1 Graph metrics

The following metrics shown in Table 4.1 and Table 4.2 we got after processing the graph with the python program we wrote, and they match with the ones obtained by Gephi. Note that we had to convert our directed weighted graph into undirected to be able to use some Networkx methods, not available for directed graphs.

From the directed graph (Table 4.1) we can point out some facts about the graph consistency and the outcomes from the degree measurements:

The directed graph has more than 50% bidirectional edges. This reflects that central paths are used in both directions. Users can go from the main building to a bus stop and the opposite, but still paths addressing the periphery are less likely to be required not leading to the central positions. And also

Directed
Directed nodes: 11297
Directed edges: 15865
Bidirectional edges: 8058
Non bidirectional edges: 7807
Average in-degree: 1.40435513853 total inDegree: 15865
Average out-degree: 1.40435513853 total outDegree: 15865
Max InDegree: 6
Min InDegree: 0
Nodes with max inDegree
1158011.805,9203507.1281,0.0
1158447.7945,9202928.1989,1.0
1158213.3523,9203068.4147,0.0
1158219.6257,9203252.4184,2.0
Number of nodes with min inDegree 961
Max OutDegree: 6
Min OutDegree: 0
Nodes with max outDegree
1157918.3114,9203478.0101,1.0
1158091.5894,9203329.7947,0.0
Number of nodes with min outDegree 636

Table 4.1: Graph metrics calculated by our python program with the Networkx library for the directed graph.

the last segments of the paths, for example just a small corridor in front of a classroom, are more likely to be just ending points not eager to start a route in the opposite direction.

There are 4 points with in degree 6, while 961 points have zero. We can highlight those four authorities as places where many routes lead, the high number of points without any income segment can be easily identify as the points where some requests took place but nobody asked to go there.

There are 2 points with out degree 6, while 636 points have zero. We can highlight those two hubs as more divergent places, more different routes can start from these points. The points with no out degree represent those target points, once the users reach their destinations there is no next point.

The total in and out degrees are the same. Every edge in a directed graph without loop edges, starts in one node and ends in a different one. So for each edge we should have one in-degree and one out-degree (no matter directionality). Indeed we have the same in and out degree, and that amount is equal to the number of edges: our graph is consistent.

From the undirected graph Table 4.2 we can still point out facts about our graph consistency and how interconnected is:

The number of edges has been reduced. We had more than a half of edges

UnDirected
UnDirected nodes: 11297
UnDirected edges: 11836
Average degree: 2.09542356378 total Degree: 23672
Max Degree: 7
Min Degree: 1
Nodes with max degree
1158213.3523,9203068.4147,0.0
1158213.3523,9203068.4147,1.0
1158091.5894,9203329.7947,0.0
Number of nodes with min Degree 1671
Average Clustering Coefficient: 0.00446389053984
Number of strong connected components: 47
Giant Component
Average shortest path of the giant component: 49.8695522356
Number of Nodes of the giant component: 10524
Number of Edges of the giant component: 11107
Percentage of nodes of the the giant component: 93.157475436
Percentage of edges of the the giant component: 93.8408246029
Average degree in the giant component: 2.11079437476 total Degree: 22214
Max Degree in the giant component: 7
Min Degree in the giant component: 1
Nodes with max degree
1158091.5894,9203329.7947,0.0
1158213.3523,9203068.4147,0.0
1158213.3523,9203068.4147,1.0
Number of nodes with min Degree 1532
Biggest Diameter 156 Diameter of the giant Component: 156

Table 4.2: Graph metrics calculated by our python program with the Networkx library for the now undirected graph. Since our graph is not completely connected, some metrics can only be measured for the giant component.

bidirectional -per each pair of them there will be just one now- so at first sight the number of edges should be lower than in the directed. In fact, the number of edges now should be half the previous bidirectional plus the non bidirectional, and so it is.

The total degree is two times the number of edges. Every edge we have now add one degree in two nodes, so the total amount of degrees has to be twice the number of edges: our graph is consistent again.

The highest and lowest degree measures make sense. No we don't distinguish between in or out degree, so at least every node has to have one edge since we are creating the graph from edges. The highest in/out degree value we had before was 6, we could have expected more nodes with the highest value -7- assuming that for example a node with 6 outcomes should at least have one

point leading to it, even more with our highly connected graph. But some of those edges must have been two directed bidirectional ones, so they are only one undirected edge now, adding just one total degree to both nodes. A node that had in/out degree 6 before, now needs to be connected at least to more than 6 nodes to have a higher degree.
The degree now means the number of neighbours a node has. And that's why the minimum degree is one -every node has at least one neighbour, otherwise it wouldn't exist in the graph.

We have 47 components but the core gathers 93% of the initial graph. We will see again the number of unconnected components we have in Figure 4.1. We have some unconnected routes outside the core of the campus but still the main of the routes remain connected. Probably with more data we would have a higher connected giant component and less disconnected components.

The degree laws are fulfilled again within the giant component. The degree is twice the number of edges. The nodes with more neighbours stay in the giant component, and an 8,3% of the nodes with just one neighbour -ends of routes- stay in the isolated routes.

Average degree. The higher the average degree is the more frequent segments are, and the more diverse are the routes. A lower number shows the presence of many repeated paths in the routes, because if the nodes are less connected to each other there are less choices at the time of picking a route between two nodes. On average we have two edges per node, but considering that we have about 15% of the nodes (1671/11297) with just one edge, we must have some authorities and hubs in our graph. We will locate those nodes with the visualizations of Gephi in the following subsections.

The diameter of the giant component is 156 hops. The diameter of a graph is the maximum distance of all its vertices, and the distance is the minimum length in number of hops between each pair of nodes. So in the giant component the maximum distance we can find between any pair of nodes is 156, the diameter. A high graph distance value indicates that there are deviations within the routes, for instances paths to the bus stops, the international house or the gym. These distance-metrics by their own don't give us a pivotal insight, we need to picture them in a graph visualization because they count the number of nodes not real distances.

In the core we can go between a pair of nodes in 50 hops on average. The average shortest path is the mean among all the distances between every pair of vertices in the connected component. The lower the average path length is, the more interconnected the network is (meaning that certain routes were followed quite often in the campus and some central segments often appear in different clusters being joint paths). We can approach an idea of how interconnected the graph is, but we have to keep in mind that the majority of the nodes are distributed with geographic constraints. Inside a building we will have much more nodes than within an open area, covering maybe inside the building less distance. We think this metric is not a key one for our network.

4.1.2 Graph visualizations

The graph visualizations corroborate some assumptions with the layouts help. Firstly, we can see in Figure 4.1 that we have a highly connected core -a giant component- that groups most of the nodes, remaining some peripheral smaller routes not connected to it. And from Figure 4.2 we notice some graph hierarchy: several thin long routes come to join thicker ones as more close they get to the core.

These two observations could be reasoned: across the campus exist several buildings, some quite far from the rest (like the hospital or the gym), so routes to those points could take place by alternative paths distinct from the common walkways. That's why we have so few unconnected paths, because requests inside the 'heart' of the campus will tend to be connected (even if people from electronics never go to the chemistry building, with just one request between both buildings they will be linked in our graph, and considering the number of students that is not so weird). We had also corroborated this before in Table 4.2, watching that the giant component -the core, the heart of the campus- gathers the 93% of the graph.

With the last Figure 4.3 we can give more sense to the edges thickness, since they are placed in their real location. The thicker edges -higher weight, the more used- go through the highways that shape the backbone of the campus, flowing thinner walkways that at the same time lead to more thinner ones. This Figure 4.3 matches the hierarchy we intuited from Figure 4.2 and the spatial distribution from Figure 4.1, placing the graph insights on their real location inside the campus.

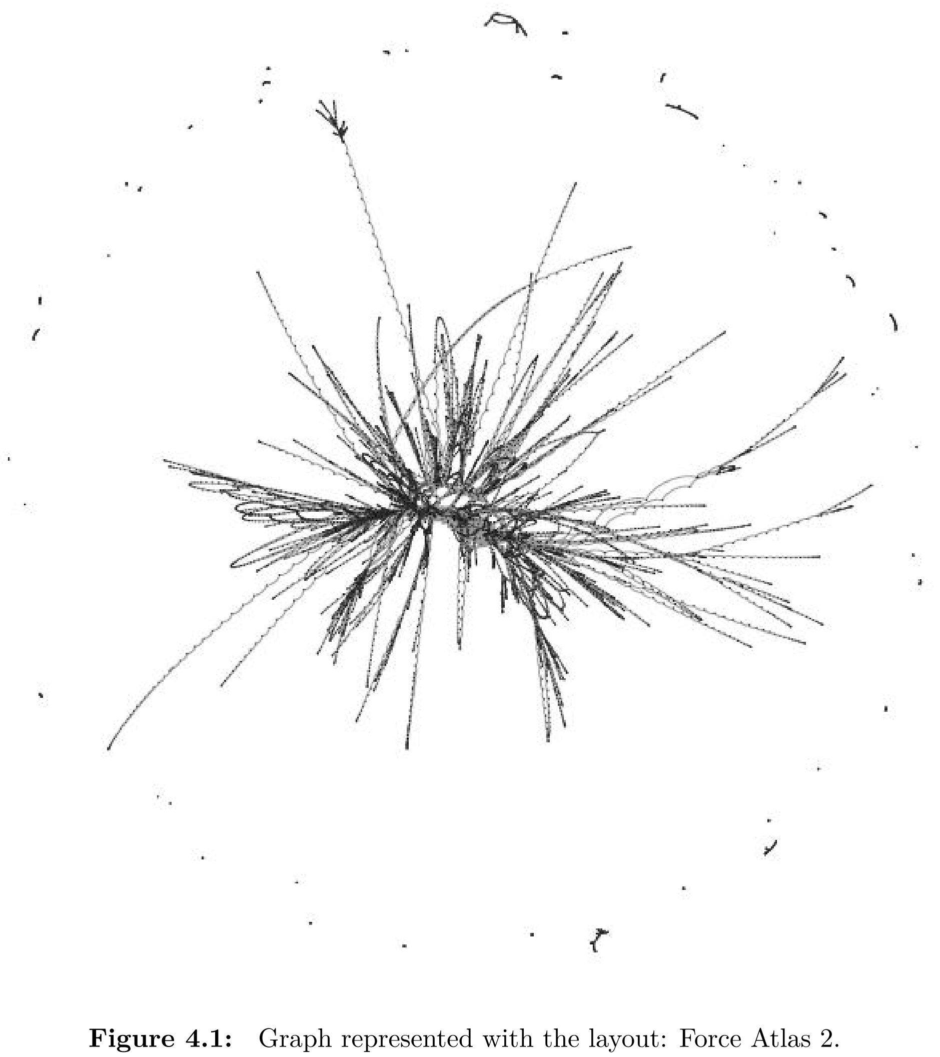


Figure 4.1: Graph represented with the layout: Force Atlas 2.

More connected nodes attract each other while the disconnected push away themselves. It tends to push hubs (high out-degree nodes) at the periphery and push the authorities (high in-degree) more central.

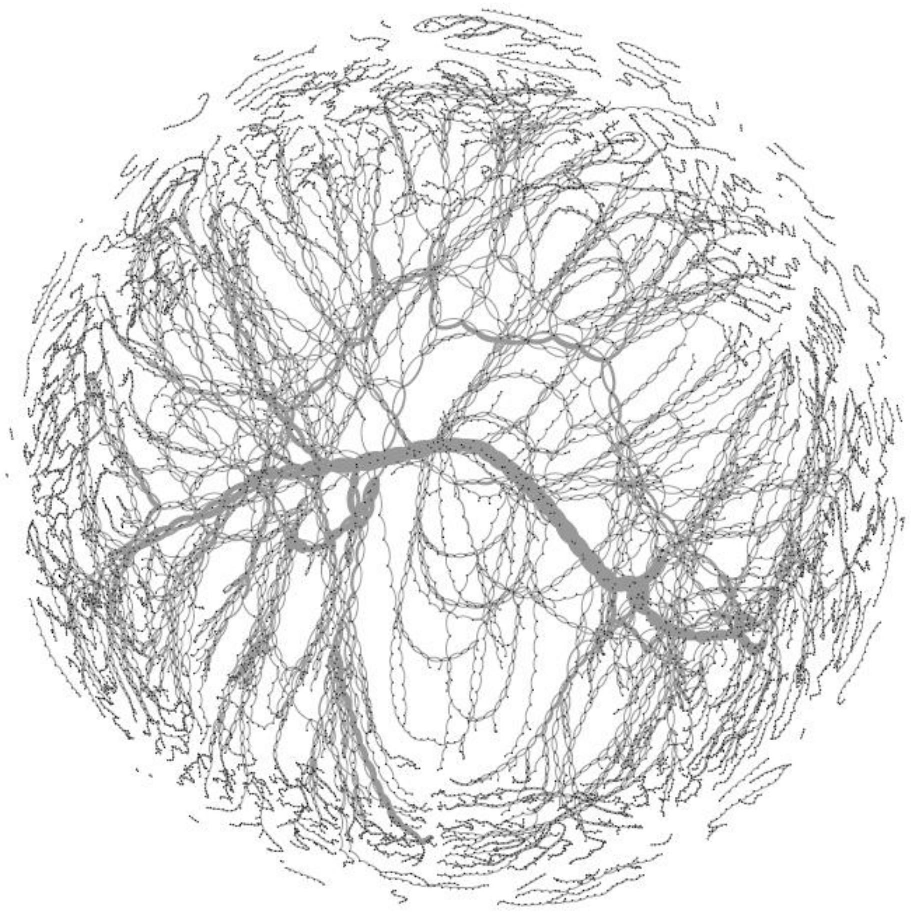


Figure 4.2: Graph represented with the layout: Yifan Hu.

Very similar to the Force Atlas 2 (both are forced-based) but more simple and quick.

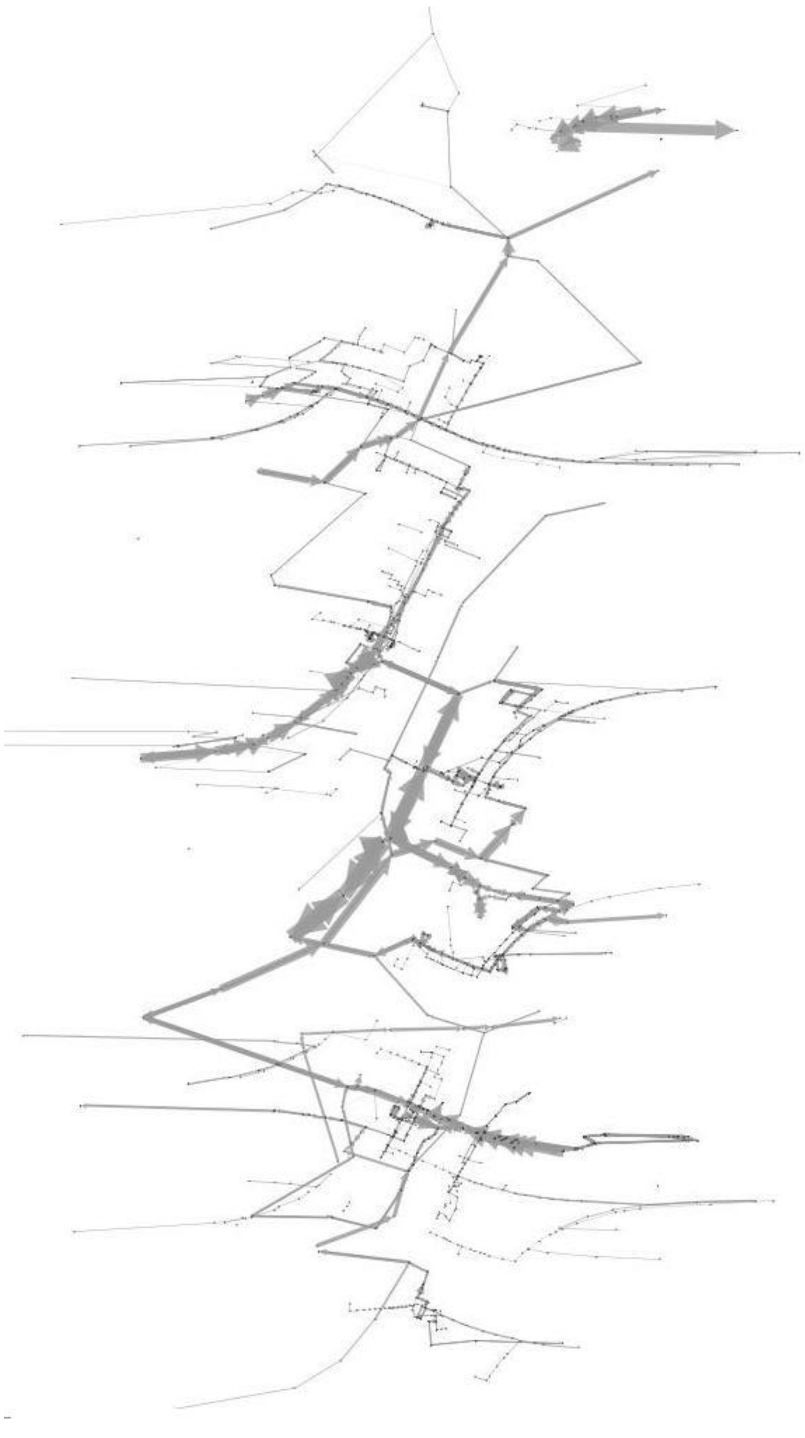


Figure 4.3: Graph metrics represented with the layout: GeoLayout.

This layout uses latitude and longitude coordinates to set nodes position, so we can place in their actual Gløshaugen location the hubs, authorities, communities, weighted edges and other graph key metrics.

4.2 Campus-level interpretation

This section starts introducing a statistical view of the relative request popularity of each building. After exposing how the requests are distributed among the buildings, we are going to analyze how the different buildings are related to each other shaping the campus structure.

4.2.1 Building popularity

We can see in the Figure 4.4 the result of gathering together all the requests that pointed to every target belonging to the same building. The Figure 4.4 shows the approximate percentage of requests per building.

Realfabygget seems to be the building where more rooms were requested, so we are going to see how those requests are distributed among the different rooms Figure 4.6. We are also following the same procedure with our building -Elektro- but this time just differing between wings Figure 4.7.

With the Figure 4.4 and Figure 4.5 we can imagine how the traffic is going to flow addressing the most popular buildings. But there are more interesting insights coming from these data we can highlight:

There are big differences between buildings (Figure 4.5). Realfabygget has more than a quarter of the requests. If we add Realfabygget, Elektro, Sentral-bygg, Fisykk and IT-byggene+syd requests we have almost three quarters of the total (only five between twenty three buildings). These differences can highlight that either some buildings are more familiar to their students, they have more students or it just mean that our application is more known so the people use it more when they don't know how to get to a place. In any case, since those buildings are the trendy destinations, they would be also the more likely to launch new promotions or services. And it could be interesting advertising more the application at the beginning of next course in the buildings with less requests to check if the number of requests increase there (to know if the low number is because the people are familiar with those buildings or because they just don't know about our application).

Looking into Realfabygget (Figure 4.6). The rooms 'R2', 'R10' or 'R7' have more requests, but the distribution among the others is almost uniform. That could mean that more courses take place in those most popular rooms, or that most courses for first-year students take place in those rooms.

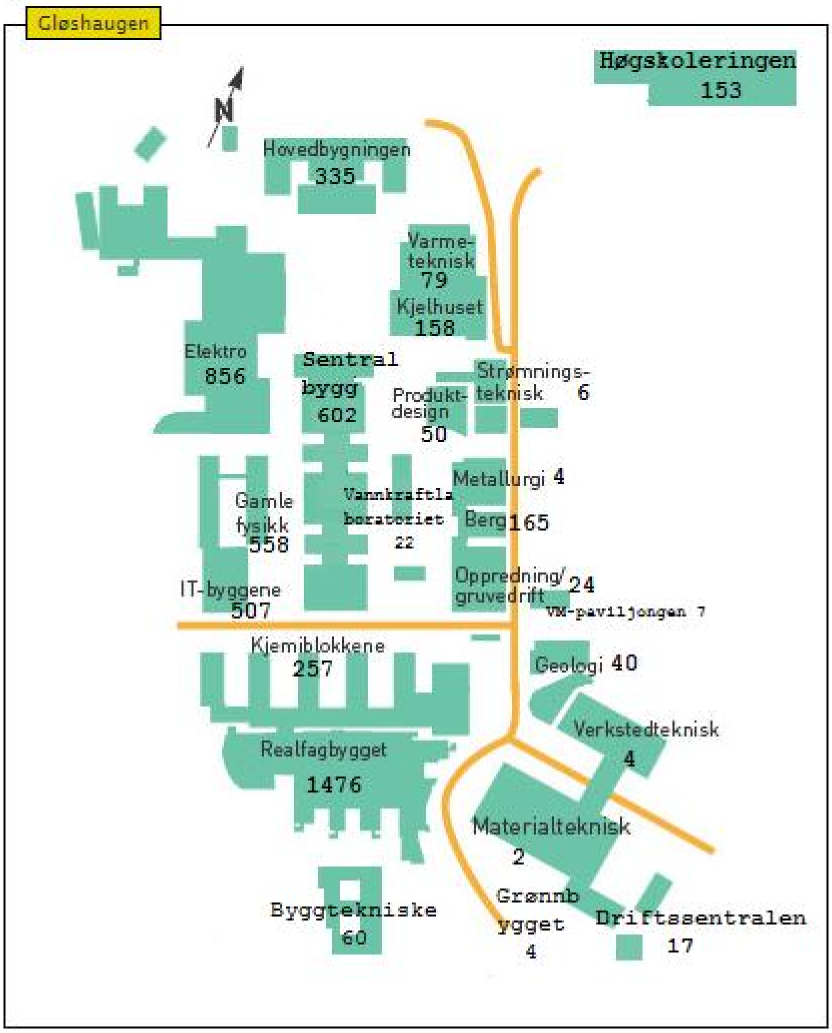


Figure 4.4: Gløshaugen buildings, each one with the number of times that a room has been requested inside.

4.2.2 Foundation places

The Figure 4.8 shows the natural communities of the graph, utilizing different colours according to the building identifier of the nodes (with python we added the building identifier as another attribute to the nodes). The Figure 4.8 pictures how the main traffic flows through the Sentralbygg (buildingId 32, blue), Realfabygget (buildingId 67, green) and outdoors (buildingId 0, brown). The edges across these communities

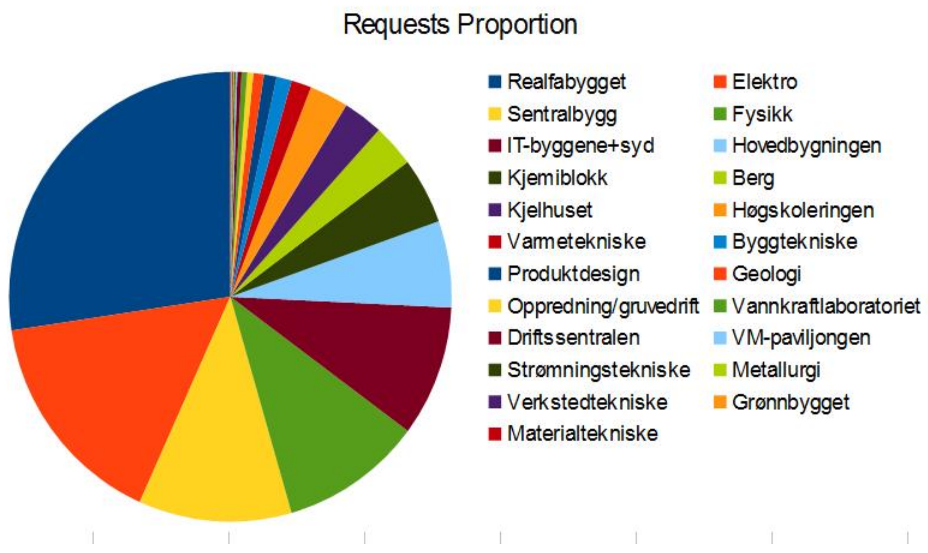


Figure 4.5: Relative request popularity per building in Gløshaugen.

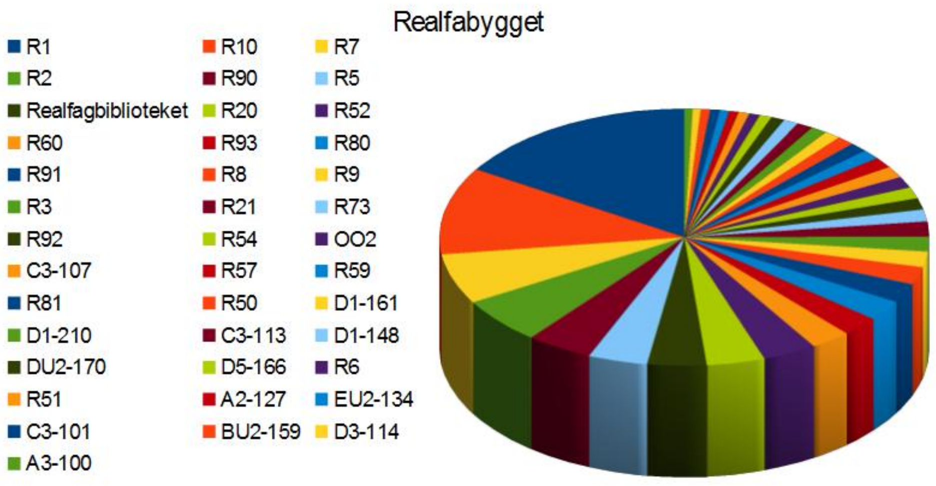


Figure 4.6: An approximate percentage of the times that a room was requested inside Realfabygget: the building with the highest number of requests.

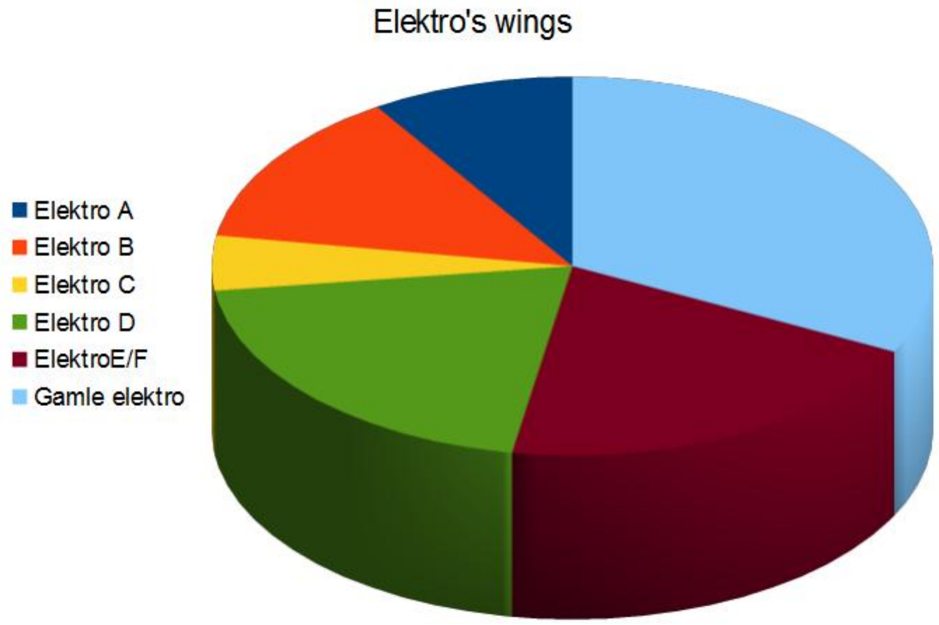


Figure 4.7: An approximate percentage of the times that a room was requested inside each wing of the Elektro building.

are thicker because the weight is higher, what means more traffic inside those buildings (those segments were used more times).

We can also see how the nodes are actually distributed among each building in Figure 4.9. Realfabygget has the highest number of nodes because despite being a big building, as we saw in Figure 4.5 a large number of requests were made to a lot of its rooms Figure 4.6. So it has to be the most explored building (the one with more different points visited) and that is why we have so many segments there (with their corresponding nodes). By the other hand we also have a lot of nodes outdoors since all the buildings can be reached from outdoors, some of them can only be reached that way.

However, the visualization of the different communities doesn't give a clear overview about the relationships between them. So we collapsed all the nodes belonging to the same building, obtaining the Figure 4.10, a new weighted graph where we can accurately describe the current users mobility patterns. Discovering the group of points that tend to be more sequentially followed can tell us the structure of the users movement among the different communities.

The strongest edges go from outdoors to Realfabygget and Sentralbygg, being the traffic in the opposite direction lower but not either negligible. We remark also the strong connection between Sentralbygg and Realfabugget, a lot of people should have used that high walkway. These facts come to sustain what we just saw: the buildings whose rooms were more requested Figure 4.5 are better connected to each other. Those buildings gather stronger connections between outdoors and each other.

We can also see that the edges that go from outdoors to the buildings are stronger than the corresponding ones following the opposite direction. Because no matter where the requests were made we can reach any building from outdoors and also because different routes leading places in the same building are more likely to share the entrance of the building (the closest streets to the door, because since we collapsed the communities we are not considering the routes within the buildings). So our Figure 4.10 gives us an approximation of how many people come/leave each building heading to another building or going just outdoors.

We also appreciate that some buildings are (maybe just weakly) connected to more buildings, while others are only connected to outdoors. Comparing with social network analysis, if the points in a community have more ties within the the own building community than outside, they can be seen 'like-minded' points, persisting equally even if the rest of the network changes. Applied to our case, those communities could stay isolated against events or campaigns launched in other buildings, it is more difficult to reach people inside them, so an special effort is needed there to avoid that. Those buildings that need more attention are for example 28, 54 or 53

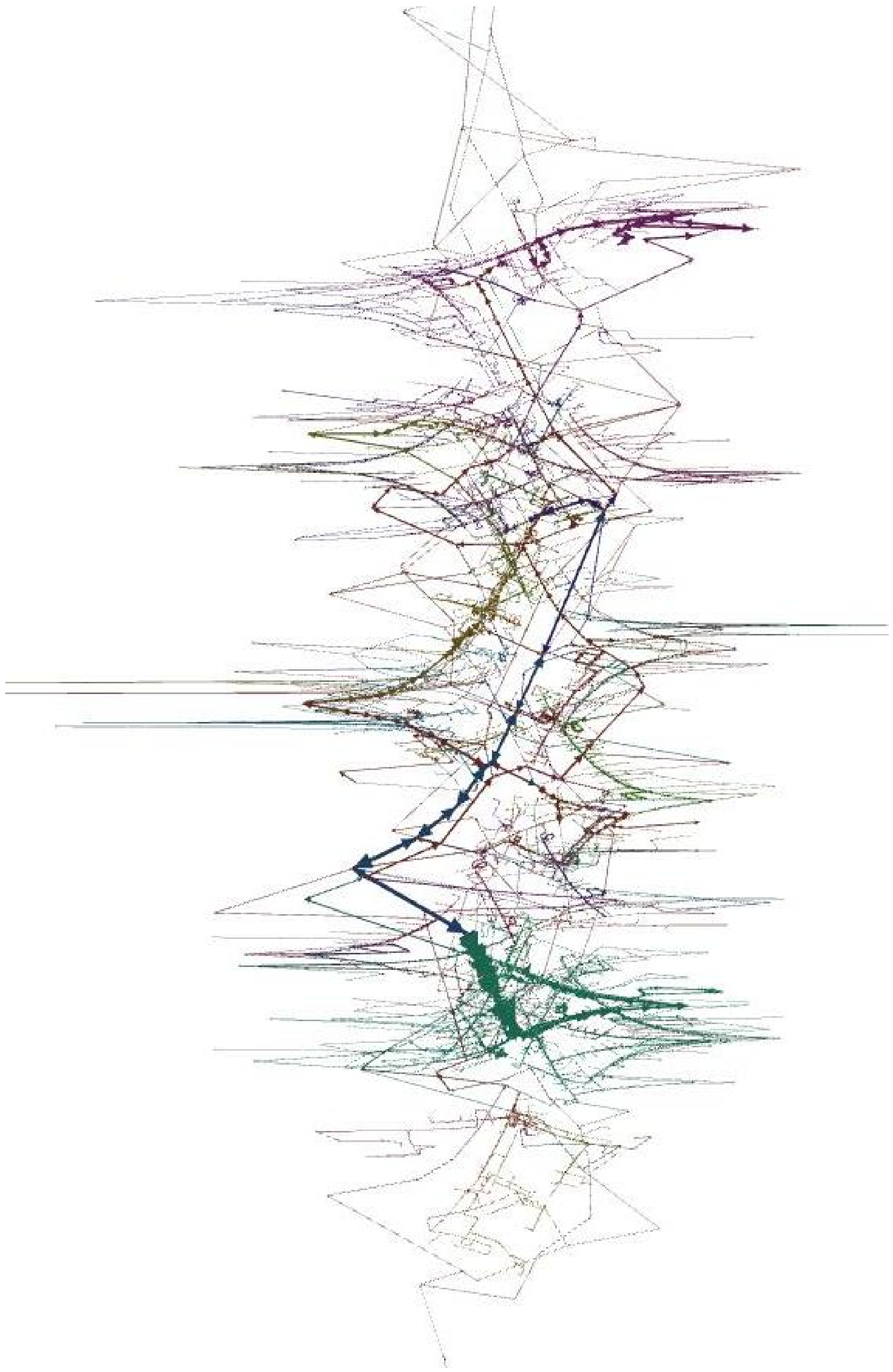


Figure 4.8: Natural communities of our graph: the different buildings. Graph representing CampusGuiden traces, employing different colours according to the building identifiers.

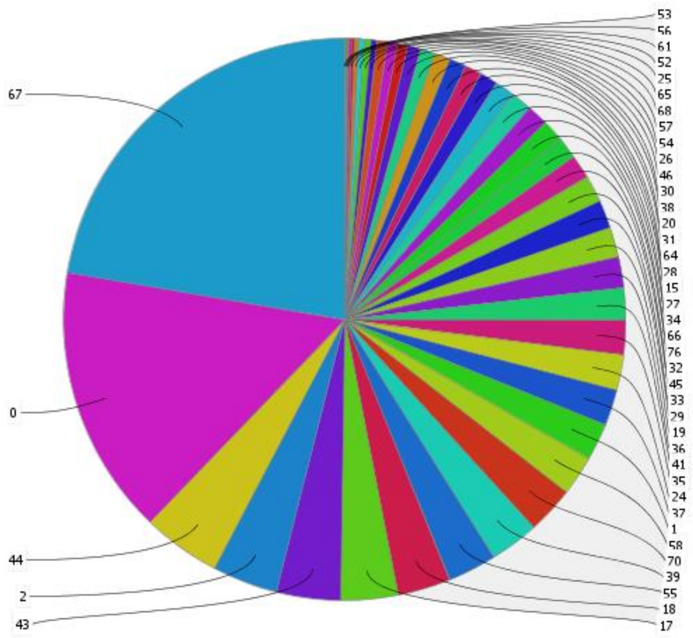


Figure 4.9: Nodes distribution among the natural communities of our graph: the buildings. Each number is the identifier of a building, zero represents the outdoor nodes.

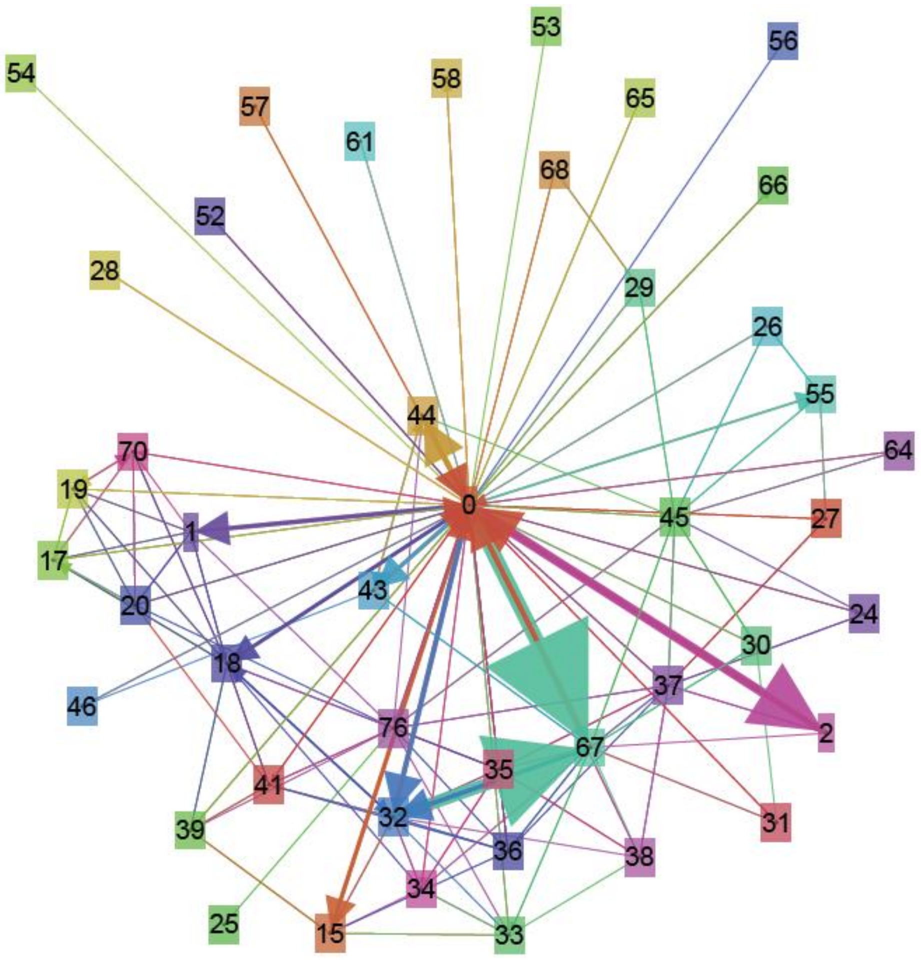


Figure 4.10: Meta-nodes representing the different communities and the weighted edges between them. The edges have the color of the destination node, and the weight sets the size of the arrow. We can picture the traffic that took place between buildings.

f2 341	R1 Realfagbygget 181	R10 Realfagbygget 123
H3 Hovedbygningen 111	F1 IT-syd 98	Lesesal Gul Sentralbygg 1 79
EL5 Gamle elektro 78	R7 Realfagbygget 71	R2 Realfagbygget 65
F2 Gamle fysikk 64	Søk etter bygning eller rom 61	K25 Kjemiblokk 3 50
R90 Realfagbygget 49	H3 49	K27 Kjemiblokk 1 48
R5 Realfagbygget 44	Realfagbiblioteket Realfagbygget 44	F6 Gamle fysikk 43
R20 Realfagbygget 43	ITS204 IT-syd 41	1 Høgskoleringen 3 (P15) 41
Trådløse Trondheim IT-syd 40	ITS206 IT-syd 40	R52 Realfagbygget 40
H1 Hovedbygningen 40	KJEL1 Kjelhuset 40	EL23 ElektroE/F 39
B23 Berg 39	212 Sentralbygg 2 36	G022 Gamle elektro 35

Table 4.3: Top 30 requested rooms in Gløshaugen.

(the ones with just one edge to outdoors).

In the opposite case we can find communities that are highly connected, for example building 45, 76 or 37. Those buildings that bridge different communities must hold the locations of the main corridors, crossroads or stairs. Comparing again with social network analysis, those buildings are more likely to spread some information to their neighbours: more people can go to/come from different buildings, they are natural crossroads. So we should settle the advertisement campaigns, organize the events or test new services in the main hall of those buildings, with the aim that much more people could come more easily from their current locations or just pass by (Being connected to more buildings makes easier to get there, at least coming on purpose from the neighbour buildings or arriving from them just by chance).

4.3 Room-level interpretation

This section starts introducing a statistical view of the popularity of the requested targets, taking a look to what the popularity of the targets discloses. Then we introduce how some centrality measures expose the graph structure. This time we report the graph structure in a deeper level, not talking about the communities that shape the graph but exposing those nodes that set up the foundations of every building and their floors.

4.3.1 Target popularity

We can see the popularity of every target with more than ten request in the appendix Section A, but we are going to start watching the first thirty most-requested rooms in the Table 4.3. As we can see there are really big differences just between this first selection, so the popularity of the rooms is far away from being uniform.

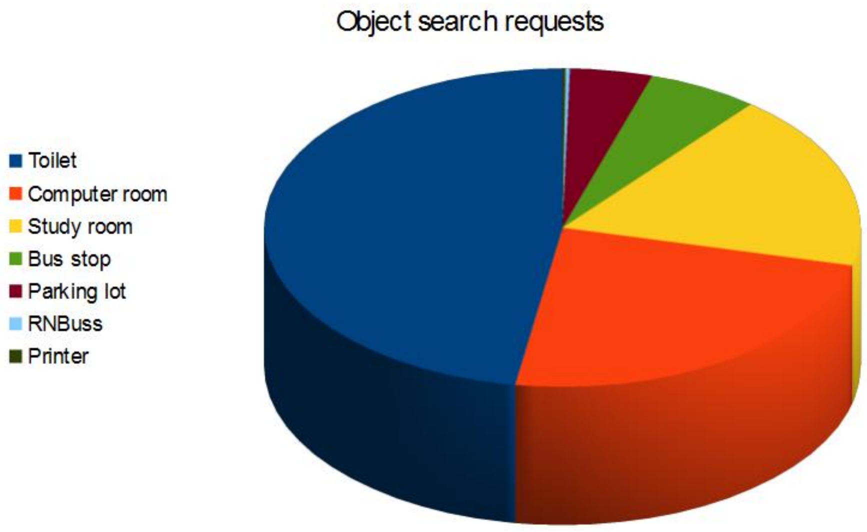


Figure 4.11: Percentage of the interest for some general objects ('objectsearch' type of requests).

But CampusGuiden not only allows specific room searches, from the 'objectsearch' type of requests we can also know which kind of objects rise more interest among the users. Picking every different object and ordering all of them by the number of times they were requested we obtain the percentage shown in Figure 4.11.

We can highlight the following items from the results shown in Figure 4.5 and Figure 4.11:

There are rooms really requested frequently (Table 4.3). 'F2 Gamle fysikk' gathers 405 requests, 'R1 Realfagbygget' gathers 181 or 'R10 Realfagbygget' with 123. The difference with the rest is quite huge, because there are rooms with no more than one request. This fact can reveal either those rooms are really hidden, popular because they hold courses for new students or just a lot of courses take place there. For this resolution we ignore the usage of the application in the corresponding building, because as we saw before those buildings are the most popular (what could also be ought to these rooms).

"Internasjonalt hus" has only 5 requests. Normally the exchange students have problems to find this building and they are the ones who have to go there more to handle paperwork. Having so few requests can expose the CampusGuiden unfamiliarity between the international students, so heading to them more advertisement campaigns at the beginning of next course could be interesting. If new services are included (or just the existing guidance) the international students are a secure market niche: they probably ignore the rooms location, the activities and the promotions that can take place.

“Biblioteket” popularity. While Realfagbygget’s library has 44 requests, the architecture’s one was requested 4 times and the technology’s only 1 time. Assuming their locations are equally known and that all the faculties have about the same number of students: the Realfagbygget’s library is more popular. Since it seems people prefer gathering there to study, it is a good place to launch services or advertisement promotions.

“SiT” interest. Sit could be interested on our outcomes. Their kafes were requested 36 times (Realfagbygget:29 and Kjelhuset:7) and their kiosks 13 times (all in the Sentralbygg). Knowing the interest each of their facilities rise among the students can give them a feedback about their services. We could also arrange a partnership with them, suggesting special promotions or advertising the prizes and offers of every facility.

Anecdotes. There were also requests interested on how to get to places in Trondheim like Samfundent, Torg or Nidarosdomen. But we found also people looking for BigBen, TowerofLondon or Eiffeltårnet, places that turn out to be rooms in the Byggtekniske building instead of being far away.

4.3.2 Foundation nodes

The centrality analysis of our graph identifies the critical nodes within its structure. When we analyzed the structure under a campus level, we identified the buildings and highways that shaped the backbone of the campus in a general view. But now we are going to take a deeper look into the graph structure to locate the nodes that set up the foundation of the communication across every building and their floors.

Degree. The Figure 4.12 highlights the same four authorities the Table 4.1 did. In the in-degree visualization we can appreciate where are located the nodes where more edges lead. On the other hand, the Figure 4.12 highlights the same two hubs the Table 4.1 did. And we can locate in this out-degree visualization where are the nodes where more edges leave.

We can not forget that the paths -streets or corridors- are bidirectional in the space we are modeling: the points with the highest in/out degree values are for sure interesting crossroads no matter the direction.

All the locations we can see in Figure 4.12 and Figure 4.13 where many pathways lead or leave (not only the ones with highest in/out degree) are important connecting

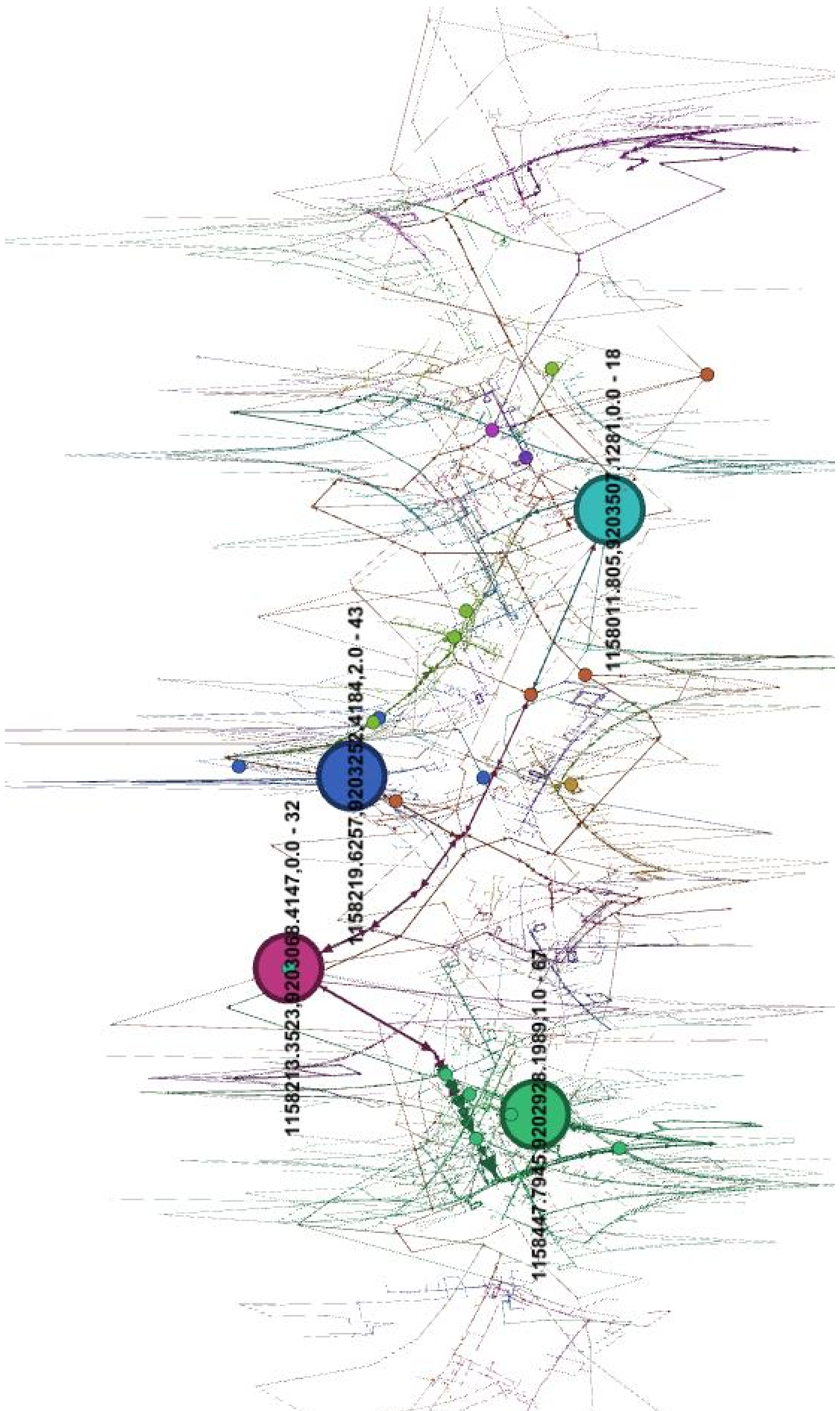


Figure 4.12: Nodes seized by their in-degree centrality. The spline employed to determine the size is given by Figure 3.8. The highest in degree value is 6 and the lowest 0. The highest values are labeled with their coordinates and building identifier.

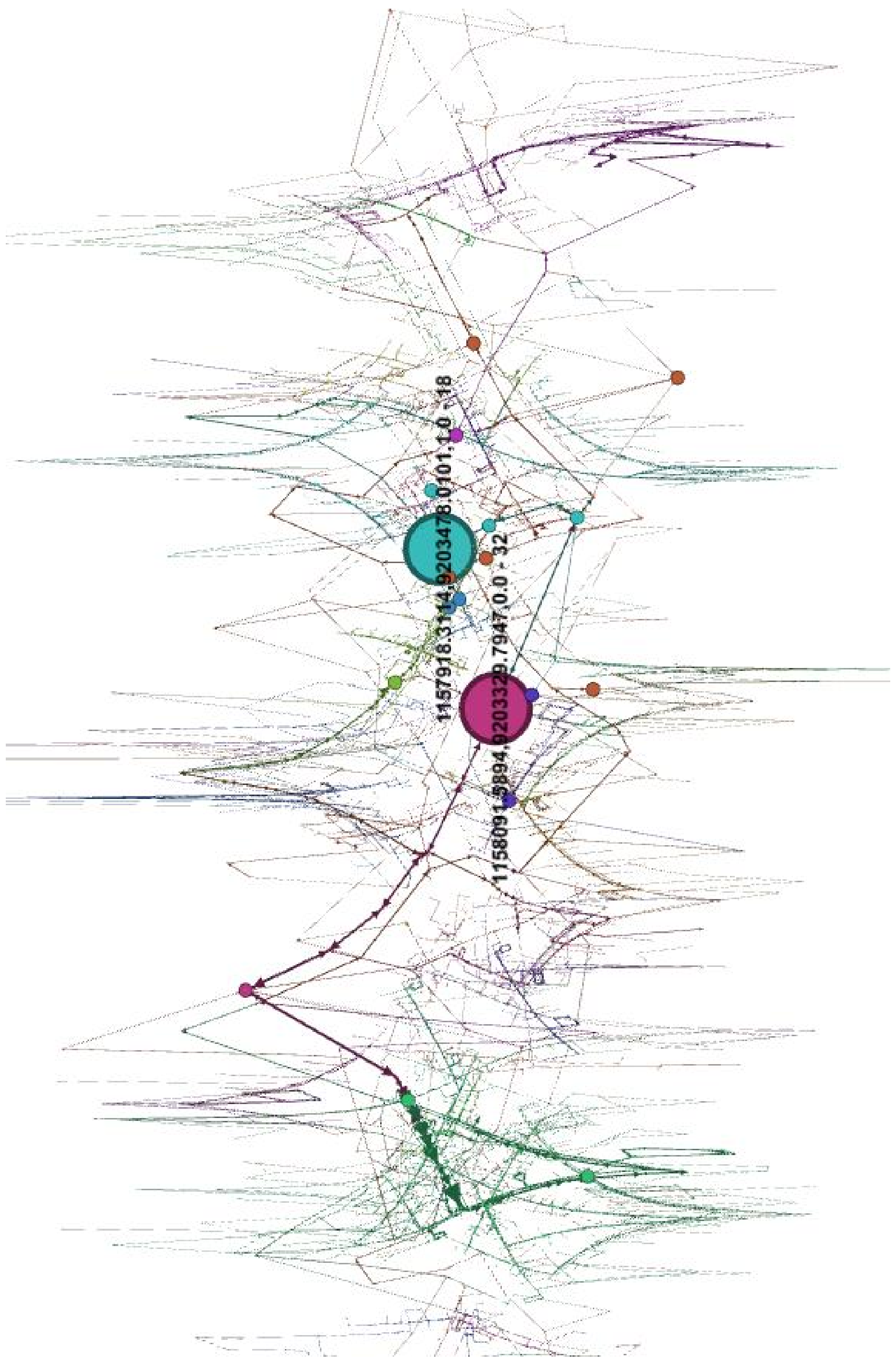


Figure 4.13: Nodes seized by their out-degree centrality. The spline employed to determine the size is given by Figure 3.8. The highest out degree value is 6 and the lowest 0. The highest values are labeled with their coordinates and building identifier.

crossroads. It is very likely that somebody pass by them when going/coming anywhere outside or inside their corresponding building. The analysis is similar to the one we made with the high-connected buildings when analyzing community structure, but in a deeper level. If we want to launch events, testing new services or settle advertisement campaigns (like just sticking a banner in a wall) this points in each building, by the own mobility pattern structure, are more likely to be visited (no matter the weights we had at the moment we run this analysis, because with more data the weight could change but no the main route structures).

Betweenness. In the Figure 4.14 we can see the nodes with higher betweenness centrality. Most of them are located in the Sentralbygg, and watching the long highly-weighted edges they are connected to, we can say that those nodes are the main doors of the building (the floor -z- is zero). The strong edges between those points suggest that that route is a network backbone, what this graph structure-measure is actually supporting.

The edges between these points are flow constrictors, they have the control power of keeping several locations linked. The central location within the campus plus the high level of connection to other buildings, the huge-weighted edges that it has and its role as flow constrictor (due to the betweenness centrality) make Sentralbygg ground floor a key meeting-place in Gløshaugen.

Closeness. Gephi calculates closeness as the average number of hopes -distance- between each node and the rest of the network. So the highlighted nodes in Figure 4.15 have the most difficult access to the rest of the network. From these nodes it is more difficult reaching the rest, what makes them the last points where people could gather by chance. Because if we have a slow access from these points to the rest, reciprocally it is more difficult to end up there from anywhere in the campus. We can see that they refer mostly to high floors, those are nodes to avoid.

Eccentricity. Eccentricity gives us an idea of the most peripheral places in the campus, as we can see in Figure 4.16. Those points are located in the perimeter, inside buildings with several floors, and most of them are indeed on the highest floors (where going through more nodes -floors- is required in order to leave or come).
Using the filter module, we deleted the features that are not on the ground floor (z=0) obtaining the Figure 4.17. That image confirms that the more peripheral points are located in highest floors of the less central buildings.

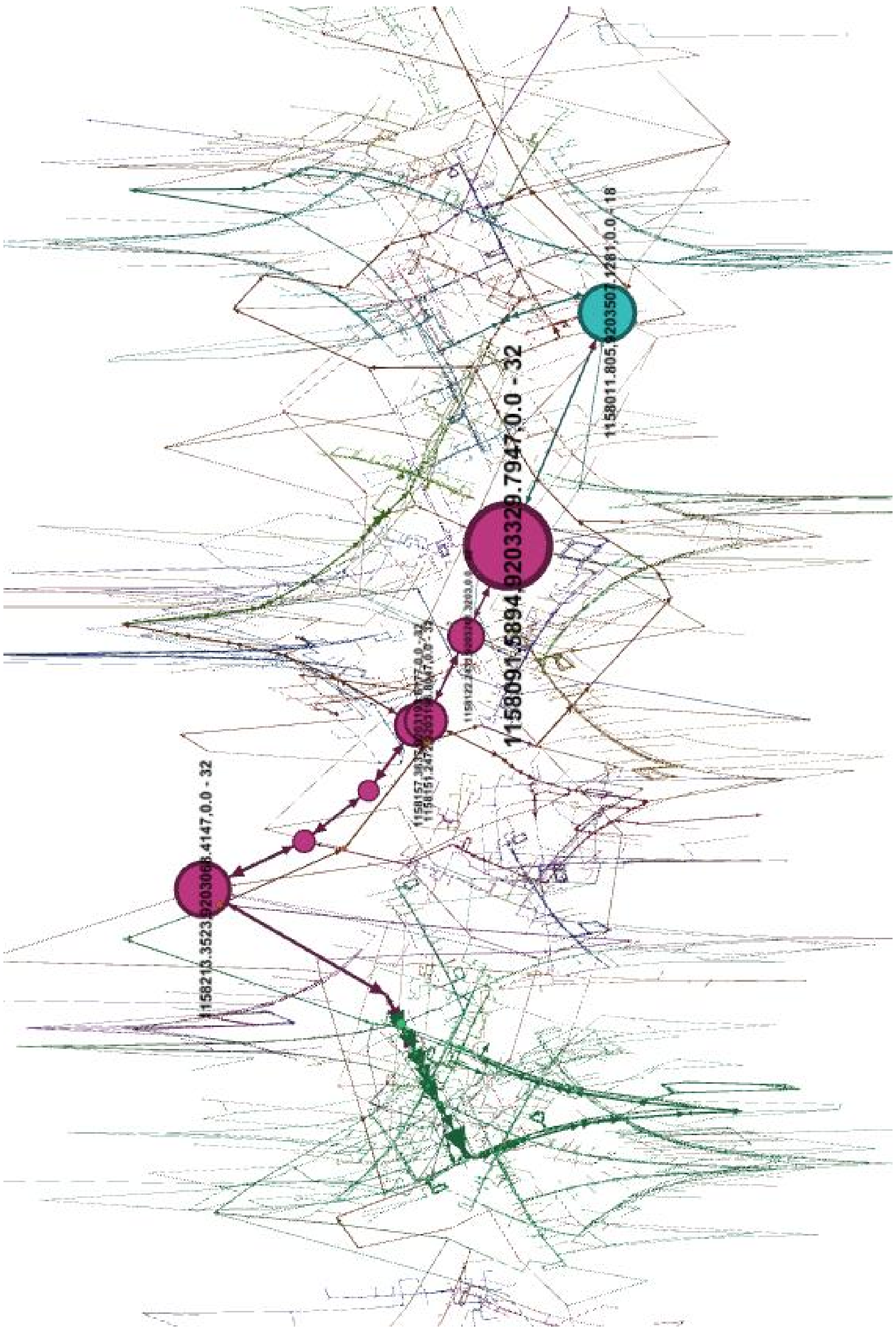


Figure 4.14: Nodes seized by their betweenness centrality.

The spline employed to determine the size is given by Figure 3.8. The highest betweenness value is 1.56 and the lowest 0. The highest values are labeled with their coordinates and building identifier.

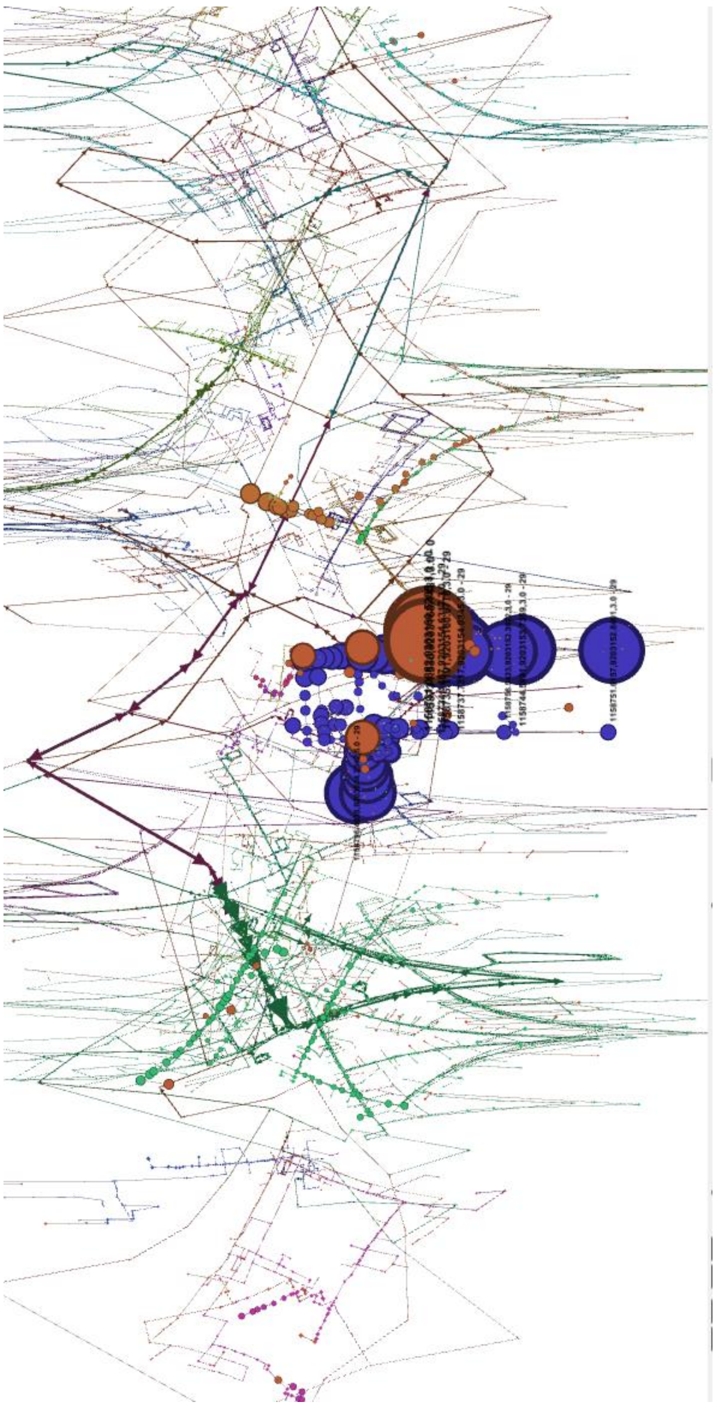


Figure 4.15: Nodes seized by their closeness centrality. The spline employed to determine the size is given by Figure 3.8. The highest closeness value is 159.53 and the lowest 1. The highest values are labeled with their coordinates and building identifier.

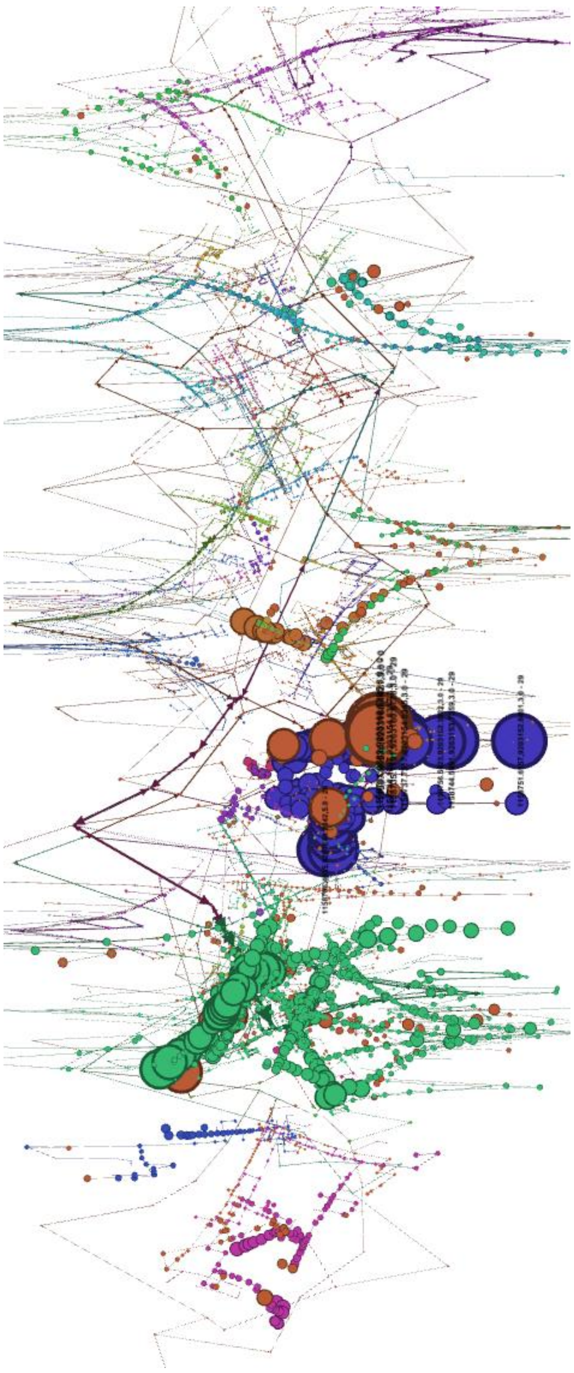


Figure 4.16: Nodes seized by their eccentricity centrality. The spline employed to determine the size is given by Figure 3.8. The highest eccentricity value is 177 and the lowest 0. The highest values are labeled with their coordinates and building identifier.

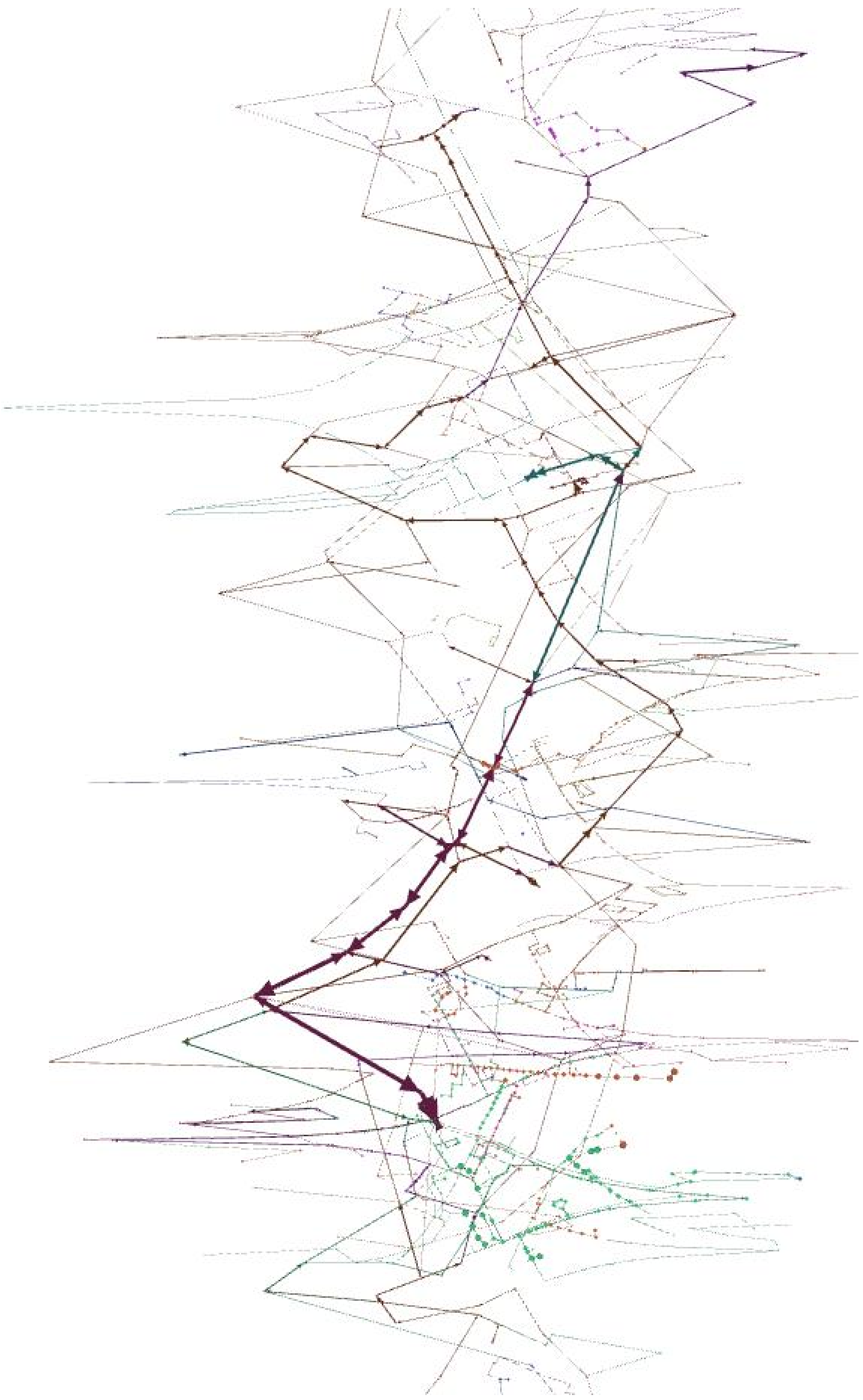


Figure 4.17: Nodes seized by their eccentricity centrality. The spline employed to determine the size is given by Figure 3.8. The highest eccentricity value is 177 and the lowest 0. This image contains the same measures as Figure 4.16 but keeping only the ground floor features (features with 'z' equal to zero).

4.4 Summary

Request distribution

Counting the number of times each target is requested showed that those inquiries are far away of being uniformly distributed among the possible destinations. We can appreciate the difference: the first in the list with about 400 requests -F2 Gamle fysikk- accumulates more than twice as much as the second does -R1 Realfagbygget- and at least more than 40 times than the 86% of the other searched targets, which group 10 or less requests.

Putting together the number of times every target belonging to the same building has been requested, we came to an irregular distribution again. As we can see in Figure 4.5, Realfagbygget has more than a quarter of the requests. If we add that number the ones that Elektro and Sentralbygg gather we have more than the half. And if we also count Fisykk and IT-byggene+syd we have -between only five of twenty three buildings- almost three quarters of all the inquiries' targets.

In addition to the irregularity between rooms and buildings in general, in the most popular buildings we also have some places much more popular than the rest. In Figure 4.6 an Figure 4.7 we can observe how some rooms or building wings are more solicited than others.

Graph structure

Every distinct segment that appeared in the followed requested routes and the number of times they did, depict the weighted arcs of the directed graph we worked with. From the arcs -directed edges- we revealed the graph's structure and from the nodes we distinguished the bridge points in the network.

Edges

Visualizing our graph with the layout 'Force Atlas 2' (Figure 4.1) allowed us to identify a highly connected core that groups most of the nodes, remaining some peripheral smaller routes not connected to it. So we get a first impression of the network's structure: most of the points are linked, leaving unconnected some outsider routes (the giant component concentrates the 93% of the graph Table 4.2).

Meanwhile, from the visualization with 'Yifan Hu' layout (Figure 4.2) we notice some graph hierarchy: several thin long routes come to join to thicker (higher weighted edges) ones as more close they get to the core. So we actually have some highways across the campus where different routes converge.
Finally, with the 'Geo Layout' (Figure 4.3) we aggregate the real location to the nodes, drawing the features keeping the campus shape. With this visualization we identify

the backbone of the network, the flow highways and the peripheral unconnected routes.

Nodes

Once we have pictured the graph structure we look into the relations that take place and the bridge points, using for that the node attributes.

After collapsing all the nodes according to the building they belong, we saw the traffic between them in Figure 4.10. This figure highlights the traffic flow between buildings.

Most of the edges lead from outdoors to the buildings' inside, being also relevant the strong flow between Realfagbygget and Sentralbygg (both also diversely high connected to others). Despite the width of the relations between buildings, some of them lack of relations with any other like 28, 54 or 53 (isolated with just one edge with outdoor), while also highly connected communities exist like 45, 76 or 37, the bridge buildings.

The representations of the degree in Figure 4.12 and Figure 4.13 bring the attention to the nodes with more neighbours: the crossroads, the points were more different possible paths come to join, the points that actually bridge the network.

Sizing the nodes by their betweenness centrality in Figure 4.14 points up possible flow constrictor places, nodes that link several routes. Majority of them are located in the Sentralbygg, and watching the long highly-weighted edges they are connected to, we can say those nodes are the main doors of the building (the floor -z- is zero). The strong (highly weighted) edges between those points suggest that that route could is a network backbone, what actually this graph structure-measure is supporting.

Finally, we identify the places where more hops are needed to get to their farthest point within the whole network in Figure 4.15, Figure 4.16 and Figure 4.17. Most of those nodes are located in the highest floors of the peripheral buildings.

Chapter 5

Location-Based Business Opportunities

5.1 State-of-the-art

5.1.1 The mobile landscape

The development of network and device technologies is ready to make mobile revolution real. Nowadays, about 35% of all the telephones are smartphones, and the penetration is expected to rise to 47% in 2015. More affordable data pricing and device convergence make mobile become more than just voice. Manufacturers and providers compete strong to place their products in this vivid market.

The device convergence started equipping the smartphones with newer sensor arrays. The revolution came deploying inertial sensors in mobile devices. Accelerometers and gyroscopes help to follow a relative position from an initial one. The accelerometer alone can't tell the difference between motion and gravity, but adding a gyroscope the devices can immediately sense the onset of motion. The next step is utilizing humidity, sound or pressure sensors to perform more accurate indoor navigation. The information these sensors collect enhance the mobile handset response, enabling augmented-reality applications [Inc10].

Multiple type of services are emerging due to the smartphones capabilities. Most of them offer a continuous service thank to the connectivity provided by wireless infrastructures. The content can be distributed either by an app permanently stored on the device, or by accessing every time to the mobile website version. The app environment is currently preferred. This over saturated market make application providers compete against each other to find users and engage them. But developers also have to care about a high number of platforms, so the standard HTML5 is likely to take off soon. And it will overcome app as a browser based solution.

Location based systems bring a growing area for m-commerce strategies. New services must anticipate and meet the needs of customers at the point they arise. They can improve the consumption experience by suggesting useful and actionable information. The real-time location of the customers allows a higher quality advertising strategy. The traces of the users can reveal their mobility patterns and behaviours. Companies can integrate that information with their own customer database, letting that location and personal knowledge trigger new challenges to attract and engage the customer needs.

#socialEra. The information age focused on the value data provided, but nowadays companies use social to stethoscope the market pulse in order to thrive. Social is already more than media, giving a new set of rules to success. Companies have to work with others, collaborate and co-create value instead of grow bigger. They have to reinforce the partnerships with other companies to provide a wider offer. But they also should listen to the social networks, monitoring them to discover the people needs and likes. Social networks bring the opportunity of integrate location information with their users profile information. People look forward to the approval of the others and we share really valuable data about ourselves in the social networks (likes, interests or past behaviour). Facebook is the biggest banner of how much we want that social validation.

Qualcomm and Cisco recently announce Indoor Location Technology collaboration to enable venues with five meters accuracy. And Foursquare holds partnerships with HBO, Bravo or Starbucks (you can earn access to daily promotions by checking in on Starbucks via Foursquare). These agreements enable third-party advertisements through promotions, multimedia downloads or games. Because "gamification" is growing popular. People enjoy playing on their hand devices because they can share their scores and interact through their social profiles (FarmVille, AngryBirds or Scrabble).

Privacy. Potential services based on the user's location can target the useful information at the right place in time. The storage cost of those large datasets is not a problem any more. The analysis of those datasets can identify consumer trends, profiles or habits. Companies will have to conduct data mining and business intelligence activities. They must squeeze the data at their disposal to draw lines in the right direction. However, the personalization of any service relies on the customers willingness to share their personal and location information [TKCS09]. So providers must earn customers trust, letting them control their own personal information. Otherwise no application would have users sharing their data.

There has to be a trade-off between the concern of privacy and the extra-value customers get in exchange. It has been proved that monetary incentives, like free services or promotions, influence consumers to disclose their data when they are reticent. For example: making free registrations to give more personal data than the paid ones, offering discounts to costumers when they give more data or any other of

the privacy-friendly business models for mobile location-based services described in [LBFP11].
The collection of data is constrained by the privacy right. Users know that companies are not collecting data unless they are planning to use them. They need to know how their data is going to be used, retained and the implications. Companies need to define clearly and resolve data ownership issues. The Cellular Telephone Industry Association published a document with some guidelines to promote and protect users privacy within the new developing location based services [Ass] [Ass10]. Those guidelines follow two fundamental principles: user notice and consent.

First, providers have to ensure that users notice meaningfully how their location information will be used, disclosed and protected. So then users can decide whether allowing the system to access that information. And secondly providers have to get the users consent to use or disclose their location data. The users must have the right to revoke or terminate the agreement any time. These guidelines apply both, mobile devices or specific personal location.

5.1.2 Market scope

Smaller and more accurate location receivers allow manufacturers to increase the number of available wireless handset devices equipped with location services. Internet mapping and navigation services emerged to cover the need of guidance through unfamiliar roads. They started as an outdoors game, continuously improving with extra features and travel information.

Despite the challenges, alternative technologies are being satisfactorily addressed by the indoor location industry to solve the GPS limitations. This industry is expected to grow multi-billion dollar revenues over next years. The potential opportunities can be reflect in the agreement launched on August 2012 [Nok12]. Twenty two giant companies (Broadcom, Nordic Semiconductor, Nokia...) want to set open interfaces and a standard-based approach next years. This alliance reflects the expectation on a growing market, they gamble for joining indoor positioning systems to mobile services enhancing consumer experience.

The activity is focusing on location consumers in major venues around the world, paying special attention to large open indoor areas. About 130 companies are working on indoor location, navigation or mapping venues [LLC12]. The industry lacks of mapping standards for indoors. But the Open Geospatial Consortium has already initiated a working group with the aim of provide a common schema (IndoorGML Indoor Location Standard). Huge companies that previously mapped outdoor areas are starting to map indoors creating guides for malls, walkways or airports (Nokia, TomTom or Google).

Amazingly, many of these popular location services come for free, for example Nokia-OviMaps or Android-GoogleMaps. Google triggered the growth when their venue maps and indoor location entered, but smaller companies shouldn't fear the competence. Google wants venues to provide their indoor maps to Google in exchange of just being included on Google maps, without sharing any of the revenues coming from advertising or business intelligence analysis. Google depends on the venues to get their indoor map rights or the allowance to map the venues.

The venues know how valuable the analytic data is, and they can be resistant to loose the control of the content and space monetization (for example personalized adds or coupons to users using the location service). And this is where the opportunity lays for non-Google companies. Venues keen on keeping some revenues from the potential data could set agreements to share the profits with any of those companies with their own developed system (something that Google is not willing to do). And anyway, there are still opportunities in a vast amount of indoor areas willing to be plotted and provide with navigation systems.

5.1.3 Revenue models

The main sources of profit come from the application sells, advertising and the payment received from provided services. Developers have to keep in mind how much the costumers would be willing to pay for the application service. A profitable application covers a customer mass by maintaining a service at reasonable cost. But the amount of downloads is not enough, an active user base is also needed. And the time window to achieve that is really small (long term audiences are on average a 1% of the downloads and only 20% of the free applications are used after the first day, and less than 5% after a month). We are going to see now different ways to recover the investment [RM04].

Companies have to decide whether offering a free download or charge a determinate fee. Only the most successful apps offer something new or addictive enough to maintain a big user community, so they get more revenue for advertising. Decreasing price is often worthwhile to increases the downloads at expense of some incomes. Appearing on the top lists also increases the downloads, but extra value has to be provided if we want costumers eager to pay for some service. We can find more details in this study [Yar09].

The subscription model is starting to be exploded by the media content providers. Consumers demand media content through all the devices they have (music, video and television). Media producers deliver high quality content for the new platforms, but they have to compensate the other affected parts of their business. Network operators bill customers the higher bandwidth consumption these services require.

This model provides access to content through any device by paying a fee. The subscription can offer a content to download or one watch on streaming. Costumers can pay a periodical fee for a whole access, or prefer 'Á la carte' model, where they pay just for singular access.

The premium model is in between the subscription and the free model. Some services are offered for free towards a large number of users to get the application well known. And if costumers want access to more features or just skip the adverts they have to pay a subscription. i.e LinkedIn gives access to extra services and Dropbox more space for premium users.

The advertising model reduces the subscription fee or just replaces it through advertising. Customers are more likely to tolerate advertising in exchange of reducing the service price. Ad-supported models are more profitable when they utilize costumer profiles to offer personalized ads that users can find useful. Advertisers do not value yet the mobile ads as much as the television ones, brands are slow to move advertising money. However, mobile advertising is growing, while television and internet are slightly dropping. The screen size of the devices constraint the advertising methods, forcing more dynamic and interactive ads. When providers have data strongly indicating success, they will get better advertising rates.

The key value from mobile advertising is to effectively target the proper audience. The growing convergence between social networks and all kind of applications reflects consumers behaviour. Behaviour patterns that marketers have been always looking forward to understand in order to provoke a response with their campaigns. Mobile platforms bring the opportunity to establish a direct contact, allowing a real-time answer. Location information can ensure offering to the right person in the right context, advertise only the promotions that are relevant to the customer. Avoid spamming is essential.

Furthermore personalized advertising, mobile devices allow costumers to share their experiences through the social networks. They share in their communities their conformance or disagreement in real-time, tagging the locations. Social networks leverage a tool where potential customers can get to know any products they want from reliable sources they know, the feedback from their own contacts.

Business travelers and tourists crave a killer venue navigation application. Venues like airports, stations or malls are growing really bigger hosting inside all kind of stores. Location interfaces allow also "check-in" features where people reflect their consumption experiences tagging the location and the time. They can share reviews of restaurants and bars, helping other customers to find the more suitable places for them. These platforms could pop up on your mobile advertising messages and collect data about customer's daily routines. That suggests pushing adds to potential customers, engaging the user at the right place in a real-time channel. Going to the cinema with your partner or looking for new sneakers are really different contexts, even if both take place in the same corridor of the mall.

Providers acquire traces over the time that can be used to improve the own service or to sell as an asset to other companies. The investments made on developing a location system can generate revenues from the potential services born from the collected data. Costumers will change from their "sit and search" by their initiative on their personal computers, to a "roam and receive" provided by the servers. Costumers interested in a certain clothes trend could receive alerts or promotion notifications in a venue when they are passing by the shops(timely and relevant info at a point ready to consume, a dream of every brand).

Nevertheless, these systems won't really boom until they rely on trustful "check-ins". There has to be differences between the comments written 'on the ground' and those written afterwards, highlighting those written by the own contacts. Marketers and retailers in venues can be really interested on location based services. They could offer the navigation service for free expecting revenues from the customers consumption due to the new advertising strategy.

The brand supported model is highly correlated to the previous one. Big brands are highly interested on collecting data from their customers, so they have to find some service to offer in exchange. For example, Starbucks offer promotions rewarding loyalty through the social networks and location features. Disney, Sony or Adidas have their own apps in the top markets [Dis12]. Brands want to reach big active communities. They need 'sticky' applications that make the users spend more time with them. They incentive people to keep switched their location service without alienate them. That way brands support economically the application expecting revenues from potential customers. These marketing strategies bridge the gap between brands and technology providers. The purpose is to obtain data without alienate customers.

We have mostly talked about advertising in the costumer mobile devices on the ground in order to enhance them to consume. And the next step is closing the loop allowing them also to pay with their devices. If costumers are comfortable making purchase on internet, it stands to reason that mobile could follow the same path. M-payment techniques like NFC are likely to allow catching the whole transaction. Partnering with banks means investments increase, and providers can demonstrate how valuable are their marketing campaigns through the real impact they have over the clients.

The ecosystem problem -not technology- requires the cooperation of a loot of different actors. Compared to plastic cards that companies control, mobile is much of an open platform where many pieces of the puzzle are controlled by more than one entity. If a bank company chooses a partnership with just one operator, what will happen with costumers from other operators? There is a need to get standards to trigger this market over than PayPal or bank transactions through internet browser.

5.2 CampusGuiden

5.2.1 General guidelines

Every PC, tablet or smartphone user can search for any point of interest at NTNU Gløshaugen campus and the CampusGuiden application will guide them to get there. The main work of this thesis was analyze the traces of this navigation app Section 4. Before finding practical employments to those outcomes, we are going to bring some lines of action that improve the service and try to generate more revenues.

The popularity of the application reveals how big is the users community. In addition to count the downloads, collect the requests reports when the sessions took time. Measure how long are those sessions tells how much time do we have for advertising or suggesting any further activity. These measures proves not only how many users are interested on the app but also the real interest it arises (number of downloads vs number and duration of the requests).

Accuracy and quality of the maps are key issues regarding customers engagement. In addition to keep updated the maps, we also see convenient making the maps more interactive. For example allowing users to save favourite locations or a history record with places they use to go. Manual location inputs like streets intersection or room codes are also useful when you want to consult a route. New functionalities can informing about free spots on the parking areas, study rooms or an approximation of gym occupancy. People would definitely check how occupied is the library or the gym before going there for nothing. These kind of data add extra value to the application. It makes it useful not only for people looking for navigation tools but also for people who actually knows well the campus and may use the app just because of these extra features.

If the application really engages the users community, adding extra services, big branches or local companies can be interested on support the application with advertising. In order to make the application more "sticky" we need to keep costumers updated, offering navigation and campus related services together. Then we can go for better advertising rates or partnerships.

5.2.2 Partnerships

The current mobile market require partnerships. In the social age we are, small companies must work together in order to improve their services and enhance their reachability. The wider services provided, the more people can find the application interesting. Once we have data strongly indicating success we can arrange pleasing deals with other entities.

The killer applications merge their services with the social networks. We can let users know where their friends are by sharing their location inside the campus. CampusGuiden should integrate with social networks, specially with Facebook. The application would be more interactive in a community if the users know where the people are or where are planning to go, sharing also their routes. We could even allow users to tag different rooms and share those tags with their friend (for example tag a group of friends in a room with “Guys we are gonna meet here at 3 p.m.”). Foursquare, Facebook or Twitter already support already geo-tagging but not in such an accurate way. This is something completely new that no other application offers. The future of mobile applications comes through integration with other services and social networks.

We can create a costumer profiles database where users could subscribe the kind of events they are interested on. When those events take place the application can alert the subscribers tagging the location in the campus where they will happen. This event subscription service can be launched partnering with the university, NTNU. CampusGuiden would tag the location and send the events through the users mail, Facebook or Twitter profiles. That way users can share it at the same time with their own group of friends. This service will help NTNU to inform about the events and spread the location through the social networks. We have a chance to keep engaged the student and worker community during their stay in Gløshaugen by adding also the information about parking, gym or room occupancy we already proposed. If SiT provide us information about gym occupancy, we could publish it tagged in the corresponding gym location in the CampusGuiden map. For people who like to work out there is no more annoying thing that finding crowded the gym. This service would bring a lot of daily subscriptions. We can arrange more interesting partnership with SiT. We have some data from the traces that they could find interesting, for example the times the cafés were requested (36 times: Realfabygget-29 and Kjelhuset-7) and the kiosks (13 times, all in the Sentralbygg). Another subscription service could tag with CampusGuiden the daily menu or offer to every cafés locations. Allowing also the users to share this on their social networks. Since the service is for free, we can ask the users for permission to share some information from their profiles to set the basis for a potential personalized advertising system. Important branches or entertainment marketers interested on catching students attention can launch periodically promotions to the right public, taking advantage of that data. But these services require a deeper analysis first. Despite the advertising revenues, we don't want to scare the users. A previous market acceptance survey is needed before launching these advertising campaigns. The key issue is keeping users on-line. It helps to build a socially relevant consumer community without alienating the people we want to monitor. In addition to the previous services, we can launch contests or advertising campaigns rewarded with coupons, discounts or promotions at the SiT Kafes [Med12]. Better rewards like free tickets can be sponsored by local restaurants or cinemas. These social games enhance the people to play and stay connected having fun, even more if they can interact and share their score in their social profiles (i.e. AngryWords, FarmVille or AngryBirds in Facebook).

5.2.3 Traces

With the graph and requests analysis we have created a database with the most interesting points to offer information, launch new promotions or test new services. We know the places where we can target more audience and those we should avoid thank to the structure of the mobility patterns. It is interesting keep on doing these analysis with traces from different periods to check if the outcomes come similar or they change due to seasons. If the behaviour change according to the season, we would have to use different strategies depending on the month of the year.

The main idea that emerge from the analysis of the traces is the application adaptivity across the campus. How it has been accepted in the different buildings. Which are the main roads and points where the information spread faster. When in Section 4 we analyzed the graph metrics, we already exposed how they revealed the mobility patterns of the users. Now we will highlight some of those outcomes again, purposing marketing utilities.

We utilize the hierarchical data analysis we carried out Table 5.1. First we differ between inter and intra buildings. And we reveal the acceptance across each category under two points of view. By the number of times we suppose people have been there, unfold by the popularity of the targets. And by the inner structure of the network, the strategic situation unfold by the centrality measures.

Popularity: inter-building. The amount of requests targeting any room inside a particular building inform us about the popularity of the application there Figure 4.5. In the buildings with less percentage we can advertise our app more to increase the number of users. On the other hand, the popular buildings reflect the acceptance the users have about new features. So those are places eager to host new releases since people seem more active with new technologies. The campaigns should be advertised there because people use our services, so they are more likely to spread the word.

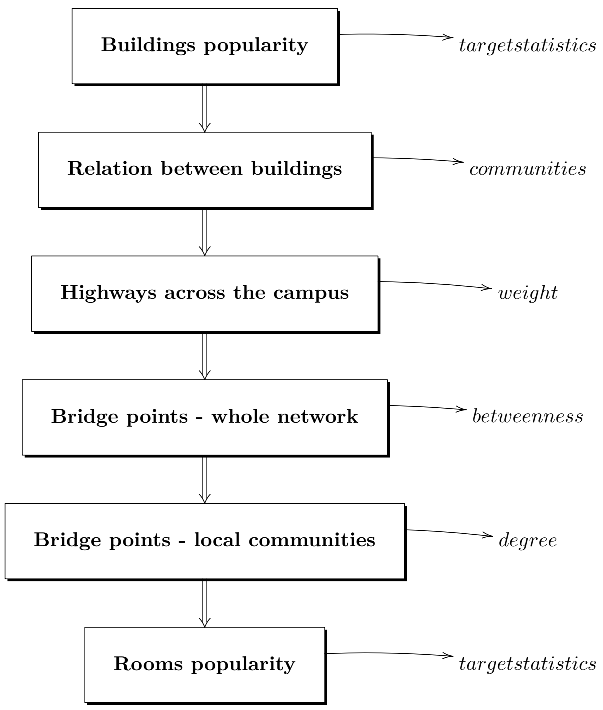

Buildings popularity
targetstatistics
Relation between buildings
communities
Highways across the campus
weight
Bridge points - whole network
betweenness
Bridge points - local communities
degree
Rooms popularity
targetstatistics

Table 5.1: Hierarchy of outcomes from network analysis.

Popularity: intra-building. Looking inside each building we found substantial differences between the number of times the rooms were requested. For example, the International house had only five requests. The international students may don't know about CampusGuiden. And at the beginning they probably don't know how to move across the campus either. So we recommend to intense the advertising campaigns launched for them. The international office could include also info about CampusGuiden when they provide the students the info they need. The popularity of the other rooms may interest NTNU, the head departments may consider that info at the time of schedule and place the courses. As well as SiT may also use the popularity of their facilities. We already proposed those partnerships.

Structure: community. The visualization of the collapsed communities let fall how the users moved between the buildings Figure 4.10. The biggest flows come to Realfabygget and Sentralbygg from the outside, being the traffic in the opposite direction lower, but not negligible. Sentralbygg and Realfagbygget are also strongly connected, they both gather many requests and a trending path between them.

Some buildings are, although weekly, at least connected to more buildings. While others are just connected to the outside. Those buildings which are less connected to the rest could remain 'like-minded' against what happens in other buildings. So events launched in other buildings have to be advertised also in buildings like 28, 54 or 53 that have only one edge if we want more people to know.

On the other side, highly connected buildings like 45,76 or 37 bridge different communities. So they are more likely to spread information to their neighbours. We can settle advertisement campaigns, organize events or test new services in the main hall of those buildings. They are crossroads where is easier to come or arrive just by chance. The campus distribution makes more people pass through these buildings when they want to go anywhere else.

Highways. Looking now the routes employed, we distinguish in Figure 4.1 and Figure 4.2 a highly connected core and a smaller periphery disconnected. But it is in Figure 4.3 where we notice that in the 'heart' of the campus, thin routes lead to a very thick ones. Those thick edges shape the backbone of the campus. The major of the people go everyday through these highways, so we can reach the most public here.

Structure: inter-building. The weighted graph represents the structure of the mobility patterns. No matter the weight, the structure of the main routes is quite static because the streets and corridors of the campus don't change their location. To find the points that bridge the campus we take the centrality metrics. The betweenness exposes the nodes that are junctions for communication in the whole network, the more influential ones Figure 4.14. Most of them are located in Sentralbygg connected to highly-weighted edges, these points must be the main doors of the ground floor. These nodes flow constrictors because keep several locations linked, this metric captures brokerage.
The ground floor of the Sentralbygg is a key meeting-place in Gløshaugen. Besides the flow constrictor role and the highly visited main corridor, is well connected to other buildings.

With closeness centrality though, we measure how far away the rest of the network is from a node. Instead of setting constraints (betweenness) or spreading information (degree), this notion looks for scope interaction, seen as access to more resources. The nodes with the most difficult access to the rest of the network makes them the last meeting points Figure 4.15. Not only because their slow access to the rest, but for being slow to come there (we can see that they refer to high floors). Those are places to avoid if we want many people to reach them easily. The eccentricity gives us an idea of the most peripheral places in the campus Figure 4.16. Points to avoid too, because they are located in peripheral buildings with several floors, most of them in the highest floors.

Structure: intra-building. The paths are bidirectional in the space we are modeling. That is why so no matter the followed direction, the points with the highest degree values are the crossroads where more pathways leave or lead: Figure 4.12 and Figure 4.13. These points are more likely to be visited in a community, the best points for example to stick a banner in the wall.

5.2.4 Extrapolation: from Gløshaugen to other venues

CampusGuiden can be exported to other venues like malls or airports. An indoor positioning system increases the value of the assets, bringing a platform where social network data can converge with the customers mobility patterns. In addition to navigation services, we can exploit again the possibility of tagging the events location and share them through the social networks. Companies can provide extra personalized services thank to the collected data. The key is overtake the competence with a rapid adoption system. CampusGuiden is an already developed and working tool, so if it wants to stay on the cusp of the wave, the application needs to boost new services and expand to more fields of action like different venues.

CampusGuiden could create new relationships between indoor navigation and customers likes and behaviours. We can collect and analyze the traces of an actively engaged community, as well as their likes reflected by their subscriptions. That information may help to place events, detect mobility patterns or consumer preferences. Big brands and retailers are keen on these valuable assets to improve their marketing campaigns.

Chapter 6 Conclusion

We have found the mobility patterns we were looking for in the CampusGuiden traces. First, we obtained statistics about the building popularity among the users, helping to gather insights about where the application had more penetration. Afterwards, we obtained the whole followed routes from the user requests, and, after decomposing them into small segments, we created a weighted directed graph representing all mobility data. We verifies the resulting graph made up by those segments in two different ways. First, we compared the representation to a ground truth of available paths; they matched. Second, we saw how the metrics of the graph accomplished the graph degree rules.

We experimented with different centrality metrics and graph visualizations in order to expose the bridge points and most used highways at campus. Request popularity and structural bridge points can enable future strategies; first, one should increase advertisement campaigns of Campusguiden in places where the application is not well-used. Second, spreading information physically in campus (fliers, magazines) should be done in places whose location make them more likely to be visited by many people; according to mobility patterns and popularity.

After giving a look at the mobile location based market, we suggest to integrate the application with social networks. Users would interact with the maps tagging information and sharing locations and tags through social networks. We also propose the creation of a customer profile database, allowing the users to subscribe to events they are interested in; these events then would have their location tagged in Campusguiden. In order to improve on the usefulness of Campusguiden, it would be really beneficial to provide information about parking, libraries and gym occupancy. Furthermore, to raise and maintain awareness in the user community, we suggest weekly contests rewarded with coupons which could used on campus (SiT cafeterias, bookstore) or downtown (stores).

These potential improvements naturally need a deeper viability study. In a shorter term, we could measure how long Campusguiden user sessions are active. Moreover, we could compare the mobility patterns in different seasons to check if and how they change. It would be also interesting to classify the users into several groups (e.g., professors, first year students, staff, international students) based on their individual mobility patterns; this could enable more targeted advertising campaigns and services. We leave these challenges open for future studies.

References

[Ass] Cellular Telephone Industry Association. Examples of Location Based Services Privacy Policies . "http://www.ctia.org/business_resources/wic/index.cfm/AID/11924".

[Ass10] Cellular Telephone Industry Association. Best Practices and Guidelines for Location Based Services . "http://www.ctia.org/consumer_info/service/index.cfm/AID/11300", 2010.

[BGLL08] V.D. Blondel, J.L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, 2008.

[BHJ09] Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy. Gephi: An open source software for exploring and manipulating networks. 2009.

[Bre97] P. Brenner. A technical tutorial on the ieee 802.11 protocol. *BreezeCom Wireless Communications*, pages 1–24, 1997.

[CNM04] A. Clauset, M.E.J. Newman, and C. Moore. Finding community structure in very large networks. *Physical review E*, 70(6):066111, 2004.

[Con12] Gephi Consortium. Official tutorials. "http://gephi.org/users/", 2012.

[CPIP10] Krishna Chintalapudi, Anand Padmanabha Iyer, and Venkata N. Padmanabhan. Indoor localization without the pain. In *Proceedings of the sixteenth annual international conference on Mobile computing and networking*, MobiCom '10, pages 173–184, New York, NY, USA, 2010. ACM.

[Dev12] NetworkX Developers. "http://networkx.lanl.gov/", 2012.

[Dis12] Distimo. "http://www.distimo.com/publications/archive/Distimo%20Publication%20-%20October%202012.pdf", 2012.

[EK10] D. Easley and J. Kleinberg. *Networks, crowds, and markets.* Cambridge Univ Press, 2010.

[GKH09] Emden Gansner, Stephen Kobourov, and Yifan Hu. GMap: Visualizing Graphs and Clusters as Maps, 2009.

[GLN09] Y. Gu, A. Lo, and I. Niemegeers. A survey of indoor positioning systems for wireless personal networks. *Communications Surveys & Tutorials, IEEE*, 11(1):13–32, 2009.

[Hal11] Christian Halvorsen. Campusguiden : En navigasjonstjeneste for innendørs bruk, 2011.

[HVW09] D. Holten and J.J. Van Wijk. Force-directed edge bundling for graph visualization. In *Computer Graphics Forum*, volume 28, pages 983–990. Wiley Online Library, 2009.

[IEE11] IEEE. Ieee standard for information technology– local and metropolitan area networks– specific requirements– part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications amendment 8: Ieee 802.11 wireless network management. *IEEE Std 802.11v-2011 (Amendment to IEEE Std 802.11-2007 as amended by IEEE Std 802.11k-2008, IEEE Std 802.11r-2008, IEEE Std 802.11y-2008, IEEE Std 802.11w-2009, IEEE Std 802.11n-2009, IEEE Std 802.11p-2010, and IEEE Std 802.11z-2010)*, pages 1 –433, 9 2011.

[IH92] M.D. Irwin and H.L. Hughes. Centrality and the structure of urban interaction: measures, concepts, and applications. *Social Forces*, 71(1):17–51, 1992.

[Inc10] IHS Inc. Games and Navigation Driving Rapid Gyroscope Growth in Mobile Handsets . "http://imsresearch.com/news-events/press-template.php?pr_id=1500", 2010.

[ISO07] ISO. "http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=32554", 2007.

[LBFP11] Z. Liu, R. Bonazzi, B. Fritscher, and Y. Pigneur. Privacy-friendly business models for location-based mobile services. *Journal of theoretical and applied electronic commerce research*, 6(2):90–107, 2011.

[LDBL07] H. Liu, H. Darabi, P. Banerjee, and J. Liu. Survey of wireless indoor positioning techniques and systems. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 37(6):1067–1080, 2007.

[LLC12] IndoorLBS LLC. Accurate Mobile Indoor Positioning Industry Alliance, called In-Location, to promote deployment of location-based indoor services and solutions. "http://www.indoorlbs.com/p/market-report.html", 2012.

[Med12] Millennial Media. "http://www.millennialmedia.com/mobile-intelligence/smart-report/", 2012.

[Nok12] Nokia. Accurate Mobile Indoor Positioning Industry Alliance, called In-Location, to promote deployment of location-based indoor services and solutions. "http://press.nokia.com/2012/08/23/accurate-mobile-indoor-positioning-industry-alliance-called-in-location-to-promote-deployment-of-location-based-indoor-services-and-solutions/", 2012.

[Par] D. Paranyushkin. Identifying the pathways for meaning circulation using text network analysis.

[RM04] B. Rao and L. Minakakis. Assessing the business impact of location based services. In *System Sciences, 2004. Proceedings of the 37th Annual Hawaii International Conference on*, pages 8–pp. IEEE, 2004.

[SDS09] T. Sutton, O. Dassau, and M. Sutton. A gentle introduction to gis. *Chief Directorate: Spatial Planning & Information, Eastern Cape*, 2009.

[TKCS09] J. Tsai, P. Kelley, L. Cranor, and N. Sadeh. Location-sharing technologies: Privacy risks and controls. TPRC, 2009.

[VWG$^+$03] M. Vossiek, L. Wiebking, P. Gulden, J. Wieghardt, C. Hoffmann, and P. Heide. Wireless local positioning. *Microwave Magazine, IEEE*, 4(4):77–86, 2003.

[Yar09] Greg Yardley. 'AppStore Secrets', What we've learned from 30,000,000+ downloads. "http://www.slideshare.net/pinchmedia/iphone-appstore-secrets-pinch-media", 2009.

Appendix A

Target Popularity

f2 341	R1 Realfagbygget 181	R10 Realfagbygget 123
H3 Hovedbygningen 111	F1 IT-syd 98	Lesesal Gul Sentralbygg 1 79
EL5 Gamle elektro 78	R7 Realfagbygget 71	R2 Realfagbygget 65
F2 Gamle fysikk 64	Søk etter bygning eller rom 61	K25 Kjemiblokk 3 50
R90 Realfagbygget 49	H3 49	K27 Kjemiblokk 1 48
R5 Realfagbygget 44	Realfagbiblioteket Realfagbygget 44	F6 Gamle fysikk 43
R20 Realfagbygget 43	ITS204 IT-syd 41	1 Høgskoleringen 3 (P15) 41
Trådløse Trondheim IT-syd 40	ITS206 IT-syd 40	R52 Realfagbygget 40
H1 Hovedbygningen 40	KJEL1 Kjelhuset 40	EL23 ElektroE/F 39
B23 Berg 39	212 Sentralbygg 2 36	G022 Gamle elektro 35
EL4 ElektroB 34	K26 Kjemiblokk 4 34	Hangaren Sentralbygg 1 33
S2 Sentralbygg 1 32	S21 Sentralbygg 2 31	GK1 Gamle Kjemi 30
S22 Sentralbygg 2 30	EL6 Gamle elektro 30	R60 Realfagbygget 30
S23 Sentralbygg 2 29	R80 Realfagbygget 29	SiT Kafe Realfag Realfagbygget 29
L11 Byggtekniske laboratorier 28	R93 Realfagbygget 28	R91 Realfagbygget 27
K5 Kjemiblokk 5 27	EL1 Gamle elektro 27	324 Høgskoleringen 3 (P15) 26
R8 Realfagbygget 25	B21 Berg 25	B22 Berg 25
SÃ¸k etter bygning eller rom 24	S8 Sentralbygg 2 24	B1 Berg 24
KJEL21 Kjelhuset 23	R9 Realfagbygget 22	G1 Geologi 22
EL3 Gamle elektro 22	K24 Kjemiblokk 3 21	S5 Sentralbygg 2 21
F1 21	R3 Realfagbygget 21	000 Hovedbygningen 21
001 Gamle fysikk 21	R21 Realfagbygget 20	VA2 Varmetekniske laboratorier 20
S24 Sentralbygg 2 20	F4 Gamle fysikk 20	S1 Sentralbygg 1 19
127 Sentralbygg 1 18	000 Gamle Kjemi 18	100 Produktdesign 18
Drivhuset IT-syd 17	R73 Realfagbygget 17	Find building or room 17
G034 Gamle elektro 17	Zevs Byggtekniske laboratorier 17	454 IT-bygget 17
R92 Realfagbygget 16	R54 Realfagbygget 16	002 Realfagbygget 16
G038 Gamle elektro 16	S7 Sentralbygg 2 16	B2 Berg 16
EL2 Gamle elektro 16	A168 ElektroA 16	C3-107 Realfagbygget 16
TrÃ¥dlÃ¸se Trondheim IT-syd 15	B113 ElektroB 15	R57 Realfagbygget 15
Hangaren 15	F2 14	R8 14
R59 Realfagbygget 14	A3-100 14	S3 Sentralbygg 1 14
F3 Gamle fysikk 14	KJEL2 Kjelhuset 13	G112 Gamle elektro 13
A172 ElektroA 13	S4 Sentralbygg 1 13	054 IT-bygget 13
R81 Realfagbygget 13	B3 Oppredning/gruvedrift 13	R50 Realfagbygget 13
205 Kjemiblokk 4 13	008 - IT-syd 13	KJL1 Kjelhuset 13
D240 Elektro D B2 12	ITS204 - IT-syd 12	1056 Sentralbygg 2 12
D1-161 Realfagbygget 12	D1-210 Realfagbygget 12	G144 Gamle elektro 12
G238 12	02 Vannkraftlaboratoriet 12	C3-113 Realfagbygget 12
311 HÃ¸gskoleringen 3 (P15) 12	E404 ElektroE/F 11	100 11
242 IT-bygget 11	SiT Storkiosk Sentralbygg 2 11	D1-148 Realfagbygget 11
D429 Elektro D B2 10	B25 Berg 10	G122 Gamle elektro 10
B145 - Elektro D B2 10	314 Høgskoleringen 3 (P15) 10	546 10
H2 Hovedbygningen 10	S6 Sentralbygg 2 10	G21 Geologi 10
311 Høgskoleringen 3 (P15) 10	D238A Elektro D B2 10	DU2-170 Realfagbygget 10
203 Gamle fysikk 10		

Table A.1: All the Gløshaugen rooms requested by the users at least ten times.

Appendix B Python Programs

B.1 readTrackposition.py

```
#!/usr/bin/python -tt

import sys


__author__="Santiago␣Diez␣Martinez"
__email__="santydm2002@hotmail.com"


"""Read the file of location requests

First:
        Read trackpostion-31072012.csv and try to store each
            field of the request in a dictionary.
        requests [time][lon][lat][z][q][tonodeid][qtype]

Then store each kind of request in a different file based on
    the qtype.

"""



def readFile(filename):
  """Stores the fields of each request in a dictionary"""

  #Create a list to store all the requests in tupples with
      the common format
  listOfRequests = []
```

```
request=('time','lon','lat','z','q','tonodeid','qtype')
inputFile=open(filename,'rU')
for line in inputFile:
  # In order to delete the last \n of each line (we also
      separate each field between ;
       fields=line[:-1].split(';')
       request=fields
       listOfRequests.append(request)

inputFile.close()

#Create thre new different lists to store different kind
    of requests depending on the field qtype
listOfGeopos = []
listOfSearch = []
listOfObjectsearch= []
#Create also a list with the requested targets to make
    statistics.
listOfTargets=[]

for i in listOfRequests:
       if i[6]=='geopos': listOfGeopos.append(i)
       if i[6]=='search':
         listOfSearch.append(i)
         listOfTargets.append(i[4])
       if i[6]=='objectsearch': listOfObjectsearch.append(i
           )

#Create a file.txt to store each kind of list
# GEOPOS
#printFile('geopos',listOfGeopos)
# SEARCH
#printFile('search',listOfSearch)
# OBJECTSEARCH
#printFile('objectsearch',listOfObjectsearch)

#Print a file with the targets in order to highlight the
    most visited ones
buildings={}
for target in listOfTargets:
  word=target.split('␣')
```

```
        for i in word:
                if i not in buildings: buildings[i]=1
                else: buildings[i] += 1
      orderBuildings=sorted(buildings.items(),key=getCount,
         reverse=True)
      outputFile=open('buildingsSearched.txt','w')
      for i in orderBuildings: print>>outputFile, i[0],i[1]
      outputFile.close()


# Function to store each kind of list
def printFile(type,listToPrint):
        """Stores in a new file the list passed as a parametre,
           that list contains  qtype = type requests"""
        name = raw_input("Please give a file name to store "+
           type +" requests: ")
        name = name + '.txt'
        print name
        outputFile= open(name,'w')
        #a mode writes at the end of the file an creates it if
           it didn't exit
        #w mode over writes everything
        for item in listToPrint: print>>outputFile, item
        print>>outputFile, '\n \n \n'
        # Print also at the end of the file the different
           destinations required and number of times if type !=
           objectsearch
        if type is not 'geopos':
          dictionary = {}
          for i in listToPrint:
                if not i[4] in dictionary:
                        dictionary[i[4]]=1
                else:
                        dictionary[i[4]]= dictionary[i[4]] + 1
              # Sort them so the big counts are first using key=
                 get_count() to extract count.
          orderTargets = sorted(dictionary.items(),key=getCount,
             reverse=True)
          for j in orderTargets: print>>outputFile, j[0], j[1]
        outputFile.close()
```

```
def getCount(word_count_tuple):
  """Returns the count from a dict word/count tuple -- used
      for custom sort."""
  return word_count_tuple[1]


def main():
  if len(sys.argv) != 2:
    print 'usage: ./readTrackposition.py file'
    sys.exit(1)

  filename = sys.argv[1]
  readFile(filename)

if __name__ == '__main__':
  main()
```

B.2 writeQueries.py

```
#!/usr/bin/python -tt

import sys
import urllib
import urllib2

__author__="Santiago Diez Martinez"
__email__="santydm2002@hotmail.com"


"""Write the proper queries for the server from the "search"
      qtype requests from the dump

First:
            Read a file and try to store each field of the
                request in a dictionary.
            requests [time][lon][lat][z][q][tonodeid][qtype]

            Then create a query per request with the proper
                gramar to send the campusguiden server.
            After sending each url save the geoJSON resultant
                string in another file.
```

```
"""


def readFile(filename):
  """Stores the fields of each request in a dictionary"""

  #Create a list to store all the requests in tupples with
     the common format
  listOfRequests = []
  request=('time','lon','lat','z','q','tonodeid','qtype')
  inputFile=open(filename, 'rU')
  for line in inputFile:
    # In order to delete the last \n of each line (we also
        separate each field between ;
         fields=line[:-1].split(',')
         request=fields
         listOfRequests.append(request)

  inputFile.close()
  outputformat=[]
  #Write the proper queries to the server with the given
     format.
  for i in listOfRequests[1:]:
    string='https://app.campusguiden.no/routing?type=
        shortestpath&source_lon='+i[1]+'&source_lat='+i[2]+'&
        source_z='+i[3]+'&source_proj_type=ll4326&target_poi=
        '+i[5]+'&guid=test'
    outputformat.append(string)
  # for j in outputformat[:5]:
        # print j
  printFile(outputformat)
  return outputformat


# Function to store a list into a file.
def printFile(listToPrint):
  """Stores in a new file the list passed as a parametre"""
  outputFile= open('requests.txt','w')
  #a mode writes at the end of the file an creates it if it
     didn't exit
  #w mode over writes everything
```

```
    for item in listToPrint: print>>outputFile, item
    outputFile.close()

def connect(requests):
    """ Connects to each url (query) saving the returning
        geoJSON string. """
    outputFile=open('pathsGeoJSON.geojson','a')
    print len(requests)
    contador=0
    for i in requests[:]:
        try:
            f = urllib2.urlopen(i)
            text=f.read()
            # print f.readline()
            print>>outputFile,text
            f.close()
        except urllib2.HTTPError, e:
            print "Error"
            print>>outputFile,'HTTPERROR'+i
            print e.code
        except urllib2.URLError, e:
            print "Error"
            print e.reason
            print>>outputFile,'HTTPERROR'+i
    outputFile.close()




def main():
    if len(sys.argv) != 2:
        print 'usage:␣./writeQueries.py␣file'
        sys.exit(1)

    filename = sys.argv[1]
    #Write the queries.
    requests=readFile(filename)
    #Connect to the urls and save the results.
    connect(requests)

if __name__ == '__main__':
  main()
```

B.3 createGraphDictionary.py

```
#!/usr/bin/python -tt

import sys
import urllib
import urllib2
import re


__author__="Santiago␣Diez␣Martinez"
__email__="santydm2002@hotmail.com"

""" This program reads the GeoJSON features obtained from
    the querys to the location server.
        Read a file and try to store each feature in a
            dictionary, increasing a weigth every time the
            segment appears.
        path [x1][y1][z1][x2][y2][z2][weight]

        We also will deal with the stairs, creating new
            segments with them, and splitting them in the
            smallest possible hops.
    We will create also new segments in order to keep linked
         a route when a jump takes place from the outside to
        a building without a stair, but yet changing 'z'.
"""




def readFile(filename):
   """Stores each segment in a dictionary"""

  #Create a list to store all the requests in tupples with
     the common format
  segment=('x1','y1','z1','x2','y2','z2','buildingId')
  stair=('floor','stairsDirection','x','y','z','buildingId')
  listOfSegments = []
  listOfSegments.append(segment)  # Contain segments
  listOfStairs = []
```

```
listOfStairs.append(stair)  # Contain stairs

listOfEdges=[] # Store all the edges, without splitting
    the stairs in steps (new list including stairs and
    segments: segments + stairs withoutSplit)
auxiliarStairs=[]
auxiliarStairs.append(segment)# Store the edges that come
    from stairs.

# We store in paths all the segments, no matters what
    request they belong to.
# We have every path from each route separated by floor.
inputFile=open(filename, 'rU')
# The regular expresion: '"type":"Feature", "id":".*' is
    the begining of each GeoJSON feature (corresponding to
    a path in the same floor)
paths = re.findall(r'"type":"Feature",␣"id":".*',inputFile
    .read())
printFile(paths, 'pathsFromGeoJSONFile') # Store all the
    paths in a file for debbuging.

# Now we want to process each line
# In each path we take the common floor and building for
    all the points in the path.
# Then we take every pair of coordinates as a tupple (lon,
    lat).
# If there is a stair in the path it is included as an
    extra point. We can notice that thanks to '
    stairsDirection'.
# Then that point will appear twice -the end of the stair
    + begining of the path in the current floor.
for path in paths[:]:

  buildingId = 0 # When the path has no assigned building
      we are going to use this reference -> outside.
  floor=re.search(r'"floor":"\d"',path)
  building=re.search(r'"building_id":"\d+"',path)
  stairs=re.search(r'"stairsDirection":␣"[-\d\.]+"',path)
  points = re.findall(r'([\d\.]+),([\d\.]+)',path)
  if floor:
    z = floor.group()
    high=re.search(r'\d',z)
    if high: coordinateZ=float(high.group())
  if building:
    buildingSearch = building.group()
    buildingNumber=re.search(r'\d+',buildingSearch)
    if buildingNumber:
      buildingId=buildingNumber.group() #if we have a
          buildingId in the path we replace 0 with the
          actual one.
  if stairs:
    stairs=stairs.group()
    stairsDirection=re.search(r'[-\d\.]+',stairs)
    if stairsDirection: stairsDirection=float(
        stairsDirection.group())
      #
          ##########################################################################
              STAIRS
      # If we have an stair as starting point, that point
          will appear two times in points.
  if stairs:
    for point in points[0:1]:
      stairLong= point[0]
      stairLat= point[1]
    newStair=(z,stairsDirection,stairLong,stairLat,
        coordinateZ,buildingId) # Stairs saved for
        debugging.
    listOfStairs.append(newStair)
    points = points [1:] # We delete the repeated point.

        # Create a new segment, edge, from this stair.
        # This stairs coordinates represent the user start
            point in the current floor -the end of the
            stairs.
        # So we will take the last point from the previous
            path (in the different floor) as the begining
            of the stairs.
        # The floor indicates the actual floor, the
            stairsDirection is the action that has been
            followed (going up/down X floors).
    # First point is the last in previous path, and second
        point is the starting point after the stair.
        lastPoint=listOfEdges[-1]
        x1= lastPoint[3]
        y1= lastPoint[4]
        z1= float(coordinateZ) - float(stairsDirection)
        newSegment=(x1,y1,z1,stairLong,stairLat,coordinateZ,
            buildingId)
        listOfEdges.append(newSegment)
        auxiliarStairs.append(newSegment)
    #
      ################################################################################
          SEGMENTS
    # Now we store every segment that shapes the path.
        # In case we have a segment that is not linked
            because a change of reference inside buildings
            when coming from outside
        # (different 'z' without stairs, but exactly the
            same lon and lat)
    if len(listOfEdges)>1 and points[0][0]==listOfEdges
      [-1][0] and points[0][1]==listOfEdges[-1][1] and
      points[1][0]==listOfEdges[-1][3] and points[1][1]==
      listOfEdges[-1][4] and not coordinateZ==listOfEdges
      [-1][5]:
        newSegment=(listOfEdges[-1][3],listOfEdges[-1][4],
            listOfEdges[-1][5],points[0][0],points[0][1],
            coordinateZ,buildingId)
        listOfSegments.append(newSegment)

        # Normal segment
    i=0
    for point in points[:-1]:
        i=i+1
        x1=point[0]
        y1=point[1]
        x2=points[i][0]
        y2=points[i][1]
        newSegment=(x1,y1,coordinateZ,x2,y2,coordinateZ,
            buildingId)
        listOfSegments.append(newSegment)
        listOfEdges.append(newSegment)

#printFile(listOfStairs,'listOfStairs1')
```

```
  #putWeight(listOfStairs, 'stairsWeight')
  #printFile(listOfEdges, 'listOfEdgesWithOutSplittingStairs1
      ')
  #putWeight(listOfEdges, 'edgesWeightWithoutSplittingStairs
      ')
  printFile(listOfSegments, 'initialSteps')
  # putWeight(listOfSegments, 'segmentsWeight')

#
    ##############################################################################
        SPLIT STAIRS
# Splitting the stairs into smaller steps according to
    floors
  steps05=[] # List with the minimum stair height for
      buildings with half floors.
  steps1=[] #List with the minimum stair height for
      buildings with no half floors.
  halfFloors=[]
  bigStairs=[]
  newAuxiliarStairs=[]

  # First identify the Buildings that have half floors
  buildingsWithHalfFloors=[]
  for i in auxiliarStairs[1:]:
    jump=float(i[5])-float(i[2])
    if re.search(r'\.5',str(jump)):
      if not i[6] in buildingsWithHalfFloors:
        buildingsWithHalfFloors.append(i[6])
  print buildingsWithHalfFloors

  # Separate the stairs in floors.
  for i in auxiliarStairs[1:]:
    jump=float(i[5])-float(i[2])
        #
            #####################################################################
                Building has half floors.
    if i[6] in buildingsWithHalfFloors:
      if abs(jump)==0.5: #Smallest step in the building.
        listOfSegments.append(i)
        newAuxiliarStairs.append(i)
        steps05.append(i)
        else: #They can be split in smaller steps.
          halfFloors.append(i)
          #
             ##########################################################################
               Building with no half floors.
      else:
        if abs(jump)==1: #Smallest step in the building.
          listOfSegments.append(i)
          newAuxiliarStairs.append(i)
          steps1.append(i)
        else: #They can be split in smaller steps.
          bigStairs.append(i)

  # printFile(bigStairs,'bigStairs')
  # printFile(halfFloors,'halfFloors')

  pointsWithoutFirstSolution=[]
  pointsWithoutSecondSolution=[]
  splitAgain=[]
  #
     ##########################################################################
       Buildings without half floors.
  #First we try to get the minimum jump.
  print 'The␣following␣steps␣have␣been␣split␣in␣buildings␣
      without␣half␣floors'
  for i in bigStairs:
    solution=splitInTwo(i,steps1)
        #If we can't split in a minimum height jump, check
            if at least we can split in one smaller (even if
            it is not the smallest)
    if not solution[0]:
      solution=tryAgain(i,bigStairs)
      if not solution[0]:
        pointsWithoutFirstSolution.append(i)
        listOfSegments.append(i) #We cannot split this
            stair.
        newAuxiliarStairs.append(i)
      else:
        print solution[0], '\n', solution[1]
        listOfSegments.append(solution[0]) #There was no a
            smaller jump before, so this is the smallest we
            can have.
        newAuxiliarStairs.append(solution[0])
        if abs(float(solution[1][5])- float(solution[1][2]))
            ==1:
          listOfSegments.append(solution[1])
          newAuxiliarStairs.append(solution[1])
        else:
          bigStairs.append(solution[1]) #We will try again
              putting it at the end of the list.
        #We have found the smallest first jump.
    else:
      print solution[0], '\n', solution[1]
      listOfSegments.append(solution[0])
      newAuxiliarStairs.append(solution[0])
      if abs(float(solution[1][5])- float(solution[1][2]))
          ==1:
        listOfSegments.append(solution[1])
        newAuxiliarStairs.append(solution[1])
      else:
        solution2=splitInTwo(solution[1],steps1)
        if not solution2[0]:
          listOfSegments.append(solution[1])
          newAuxiliarStairs.append(solution[1])
        else:
          print solution2[0], '\n', solution2[1]
          listOfSegments.append(solution2[0])
          newAuxiliarStairs.append(solution2[0])
          if abs(float(solution2[1][5])- float(solution2
              [1][2]))==1:
            listOfSegments.append(solution2[1])
            newAuxiliarStairs.append(solution2[1])
          else:
            listOfSegments.append(solution2[1])
            newAuxiliarStairs.append(solution2[1])
            splitAgain.append(solution2[1])
            print 'This␣part␣is␣not␣implemented␣because␣this
                ␣case␣never␣happened␣before-we␣are␣not␣
                loosing␣data-It␣just␣could␣be␣optimized.'

#
    ##################################################################
```

```
      Buildings with half floors.
  #First we try to get the minimum jump.
  print 'The␣following␣steps␣have␣been␣split␣in␣buildings␣
      with␣half␣floors'
  for i in halfFloors:
    solution=splitInTwo(i,steps05)
        #If we can't split in a minimum height jump, check
            if at least we can split in one smaller (even if
            it is not the smallest)
    if not solution[0]:
      solution=tryAgain(i,halfFloors)
      if not solution[0]:
        pointsWithoutFirstSolution.append(i)
        listOfSegments.append(i) #We cannnot split this
            stair.
        newAuxiliarStairs.append(i)
      else:
        print solution[0], '\n', solution[1]
        listOfSegments.append(solution[0]) #There was no a
            smaller jump before, so this is the smallest we
            can have.
        newAuxiliarStairs.append(solution[0])
        if abs(float(solution[1][5])- float(solution[1][2]))
            ==0.5:
          listOfSegments.append(solution[1])
          newAuxiliarStairs.append(solution[1])
        else:
          halfFloors.append(solution[1]) #We will try again
              putting it at the end of the list.
        #We have found the smallest first jump.
    else:
      print solution[0], '\n', solution[1]
      listOfSegments.append(solution[0])
      newAuxiliarStairs.append(solution[0])
      if abs(float(solution[1][5])- float(solution[1][2]))
          ==0.5:
        listOfSegments.append(solution[1])
        newAuxiliarStairs.append(solution[1])
      else:
        solution2=splitInTwo(solution[1],steps05)
        if not solution2[0]:
          listOfSegments.append(solution[1])
          newAuxiliarStairs.append(solution[1])
        else:
          print solution2[0], '\n', solution2[1]
          listOfSegments.append(solution2[0])
          newAuxiliarStairs.append(solution2[0])
          if abs(float(solution2[1][5])- float(solution2
              [1][2]))==0.5:
            listOfSegments.append(solution2[1])
            newAuxiliarStairs.append(solution2[1])
          else:
            listOfSegments.append(solution2[1])
            newAuxiliarStairs.append(solution2[1])
            splitAgain.append(solution2[1])
            print 'This␣part␣is␣not␣implemented␣because␣this
                ␣case␣never␣happened␣before-we␣are␣not␣
                loosing␣data-It␣just␣could␣be␣optimized.'

  # printFile(auxiliarStairs,'AuxStairs')
  # printFile(newAuxiliarStairs,'NewAuxStairs')
  # printFile(splitAgain,'splitagain')
  # printFile(pointsWithoutFirstSolution,'
      pointsWithoutFirstSolution')
  # printFile(pointsWithoutSecondSolution,'
      pointsWithoutSecondSolution')
  # putWeight(steps05,'smallSteps05')
  # putWeight(steps1,'smallSteps1')
  printFile(listOfSegments,'finalSteps')
  putWeight(listOfSegments,'edgesWeighted')

#
    ##########################################################################


def putWeight(list,fileName):
  """ This function takes a list and creates a file storing
      only one time each item"""
  dictionary = {}
  for i in list:
    if not i in dictionary:
```

```
                dictionary[i]=1
            else:
                dictionary[i]= dictionary[i] + 1
    # Sort them so the big counts are first using key=
        get_count() to extract count.
    orderTargets = sorted(dictionary.items(),key=getCount,
        reverse=True)
    printFile(orderTargets,fileName)


# Function to store each kind of list
def printFile(listToPrint,fileName):
    """Stores in a new file the list passed as a parametre"""
    file=fileName+'.txt'
    outputFile= open(file,'w')
    #a mode writes at the end of the file an creates it if it
        didn't exit
    #w mode over writes everything
    for item in listToPrint: print>>outputFile, item
    outputFile.close()

def getCount(word_count_tuple):
    """Returns the count from a dict word/count tuple -- used
        for custom sort."""
    return word_count_tuple[1]

def splitInTwo(segment,listSmallSteps):
    """ The function split the segment into two new ones. The
        first will belong to listSmallSteps, the second will
        contain the remaining way. The condition is the first
        jump has to be sizeOfJump"""
    # Same start point, same building and one more floor
    # Segment format -> x1,y1,z1,x2,y2,z2,buildingId
    first=second=0
    for j in listSmallSteps:
            if segment[0]==j[0] and segment[1]==j[1] and segment
                [2]==j[2] and segment[6]==j[6] and ((float(
                segment[2])<float(j[5])<=float(segment[5])) or (
                float(segment[2])>float(j[5])>=float(segment[5]))
                ):
```

```
            first=(segment[0],segment[1],segment[2],j[3],j[4],
                j[5],j[6])
            second=(j[3],j[4],j[5],segment[3],segment[4],
                segment[5],segment[6])
  return first,second

def tryAgain(segment,listSmallSteps):
  """ """
  # Same start point, same building and one more floor
  # Segment format -> x1,y1,z1,x2,y2,z2,buildingId
  first=second=0
  for j in listSmallSteps:
    if not j==segment:
            if segment[0]==j[0] and segment[1]==j[1] and
                segment[2]==j[2] and segment[6]==j[6] and ((
                float(segment[2])<float(j[5])<float(segment[5])
                ) or (float(segment[2])>float(j[5])>float(
                segment[5]))):
              first=(j[0],j[1],j[2],j[3],j[4],j[5],j[6])
              second=(j[3],j[4],j[5],segment[3],segment[4],
                  segment[5],segment[6])
  return first,second



def main():
  if len(sys.argv) != 2:
    print 'usage:␣./createGraphDictionary.py␣file'
    sys.exit(1)

  filename = sys.argv[1]
  readFile(filename)



if __name__ == '__main__':
  main()
```

B.4 createCSVFile.py

```
#!/usr/bin/python -tt
```

```
import sys
import urllib
import urllib2
import re

__author__="Santiago␣Diez␣Martinez"
__email__="santydm2002@hotmail.com"


"""
 This program reads a CSV (comma separated value) file and
    creates the files needded to use MyGeoDataConverter. We
    introduce as parameters the filename
"""

def readLineFile(filename):
   """Stores each segment in a dictionary"""
   # CREATE THE CSV FILE the .ovf is created by hand before.
   #Create a list to store all the segments in tupples with
      the common format
   listOfSegments = []
   segment=('x1','y1','z1','x2','y2','z2','buildingId','
      weight')
   inputFile=open(filename,'rU')
   for line in inputFile:
      # In order to delete the last \n of each line (we also
         separate each field between ',' (comma separated
         values CSV)
      fields=line[:-1].split(',')
      segment=fields
      listOfSegments.append(segment)
      # print segment
   inputFile.close()
   print len(listOfSegments)
   #Write the head of the file
   outputformat=[]
   outputformat.append('id,name,geometryProperty,floor,
      buildingId,weight')
   #Write each segment in the proper grammar
   index=0
   for i in listOfSegments[:]:
    index=index+1
    id=str(index)
    string= id+',"Line␣#'+id+'","'+i[0]+i[1]+','+i[3]+i[4]+'
        ","'+i[2]+'","'+i[6]+'","'+i[7]+'"'
    outputformat.append(string)
  printFile(outputformat,'segmentsInCSVFormat.csv')




def putWeight(list,fileName):
  """ This function takes a list and creates a file storing
      only one time each item"""
  dictionary = {}
  for i in list:
    if not i in dictionary:
      dictionary[i]=1
    else:
      dictionary[i]= dictionary[i] + 1
  # Sort them so the big counts are first using key=
      get_count() to extract count.
  orderTargets = sorted(dictionary.items(),key=getCount,
      reverse=True)
  printFile(orderTargets,fileName)


# Function to store each kind of list
def printFile(listToPrint,fileName):
  """Stores in a new file the list passed as a parametre"""
  outputFile= open(fileName,'w')
  #a mode writes at the end of the file an creates it if it
      didn't exit
  #w mode over writes everything
  for item in listToPrint: print>>outputFile, item
  outputFile.close()

def getCount(word_count_tuple):
  """Returns the count from a dict word/count tuple -- used
      for custom sort."""
  return word_count_tuple[1]
```

```
# This basic command line argument parsing code is provided
    and
# calls the print_words() and print_top() functions which
    you must define.
def main():
  if len(sys.argv) != 2:
    print 'usage:␣./createCSVFile.py␣file'
    sys.exit(1)

  filename = sys.argv[1]
  type=sys.argv[2]
  requests=readLineFile(filename)


if __name__ == '__main__':
  main()
```

B.5 loadWeightedArcsIntoGraph.py

```
#!/usr/bin/python -tt

import networkx as nx #Library to work with graphs
import csv #Library to load and read coma separated values (
    CSV) files
import sys #System library
import os #Operational System library

__author__="Santiago␣Diez␣Martinez"
__email__="santydm2002@hotmail.com"


def loadGraph(filename):
  """ This function loads a directed graph from a CSV file.
      That file has every arc with the following format:
  sourceCoordinates;targetCoordinates;buildingId;weight   (
      the coordinates are also separated by a single coma)"""

  #Create an empty new directed graph
  directedGraph=nx.DiGraph()
  #Open the file in read mode
  file=open(filename,"r")
  nodes=[] #List to store the nodes we already have
  nodesForGephi=[]
  edges=[] #List to store the edges we have
  for line in file.readlines()[:]:
  #We separate now each field by just a semicolom (';')
    fields=line.split(";")
    buildingId=fields[2]
    source=fields[0]
    target=fields[1]
    #We delete the line change at the end of the last field
    aux=fields[3].split("\n")
    weight=aux[0]
    #If the nodes are not still in the graph we add them
    if source not in nodes:
      directedGraph.add_node(source)
      source1=source+','+buildingId
      nodes.append(source)
      nodesForGephi.append(source1)
    if target not in nodes:
      directedGraph.add_node(target)
      target1=target+','+buildingId
      nodes.append(target)
      nodesForGephi.append(target1)

        #Now we add the arc-directed edge.
    directedGraph.add_edge(source,target,weight=weight,
        buildingId=buildingId)
    edge=source+';'+target+';'+weight+';'+buildingId
    edges.append(edge)

  printFileNodes(nodesForGephi,'nodesForGephi.txt')
  printFileEdges(edges,'edgesForGephi.txt')
  return directedGraph


def edgesCaracterization(directedGraph):
  """ This function calculates the number of directed/
      undirected edges """
  output=[]
  #Print the number of edges and nodes
  char='################################## Directed'
```

```
    output.append(char)
    char='Directed␣nodes:␣'+str(directedGraph.number_of_nodes
        ())
    output.append(char)
    char= 'Directed␣edges:␣'+str(directedGraph.number_of_edges
        ())
    output.append(char)

    #Now we are going to print how many edges are bi-
        directional
    bidirectional=0
    non=0
    for edge in directedGraph.edges():
      #We check if the edge exits in the opposite direction
      if(directedGraph.has_edge(edge[1],edge[0])):
        bidirectional += 1
      else:
        non +=1
    char='Bidirectional␣edges:␣'+ str(bidirectional)
    output.append(char)
    char='Non␣bidirectional␣edges:␣'+ str(non)
    output.append(char)
    # Print the results
    for i in output:
      print i

    return output


def averageDegree(directedGraph):
    """ This function calculates the average degree of the
        graph. For doing that, it goes through all the nodes
        addind their
    degree, and finally it normalizes by the ammount of nodes.
        """
    output=[]

    #IN-Degree
    degree=0
    averageDegree=0
    #We obtain first a list with every node inDegree.
```

```
degreeList=(directedGraph.in_degree()).values()
for degree in degreeList:
  averageDegree += degree
#Now we calculate the average
averageInDegree= float(averageDegree)/directedGraph.
    number_of_nodes()
char="Average␣in−degree:␣"+str(averageInDegree)+"␣total␣
    inDegree:␣"+str(averageDegree)
output.append(char)

#OUT−Degree
degree=0
averageDegree=0
degreeList=(directedGraph.out_degree()).values()
for degree in degreeList:
  averageDegree += degree
averageOutDegree= float(averageDegree)/directedGraph.
    number_of_nodes()
char="Average␣out−degree:␣"+str(averageOutDegree)+"␣total␣
    outDegree:␣"+str(averageDegree)
output.append(char)

#Now we calculate the highest and lowest INdegree between
    all the nodes. Then we create a list with the nodes
    that have those degrees.
degreeValues=directedGraph.in_degree()
min=100
max=0
nodeMin=[]
nodeMax=[]
#We search the end values
for node in degreeValues:
        if degreeValues[node]>=max:
                max=degreeValues[node]
        if degreeValues[node]<=min:
                min=degreeValues[node]
char='Max␣InDegree:␣'+str(max)+'\n'+'Min␣InDegree:␣'+str(
    min)+'\n'+'Nodes␣with␣max␣inDegree'
output.append(char)
#Now we save the nodes whith those degrees
for node in degreeValues:
        if degreeValues[node]==max:
                nodeMax.append(node)
                output.append(node)
        if degreeValues[node]==min:
                nodeMin.append(node)
char='Number␣of␣nodes␣with␣min␣inDegree␣'+str(len(nodeMin)
    )
output.append(char)

#Now we calculate the highest and lowest OUTdegree between
     all the nodes. Then we create a list with the nodes
    that have those degrees.
degreeValues=directedGraph.out_degree()
min=100
max=0
nodeMin=[]
nodeMax=[]
#We search the end values
for node in degreeValues:
        if degreeValues[node]>=max:
                max=degreeValues[node]
        if degreeValues[node]<=min:
                min=degreeValues[node]
char='Max␣OutDegree:␣'+str(max)+'\n'+'Min␣OutDegree:␣'+str
    (min)+'\n'+'Nodes␣with␣max␣outDegree'
output.append(char)
#Now we save the nodes whith those degrees
for node in degreeValues:
        if degreeValues[node]==max:
                nodeMax.append(node)
                output.append(node)
        if degreeValues[node]==min:
                nodeMin.append(node)
char='Number␣of␣nodes␣with␣min␣outDegree␣'+str(len(nodeMin
    ))
output.append(char)

for i in output:
  print i

return output
```

```
def averageShortestPathLength(directedGraph):
  """ This function calculates the average shortest path
      between all the graph nodes. But if the graph is not
  connected -we have several componentes- we launch an
      exception to calculate it only in the giant component (
      the biggest)."""

  output=[]
  #We need to have an un-directed graph to calculate these
      measures (library specifications)
  graph=directedGraph.to_undirected()
  #Print the number of edges and nodes
  char='#################################␣UnDirected'
  output.append(char)
  char='UnDirected␣nodes:␣'+str(graph.number_of_nodes())
  output.append(char)
  char= 'UnDirected␣edges:␣'+str(graph.number_of_edges())
  output.append(char)
  #Degree
  averageDegree=0
  degreeList=(graph.degree()).values()
  for degree in degreeList:
    averageDegree += degree
  averageTotalDegree= float(averageDegree)/directedGraph.
      number_of_nodes()
  char="Average␣degree:␣"+str(averageTotalDegree)+ "␣total␣
      Degree:␣"+str(averageDegree)
  output.append(char)
  #Now we calculate the highest and lowest degree between
      all the nodes. Then we create a list with the nodes
      that have those degrees.
  degreeValues=nx.degree(graph)
  min=100
  max=0
  nodeMin=[]
  nodeMax=[]
  #We search the end values
  for node in degreeValues:
        if degreeValues[node]>=max:
                max=degreeValues[node]
        if degreeValues[node]<=min:
                min=degreeValues[node]
  char='Max␣Degree:␣'+str(max)+'\n'+'Min␣Degree:␣'+str(min)+
      '\n'+'Nodes␣with␣max␣degree'
  output.append(char)
  #Now we save the nodes whith those degrees
  for node in degreeValues:
        if degreeValues[node]==max:
                nodeMax.append(node)
                output.append(node)
        if degreeValues[node]==min:
                nodeMin.append(node)
  char='Number␣of␣nodes␣with␣min␣Degree␣'+str(len(nodeMin))
  output.append(char)
  #First we calculate the Average Clustering Coefficient (
      metric defined only for undirected graphs)
  char="Average␣Clustering␣Coefficient:␣"+str(nx.
      average_clustering(graph))
  output.append(char)

  #We try first with the whole graph
  try:
    avShortestPath=nx.average_shortest_path_length(graph)
    char="Average␣Shortest␣Path␣Length:␣"+str(avShortestPath
        )
    output.append(char)
  #But if the graph is not connected we launch an exception
      to catch the error
  except nx.NetworkXError:
    aux=severalComponents(graph)
    for i in aux:
          output.append(i)

  for i in output:
    print i

  return output

def severalComponents(graph):
  """ This function calculates the average shortest path
```

```
        between the giant component nodes. It also calculates
        the highest diameter."""
    aux=[]
    #Number of strong connected components
    num=nx.number_connected_components(graph)
    char="Number␣of␣strong␣connected␣components:␣"+str(num)
    aux.append(char)

    #List with those connected components
    componentsList=nx.connected_components(graph)
    #Now we choose the giant component (the methor
        connected_components return a list already ordered by
        size, starting with the biggest)
    component=componentsList[0]
    H=graph.subgraph(component)
    if (H.number_of_nodes()>1):
        max=nx.average_shortest_path_length(H)
        char='####␣Giant␣Component'
        aux.append(char)
        char="Average␣shortest␣path␣of␣the␣giant␣component:␣"+
            str(max)
        aux.append(char)
        char="Number␣of␣Nodes␣of␣the␣giant␣component:␣"+str(H.
            number_of_nodes())
        aux.append(char)
        char="Number␣of␣Edges␣of␣the␣giant␣component:␣"+str(H.
            number_of_edges())
        aux.append(char)
        percent=(float(H.number_of_nodes())/float(graph.
            number_of_nodes()))*100
        char="Percentage␣of␣nodes␣of␣the␣the␣giant␣component:␣"+
            str(percent)
        aux.append(char)
        percent=(float(H.number_of_edges())/float(graph.
            number_of_edges()))*100
        char="Percentage␣of␣edges␣of␣the␣the␣giant␣component:␣"+
            str(percent)
        aux.append(char)

            #Degree in the giant component
        averageDegree=0
    degreeList=(H.degree()).values()
    for degree in degreeList:
      averageDegree += degree
    averageTotalDegree= float(averageDegree)/H.
        number_of_nodes()
    char="Average␣degree␣in␣the␣giant␣component:␣"+str(
        averageTotalDegree)+ "␣total␣Degree:␣"+str(
        averageDegree)
    aux.append(char)
    #Now we calculate the highest and lowest degree between
        all the nodes. Then we create a list with the nodes
        that have those degrees.
    degreeValues=nx.degree(H)
    min=100
    max=0
    nodeMin=[]
    nodeMax=[]
    #We search the end values
    for node in degreeValues:
            if degreeValues[node]>=max:
                    max=degreeValues[node]
            if degreeValues[node]<=min:
                    min=degreeValues[node]
    char='Max␣Degree␣in␣the␣giant␣component:␣'+str(max)+'\n'
       +'Min␣Degree␣in␣the␣giant␣component:␣'+str(min)+'\n'+
        'Nodes␣with␣max␣degree'
    aux.append(char)
    #Now we save the nodes whith those degrees
    for node in degreeValues:
            if degreeValues[node]==max:
                  nodeMax.append(node)
                  aux.append(node)
            if degreeValues[node]==min:
              nodeMin.append(node)
    char='Number␣of␣nodes␣with␣min␣Degree␣'+str(len(nodeMin)
        )
    aux.append(char)

#We search for the biggest diameter.
maxDiameter=0
for component in componentsList:
        H=graph.subgraph(component)
        if(H.number_of_nodes()>1):
                diameter=nx.diameter(H,e=None)
        if diameter>maxDiameter: maxDiameter=diameter
    char="Biggest␣Diameter␣"+str(maxDiameter)+"␣Diameter␣of␣
        the␣giant␣Component:␣"+str(nx.diameter(graph.subgraph(
        componentsList[0]),e=None))
    aux.append(char)
    return aux


# Function to store the nodes in CSV format
def printFileNodes(listToPrint,fileName):
    """Stores in a new file the list passed as a parametre"""
    outputFile= open(fileName,'w')
    #a mode writes at the end of the file an creates it if it
        didn't exit
    #w mode over writes everything
    print>>outputFile,'Id;x;y;z;buildingId'
    for item in listToPrint:
        field=str(item).split(',')
        id=field[0]+','+field[1]+','+field[2]
        print>>outputFile, id+';'+field[0]+';'+field[1]+';'+
            field[2]+';'+field[3]
    outputFile.close()

# Function to store the edges in CSV format
def printFileEdges(listToPrint,fileName):
    """Stores in a new file the list passed as a parametre"""
    outputFile= open(fileName,'w')
    #a mode writes at the end of the file an creates it if it
        didn't exit
    #w mode over writes everything
    print>>outputFile,'source;target;weight;buildingId'
    for item in listToPrint:
        field=str(item).split(';')
        print>>outputFile,field[0]+';'+field[1]+';'+field[2]+';'
            +field[3]
    outputFile.close()
```

```
# This basic command line argument parsing code is provided
    and calls the functions to use.
def main():
  if len(sys.argv) != 2:
    print 'usage:␣./loadWeightedGraph.py␣file'
    sys.exit(1)
  filename = sys.argv[1]
  graph=loadGraph(filename)

  #Create a file to store the graph's metrics.
  resultsFile=open('graphMetrics.txt','w')
  edges=edgesCaracterization(graph)
  for i in edges: print>>resultsFile, i
  directedDegree=averageDegree(graph)
  for i in directedDegree: print>>resultsFile, i
  nonDirectedMetrics=averageShortestPathLength(graph)
  for i in nonDirectedMetrics: print>>resultsFile, i
  resultsFile.close()


if __name__ == '__main__':
  main()
```