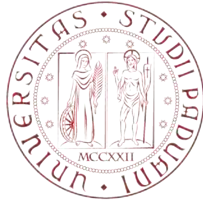

Università degli Studi di Padova
Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea Magistrale in Matematica

Community Detection on Temporal Networks

Relatore:
Giovanni Da San Martino

Correlatrice:
Giorgia Callegaro

Laureanda:
Sara Comelli
Matricola 2006655

16 dicembre 2022

Abstract

Temporal networks are widely used nowadays to represent dynamic systems in various contexts, from physics to biology, technology, economics and sociology. A well known example are social networks: the nodes represent the users, while the edges represent the connections between them, changing over time depending on their interactions.
Community detection is an important analysis that can be done on the network in order to understand if the nodes are organized into groups or communities and how these evolve during time. This might be useful for real life application, for instance to discover disinformation campaigns on social networks.
We analyse the current state-of-the-art algorithms. Specifically we focus on the trade-off between the stability of the algorithms over time and their ability to adapt to rapid changes in the communities structure.

Contents

Introduction **3**

1 Networks and Communities **5**
- 1.1 Static networks . . . 5
- 1.2 Temporal networks . . . 8
  - 1.2.1 Classification of temporal networks . . . 8
  - 1.2.2 Undirected and unweighted temporal networks . . . 9
  - 1.2.3 Snapshot networks . . . 10
  - 1.2.4 Conversion between different classes . . . 11
- 1.3 Communities . . . 12
  - 1.3.1 Motivation . . . 12
  - 1.3.2 Definition of community . . . 14
  - 1.3.3 Notation . . . 16
- 1.4 Evolution of communities . . . 17
  - 1.4.1 Partition similarity . . . 17
  - 1.4.2 Tracking the communities across time steps . . . 19
  - 1.4.3 Community events . . . 21
- 1.5 Evaluating the quality of a partition . . . 22
  - 1.5.1 Quality measures . . . 22
  - 1.5.2 Modularity . . . 25
  - 1.5.3 Modularity-driven approaches . . . 28

2 Community Detection Algorithms **29**
- 2.1 Community detection on static networks . . . 29
  - 2.1.1 Related works - static case . . . 29
  - 2.1.2 Louvain algorithm . . . 30
- 2.2 Community detection on temporal networks . . . 36
  - 2.2.1 Related works - temporal case . . . 36
  - 2.2.2 Aynaud and Guillaume's algorithm . . . 37
  - 2.2.3 ECSD algorithm . . . 41

3 Experimental Evaluation **49**
3.1 Experimental setup 49
3.1.1 Building the network 50
3.1.2 Detecting the communities 51
3.1.3 Modularity analysis 51
3.1.4 Smoothness analysis 53
3.2 Sociopatterns network 55
3.2.1 Description of Sociopatterns network 55
3.2.2 Results on Sociopatterns network 55
3.3 Artificial random generated network 64
3.3.1 Description of the artificial network 64
3.3.2 Results on the artificial network 65
3.4 DBLP network 72
3.4.1 Description of the DBLP network 72
3.4.2 Results on the DBLP network 72

Conclusions and future work **81**

Appendix A **83**

List of Figures **91**

Bibliography **93**

Ringraziamenti **97**

Introduction

The importance of networks has increased more and more in recent years. Nowadays they are usefully used to represent many dynamics and systems of various disciplines, such as physics, engineering, biology, technology, economics and sociology. Facebook, for instance, is a large social network in which the users are connected by links that represent their interactions.
When dealing with networks it might be interesting to discover if the network is organized into groups or *communities*: they are intuitively dense subnetworks that are well separated from each other. Discovering the community structure of a network might be really useful for real life applications. It allows to classify nodes, based on their role with respect to the communities they belong to. For instance we can distinguish nodes that are totally embedded within their communities from nodes at the boundary between communities, which could play a major role in holding the network together and in spreading information across it. We can unveil if the network is organized into a sort of hierarchy or if some communities have more influence than others, for example if they are larger or more cohesive.
As we will see later, community detection is a very challenging problem. The main reason behind this is that the problem is ill-defined: there is not a universal definition of what is a community and there are not guidelines about how to evaluate the performance of an algorithm or to validate its outputs. On one hand this has brought a lot of confusion in the field, but on the other hand it inspired many interesting and original approaches to the problem.
In particular we will focus on the problem of partitioning the network into communities: we want to divide the nodes into communities, whose number and sizes are not given a priori, so that each node belongs to exactly one community. Moreover we will work with temporal networks, which are networks that change over time. The problem, then, is not only to detect the communities on a fixed network, but also to study their evolution as time goes by.

This master thesis is divided into three chapters.

In Chapter 1 we will present the basic tools and notation for static and temporal networks. We will define more precisely the problem of community detection, with an highlight on the events that characterize the evolution of dynamic communities. We will finally present a widespread measure to evaluate the quality of a partition into communities: the modularity function.

In Chapter 2 we will make a rapid overview of the most used algorithms for community detection on static and temporal networks. In particular we will focus on one of the most popular and fast algorithms for static networks, the Louvain algorithm, and on two algorithms for dynamic community detection: the Aynaud and Guillaume's algorithm and the ECSD algorithm. We will present them, by explaining how they work and reporting a pseudo-code for their implementation.

In Chapter 3 we will show the results that we obtained by testing the Aynaud and Guillaume's algorithm and the ECSD algorithm, presented in Chapter 2. After the description of the experimental setup, we will present the three networks on which we tested the algorithms: the first one is the *Sociopatterns network*, already implemented in the library `tnetwork`, the second one is an artificial graph, that we generated using a function of the same library, and the third one, the *DBLP network*, is a co-authorship network between authors of scientific papers from DBLP computer science bibliography.
The two algorithms perform quite well on both the Sociopatterns and the DBLP networks, while they behave differently and achieve worse results on the artificial network, suggesting that the algorithms are more suitable for networks that do not change dramatically over time.
After the analysis of the results, we draw some conclusions and suggest some possible improvements.

Chapter 1

Networks and Communities

1.1 Static networks

We denote a network with G or $G = (V, E)$, where V represents the set of nodes and E is the set of edges. We define the following quantities:

- $n = |V|$ is the number of nodes of the network.
- $m = |E|$ is the number of edges of the network.
- e_{ij} is the edge that connects two nodes i and j; we consider only undirected graphs so $e_{ij} = e_{ji}$.
  If there exists the edge e_{ij}, the nodes i and j are said to be *adjacent* and they are the *ends* of the edge e_{ij}. The edge e_{ij} is said to be *incident* in i and j.
- $N(i)$ is the set of neighbours of the node i.
- In case of weighted network: $W(e_{ij})$, or for brevity w_{ij}, is the weight of the edge e_{ij}, where the weight is a function $W : E \to \mathbb{R}_+$.
  In case of unweighted network: we can ignore the weight function or simply consider it as a function that assumes values in $\{0, 1\}$, i.e. $W(e_{ij})$ is 1 if the edge e_{ij} exists, 0 otherwise.
- A is the *adjacency matrix*. A is an $n \times n$ matrix, whose entry A_{ij} is 1 if the nodes i and j are adjacent, 0 otherwise. Notice that A is symmetric since we are dealing with undirected networks.
- W is the *weighted adjacency matrix*. W is an $n \times n$ matrix, whose entry W_{ij} is w_{ij} if the nodes i and j are adjacent, 0 otherwise. Notice that W is symmetric since we are dealing with undirected networks.

- w is the sum of the weights of all the edges in the network. We will call it the *weight* of the network and it is given by:

$$w = \frac{1}{2} \sum_{i,j=1}^{n} W_{ij}. \tag{1.1}$$

- k_i is the *degree* of node i, i.e. the number of its incident edges:

$$k_i = \sum_{j=1}^{n} A_{ij}. \tag{1.2}$$

- K_i is the *weighted degree* of node i, i.e. the sum of the weights of its incident edges:

$$K_i = \sum_{j=1}^{n} W_{ij}. \tag{1.3}$$

Actually most of the authors, instead of explicitly defining also the weighted adjacency matrix W and the weighted degree K_i, just use the adjacency matrix A and the node degree k_i even for the weighted case, substituting the weighted quantities in their definitions. In many contexts this is automatic and our notation might seem redundant, but we prefer to keep the two definitions separated mainly for two reasons:

1. to be more precise, since in a weighted graph all the quantities are defined and different;

2. for historical reasons, because in the literature a lot of algorithms and formulas about community detection have been defined in the unweighted case and successively extended by other authors to the weighted case.

In Figure 1.1 there is an example of an unweighted static network with 13 nodes. The adjacency matrix that corresponds to this network is reported in Table 1.1. In particular, notice that the matrix is symmetric and on the diagonal there are only 0 because the network has no loops. If we consider the blue node number 7, its degree k_7 would be:

$$k_7 = \sum_{i=1}^{13} A_{i,7} = \sum_{i=1}^{13} A_{7,i} = 0+0+0+0+1+0+0+1+1+1+0+1+0 = 5,$$

that is precisely the number of its neighbours.

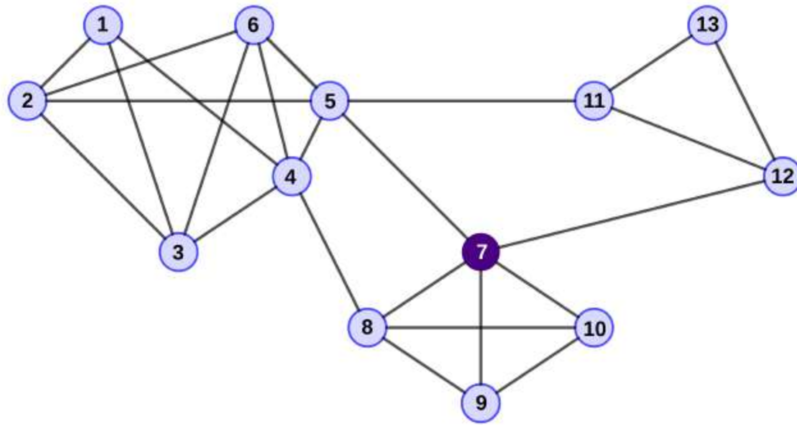

Figure 1.1: An example of a static network.

0	1	1	1	0	0	0	0	0	0	0	0	0
1	0	1	0	1	1	0	0	0	0	0	0	0
1	1	0	1	0	1	0	0	0	0	0	0	0
1	0	1	0	1	1	0	1	0	0	0	0	0
0	1	0	1	0	1	1	0	0	0	1	0	0
0	1	1	1	1	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	1	1	1	0	1	0
0	0	0	1	0	0	1	0	1	1	0	0	0
0	0	0	0	0	0	1	1	0	1	0	0	0
0	0	0	0	0	0	1	1	1	0	0	0	0
0	0	0	0	1	0	0	0	0	0	0	1	1
0	0	0	0	0	0	1	0	0	0	1	0	1
0	0	0	0	0	0	0	0	0	0	1	1	0

Table 1.1: Adjacency matrix of the static network in Figure 1.1.

In the following we will use the terms network or graph interchangeably and we will call *static* a network when we want to explicitly distinguish it from a temporal network. A static network indeed is a network fixed over time. A temporal network instead, as we will see in the next section, is a network that changes over time and all the previously defined quantities will depend on the variable t.

1.2 Temporal networks

1.2.1 Classification of temporal networks

Temporal networks[1] are networks that change over time.
There is not a universally accepted representation of temporal networks because the scenario becomes considerably more complicated compared to the static case and each author adopts his or her preferable notation depending on the purpose.

Consider for example a social network, in which the users are represented by nodes and their friendship interactions by edges: an edge is built from one node to another if two users become friends and it is removed if they are not friends anymore (let us consider friendship an undirected relation).
There are mainly two different ways to represent this temporal network:

1. We can think of each user as a node and say that it "appears" in the network when the user creates the profile, and "disappears" when the user deletes it. The same happens for each edge: when two users become friends an edge appears between them and it stays as long as they are friends.

2. We can build a static network every day, checking which users are present, and which of them are friends with each other. Doing this network day by day it will naturally change over time, reflecting the actions that happened in the real social network.

However, the second way might lose some information: if, for example, a user creates and deletes his or her profile during the same day, this will not be noticed in the representation. In fact, with the second representation we can only see changes that happened from one day to the next but not during the same day. One can then think of reducing the window and maybe create a snapshot every hour, or every second.

[1] We say equivalently dynamic/temporal graphs/networks.

Let us now complicate the things and add another type of relation, a sort of reaction that links two users for an instant, for example a "like" (consider it as if it was undirected). In this way, besides having edges with a duration that represent friendships, we also have instantaneous edges. The second representation would not be able to catch this type of information. More precisely, it would ignore it if it happens in an instant out of our temporal grid, or it would notice it if it happens in an instant in the grid, but it would be unable to distinguish it from a more stable link of the first type.

This basic example gives us an idea of the complexity of representing dynamic networks and justifies the classification of the representation of temporal networks into two different classes, similar to the ones defined by Rossetti and Cazabet (see [1]). The first one is the most accurate and will be presented in Section 1.2.2. However, it is not suitable for weighted networks and it is difficult to handle, that is why we will use the second type of representation, described in Section 1.2.3. We will make a brief comparison between them in Section 1.2.4.

1.2.2 Undirected and unweighted temporal networks

At first let us consider an undirected and unweighted network. In this situation, in a temporal network, the nodes can appear or disappear and the same holds for the edges, with the only constraint that their end nodes must be present when the edge is present.
We can then formulate this general and high-level definition of undirected and unweighted temporal networks. It does not lose any information about the time of existence and the relations between the elements of the graph.

Definition 1.2.1 (Undirected and unweighted temporal network)**.** An ***undirected and unweighted temporal network*** is a pair $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where:

- $\mathcal{V}$ is a set of pairs of the form (i, τ_i): i is a node of the graph and τ_i is its time of existence.
  τ_i can be an open interval $(start_i, end_i)$, a closed interval $[start_i, end_i]$ or an half open - half closed interval $(start_i, end_i]$ or $[start_i, end_i)$, with $start_i \leq end_i \in \mathbb{R}_+$. If $start_i = end_i$ the existence τ_i of the node is instantaneous.

- $\mathcal{E}$ is a set of triplets of the form (i, j, τ_{ij}): i and j are two nodes of the graph and they are connected by the edge e_{ij}, whose time of existence is τ_{ij}.
  Again τ_{ij} can be an open interval $(start_{ij}, end_{ij})$, a closed interval

$[start_{ij}, end_{ij}]$ or an half open - half closed interval $(start_{ij}, end_{ij}]$ or $[start_{ij}, end_{ij})$, with

$$start_i, start_j \leq start_{ij} \leq end_{ij} \leq end_i, end_j$$

and $start_{ij}, end_{ij} \in \mathbb{R}_+$. If $start_{ij} = end_{ij}$ the existence τ_{ij} of the edge is instantaneous.

In some articles a stricter classification is made between the networks in which the edges and the nodes have a non-zero duration, or in which they are instantaneous (see for example [2]). Hence, in the specific case in which all the edges have non-zero duration, we refer to ***interval networks*** (here again some authors consider just closed intervals, others open, others yet half open - half closed). In the other case in which all the edges have instantaneous duration we refer to ***contact sequences*** or ***link stream networks***. Here we prefer to keep the definition as general as possible, allowing some of them to be instantaneous and others not.

Definition 1.2.1 is very detailed and takes into account all possible temporal information but this is also its major drawback: considering the time as a continuous variable makes the temporal network really difficult to handle and furthermore such precision usually is not needed in real cases, at least in the ones for which data is publicly available, for which an instantaneous link is in any case not feasible.
What is even worse is that it is quite hard to extend it to the weighted case: how would we regulate the changing of the edge weights during time? In the literature there is not yet a fulfilling definition of weighted temporal networks that takes into account all the temporal information, but the most popular is the one presented in the next section.

1.2.3 Snapshot networks

For the reasons explained above we have decided to adopt another definition of temporal networks, the one of *snapshot networks*. It will be used during the all work, especially in the experimental section.

Definition 1.2.2 (Snapshot Network)**.** A ***snapshot network*** $\mathcal{G}$ is an ordered sequence of static graphs, each of them associated to an instant $t \in \{t_0, t_1, \ldots, T\}$. We indicate it with:

$$\mathcal{G} = (G^{t_0}, G^{t_1}, \ldots, G^T)$$

where each snapshot is a static graph with its nodes and edges, $G^t = (V^t, E^t)$.

Notice that the times of the snapshots $t \in \{t_0, t_1, \dots, T\}$ are left intentionally general. They could be integers of a sequence ($t_0 = 0, t_1 = 1, t_2 = 2, \dots$), dates ($t_0 = 19^{th}$ *January* 2022, $t_1 = 24^{th}$ *February* 2022, ...). A commonly used choice in datasets and experimental papers is to use POSIX time[2].

If all the snapshots are unweighted graphs, the network will be an unweighted snapshot network. Conversely, if any of the snapshot G^t is a weighted graph, it will be a weighted snapshot network. However, in the following we will just say snapshot network, adding the terms *unweighted* or *weighted* if we want to emphasize it.
Notice that at any time t a static graph G^t is well defined, so all the notations presented in Section 1.1 can be used for dynamic graphs just by adding the apex with the time t. For example an element of the adjacency matrix of snapshot t will be denoted with A^t_{ij}, and so on for all the other quantities.

As we can see, Definition 1.2.2 is more manageable and intuitive than Definition 1.2.1 and it is very popular in the literature. This corresponds precisely to the idea of taking a photograph, a snapshot, of the network at a precise time. The major drawback in this scenario is that we are loosing temporal information, because we are discretizing the time variable, but we can certainly refine the grid time at the desired resolution.
Definition 1.2.2 is more convenient for us also because we will focus on the problem of community detection and this representation makes it more intuitive, as we will see later. Moreover, the algorithms we selected in the experimental section explicitly require a snapshot representation for their implementation.

1.2.4 Conversion between different classes

As said before, the main difference that distinguishes the two classes is that in the first one time is considered as a continuous variable, while in the second one as a discrete variable.
Another important difference between the two representations, especially in the experimental section, consists in the way we access network information: the snapshot representation is more convenient if we want to know the situation at a given time, the interval representation instead is better when we

[2] POSIX time (also known as Unix time or Epoch time) is a system for describing instants in time, defined as the number of seconds that have elapsed since 00:00:00 Coordinated Universal Time (UTC), Thursday, 1st January 1970, not counting leap seconds.

want to access the information of an element without having to search for it among all the snapshots.

If we are dealing with closed interval networks or with contact sequences the two representations might be considered interchangeable. In fact it will be sufficient to discretize the time with a fine enough grid, or vice versa to extend discrete times to intervals. A key point for the choice of the grid is to look for the *atomic changes* that happened during the evolution of the network: they are those modifications in the graph (creation of a node, removal of an isolated node, creation of an edge, removal of an edge, weight increase and weight decrease) that cannot be subdivided into smaller changes. They are well explained in [3], but we will not dwell further on them nor on the conversion between the classes. We will adopt the snapshot representation for the rest of the work and especially in the experimental part.

1.3 Communities

1.3.1 Motivation

When looking at the layout of a static network, sometimes we notice that the nodes are grouped into communities[3]: they are intuitively dense subgraphs that are well separated from each other.
Figure 1.2 gives us an example of a graph and its communities. However, it is seldom possible to represent the network and visualize such an intuitive partition into communities: the network might be huge or in other cases the visualization may not intuitively suggest a community structure if the nodes are badly disposed on the plane.

The organization of the network into communities is natural for a lot of real networks: for example we can think of a co-authorship network in which each node represents an author and an edge is built between two authors if they publish a paper together. The authors are grouped into clusters that roughly represent the area of their research, and the size of the clusters might vary a lot from a popular topic to a less known one. We used a co-authorship network in the experimental part of Section 3.4. Similarly, it is useful to select groups in a social context, for example to regulate the spreading of a pandemic or to control the diffusion of certain opinions in social networks.
In particular, the authors Weber and Neumann made a very interesting research about the amplification of political news on Twitter (see [5] and [6]).

[3] Also called clusters or modules.

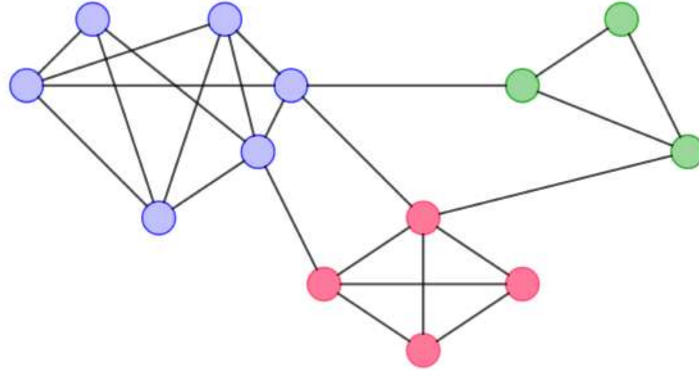

Figure 1.2: A partition into three communities of the static network in Figure 1.1.

They collected various political tweets from the platform, organized the data into temporal networks, by inferring links between the users, and subsequently used community detection algorithms to discover groups of accounts engaging in coordinated behaviours, that might become damaging and misinformative.

Knowing the partition of the network into communities might be very useful to know also the function that a precise node or person plays in the network. There are nodes that are fully embedded in a cluster, so all their neighbours belong to the same cluster, and they represent the core of the group. There are nodes though that lie at the boundary of the cluster, meaning that they have neighbours that belong to other communities. They play an important role as "gatekeepers" between different parts of the network and they are important for the spreading of the information in the whole network. These concepts are related to the *belonging degree* of a node to a certain community, see Definition 1.5.2.

Identifying the communities is an ill-defined problem. Indeed there is not a universal definition of community and consequently there are not clear guidelines on how to assess the performance of different algorithms nor how to compare them. On one hand such ambiguity has generated a lot of noise and confusion in the field. On the other hand it leaves a lot of freedom to

propose diverse approaches to the problem, which often depend on the aim of the research.
For example one can think of finding the best community in the surroundings of a certain node. In this case we usually speak of *local community detection*, underlying the fact that we are focused on a local part of the graph (see [16]).
In other situations one might look for the best balanced partition of the network into a given number of groups, useful especially for parallel computing tasks (see [7]).
We will speak about *covering with communities* when we want all our nodes to be assigned to at least one community and the communities might be overlapping, i.e. share some nodes (see [15]).

In particular we will focus on the problem of *network partitioning into non-overlapping communities*. We want to assign every node to exactly one community, but we have no constraints about the number of communities or their sizes. This is also connected to the topic of *data clustering*, i.e. grouping data elements into clusters based on some notion of similarity, such that elements in the same cluster are more similar to each other than they are to elements of different clusters. Often the tools for data clustering might be used for community detection on networks (see the survey [8]), but we will follow another direction.
Actually the problem is even more complicated for us because we are not dealing only with community detection on static networks, but we will have to discuss it in a time varying scenario. So, besides assigning each node to a community at the first snapshot, we will have to track the communities evolution, trying to understand how they change with the passing of time. This will be explained in Section 1.4.2.
First, let us start with the static case, giving some definitions and notation.

1.3.2 Definition of community

As we said above there is not a universally accepted definition of network community and it varies a lot depending on the aim of the researcher (see [9]). Nevertheless, ideal communities should have:

1. high cohesion, i.e. many internal links;
2. high separation, i.e. few links between them.

Classical definitions of community structure are based on cohesion[4], separation or a trade-off between them. We will adopt the definitions of Fortunato

[4] Like cliques.

et al. because they reflect very well our intuition (see [4]).

Definition 1.3.1 (Strong and weak communities)**.** Let us consider a network $G = (V, E)$ and a subset of the nodes $C \subseteq V$. We identify C with its node induced subgraph of G, that is $(C, E(C))$. We say that:

- C is a ***strong community*** if the internal degree of each node in C exceeds its external degree.

- C is a ***weak community*** if the sum of the internal degrees of all the nodes in C exceeds the sum of their external degrees.

Remark. The internal degree of a node is the number of neighbours that belong to its own community:

$$k_i^{int} = \sum_{j \in C} A_{ij}, \tag{1.4}$$

while the external degree is the number of neighbours that belong to the rest of the network:

$$k_i^{ext} = \sum_{j \in V \setminus C} A_{ij}. \tag{1.5}$$

Notice that the sum of the internal and the external degrees is equal to the degree of the node:

$$k_i^{int} + k_i^{ext} = \sum_{j \in V} A_{ij} = k_i.$$

A drawback of Definition 1.3.1 is that it separates the subgraph at study from the rest of the network, which is considered as a single object. But also the rest of the network might be divided into communities, so it makes more sense to say that the nodes in C are more attached to C than to the other communities, instead of considering all the rest of the network as a whole. This idea has inspired the following definition.

Definition 1.3.2 (Strong and weak communities in a partition)**.** Let us consider a network $G = (V, E)$ and a partition $\mathcal{C} = (C_1, C_2, \ldots, C_p)$ of the nodes into p sets, i.e. $\bigcup_{h=1}^{p} C_h = V$ and $C_h \cap C_l = \emptyset$ for $h \neq l$. Each set C_h can be identified with its nodes-induced subnetwork of G, that is $(C_h, E(C_h))$.

- $C_h \in \mathcal{C}$ is a ***strong community*** if the internal degree of each node in C_h exceeds its external degree to any other community $C_l \in \mathcal{C}$, $l \neq h$.

- $C_h \in \mathcal{C}$ is a ***weak community*** if the sum of the internal degrees of the nodes in C_h exceeds the sum of their external degrees to any other community $C_l \in \mathcal{C}$, $l \neq h$.

Remark. In case of a weighted network, in Definition 1.3.1 and 1.3.2 we just replace the degree of a node with its weighted degree.

Defining the communities beforehand is a useful starting point to understand the topic. However, Definitions 1.3.1 and 1.3.2 are quite hard to use in practice, because partitions into communities are quite hard to detect. Almost all the techniques to detect a partition into communities in networks do not require a definition of community. The authors usually suggest an algorithm that works on a specific class of networks (like weighted/unweighted, directed/undirected, simple, etc.) and they call communities the output of their algorithm, regardless of whether or not the partition produced by the algorithm satisfies any precise definition (see [18], [19], [20]). Only later they check "how good" are the communities found, namely how they correspond to our intuitive idea of being dense and separated from each other. To evaluate the quality of the communities found one could check, community by community, if they satisfy Definition 1.3.2, but it is generally more convenient to use an evaluation function that takes as input the entire partition of the graph into communities and returns a numerical value, as we will show later in Section 1.5.2.
In this work we follow the same approach: we will say that *any* partition of the network is a partition *into communities*, and we will evaluate at a later stage the quality of the partition we found.

1.3.3 Notation

We denote with $\mathcal{C} = (C_1, C_2, \ldots, C_p)$ a partition or clustering of the nodes into p sets. In particular $\bigcup_{h=1}^{p} C_h = V$ and $C_h \cap C_l = \emptyset$ for $h \neq l$, so each node of the network belongs to exactly one community.
Each set C_h, or sometimes just C, will be called community or cluster. With a little abuse of notation, we will denote with the lowercase letter c_i the community of the node i.
A clustering is trivial if either $p = 1$, i.e. there is only one cluster that contains all the nodes, or $p = n$, so each cluster is composed only by one node, i.e. the clusters are the singletons.
We identify a community C_h with its node-induced subgraph of G, which is $G(C_h, E(C_h))$.
We will indicate with $\mathcal{P}$ the set of all the possible partitions of the nodes into communities. The cardinality of $\mathcal{P}$ is the Bell number. It is denoted by B_n, where n is the number of nodes, and it is recursively[5] calculated with

[5] Remember that $0! = 1$.

the formula:

$$\begin{aligned} B_0 =& B_1 = 1, \\ B_n =& \sum_{k=0}^{n} \binom{n}{k} B_k. \end{aligned}$$

Remark. The Bell number is huge and it grows faster than e^n: for example, in a graph of 15 nodes we have $1,382,958,545$ possible partitions.

When dealing with temporal networks we will just add the apex t relative to the time step, as we did before, and we define a dynamic partition of a snapshot network as follows.

Definition 1.3.3 (Dynamic partition)**.** Given a snapshot network $\mathcal{G}$, we will say that a ***dynamic partition*** is a sequence, whose elements correspond to the partitions of the snapshots of the network.
More precisely, we denote a dynamic partition of $\mathcal{G} = (G^1, \dots, G^T)$ with:

$$\mathcal{C} = (\mathcal{C}^1, \dots, \mathcal{C}^T), \tag{1.6}$$

where the t-th element of the sequence is a partition of the t-th snapshot G^t:

$$\mathcal{C}^t = (C_1^t, C_2^t, \dots, C_{p^t}^t). \tag{1.7}$$

Notice that in this case even the number of communities p^t depends on the time step t.

1.4 Evolution of communities

1.4.1 Partition similarity

In this section we deal with the problem of measuring the similarity between two partitions. It might be useful in particular in two cases in the field of community detection. Firstly, if we know the ground truth communities of the network and we want to compare them with the partition obtained with a given algorithm. Secondly, when we want to track the evolution of communities across time steps, as we will see in Section 1.4.2.
Suppose we have a network $G = (V, E)$ and two different partitions $\mathcal{C} = (C_1, C_2, \dots, C_p)$ and $\mathcal{D} = (D_1, D_2, \dots, D_q)$. How can we quantify how much they are similar or different from each other? Several measures have been proposed to quantify the similarity of two partitions and they can be broadly

classified into *node-structural measures*, which depend only on the partition of the nodes in V, and *graph-structural measures*, that take into account the edge structure of the graph. These categories can be further divided into measures relying on pair-counting, cluster overlap or entropy (for an overview and further references see [13]).
We will focus on two well known measures: the Rand index and the Jaccard index. They are based on pair-counting and we will use the node-structural formulation even if it is possible to formulate a graph-structural variant. To define the indices we need the following quantities:

$$\begin{aligned} V_{11} =& \{i, j \in V | c_i = c_j \text{ and } d_i = d_j\} \\ V_{10} =& \{i, j \in V | c_i = c_j \text{ and } d_i \neq d_j\} \\ V_{01} =& \{i, j \in V | c_i \neq c_j \text{ and } d_i = d_j\} \\ V_{00} =& \{i, j \in V | c_i \neq c_j \text{ and } d_i \neq d_j\} \end{aligned}$$

and:

$$\begin{aligned} n_{11} =& |V_{11}| \\ n_{10} =& |V_{10}| \\ n_{01} =& |V_{01}| \\ n_{00} =& |V_{00}|. \end{aligned}$$

In other words, n_{11} indicates the number of pair of nodes that are classified in the same community in both partitions, $n_{10} + n_{01}$ the number of pair of nodes that are classified into the same community in one partition but not in the other one and finally n_{00} the number of pair of nodes that are classified in different communities in both partitions.
These values are needed to define the Rand and Jaccard indices for the node-structural formulation. If one defines the previous values using edges instead of nodes, their would obtain the graph-structural variant of the two indices.

Remark. When we compare two partitions of different snapshots usually $V^t \neq V^{t+1}$ so the previous indices are not well defined. It is not completely obvious how the partition similarity on two different snapshots (or more generally on two different static graphs) should be defined. A canonical solution (see [14]) is to consider the intersection of the two graphs and calculate the indices using only the nodes that belong to $V = V^t \cap V^{t+1}$.

Definition 1.4.1 (Rand index)**.** Given a network $G = (V, E)$ and two partitions $\mathcal{C}$ and $\mathcal{D}$, we define the ***Rand index*** as:

$$R(\mathcal{C}, \mathcal{D}) = \frac{n_{11} + n_{00}}{n_{11} + n_{10} + n_{01} + n_{00}}. \tag{1.8}$$

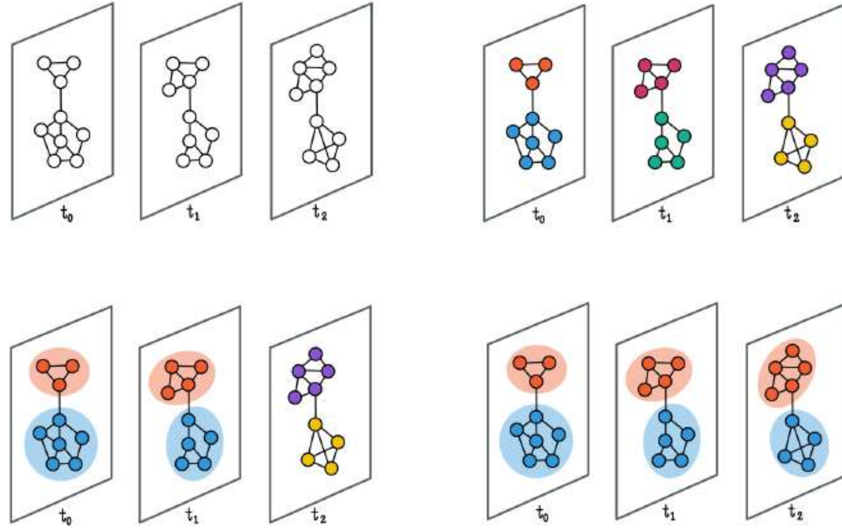

Figure 1.3: Tracking the communities across three time steps.

Definition 1.4.2 (Jaccard index)**.** Given a network $G = (V, E)$ and two partitions $\mathcal{C}$ and $\mathcal{D}$, we define the ***Jaccard index*** as:

$$J(\mathcal{C}, \mathcal{D}) = \frac{n_{11}}{n_{11} + n_{10} + n_{01}}. \tag{1.9}$$

Remark. Both of them take values in $[0, 1]$ where 1 indicates that they are the same partitions and 0 that they differ a lot from each other. Indeed, if $\mathcal{C}$ and $\mathcal{D}$ coincide, we would have that $n_{01} = n_{10} = 0$ because it will not happen that two nodes are classified in the same cluster in the first partition and in different clusters in the second one. So $R(\mathcal{C}, \mathcal{D}) = \frac{n_{11}+n_{00}}{n_{11}+n_{00}} = 1$ and $J(\mathcal{C}, \mathcal{D}) = \frac{n_{11}}{n_{11}} = 1$.
Usually, due to the dominance of the term n_{00}, the Rand index takes values very close to 1, whereas the Jaccard index usually assumes a broader range of values.

1.4.2 Tracking the communities across time steps

As we noticed in Section 1.3.1, the problem of community detection is ill-posed and the situation is even more complicated in the temporal scenario. Indeed, besides having to find good communities at each snapshot, it is reasonable to track them across different time step, giving rise to a *dynamic community.*
In Figure 1.3 we show a three-snapshot network (up-left). We firstly detect

the best communities independently from the snapshots, marking them with different colours (up-right). But this is meaningless since we are interested in the evolution of the communities across time, treating the network as a whole temporal object and not as a collection of static networks that have nothing to do with each other. That is why it is reasonable look at the communities at time t_1 and try to understand which communities at time t_1 might be the evolution of which existing communities at time t_0, and mark them with the same colour (bottom-left). We do the same between the communities at time t_2 and the ones at time t_1 (bottom-right). Thanks to this matching operation we obtain two dynamic communities, represented in red and blue.
The earliest algorithms for community detection on snapshot networks used to operate in the same way of the example above: firstly, they detected the best communities at each snapshot and only afterwards they tracked them across the different snapshots (see the survey in [11]).
This solution has evolved over time and now most of the best known algorithms use the partition at the previous time steps to find the partitions at the following ones. In this way the temporal information is taken into account during the whole process: this improves the quality of the dynamic communities and the performance of the algorithm. In the experimental part we used two algorithms that operate in this way.

The problem of tracking the communities is connected with the concept of smoothness of the algorithm. In fact, suppose that we have a snapshot network and let us focus on two consecutive time steps t and $t+1$. Once we found the partitions $\mathcal{C}^t$ and $\mathcal{C}^{t+1}$, we want to know how much they are similar, meaning how much the transition of the communities from time step t to time step $t+1$ is *smooth*.
Suppose that we have two identical snapshots: then a valid algorithm would for sure assign the nodes to the same communities obtaining the maximum smoothness and keeping high-quality communities.
Suppose now that the two snapshots have the same nodes but the links between them changed considerably. If we force our algorithm to be as smooth as possible, by assigning again the nodes into the same communities, our partition at time step $t+1$ would for sure be smooth, but it would be poor in terms of quality of the communities.
It is important for a valid algorithm to achieve a good trade-off between quality and smoothness. In the experimental part we took care of this aspect and, for each algorithm we used, we calculated the quality of the partition at each snapshot (using the function modularity of Section 1.5.2) and the smoothness of the transition respect to the previous step (using the Rand

and Jaccard indices).
Maintaining high values for both quality and smoothness is hard for common algorithms but it is always desiderable.

1.4.3 Community events

Connected with the topic of tracking the communities there is the one of *life-cycle* of a dynamic community. It aims at outlining the complete history of each community, starting from its first appearance and following all the chain of events it has been subject to.
Surprisingly in the literature there seem to be a broad consensus about the events that characterize the life of a community even though also in this case some authors consider some events that others ignore. We will follow the article by Greene et al. ([12]) and identify the following events, illustrated in Figure 1.4.

- **Birth**. It occurs when we find at time $t+1$ a community C for which there is no corresponding dynamic community at the previous time step.

- **Death**. It occurs when a community dissolves, so when it was present at time t but not anymore at time $t+1$.

- **Grow**. It occurs when the community at time step $t+1$ has a larger number of nodes with respect to the previous time step.
  It might be useful to establish a parameter to decide when the community is significantly bigger (for example if it has at least 10% nodes more than previously).

- **Shrink**. It occurs when the community at time step $t+1$ has a lower number of nodes with respect to the previous time step.
  It might be useful to establish a parameter to decide when the community is significantly smaller (for example if it has at least 10% nodes less than previously).

- **Merge**. It occurs when two or more different communities of time step t merge in a unique community at time step $t+1$.

- **Split**. It occurs when a community at time step t is divided into two or more communities at time step $t+1$.

Rossetti and Cazabet (in the survey [1]) include also the events "continue", that means that a community remains the same in the following time step (or

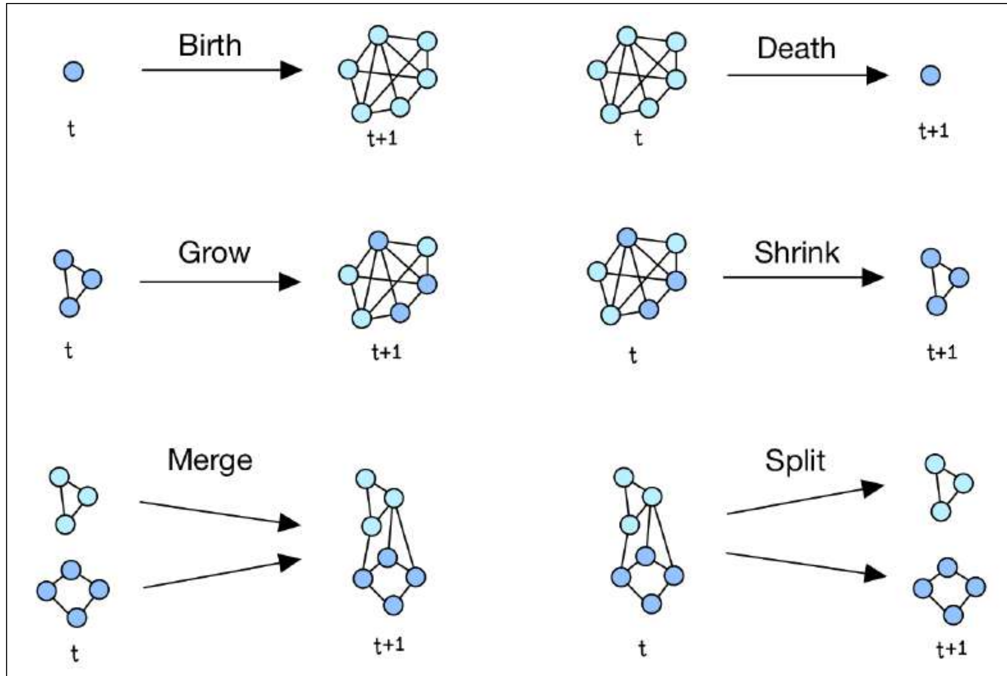

Figure 1.4: Community events.

grows/shrinks less than the threshold), and "resurgence" that occurs when a community dies and then births again after some time.
We explained the previous events to give an insight into how a network generator may work. Indeed in the experimental part in Section 3.3 we used an artificial network that was generated with a function (provided by the library we used) that makes use of the above events by randomly building a scenario with evolving communities.

1.5 Evaluating the quality of a partition

1.5.1 Quality measures

Consider a network $G = (V, E)$ and a partition $\mathcal{C} = (C_1, C_2 \ldots, C_p)$ into p communities. We are interested in knowing how good is the quality of our partition, namely how cohesive the communities are and how well they are separated from each other.
In this section we will write all the formulas in the case of a weighted network. In case of an unweighted network it suffices to replace the weighted adjacency matrix W with the adjacency matrix A (see Section 1.1).

First of all it would be interesting to know the sum of the weights of all the links inside each community C. We will call it the *weight of the community* C and is given by:

$$w(C) = \frac{1}{2} \sum_{i,j \in C} W_{ij}, \tag{1.10}$$

recalling that W_{ij} is the weight of the edge e_{ij}.
We used this notation in analogy with the weight of the network, w (see Eq. (1.1)). Indeed $w(C)$ is equal to the weight of the subnetwork of G induced by the community C.
The sum of the weights of internal edges of the partition (i.e. that connect nodes both belonging to the same community) is given by the sum of the weights of each community. More precisely:

$$w(\mathcal{C}) = \sum_{h=1}^{p} w(C_h) = \frac{1}{2} \sum_{i,j=1}^{n} W_{ij} \cdot \delta(c_i, c_j), \tag{1.11}$$

where $\delta(c_i, c_j)$ is equal to 1 if the nodes i and j lie in the same community (i.e. if $c_i = c_j$), to 0 otherwise. In this way an edge is considered in the summation if and only if its nodes belong to the same community.

The sum of the weights of edges that join a node in the community h with a node in the community l will be denoted with:

$$s(C_h, C_l) = \sum_{i \in C_h} \sum_{j \in C_l} W_{ij} = \sum_{i,j=1}^{n} W_{ij} \cdot \delta(c_i, C_h)\delta(c_j, C_l). \tag{1.12}$$

Notice that $s(C_h, C_h) = 2w(C_h)$.
The sum of the weights of the separation edges in the partition (i.e. the edges that connect nodes belonging to different communities) is given by the sum over all the communities of the weights of links separating each pair of them. More precisely:

$$s(\mathcal{C}) = \frac{1}{2} \sum_{h,l=1}^{p} s(C_h, C_l) = \frac{1}{2} \sum_{h,l=1}^{p} \sum_{i,j=1}^{n} W_{ij} \cdot \delta(c_i, h)\delta(c_j, l). \tag{1.13}$$

Now, using Eq. (1.11), we can calculate the ratio between the sum of the weights of inter-community links and the weight of the network. This corresponds to the notion of *coverage*.

Definition 1.5.1 (Coverage)**.** Given a weighted network $G = (V, E)$ and a partition into communities $\mathcal{C} = (C_1, C_2 \ldots, C_p)$, the *coverage* of the partition is given by:

$$COV(\mathcal{C}) = \frac{w(\mathcal{C})}{w} = \frac{\sum_{i,j=1}^{n} W_{i,j} \cdot \delta(c_i, c_j)}{\sum_{i,j=1}^{n} W_{ij}} \tag{1.14}$$

recalling that $w = \frac{1}{2}\sum_{i,j=1}^{n} W_{ij}$ is the weight of the network.

Coverage is a simple quality index but has a major drawback: if we want to maximize the coverage, the best partition would be the trivial one with all the nodes into one same community. As we can imagine this result is not usable in most cases and that is why we will discard this measure to introduce a more reliable one: the modularity (Section 1.5.2).

Besides coverage a lot of quality measures have been proposed in the literature to evaluate the quality of a partition, for example conductance (one of the most used), expansion, maximum out degree function, average out degree function, performance etc. (see [3] and [17]). Among all the most popular and widespread one is certainly the modularity. Despite some drawbacks, such as non local effects possibly leading to counter intuitive results, modularity indeed agrees with human intuition in various ways.
Before proceeding with the explanation of the modularity function, we will define another important quantity that reveals the degree to which a node belongs to a community.

Definition 1.5.2 (Belonging degree)**.** Given a node i and a community C, the *belonging degree* of the node i to the community C is:

$$BD(i, C) = \frac{\sum_{j \in C} W_{ij}}{K_i}, \tag{1.15}$$

where W_{ij} is the weight of the edge e_{ij} and $K_i = \sum_{j \in V} W_{ij}$ is the weighted degree of node i.

Remark. Notice that $BD(i, C)$ takes values in $[0, 1]$ and it is equal to 1 if all the neighbours of i belong to C, independently from the fact that i belongs to C or not.

The index of belonging degree will be needed for the ECSD algorithm that we will present in Section 2.2.3.

1.5.2 Modularity

Modularity is a function that evaluates the quality of the partition of a graph in communities. It is a function Q that takes values in $[-1, 1]$ and is higher if the partition into the communities of C is valuable for the network G. It measures the difference between the given division of the network into communities compared to a randomized version of the same network that is supposed not to exhibit any particular community structure.
Modularity was firstly introduced in an article by Newman and Girvan in 2004 ([18]) and it is based on a similar measure previously defined by Newman (see [21]). It has been widely used in the literature and some modifications have been proposed during years.
We now define more precisely the modularity in a unweighted graph and then extend it to the weighted case. For this first part we will refer mainly to Clauset, Newman and Girvan's paper (see [19]).

Consider a network $G = (V, E)$ and a partition $\mathcal{C} = (C_1, C_2, \dots C_p)$. A good community structure corresponds to a high fraction of within community edges, because it reveals that the edges connecting members of the same community are much more than the edges connecting different communities. Recall that this number corresponds to the coverage of the partition (Eq. (1.14)) but, as noticed before, it is not enough trying to maximize the coverage, otherwise we would put all the nodes in one community obtaining that this fraction is equal to 1 without getting any particular information about the community structure.
We consider then a randomized version of our network: more precisely suppose that the connections are made at random respecting the nodes degree. In this scenario the probability that between nodes i and j there exists an edge would be $\frac{k_i \cdot k_j}{2m}$. In this randomized network the fraction of edges within communities would be:

$$COV_{rand}(\mathcal{C}) = \frac{1}{2m} \sum_{i,j=1}^{n} \frac{k_i \cdot k_j}{2m} \cdot \delta(c_i, c_j). \qquad (1.16)$$

We now define the modularity as the difference between the actual fraction of within communities edges (Eq. (1.14)) and the same quantity in the randomized version (Eq. (1.16)):

Definition 1.5.3 (Unweighted modularity)**.** The modularity of an unweighted network $G = (V, E)$ partitioned into $\mathcal{C} = (C_1, C_2, \dots, C_p)$ communities, is

given by:

$$Q(G,\mathcal{C}) = \frac{1}{2m} \sum_{i,j=1}^{n} \left[A_{ij} - \frac{k_i \cdot k_j}{2m} \right] \cdot \delta(c_i, c_j). \tag{1.17}$$

recalling that $m = \frac{1}{2} \sum_{i,j=1}^{n} A_{ij}$ is number of edges of the network, A_{ij} are the terms of the adjacency matrix A and $k_i = \sum_{j=1}^{n} A_{ij}$ is the degree of the node i.

We will immediately extend this definition to weighted graphs as done in [22]:

Definition 1.5.4 (Weighted modularity)**.** The modularity of a weighted network $G = (V, E)$ partitioned into $\mathcal{C} = (C_1, C_2, \ldots, C_p)$ communities, is given by:

$$Q(G,\mathcal{C}) = \frac{1}{2w} \sum_{i,j=1}^{n} \left[W_{ij} - \frac{K_i \cdot K_j}{2w} \right] \cdot \delta(c_i, c_j). \tag{1.18}$$

recalling that $w = \frac{1}{2} \sum_{i,j=1}^{n} W_{ij}$ is the weight of the network, W_{ij} are the terms of the weighted adjacency matrix W and $K_i = \sum_{j=1}^{n} W_{ij}$ is the weighted degree of the node i.

Remark. As said before, the value of modularity ranges from -1 to 1. It is worth noting that, if we consider all nodes belonging to a unique community the modularity is 0. Indeed $\delta(c_i, c_j)$ is always 1, so:

$$\begin{aligned} Q(G,\mathcal{C}) =& \frac{1}{\sum_{i,j=1}^{n} W_{ij}} \cdot \sum_{i,j=1}^{n} \left(W_{ij} - \frac{(\sum_{j=1}^{n} W_{ij}) \cdot (\sum_{i=1}^{n} W_{ij})}{\sum_{i,j=1}^{n} W_{ij}} \right) = \\ =& \frac{1}{\sum_{ij} W_{ij}} \cdot \sum_{ij} W_{ij} - \frac{\sum_{ij}((\sum_j W_{ij}) \cdot (\sum_i W_{ij}))}{(\sum_{ij} W_{ij})^2} = \\ =& 1 - \frac{(\sum_i \sum_j W_{ij}) \cdot (\sum_j \sum_i W_{ij})}{(\sum_{ij} W_{ij})^2} = 1 - 1 = 0. \end{aligned}$$

In the case in which we have no loops and the communities are singletons, the modularity will be negative. Indeed $\delta(c_i, c_j) = 0$ for $i \neq j$ so:

$$Q(G,\mathcal{C}) = \frac{1}{2w} \sum_{i=1}^{n} \left[0 - \frac{(K_i)^2}{2w} \right] = - \sum_{i=1}^{n} \frac{K_i^2}{4w^2}.$$

Similarly to Definition 1.18, we can define the modularity of a single cluster, by considering only the nodes inside it.

Definition 1.5.5 (Modularity of a single cluster)**.** Given a weighted network $G = (V, E)$ partitioned into $\mathcal{C} = (C_1, C_2, \dots, C_p)$ communities, the modularity of each cluster $C_h \in \mathcal{C}$ is given by:

$$Q(G, C_h) = \frac{1}{2w} \sum_{i,j \in C_h} \left[W_{ij} - \frac{K_i \cdot K_j}{2w} \right]. \tag{1.19}$$

where w is the weight of the whole network.

We can rewrite the previous formula in a more convenient way. Recall that $w(C)$ indicates the weight of a cluster C (Eq. (1.10)) and consider the sum of the degrees of all the nodes in C, that is given by:

$$K(C) = \sum_{i \in C} K_i. \tag{1.20}$$

Then the modularity of a single cluster (Eq. (1.19)) is equal to:

$$\begin{aligned} Q(G, C) =& \frac{1}{2w} \sum_{i,j \in C} \left[W_{ij} - \frac{K_i \cdot K_j}{2w} \right] = \\ =& \frac{1}{2w} \left[\sum_{i,j \in C} W_{ij} - \frac{\sum_{i,j \in C} K_i \cdot K_j}{2w} \right] = \\ =& \frac{1}{2w} \left[2 \cdot w(C) - \frac{K(C)^2}{2w} \right]. \end{aligned} \tag{1.21}$$

Remark. The sum of $Q(G, C)$ for all the communities of the clustering returns exactly the modularity of the partition, i.e.:

$$Q(G, \mathcal{C}) = \sum_{h=1}^{p} Q(G, C_h). \tag{1.22}$$

Modularity Drawbacks

A typical problem connected with the modularity is the so called *resolution limit*. Basically it says that the modularity depends on the number of edges of the network and prevents the detection of small communities that are often merged into bigger ones (see [24]). Furthermore the modularity tends to be larger on larger networks, so that comparing values of modularities in general might be unreliable. A consequence is that computing the modularity of a partition is quite meaningless if not compared to other partitions of the same graph. Finally also partitions of random networks might reach high values

of modularity.
Despite the problems exposed we decided to adopt this quality measure because it is agrees with human intuition on many instances and it requires a low computing time to be calculated for a partition.

1.5.3 Modularity-driven approaches

Since modularity measures in some way the goodness of a partition, ideally it is natural to look for the partition with the highest modularity. However, exploring all the partition and successively picking the best one is infeasible since, as said in Section 1.3, the number of possible partition is excessively large. More formally it was proved by Brandes et al. in [23] that maximizing modularity is an NP-complete problem[6]. This means that there is no chance of finding an efficient (polynomial-time) algorithm that computes a maximum modularity partition on all problem instances. As a consequence this justifies the use of heuristics for modularity optimization.
As we will see in Chapter 2.1 some efficient and widely used algorithms proceed in building the communities by merging nodes in a greedy manner and they make use of modularity as a decision criteria to stop the merging of nodes when modularity cannot be improved (with the drawback of stopping often just in local maxima).

[6]NP-complete: "nondeterministic polynomial-time complete".

Chapter 2

Community Detection Algorithms

2.1 Community detection on static networks

2.1.1 Related works - static case

As we said in Section 1.3.1, we will focus on the problem of network partitioning into non-overlapping communities: we have to assign each node of the graph to exactly one community.
There are many community detection methods and they are often classified into categories based on the strategy used to identify the clusters. In most applications, however, just a few popular algorithms are employed. Some examples of the most adopted strategies are:

- Optimization-based methods. The goal of these algorithms is to find the maximum reached by a function that evaluates the quality of the partition, like modularity (see Definition 1.5.4). As we said previously in Section 1.5.3, finding the optimal value of modularity is an NP-complete problem, so these algorithms are heuristics that usually find just an approximation of the optimal solution. An example is the CNM method[1] in which, starting with the nodes into singleton communities, we repeatedly merge the two communities whose amalgamation produces the largest increase in modularity (see [19]). Another example is the Louvain algorithm, that we will explain in Section 2.1.2.

- Paths-based methods. We grouped into this category two types of algorithms.

    1. The first type is based on random walks: the idea is that, since clusters have high internal edge density and are well separated

[1] Named after the developers Clauset, Newman and Moore.

from each other, a random walker would be trapped in each cluster for quite some time before finding a way out and migrating to another cluster. Algorithms based on random walks are for example Walktrap (see [25]) and Infomap (see [26]).

2. The second type is based on shortest paths: the idea is that, if two communities are joined by only a few inter-community edges, then all the shortest paths from nodes in the first community to nodes of the second one must run through one of those few edges. This is the idea behind the algorithm of Girvan and Newman. It starts by placing all the nodes into the same community and by calculating the "betweenness" of each edge of the graph, that is a measure that basically counts how many shortest paths of the graph pass through that edge. They repeatedly remove the edges with the highest betweenness and update the betweenness of the remaining edges ([18]).

- Spectral methods. They typically detect the communities by using the eigenvalue spectrum or other spectral properties of the adjacency matrix (see [27]).

For further information about community detection methods one can refer to many surveys (see for example [9], [10], [28] and [29]). In the next section we will focus on a very fast and widespread algorithm based on modularity optimization: the Louvain algorithm.

2.1.2 Louvain algorithm

It was first published in the paper *Fast unfolding of communities in large networks*, written by Blondel, Guillaume, Lambiotte and Lefebvre in 2008 (see [31]). It was named Louvain algorithm because it was devised at the University of Louvain. It is a fast heuristic method to detect communities in static networks, based on modularity optimization.

It operates in two phases that are repeated iteratively. The idea is that in the first phase we find small communities by optimizing modularity locally around all the nodes, while in the second phase we build a new network by grouping each community into a unique node and then restart with phase one. Let us analyse these two phases in detail.
Given a network $G = (V, E)$, we start by assigning each node of the network to a different community, obtaining a partition $\mathcal{I} = \{\{i\} | i \in V\}$.

1. Take a node $i \in V$. Consider all its neighbours $j \in N(i)$ and evaluate the changing in modularity $\Delta Q(i \to c_j)$ that would occur if we place the node i into the community c_j. After evaluating $\Delta Q(i \to c_j)$ for each j we denote with l the neighbouring node for which the gain in modularity $\Delta Q(i \to c_l)$ is maximum (in case of a tie use a breaking rule). Now:

   - if $\Delta Q(i \to c_l) > \epsilon$, where ϵ is a threshold set at the beginning[2], we move node i into the community c_l;
   - if $\Delta Q(i \to c_l) \leq \epsilon$, we leave the node i were it was. Notice that in this case the variation of modularity $\Delta Q(i \to c_i)$ is 0 because nothing changed.

   We then take another node j and repeat the same operation, and so on for all the nodes of the graph.
   After considering all the nodes one time, we repeat the round until no improvement is possible, i.e. until for every node i we have that $\Delta Q(i \to c_j) \leq \epsilon$, $\forall j \in N(i)$. At the end of this phase we will get a partition $\mathcal{C} = (C_1, C_2, \dots, C_p)$ of the network.

   Remark. Notice that the output of the algorithm depends on the order in which the nodes are considered in this first phase. The authors stated that preliminary results on several test cases seem to indicate that the ordering of the nodes does not have a significant influence on the modularity but it may affect the computation time. The role played by the ordering though has still to be studied further.

2. We build a new network starting from the partition $\mathcal{C} = (C_1, C_2, \dots, C_p)$ we just found, with:

   - a node for each community C_h, $h \in \{1, \dots, p\}$;
   - a loop with weight $2w(C_h)$ attached to the node that represents the community C_h, recalling that $w(C_h)$ is the weight of the community C_h (see Eq. (1.10));
   - an edge between the node that represents C_h and the one that represents C_l with weight $s(C_h, C_l)$, recalling that $s(C_h, C_l)$ is the weight of the links separating C_h from C_l (see Eq. (1.12)).

   We then put each node into a singleton community and go back to phase 1.

[2] We take $\epsilon \geq 0$. As ϵ increases, the overall running time of the algorithm decreases.

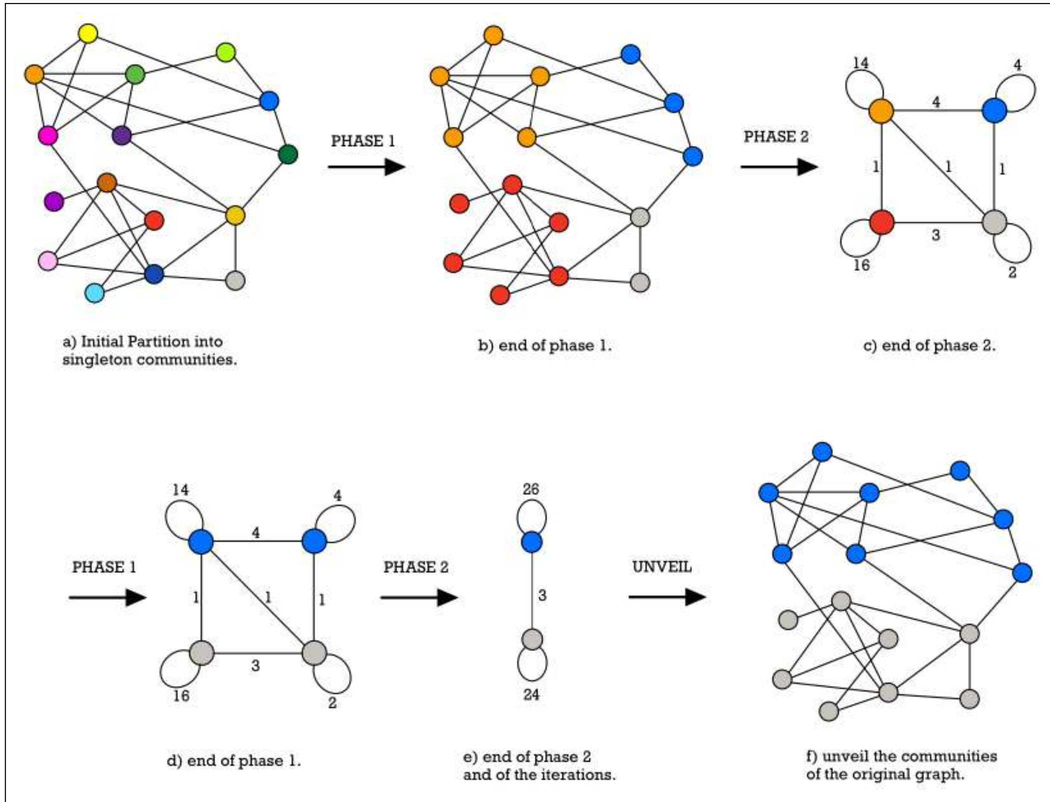

Figure 2.1: Louvain algorithm.

We repeat these phases until there are no more changes, i.e. until we start phase 1 and we end phase 1 without any modification of the communities. At this point the algorithm stops.
The result of the algorithm are the so called meta-nodes, each one representing a different community of the nodes of the previous phase. This algorithm naturally incorporates a hierarchy, each level of the hierarchy corresponding to an iteration of the combination phase1-phase2. To discover the partition of the original graph $G = (V, E)$ we have to unveil these meta-nodes and assign each node of the original graph to the community represented by the final meta-nodes. A clarifying example is shown in Figure 2.1.

Remark. Notice that the variation of modularity $\Delta Q(i \to c_j)$ that occurs if we move node i into the community of node j can easily be computed and this is one of the main advantages of the algorithm.
Let us focus on the calculations. Recall that the modularity of a partition is equal to the sum of the modularity of each cluster of the partition (see Eq. (1.22)). If we move node i into the community of node j, the only clusters that change are the one of i and the one of j. Let us indicate with C the

community of i and with D the community of j. The variation of modularity, if we remove i from C and place it into D, is given by:

$$\Delta Q(i \to D) = \; Q(D \cup \{i\}) + Q(C \setminus \{i\}) - Q(D) - Q(C). \tag{2.1}$$

Lemma 1. Notice that in general, given a cluster B and a node i, we have:

$$\begin{aligned} Q(B \cup \{i\}) =& \frac{1}{2w} \sum_{a,b \in B \cup \{i\}} \left[W_{ab} - \frac{K_a K_b}{2w} \right] = \frac{1}{2w} \sum_{a,b \in B} \left[W_{ab} - \frac{K_a K_b}{2w} \right] + \\ &+ \frac{1}{2w} \sum_{a \in B} 2 \left[W_{ai} - \frac{K_a K_i}{2w} \right] + \frac{1}{2w} \left[W_{ii} - \frac{K_i^2}{2w} \right] = \\ &= Q(B) + \frac{1}{2w} \sum_{a \in B} 2 \left[W_{ai} - \frac{K_a K_i}{2w} \right] + \frac{1}{2w} \left[W_{ii} - \frac{K_i^2}{2w} \right]. \end{aligned}$$

By letting $B = C \setminus \{i\}$, we obtain that:

$$\begin{aligned} Q(C) =& Q(C \setminus \{i\} \cup \{i\}) = \\ &= Q(C \setminus \{i\}) + \frac{1}{2w} \sum_{a \in C \setminus \{i\}} 2 \left[W_{ai} - \frac{K_a K_i}{2w} \right] + \frac{1}{2w} \left[W_{ii} - \frac{K_i^2}{2w} \right]. \end{aligned}$$

Now, if we put everything together in Eq. (2.1), we obtain that:

$$\begin{aligned} \Delta Q(i \to D) &= Q(D \cup \{i\}) + Q(C \setminus \{i\}) - Q(D) - Q(C) = \\ &= Q(D) + \frac{1}{2w} \sum_{a \in D} 2 \left[W_{ai} - \frac{K_a K_i}{2w} \right] + \frac{1}{2w} \left[W_{ii} - \frac{K_i^2}{2w} \right] + \\ &+ Q(C) - \frac{1}{2w} \sum_{a \in C \setminus \{i\}} 2 \left[W_{ai} - \frac{K_a K_i}{2w} \right] - \frac{1}{2w} \left[W_{ii} - \frac{K_i^2}{2w} \right] - \\ &- Q(D) - Q(C) = \\ &= \frac{1}{2w} \sum_{a \in D} 2 \left[W_{ai} - \frac{K_a K_i}{2w} \right] - \frac{1}{2w} \sum_{a \in C \setminus \{i\}} 2 \left[W_{ai} - \frac{K_a K_i}{2w} \right] = \\ &= \frac{1}{w} \sum_{a \in D} \left[W_{ai} - \frac{K_a K_i}{2w} \right] - \frac{1}{w} \sum_{a \in C \setminus \{i\}} \left[W_{ai} - \frac{K_a K_i}{2w} \right]. \end{aligned}$$

In conclusion, when we consider a node i and look for the neighbour j for which the gain in modularity is maximum, we just have to maximize:

$$\sum_{a \in c_j} \left[W_{ai} - \frac{K_a K_i}{2w} \right], \tag{2.2}$$

because all the other quantities do not depend on j.

Remark. This algorithm has many advantages. First of all, the steps are intuitive and easy to implement. Moreover, the algorithm is very fast. This is due to the fact that the gain in modularity is fast to compute, as we explained in the remark above. In addition, the number of communities decreases drastically after just a few iterations, so that most of the running time is concentrated on the first iteration.
One drawback of the algorithm is that it tends to produce large communities that contain a large fraction of nodes, even in cases in which we intuitively expect smaller communities. Thanks to the intrinsic multi-level nature of the algorithm though, this problem seems to be circumvented. Indeed we could just stop the iteration at any intermediate step and unveil the communities of the original graph basing on that step. Looking at Figure 2.1 for example, we could just stop at the first iteration and return the graph partitioned into four communities. This suggests us that the intermediate solutions found by the algorithm may also be meaningful.
In Listing 2.1 we report a pseudocode of the Louvain algorithm.

Listing 2.1: Pseudocode of Louvain algorithm.

```
~~~~~~~ PSEUDOCODE OF LOUVAIN ALGORITHM ~~~~~~~

INPUT: A static network G = (V, E),
       a threshold ϵ ≥ 0.
OUTPUT: A partition 𝒞 = (C_1, C_2, ..., C_p) of V.

Initial Partition: ℐ = {{i}|i ∈ V}.

Repeat:

   PHASE 1.
   Repeat:
      for i ∈ V:
         l = argmax_{j∈N(i)} ΔQ(i → c_j)
         if ΔQ(i → c_l) > ϵ:
            move i → c_l
   Until no more changes.

   PHASE 2.
   Build a network with:
   . V = {C_1, C_2, ..., C_p} found at phase 1;
   . w_ii = 2 w(C_i)  ∀i ∈ {1, ..., p};
   . w_ij = s(C_i, C_j)  ∀i < j ∈ {1, ..., p}.
   Place the nodes into singleton communities.

Until no more changes.

Return: a partition 𝒞 of the nodes of the
       initial graph (by repeatedly breaking up
       the meta-nodes of the built networks).

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

2.2 Community detection on temporal networks

2.2.1 Related works - temporal case

The literature on community detection on temporal networks is very vast and not yet unified. Many algorithms have been proposed and some attempts have been made to classify them, while regarding on their comparison the literature is still scarce (see the surveys [1], [32], [33], [34], [30]).

We present a classification into three classes, similar to the ones defined by Rossetti and Cazabet (see [1]).

1. Instant-Optimal. The idea of these algorithms is to apply at each snapshot a static algorithm for community discovery. Then they match the communities at the current time step with the ones found at the previous time steps to track the evolution of the communities. Usually this type of approach produces less smooth communities but with higher quality.
   An example is the algorithm of Green et al. (see [12]).

2. Temporal Trade-Off. The algorithms of this class take into account the communities or the network of the previous time steps to identify the communities in the current one. This can be done in different ways:

   (a) Update by Global Optimization. Methods in this subcategory use the partition found at time step $t-1$ to initialize a static algorithm at time t. An example is the algorithm of Bansal (see [35]). It is based on the CNM algorithm (see Section 2.1.1) and the idea is that, given a modified edge e_{ij} at time t, we replicate the combination steps of time step $t-1$ until node i or j are encountered. Then we switch back to the CNM algorithm and continue as in the static case. Another famous example is the algorithm of Aynaud and Guillaume, that at time step t initializes the Louvain algorithm using the communities found at time step $t-1$. We will explain it in Section 2.2.2.

   (b) Update by a Set of Rules. Methods in this subcategory consider the list of network changes that occurred between the previous step and the current one, and define a list of rules that determine how networks changes lead to communities update (see [14]).

   (c) Informed by Multi-Objective Optimization. This approach optimizes a quality function of the form: $c = \alpha CS + (1-\alpha)CT$, with

CS the cost associated with current snapshot (i.e. how well the community structure fits the graph at time t), and CT is the cost associated to the smoothness with respect to the past history (i.e. how different is the actual community structure with respect to the one at time $t-1$) and $\alpha \in [0, 1]$ is a trade-off parameter.

(d) Informed by Network Smoothing. Methods in this subcategory look for communities at t by running a static community detection algorithm, not on the graph as it is at t, but on a version of it that is *smoothed* according to the past evolution of the network, for instance by adding weights to keep track of edges' age. An example is the ECSD algorithm, that we will explain in Section 2.2.3.

3. Cross-Time. In this case dynamic community detection is done in a single process, considering simultaneously all states of the network, for example by creating a cumulative graph built by overlapping all the snapshots.

In the following sections we will analyse in detail two algorithms that belong to the second class, so they automatically look for a trade-off between smoothness and quality. They are the algorithm of Aynaud and Guillaume, see Section 2.2.2, and the ECSD algorithm, see Section 2.2.3. They are two fast algorithms that work on weighted snapshot networks.

2.2.2 Aynaud and Guillaume's algorithm

It was first published in the paper *Static community detection algorithms for evolving networks*, written by Aynaud and Guillaume in 2010 (see [36]).
It is a fast algorithm to detect evolving communities on a snapshot network and it is based on the Louvain algorithm (notice that the author Guillaume is also one of the authors of the Louvain algorithm).
Given a snapshot network $\mathcal{G} = (G^1, \dots, G^T)$, the idea of the algorithm is to apply the Louvain algorithm at each snapshot but, instead of initializing the algorithm by placing the nodes into singleton communities, we initialize it by placing each node in the communities of the previous time step. Let us analyse in detail this process and its variants.

The first version of their algorithm, that they named *static Louvain*, consists in simply applying the Louvain algorithm to each snapshot of the network, and successively tracking the evolution of the communities across time steps, as we explained in Section 1.4.2.

They tested the algorithm on an artificial network that was built by taking a static network with ca. 9000 nodes and 24000 edges and by removing one node (and its incident edges) at each time step, until the network had just one node.
They noticed that this algorithm is very unstable, in the sense that the communities found at time step $t-1$ and the ones found at time step t were very different, even if the network did not change much between the two time steps.

This suggested to the author the idea that the communities at time step t can not be searched from scratch at each time step but have, in some way, to take advantage of the previously found communities. Hence, they developed a new algorithm, the *fully stabilized Louvain*. After the communities at time step $t-1$ have been found, we apply the Louvain algorithm at time step t but, instead of placing each node into a singleton community, we place the nodes into their belonging communities of time step $t-1$. More precisely (recalling the notation of Definition 1.3.3):

- any node $v \in V^t \setminus V^{t-1}$ is placed into a singleton community $\{v\}$;
- any node $v \in V^t \cap V^{t-1}$ is placed into the community C_h^{t-1} to which it belonged in the partition $\mathcal{C}^{t-1}$.

Remark. After the nodes are placed into their starting community, the Louvain algorithm is applied: we consider one node at a time and evaluate if it is more convenient to move it from its belonging community into a neighbouring one. We do this operation repeatedly for all the nodes until no more nodes are moved and then we start phase 2 and proceed with the standard algorithm. So the differences between this stabilized version and the original algorithm happen just in the very first step of the Louvain algorithm.

The communities detected by the fully stabilized Louvain are far more stable than the ones detected by static Louvain, so we can say that the fully stabilized Louvain algorithm is very smooth. However, smoothness is not our only target: as we have said previously, we are interested in a good trade-off between smoothness and quality. The problem is that, if we look at the last snapshots of the network, the modularity of the partitions detected by the fully stabilized Louvain might be considerably lower than the modularity of the partitions detected from scratch by the static Louvain algorithm.
To overcome this issue the authors introduced a parameter $\alpha \in [0, 1]$ that balances the trade-off between smoothness and quality. This parameter limits

the constraint of the initial partition built by the fully stabilized Louvain algorithm in this way:

- any node $v \in V^t \setminus V^{t-1}$ is placed into a singleton community $\{v\}$;
- $\alpha \cdot |V^t \cap V^{t-1}|$ nodes[3] in $V^t \cap V^{t-1}$ are placed into a singleton community;
- any other node $v \in V^t \cap V^{t-1}$ is placed into the community C_h^{t-1} to which it belonged in the partition $\mathcal{C}^{t-1}$.

Remark. Notice that this version also includes the previous ones: indeed with $\alpha = 1$ we have the static Louvain and with $\alpha = 0$ the fully stabilized Louvain. The parameter α has to be chosen depending on the context, but the authors show that even a small value, like $\alpha = 0.2$, seems a good compromise between smoothness and modularity and it achieves quite better results in terms of modularity than using $\alpha = 0$.

In the following we denote this general version of the algorithm the *Aynaud and Guillaume's algorithm.* We write a pseudocode of the algorithm in Listing 2.2.

[3]We automatically approximate it to the integer part of the number instead of writing explicitely $\lfloor \alpha \cdot |V^t \cap V^{t-1}| \rfloor$.

Listing 2.2: Pseudocode of Aynaud and Guillaume's algorithm.

```
~ PSEUDOCODE of AYNAUD & GUILLAUME's ALGORITHM ~

INPUT: A temporal network 𝒢 = (G^1, G^2, ..., T),
       a threshold ε ≥ 0,
       a parameter α ∈ [0,1].
OUTPUT: A dynamic partition 𝒞 = (𝒞^t, ..., 𝒞^T).

G^1 = (V^1, E^1)
Apply Louvain to G^1 (threshold ε) and obtain 𝒞^1.

for t = 2, ..., T:
   G^t = (V^t, E^t)
   ℐ^t Initial Partition given by:
   . a singleton community ∀i ∈ V^t \ V^{t-1};
   . a community C_i^t = C_i^{t-1} ∩ V^t,  ∀i ∈ {1, ..., p^{t-1}}
   . randomly take α · |V^{t-1} ∩ V^t| nodes and
    assign them to a singleton community.

   Apply Louvain to G^t (threshold ε),
   Initial Partition ℐ^t
   and obtain final partition 𝒞^t.

Return: a dynamic partition 𝒞 = (𝒞^1, ..., 𝒞^T).

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

2.2.3 ECSD algorithm

It was first published in the paper *Evolutionary community structure discovery in dynamic weighted network*, written by Guo, Wang and Zhang in 2014 (see [37]).
It is an algorithm to detect evolving communities on a weighted snapshot network and the name ECSD, assigned by the authors, stands for "Evolutionary Community Structure Discovery".
Before proceeding with the explanation of the algorithm we have to define some useful quantities.

The input matrix U^t. The input matrix U^t is a matrix that takes into account the network at snapshot t, described by the weighted adjacency matrix W^t, and the communities found at the previous time steps. In particular:

$$U_{ij}^t = \begin{cases} W_{ij}^t & \text{if } t = 1 \\ (1-\alpha)W_{ij}^t + \alpha U_{ij}^{t-1}\delta(c_i^{t-1}, c_j^{t-1}) & \text{if } t \geq 2. \end{cases} \tag{2.3}$$

where W_{ij}^t is the weighted adjacency matrix, c_i^t is the community to which node i belongs[4] at time t and $\alpha \in [0,1]$.
The ECSD algorithm indeed, unlike the Aynaud and Guillaume's algorithm that used only the weighted adjacency matrix W and took into account the previous community by construction of the algorithm, finds its compromise between quality and smoothness by introducing the new matrix U. The parameter α is the parameter to balance this trade-off: the higher α the more importance is given to the previous communities, the lower α more importance is given to find the best partition according to the current network.
Notice also that, by construction, with the increase of t the influence of community structure at previous time steps becomes weaker and weaker.

Variation of modularity. It is useful to calculate the variation of modularity that occurs:

1. when we move a node i from its belonging community C into another community D;
2. when we merge two communities C and D.

To compute these quantities we will use the input matrix U. Now, for brevity, we do not write the apex t that indicates the time step and we will just write

[4] If the node i is not present at time t we assume that $\delta(c_i^t, c_j^t) = 0$.

U_{ij} to indicate each element of the input matrix, at a non-specified time. The weight of the graph, with respect to the input matrix U, will be denoted by u:

$$u = \frac{1}{2} \sum_{i,j \in V} U_{ij},$$

while the weighted degree of the node i will still be denoted by K_i:

$$K_i = \sum_{j \in V} U_{ij}.$$

Remark. Actually, in the definition of modularity reported in the paper, the authors exclude from the summation the indices $i = j$, using $Q(G, \mathcal{C}) = \frac{1}{2w} \sum_{i,j=1, i \neq j}^{n} \left[W_{ij} - \frac{K_i \cdot K_j}{2w} \right] \cdot \delta(c_i, c_j)$. We believe that this is a typo because it is not in line with the statements made consequently.

We now compute the variations of modularity we said above.
We already calculated the first quantity in the remark of Section 2.1.2:

$$\Delta Q(i \to D) = \frac{1}{u} \sum_{a \in D} \left[U_{ai} - \frac{K_a K_i}{2u} \right] - \frac{1}{u} \sum_{a \in C \setminus \{i\}} \left[U_{ai} - \frac{K_a K_i}{2u} \right]. \tag{2.4}$$

Let us focus on the variation of modularity that occurs if two communities C and D are merged.
Recalling that the modularity of a partition is equal to the sum of the modularity of each cluster of the partition (see Eq. (1.22)) we have that:

$$\Delta Q(C \cup D) = Q(C \cup D) - Q(C) - Q(D). \tag{2.5}$$

In particular:

$$\begin{aligned} Q(C \cup D) =& \frac{1}{2u} \sum_{i,j \in C \cup D} \left[U_{ij} - \frac{K_i K_j}{2u} \right] = \frac{1}{2u} \sum_{i,j \in C} \left[U_{ij} - \frac{K_i K_j}{2u} \right] + \\ &+ \frac{1}{2u} \sum_{i,j \in D} \left[U_{ij} - \frac{K_i K_j}{2u} \right] + 2 \frac{1}{2u} \sum_{i \in C} \sum_{j \in D} \left[U_{ij} - \frac{K_i K_j}{2u} \right]. \end{aligned}$$

So the difference in Eq. (2.5) becomes:

$$\Delta Q(C \cup D) = \frac{1}{u} \sum_{i \in C} \sum_{j \in D} \left[U_{ij} - \frac{K_i K_j}{2u} \right]. \tag{2.6}$$

Belonging degree. The belonging degree of node i to community C (recall Definition 1.5.2) with respect to the input matrix U is:

$$BD(i,C) = \frac{\sum_{j \in C} U_{ij}}{K_i}, \tag{2.7}$$

where $K_i = \sum_{j \in V} U_{ij}$ is the weighted degree of node i with respect to the input matrix.

The algorithm

Now we can move on and describe how the ECSD algorithm works. It is quite complicated and it is based mainly on three steps. We will firstly report a pseudocode of the ECSD algorithm in Listing 2.3 and then explain each step.

Listing 2.3: Pseudocode of ECSD algorithm.

```
~~~~~~~~ PSEUDOCODE of ECSD ALGORITHM ~~~~~~~~~

INPUT: A temporal network 𝒢 = (G^1, G^2, ..., T).
OUTPUT: A dynamic partition 𝒞 = (𝒞^1, ..., 𝒞^T).

for t = 1, ..., T:

   Calculate the input matrix U^t.

   Repeat:

      1. Discover the initial community.
      2. Expand the community.

   Until every node is assigned to a community.

   3. Merge the communities.

Return: a dynamic partition 𝒞 = (𝒞^1, ..., 𝒞^T).

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

1. Discover the initial community. The basic idea is that we take a node with high degree, place its neighbours into its community and clean the community by removing the nodes with low belonging degree (the idea is similar to the algorithm presented in [38]). A pseudocode is presented in Listing 2.4.

Listing 2.4: Pseudocode of "discover the initial community".

```
~~~~~~~ Discover the initial community ~~~~~~~

INPUT: An input matrix U, a threshold θ.
OUTPUT: The initial community C.

Calculate K_i for all the nodes which are
unassigned to a community.

Sort the nodes based on K_i and take i with
the largest K_i.

Find all the neighbours of i from the
unassigned nodes with U_ij > 0.

These nodes compose an initial community C.

Repeat:

   for all j ∈ C:
      calculate BD(j,C)
      if BD(j,C) < θ:
         remove j from C

Until BD(j,C) ≥ θ, ∀j ∈ C.

Return: the community C.

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

More precisely, we calculate the weighted degree K_i for all the nodes which are unassigned to a community and then sort the nodes based on K_i and take the node i with the largest degree. At the very first iteration all

the nodes are unassigned, but, as the number of iterations grows, the nodes will be assigned to communities, so we have to consider only the unassigned ones.
We then find the neighbours of the selected node, looking only among all the unassigned nodes with $U_{ij} > 0$. Notice that $U_{ij} = 0$ only if i and j are not connected at the current time step and never belonged to the same community in the previous time steps. These nodes (i and the selected neighbours) compose an initial community C.
Finally, we calculate the belonging degree of each node in C and remove the nodes for which the belonging degree is less than a chosen threshold θ. We repeat this operation until all the nodes in C have a belonging degree greater or equal to θ. Notice that it is necessary to repeat this operation since, after the removal of some nodes, the belonging degrees of the already considered nodes might change.

2. Expand the community. The basic idea is that we take the initial community and try to expand it: we look in its neighbourhood and add the nodes with a sufficiently high belonging degree. A pseudocode is presented in Listing 2.5.

Listing 2.5: Pseudocode of "expand the initial community".

```
~~~~~~~~~ Expand the initial community ~~~~~~~~

INPUT: The initial community C, a threshold γ.
OUTPUT: The expanded community C.

Repeat:

   for all j ∈ ⋃_{i∈C} N(i) \ C:
      calculate BD(j,C)
      if BD(j,C) > γ:
         calculate ΔQ(j → C)
         if ΔQ(j → C) > 0:
            move j into C

Until no more changes.

Return: the expanded community C.

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

More precisely, we take a community C and find all the neighbours in the rest of the network. So we consider all the nodes in $N = \bigcup_{i \in C} N(i) \setminus C$. Now, for any node $j \in A$ we calculate the belonging degree of j to the community C. If it is sufficiently large, i.e. it is higher than a fixed threshold γ, we calculate the variation in modularity $\Delta(j \to C)$ that would occur if we add it into the community C, thanks to Eq. (2.4). If this variation is positive[5], we move the node j into C.
We repeat this process until there are no changes in the community C. Notice indeed that, after a node is added to C, the neighbourhood of the community changed, so it is necessary to repeat the iteration again. When the iteration does not produce any change in the community C, the phase stops and the expanded community C is returned.

3. Merge the communities. The basic idea is that we start with a partition of the network. We take all the "small" communities and then merge two of them if this increases the modularity. A pseudocode is presented in Listing 2.6.

Listing 2.6: Pseudocode of "merge the small communities".

```
~~~~~~~~~ Merge the small communities ~~~~~~~~~

INPUT: A partition 𝒞, a threshold μ.
OUTPUT: A modified partition 𝒞'.

Repeat:

   for all C ∈ 𝒞 with |C| < μ:
      C' = argmax_{D∈𝒞,|D|<μ} ΔQ(C ∪ C')
      if ΔQ(C ∪ C') > 0:
         merge C and C'

Until ∀C ∈ 𝒞 : |C| > μ or
      ∀C, D ∈ 𝒞, |C|, |D| ≤ μ : ΔQ(C ∪ D) ≤ 0.

Return: the modified partition 𝒞'.

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

[5] We suggest that we can introduce a parameter ϵ, as done in the Louvain algorithm, to reduce the computing time of the algorithm.

More precisely, we start with a partition $\mathcal{C} = (C_1, \dots, C_p)$ of the graph. We find all the communities in $\mathcal{C}$ whose number of nodes is less than a threshold μ; let us call $\mathcal{S}$ the subset of the selected communities. We then take a community $C \in \mathcal{S}$ and calculate the variation of modularity $\Delta Q(C \cup D)$ that would occur if we merge it with any other community $D \in \mathcal{S}$, thanks to Eq. (2.5). We call C' the community for which the gain in modularity is maximum. If it is positive[6], i.e. if $\Delta Q(C \cup C') > 0$, we merge the two communities.
We repeat these operations until all the communities in $\mathcal{C}$ are sufficiently large, i.e. until $\forall C \in \mathcal{C}, |C| > \mu$, or until for any pair of small communities the variation in modularity that would occur if we merged them is negative, i.e. until $\forall C$ and $D \in \mathcal{C},$ with $|C|, |D| \leq \mu, \Delta Q(C \cup D) < 0$.

[6]Again we suggest that we can introduce a parameter ϵ, as done in the Louvain algorithm, to reduce the computing time of the algorithm.

Chapter 3

Experimental Evaluation

3.1 Experimental setup

In this chapter we present the experimental testing of the Aynaud and Guillaume's algorithm[1] and the ECSD algorithm, which we explained in Section 2.2. To test the algorithms we wrote a code with Python. It is organized into four steps:

1. Build the network.
2. Detect the communities with Aynaud's and with ECSD algorithms.
3. Analyse the modularity of the communities detected in step 2.
4. Analyse the smoothness of the communities detected in step 2.

In the following sections we will explain each step of the code, presenting the fundamental tools that are needed. We report the code in the Appendix A. We then applied the code to three networks: the Sociopatterns network, an artificial network and the DBLP network. We will describe the three networks and the results obtained on them in the rest of the chapter.

Libraries

We wrote the code in Python and we used mainly four libraries:

- `networkx`. It is a widespread library to handle static networks (see [39]). In the library there are many useful functions to access and plot the networks and some algorithms for community detection.

[1] In this chapter for brevity we will call it Aynaud's algorithm.

- `tnetwork`. It is a library to handle temporal networks, written mainly by Remy Cazabet (see [40]). It can handle different representations of temporal networks, access the graph at different time steps and there are some algorithms for community detection.

- `statistics`. It is a library to compute relevant statistical quantities, like mean value, median, variance and many others (see [41]). We will use it mainly for the analysis of modularity.

- `matplotlib`. It is a very popular library for creating visualization in Python (see [42]). It is used also by the libraries `networkx` and `tnetwork` to plot the networks. In particular we used it to plot the modularity and smoothness vectors in order to better analyse the results.

3.1.1 Building the network

The library `tnetwork` provides many functions to build a temporal network and it can handle different types of representations. In particular we used the snapshot representation and we used three different methods to build the three networks:

1. *Load example graph* to build the Sociopatterns network (Section 3.2).

2. *Generate random graph* to build the artificial network (Section 3.3).

3. *Reading graph from text file* to build the DBLP network (Section 3.4).

For more details about the building functions see the code in Appendix A. After building the network we can access its main properties. Notice that at each time step the network is treated as a `networkx`-object, so all the methods of the library `networkx` can be used as well. In particular for each network we calculated:

- The total number of snapshots.

- The first and last time steps.

- The mean number of nodes per time step and the minimum and maximum values obtained.

- The mean number of edges per time step and the minimum and maximum values obtained.

3.1.2 Detecting the communities

After building the network, we can detect the communities. In the library `tnetwork` there are already implemented some algorithms for dynamic community detection. In particular we used the functions `smoothed_louvain`, which is an implementation of the Aynaud's algorithm, and `smoothed_graph`, which is an implementation of the ECSD algorithm.
By applying these algorithms we obtained two different objects:

1. `comm_ayn` (relative to the Aynaud's algorithm);

2. `comm_ecsd` (relative to ECSD algorithm).

They are `tnetwork`-objects that belong to the class `DynGraphSN`.
We can manipulate the communities with various methods. In particular we used the function `DynGraphSN.communities(t)` to access to the communities at time step t. In this way we obtained the number of communities at each snapshot and the size of each community at each snapshot. In this thesis we decided to report just the following quantities:

- We plotted the number of communities at each snapshot and we computed the mean value among all the snapshots.

- We computed the mean size (i.e. the mean number of nodes) among all the communities among all the snapshots.

- The maximum and minimum sizes reached by a community among all the snapshots.

3.1.3 Modularity analysis

In this Section we will explain how we analysed the results of modularity using tools from statistics.
For each temporal network we obtained two different modularity vectors, using the function `quality_at_each_step` of the library `tnetwork`:

1. `mod_ayn` (relative to the Aynaud's algorithm);

2. `mod_ecsd` (relative to ECSD algorithm).

Each entry of `mod_ayn` represents the time step and the relative value is the modularity (see Eq. (1.18)) of the communities obtained at that time by the Aynaud's algorithm, and analogously for `mod_ecsd` and the ECSD algorithm.

To analyse the vectors we used the library `statistics`. In particular for each modularity vector $V = (v_1, v_2, \ldots, v_T)$, where T is the number of snapshots of the network, we calculated:

- The **mean value**:
$$M(V) = \frac{1}{T}\sum_{t=1}^{T} v_t. \tag{3.1}$$
This is an average over all the time steps of the values of modularity obtained at each time step. However, as noted in Section 1.5.2, the values of modularity should be compared on the same static network and it might become misleading to compare them or averaging them between graphs of different sizes.

- The **weighted mean value**:
$$WM(V) = \frac{1}{\sum_{t=1}^{T} n^{(t)}}\sum_{t=1}^{T} v_t \cdot n^{(t)}. \tag{3.2}$$
As suggested in the library `tnetwork`, we computed a weighted average of the modularity across time steps, where the weight is given by the number of nodes $n^{(t)}$ of the network at time step t.

- The **variance**:
$$var(V) = \frac{1}{T}\sum_{t=1}^{T}(v_t - M(V))^2 = \left(\frac{1}{T}\sum_{t=1}^{T} v_t^2\right) - M(V)^2. \tag{3.3}$$
This gives us information about the dispersion of the obtained values of modularity with respect to the mean value.

  Remark. We can infer a correlation between variance and instability of the algorithm: we can reasonably suppose that if the variance is high, meaning that the values of modularity can be very different from the mean value, the algorithm is having trouble in keeping a good performance in terms of quality of the partition.

- The **standard deviation**:
$$SD(V) = \sqrt{var(V)} = \sqrt{\frac{1}{T}\sum_{t=1}^{T}(v_t - M(V))^2}. \tag{3.4}$$
Together with the variance, the standard deviation gives us information about the dispersion of the values around the mean value.

- The **median**, $median(V)$, is the value that separates the higher half of values from the lower half, i.e. if we order the values is the "middle" one (or the average of the two in the middle if the number of values is even).
- The **maximum**, $max(V)$. We already know that the maximum value that modularity can reach is 1, but it may be interesting to know if it actually reaches it or not.
- The **minimum**, $min(V)$. The modularity can be negative, but it is interesting to notice if negative values are really taken, knowing that it is always possible to put all the nodes in a unique community getting a modularity of 0.

Moreover we plotted the values of modularity in two different forms:

- **Histogram:** on the x-axis there are the values that modularity assumes (from 0 to 1, divided in intervals of length 0.05) and on the y-axes the number of time steps in which modularity actually takes a value in that interval.
- **Line Plot:** on the x-axis there are the time steps and on the y-axes a line that connects the values that modularity assumes at each time step.

After doing the analysis of the two vectors of modularity `mod_ayn` and `mod_ecsd` separatedly, we compared the values obtained by plotting them into a single figure.
Finally we calculated the distance between them using the infinity norm, which, for two generic vectors $U = (u_1, \dots, u_T)$ and $V = (v_1, \dots, v_T)$, is defined as:

$$||V - W||_\infty = \max_{i=1,\dots,T} |v_i - w_i|. \tag{3.5}$$

3.1.4 Smoothness analysis

After the modularity analysis, which expresses the quality of the partitions, we focused on the analysis of smoothness. Precisely, for each snapshot, we want to compare how the previous partition is similar to the current one, using the Jaccard index (see Eq. (1.9)) and the Rand index (see Eq. (1.8)). For each temporal network we obtained four different vectors:

- `smo_jac_ayn` (relative to Aynaud's algorithm and Jaccard index);
- `smo_ran_ayn` (relative to Aynaud's algorithm and Rand index);

- `smo_jac_ecsd` (relative to ECSD algorithm and Jaccard index);
- `smo_ran_ecsd` (relative to ECSD algorithm and Rand index).

Each entry of `smo_jac_ayn` represents the time step (from 1 to $T-1$) and the relative value is the Jaccard index between the partitions found by Aynaud in that snapshot and the communities found by Aynaud in the next snapshot. Analogously for the other vectors `smo_ran_ayn`, `smo_jac_ecsd` and `smo_ran_ecsd`.
For each of the vectors we computed:

- The mean value (see Eq. (3.1)).
- The weighted mean value, with weights the number of nodes per snapshot (see Eq. (3.2)).
- The variance (see Eq. (3.3)).

Finally we plotted in a first figure `smo_jac_ayn` and `smo_jac_ecsd` to compare the smoothness of the two algorithms with Jaccard index and in second figure `smo_ran_ayn` and `smo_ran_ecsd` to compare the smoothness of the two algorithms with Jaccard index.

Remark. To compute these values we had to build two functions ad hoc. Indeed in the library `tnetwork` the smoothness between two consecutive partitions is computed using the function `similarity_at_each_step` that, instead of Jaccard or Rand indices, uses another similarity measure: the normalized mutual information. For this reason we built two new functions, `myjaccard` and `myrand`, that we used as new input of the function `similarity_at_each_step`. Our functions, given two partitions of the nodes of two consecutive snapshots (or more generally two partitions of two static graphs), automatically find the nodes that belong to both of them and compute the Jaccard and the Rand indices on this set of nodes. To build them we exploit the `jaccard_score` and the `rand_score` of the library `sklearn`. See the code in the Appendix A for more details.

Sociopatterns network S1	
Number of snapshots	87
First time step	1353301200
Last time step	1354032000
Mean number of nodes per time steps	65
Max number of nodes	123
Min number of nodes	2
Mean number of edges per time steps	81
Max number of edges	244
Min number of edges	1

Table 3.1: Description of the Sociopatterns network S1.

3.2 Sociopatterns network

3.2.1 Description of Sociopatterns network

The library `tnetwork` includes a few dynamic graphs that can be loaded with one command in the chosen format. We downloaded the "sociopatterns2012" network in the format of snapshot network.

Sociopatterns is an interdisciplinary research collaboration that adopts a data-driven methodology to study social dynamics and human activity (see [43]). The "sociopatterns2012" dataset contains the contacts between the students of 5 classes in a high school in France during one week of November 2012. The original graph has 11273 snapshots but, as shown in the library, it is possible to study dynamic network with a lesser temporal granularity than the original data, thus yielding snapshots covering larger periods. We decided to aggregate the snapshots using the function `aggregate_time_period` provided by the library and we obtained a network with 87 snapshots, one every hour. The first snapshot is taken at POSIX time 1353301200, which corresponds to the 19th November 2012, while the last snapshot at POSIX time 1354032000, which corresponds to the 27th November 2012.

The main characteristics of the Sociopatterns network, for brevity S1, are shown in Table 3.1.

3.2.2 Results on Sociopatterns network

In Table 3.2 and Figures 3.1-3.6 we present the results we obtained on the Sociopatterns network.

Notice that the mean number of communities detected by Aynaud is higher than the number of communities detected by ECSD. Consequently, the average size of communities per time step of Aynaud's communities is lower than the one of ECSD's communities.
From Figure 3.4 and from the mean and weighted mean values it seems that the modularity obtained by Aynaud's algorithm is a bit higher than with ECSD algorithm and it is visible from the histograms in Figures 3.2 and 3.3 how the values are distributed among the time steps. The infinity norm of the difference is approximately 0.2, as reported in Table 3.2.
Concerning smoothness, we notice that the values obtained with Rand index are quite high, while the mean value for the smoothness computed with Jaccard index is very low, even if it reaches some spikes of almost 0.7. We suppose that these behaviours might be originated from the fact that the set on which the indices are computed is very small. In this way, as we noted in the Remark of Section 1.4.1, when computing the Rand indices (see Eq. (1.8)) it prevails the number n_{00}, i.e. the number of pair of nodes that are classified into distinct communities at time $t-1$ and at time t.
The Aynaud's algorithm seems more smooth than the ECSD with respect to the Jaccard index, see Figure 3.5 and Table 3.2. It is the opposite with respect to the Rand index, because the ECSD algorithm reaches some spikes of smoothness and has a mean value higher than Aynaud's algorithm, see Figure 3.6 and Table 3.2.

SOCIOPATTERNS S1	Aynaud	ECSD
Mean number of communities per time step	12	9
Mean size of communities per time step	5	7
Max size of communities	28	36
Min size of communities	2	2
Mean value modularity	0.73478	0.68495
Weighted mean value modularity	0.76829	0.70792
Variance modularity	0.01645	0.01565
St. Dev. modularity	0.12826	0.12511
Median modularity	0.76389	0.70287
Min modularity	0.0	0.0
Max modularity	0.89587	0.87023
Infinity norm of the difference of modularities	0.23345	
Mean value smoothness (Jac. index)	0.06672	0.07263
Weighted mean value smoothness (Jac. index)	0.04561	0.04914
Variance smoothness (Jac. index)	0.01247	0.01515
Mean value smoothness (Rand index)	0.83007	0.78373
Weighted mean value smoothness (Rand index)	0.8673	0.81468
Variance smoothness (Rand index)	0.03067	0.02883

Table 3.2: Results for the Sociopatterns network S1.

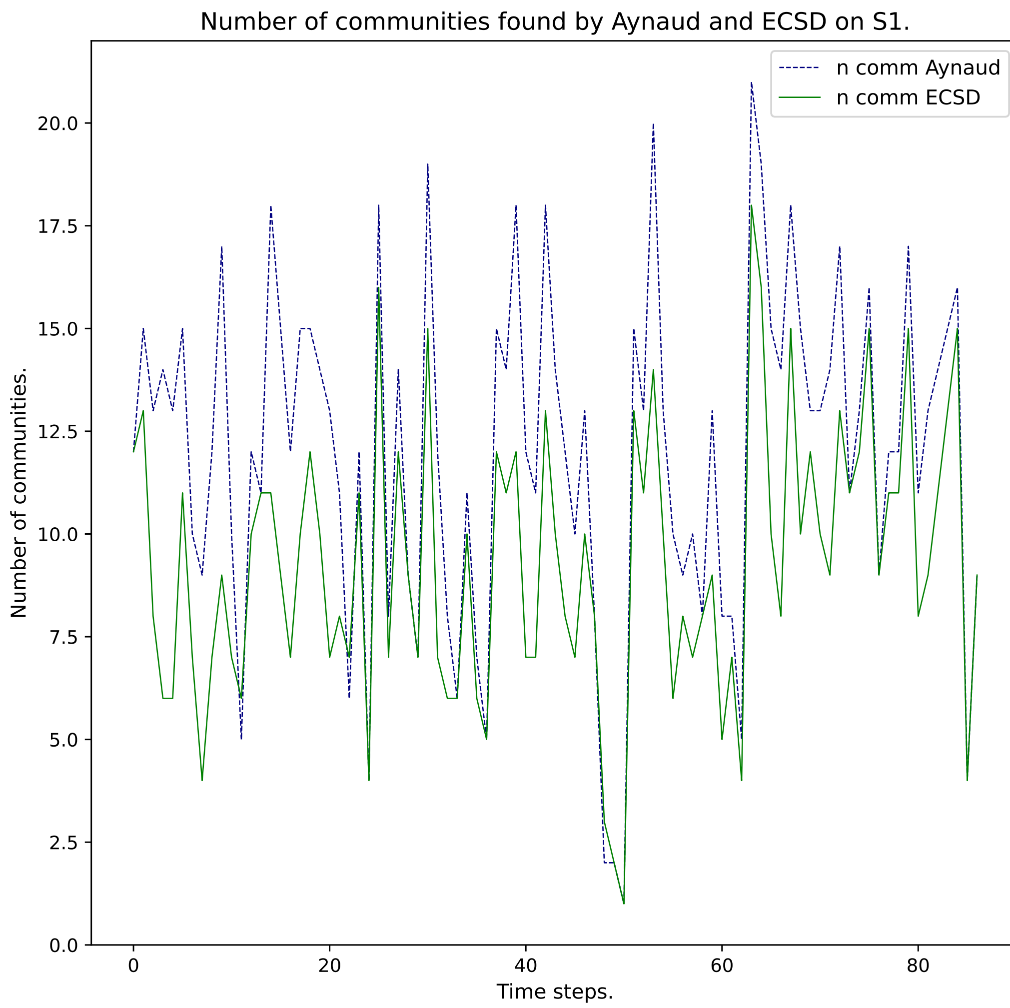

Figure 3.1: Number of communities on S1.

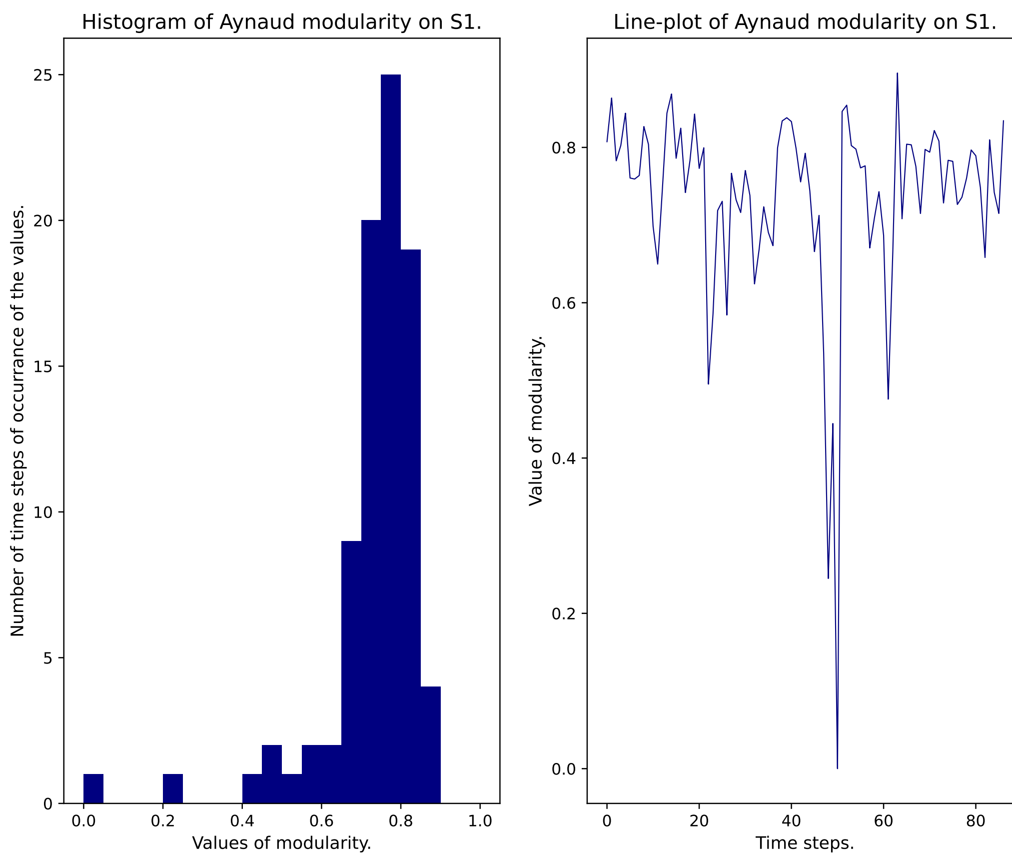

Figure 3.2: Aynaud modularity on S1.

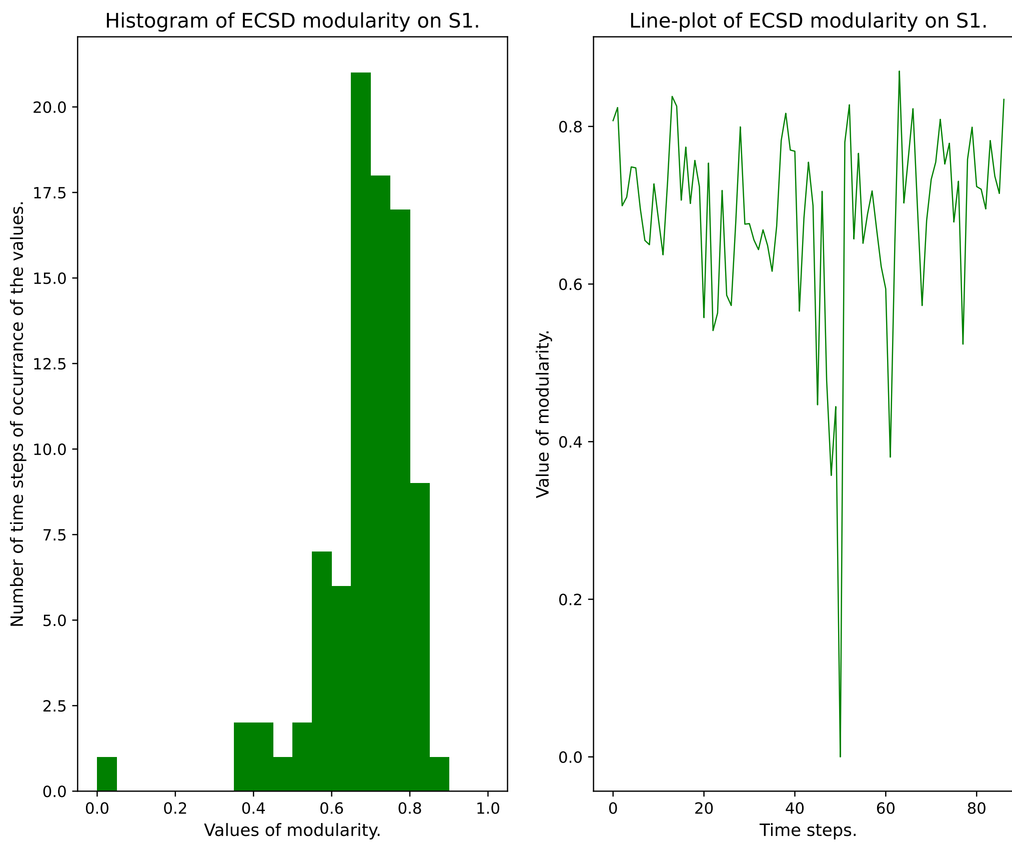

Figure 3.3: ECSD modularity on S1.

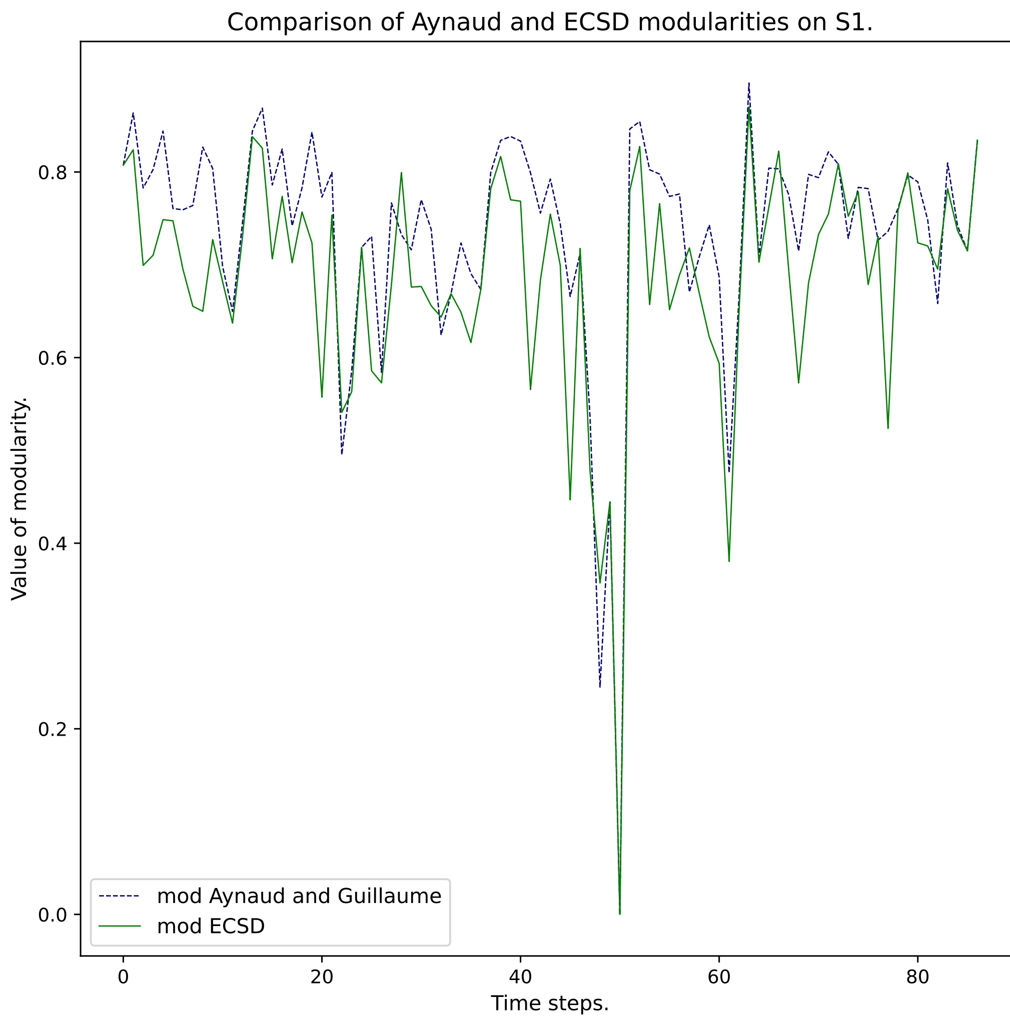

Figure 3.4: Comparison between Aynaud and ECSD modularity on S1.

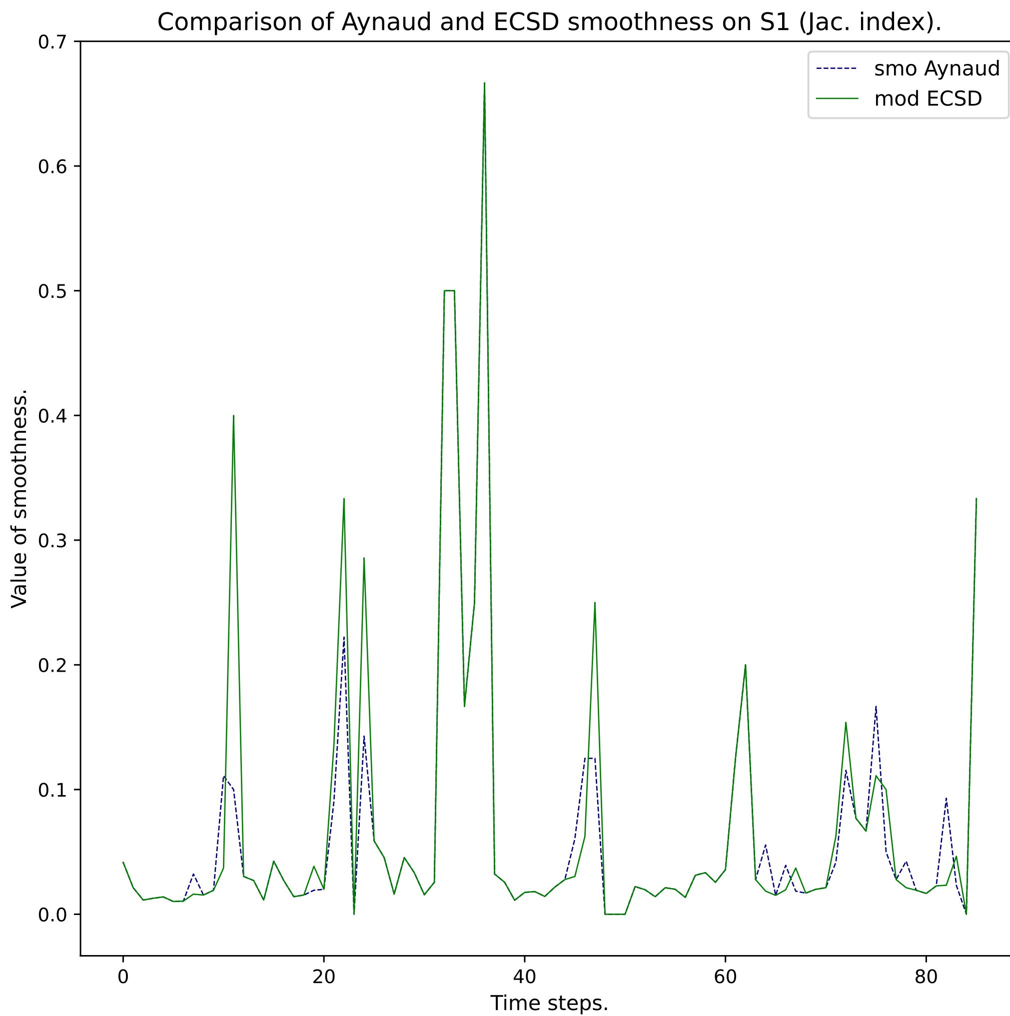

Figure 3.5: Comparison between Aynaud and ECSD smoothness (Jaccard index) on S1.

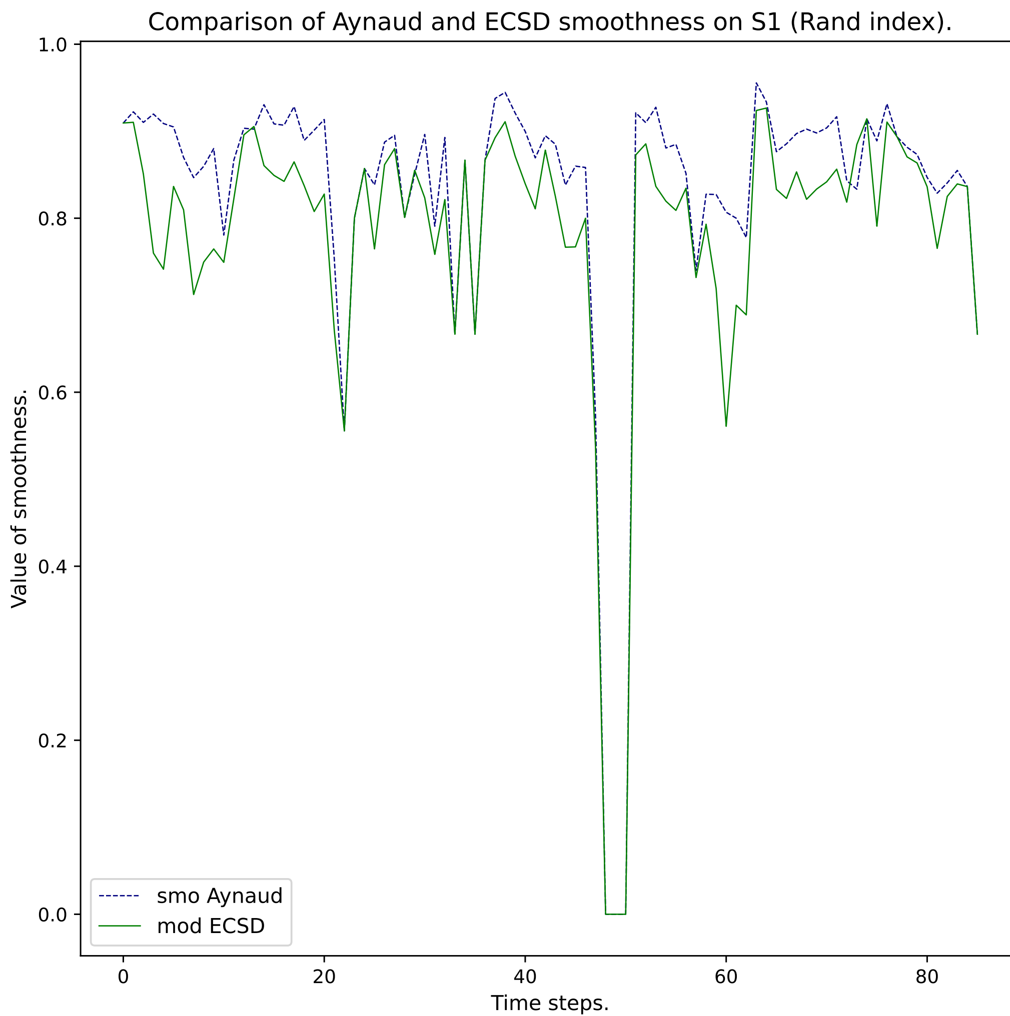

Figure 3.6: Comparison between Aynaud and ECSD smoothness (Rand index) on S1.

3.3 Artificial random generated network

3.3.1 Description of the artificial network

With the function `generate_simple_random_graph` provided by the library `tnetwork` it is possible to generate a random graph, by specifying the following parameters:

- `nb_com` = number of initial communities;
- `min_size` = size below which communities cannot be split;
- `max_size` = size above which community split;
- `operations` = number of operations to execute;
- `mu` = parameter to set how well defined is the community structure (with `mu`=0 we have cliques, as `mu` increases the density of the edges inside each communities decreases);
- `mu_noise` = parameter to set the fraction of edges randomly rewired at each snapshot.

Notice that the operations that the generator executes are the events described in Section 1.4.3. The generator automatically build a temporal networks with interval representation. With the function `to_DynGraphSN` it is possible to convert it to a snapshot network, and we can specify in input the sliding window (we used `(slices=3)`). For more details about the generator refer to the documentation of the library (see [40]).
In particular we generated several random networks using different values for the parameters. In this thesis we report the results we obtained with:

- `nb_com` = 9;
- `min_size` = 3;
- `max_size` = 35;
- `operations` = 15;
- `mu` = 0.04;
- `mu_noise` = 0.03;

The main characteristics of the generated network, for brevity G1, are shown in Table 3.3.

Generated network G1	
Number of snapshots	4094
First time step	0
Last time step	12279
Mean number of nodes per time steps	151
Max number of nodes	151
Min number of nodes	151
Mean number of edges per time steps	4110
Max number of edges	5840
Min number of edges	1816

Table 3.3: Description of the Generated network G1.

3.3.2 Results on the artificial network

As we can see the results in Table 3.4 are quite low in absolute terms and they are similar between the two algorithms. However, from Figures 3.7-3.11 we see that the algorithms behave in a strange way, even if the mean values of modularity and smoothness obtained are almost the same.
In particular, from Figure 3.10 we can see that the curve of modularity changes rapidly at some timestamps and in most of them the two algorithms assume different values. The same happens for the curve of smoothness (with Rand index) of Figure 3.11. We can infer that this happens when the network is changing; remember that the number of events that took place, determined by the parameter `operations`, is 15. So we can suppose that when the network is quite stable the algorithms behave similarly, whereas when the network is changing they behave differently and adapt differently to the change.
In particular, from Figures 3.10 and 3.11, we notice that the ECSD algorithm seems to be better both in terms of modularity and smoothness. We suppose that this might be due to the fact that placing the nodes into the previous partition for the Aynaud's algorithm might be a too strong constraint for the nodes and the algorithm is not able to rapidly catch the changes that happened, while the ECSD algorithm performs better. Both of them, however, are not optimal for this kind of network.
Finally notice from Table 3.4 that the smoothness computed with Jaccard index assumes a constant value of about 0.007, so we omit its plot.

GENERATED G1	Aynaud	ECSD
Mean number of communities per time step	2	3
Mean size of communities per time step	71	53
Max size of communities	112	112
Min size of communities	13	4
Mean value modularity	0.32016	0.36325
Weighted mean value modularity	0.32016	0.36325
Variance modularity	0.01783	0.01732
St. Dev. modularity	0.13353	0.13159
Median modularity	0.35849	0.42071
Min modularity	0.11532	0.11533
Max modularity	0.60847	0.60847
Infinity norm of the difference of modularities	0.2131	
Mean value smoothness (Jac. index)	0.00662	0.00662
Weighted mean value smoothness (Jac. index)	0.00662	0.00662
Variance smoothness (Jac. index)	0.0	0.0
Mean value smoothness (Rand index)	0.49248	0.60422
Weighted mean value smoothness (Rand index)	0.49248	0.60422
Variance smoothness (Rand index)	0.0077	0.0089

Table 3.4: Results for the generated network G1.

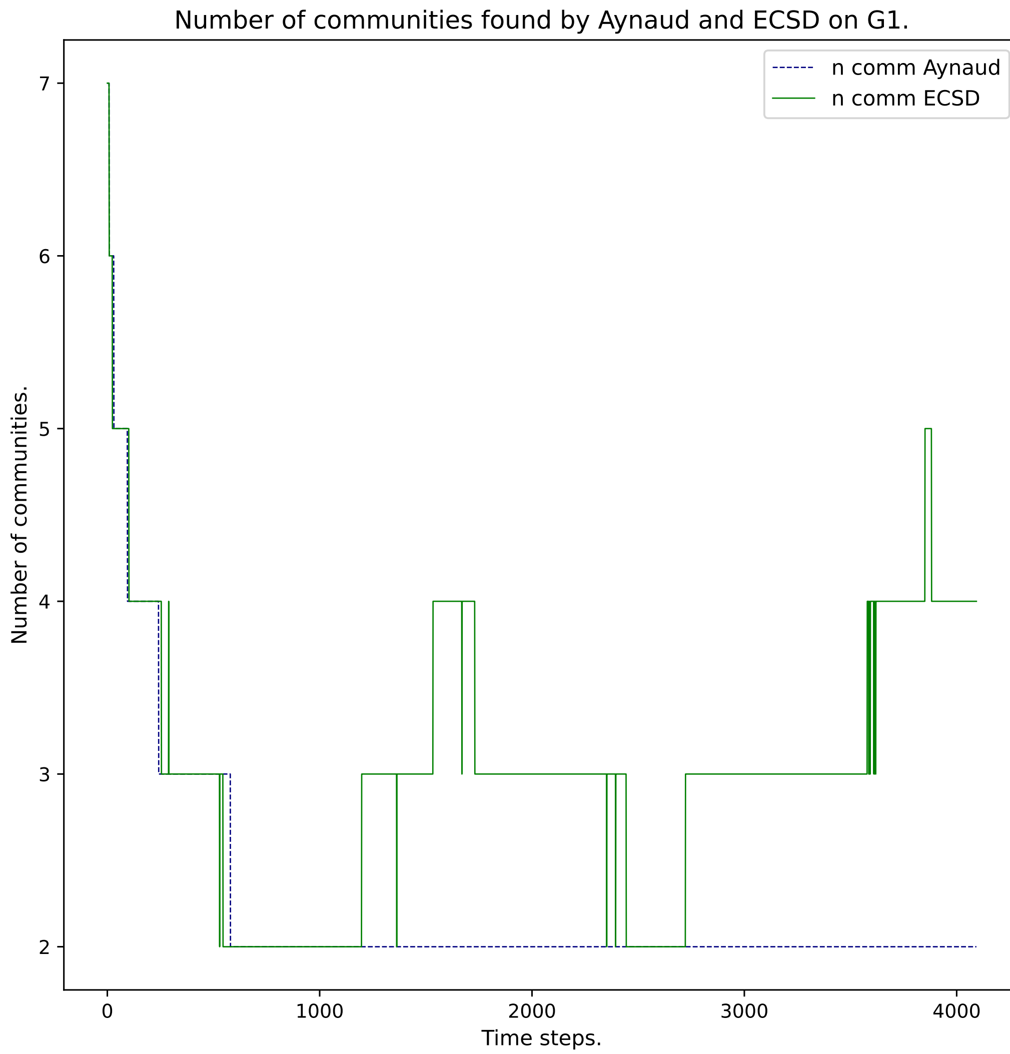

Figure 3.7: Number of communities on G1.

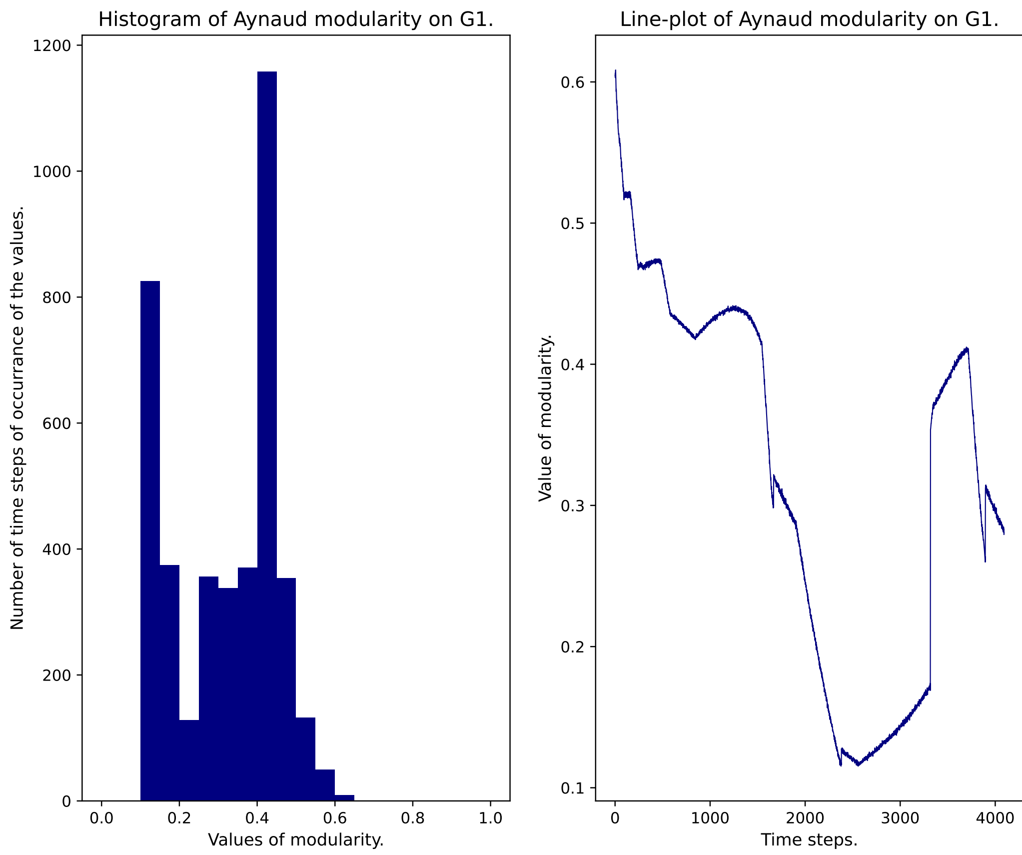

Figure 3.8: Aynaud modularity on G1.

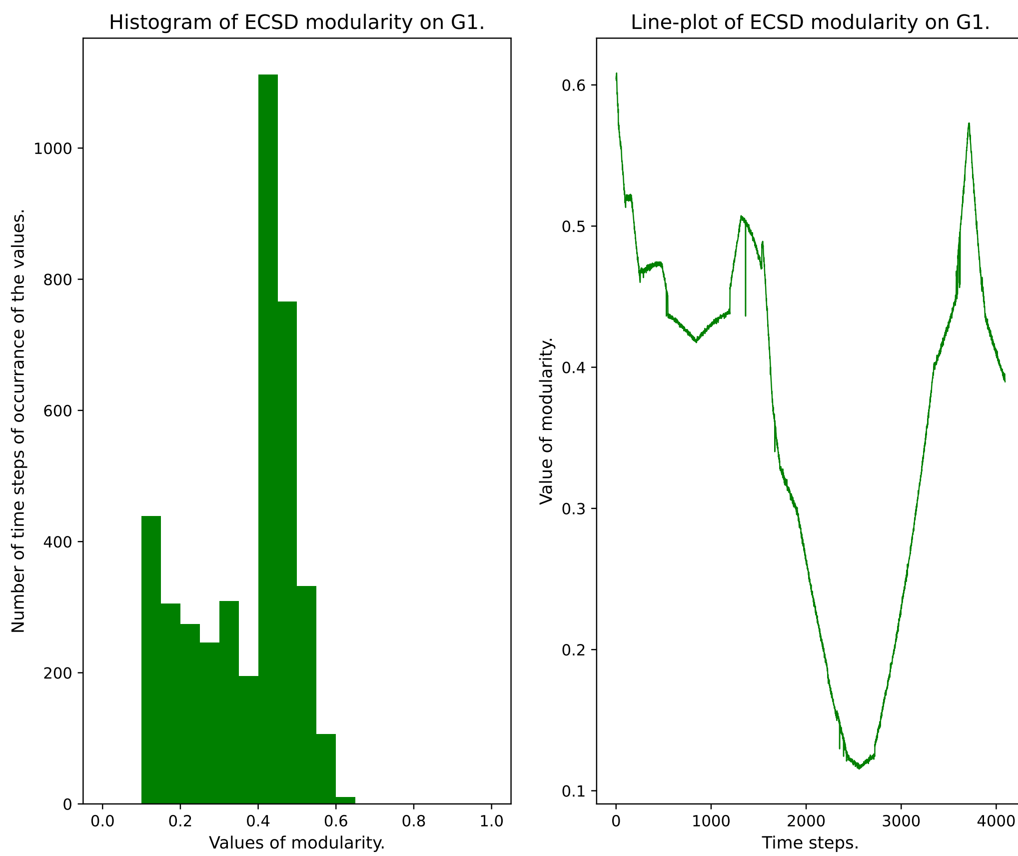

Figure 3.9: ECSD modularity on G1.

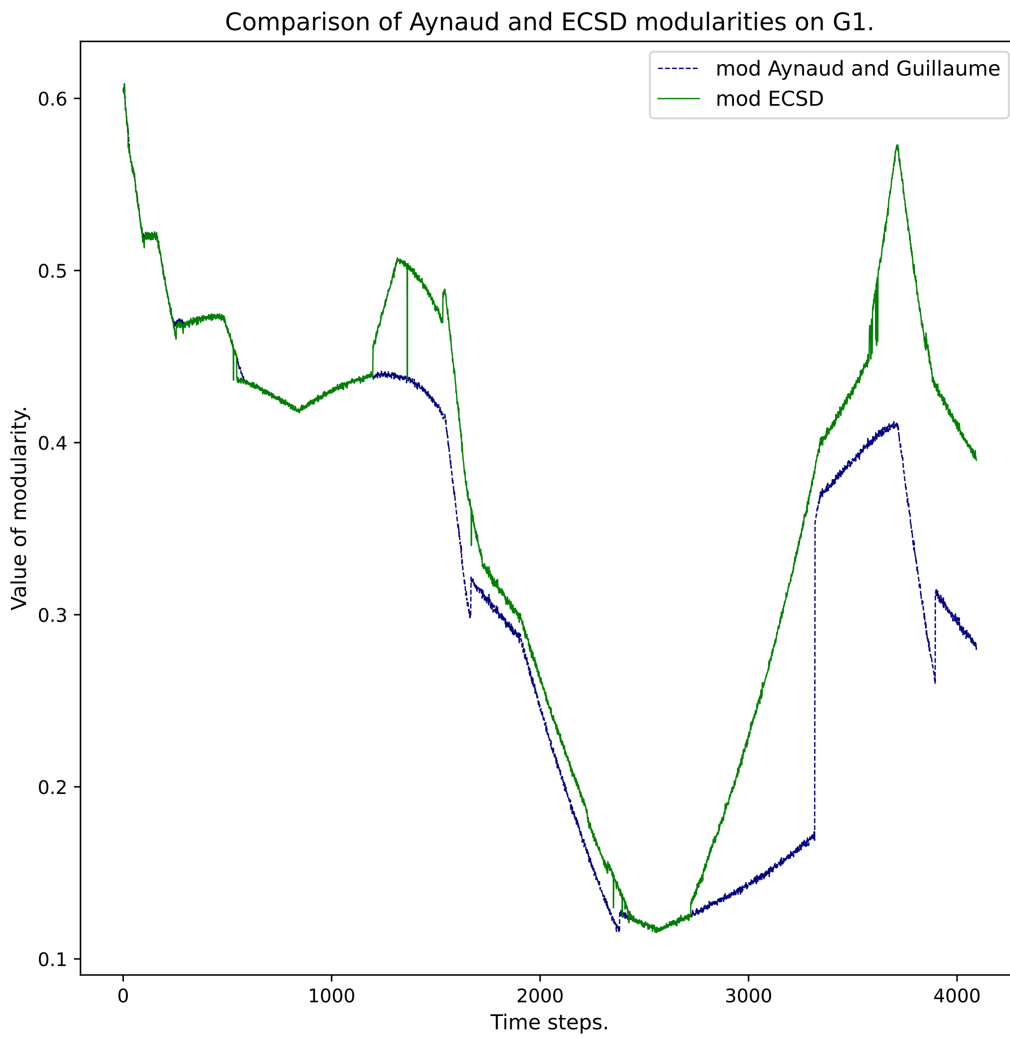

Figure 3.10: Comparison between Aynaud and ECSD modularity on G1.

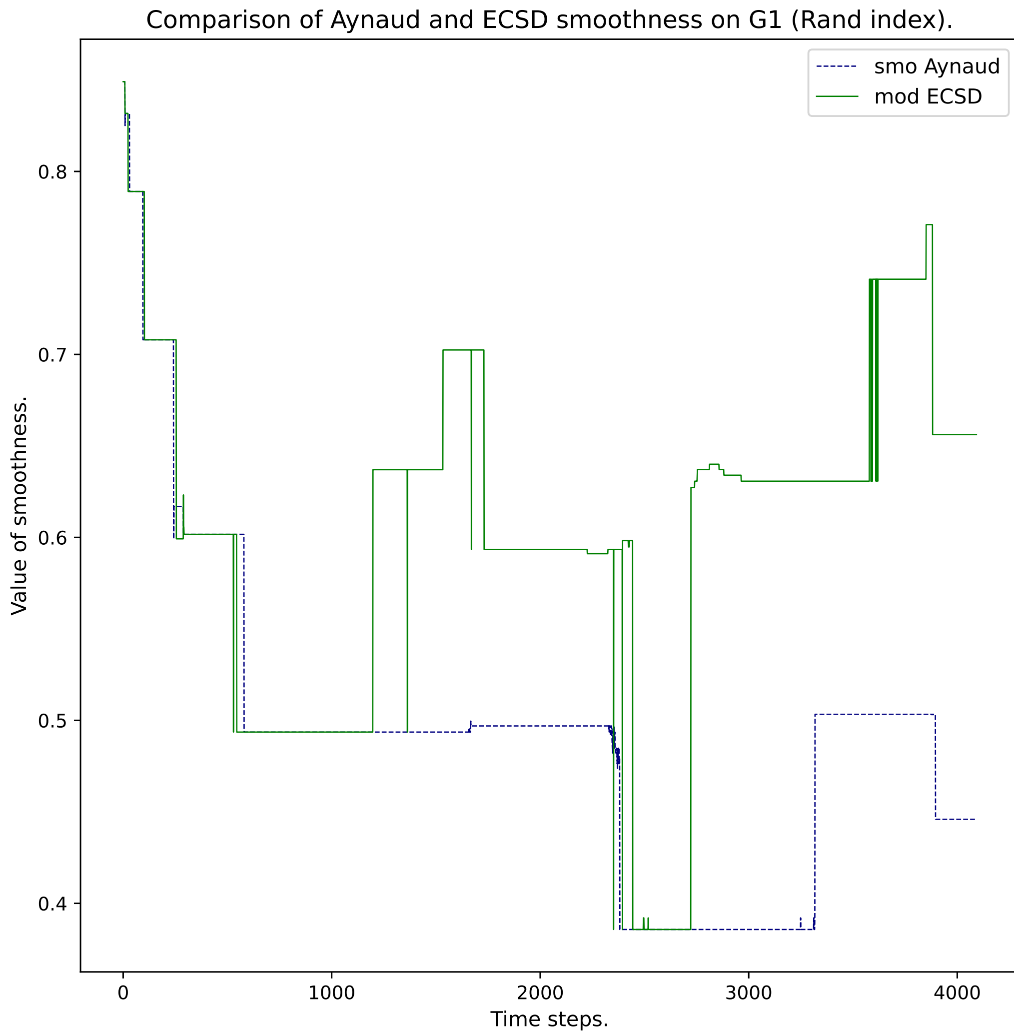

Figure 3.11: Comparison between Aynaud and ECSD smoothness (Rand index) on G1.

DBLP network D1	
Number of snapshots	30
First time step	61
Last time step	915148861
Mean number of nodes per time steps	22143
Max number of nodes	115478
Min number of nodes	1137
Mean number of edges per time steps	30412
Max number of edges	122747
Min number of edges	1016

Table 3.5: Description of the DBLP network D1.

3.4 DBLP network

3.4.1 Description of the DBLP network

The DBLP network is the collaboration graph of authors of scientific papers from DBLP computer science bibliography. Each node represents an author and an edge appears between two nodes when they publish a paper together. We found the data to build the network on the platform KONECT, which is a project in the area of network science with the goal to collect network datasets, analyse them, and make available all analyses online (see [44]). We downloaded the data, ordered them by the time stamps and then we used the function `read_interactions` of the library `tnetwork` to construct the network, by reading the first million lines of the data. With this function each snapshot contains the interactions at that moment without storing in memory the interactions of previous time steps.
The main characteristics of the DBLP network, for brevity D1, are shown in Table 3.5. Notice that the number of edges is very low with respect to the number of nodes, i.e. the network is very sparse.

3.4.2 Results on the DBLP network

The results on the DBLP network are shown in Table 3.6 and Figures 3.13-3.17.
The number of communities detected by the two algorithms is similar as shown in Figure 3.12. Since the number of edges is very low compared to the number of nodes, we suppose that the communities detected are very

small subnetworks disconnected from each other. This is confirmed knowing that the mean number of nodes per community per snapshot is 4 or 5, that is reasonable if we assume that each community is a group of authors that published a paper together.
Notice that the values of modularity obtained by the two algorithms are very high. From Figure 3.15 it is visible that the Aynaud's algorithm achieves better scores in terms of modularity, while the ECSD algorithm is loosing performance, even if both of them are really valid and get a mean value near to 1.
A strange behaviours instead is noted in terms of smoothness: both of them have a smoothness value, with respect to the Jaccard index, that is almost null, while the same quantity with respect to the Rand index is almost optimal. We suppose that the reason of this behaviour is again the fact that the two indices are computed only on a very limited set of nodes, compared to the huge number of nodes present in each snapshot. In this way if the set is very restricted and the nodes are placed into different communities the Rand index (see Eq. (1.8)) will be almost certainly 1 due to the dominance of n_{00}, while the Jaccard index (see Eq. (1.9)) will be almost certainly 0 because the term n_{11} is almost 0.
Notice that the Aynaud's algorithm on this network performs a little bit better than the ECSD one, both in terms of modularity and smoothness.

DBLP D1	Aynaud	ECSD
Mean number of communities per time step	5253	4401
Mean size of communities per time step	4	5
Max size of communities	2935	2536
Min size of communities	2	2
Mean value modularity	0.98764	0.94581
Weighted mean value modularity	0.9811	0.9036
Variance modularity	0.00005	0.00203
St. Dev. modularity	0.00741	0.04501
Median modularity	0.99042	0.96082
Min modularity	0.96845	0.85278
Max modularity	0.99623	0.99314
Infinity norm of the difference of modularities	0.12129	
Mean value smoothness (Jac. index)	0.00306	0.00176
Weighted mean value smoothness (Jac. index)	0.00046	0.00039
Variance smoothness (Jac. index)	0.00005	0.00005
Mean value smoothness (Rand index)	0.99136	0.97942
Weighted mean value smoothness (Rand index)	0.9899	0.97774
Variance smoothness (Rand index)	0.00005	0.00015

Table 3.6: Results for the DBLP network D1.

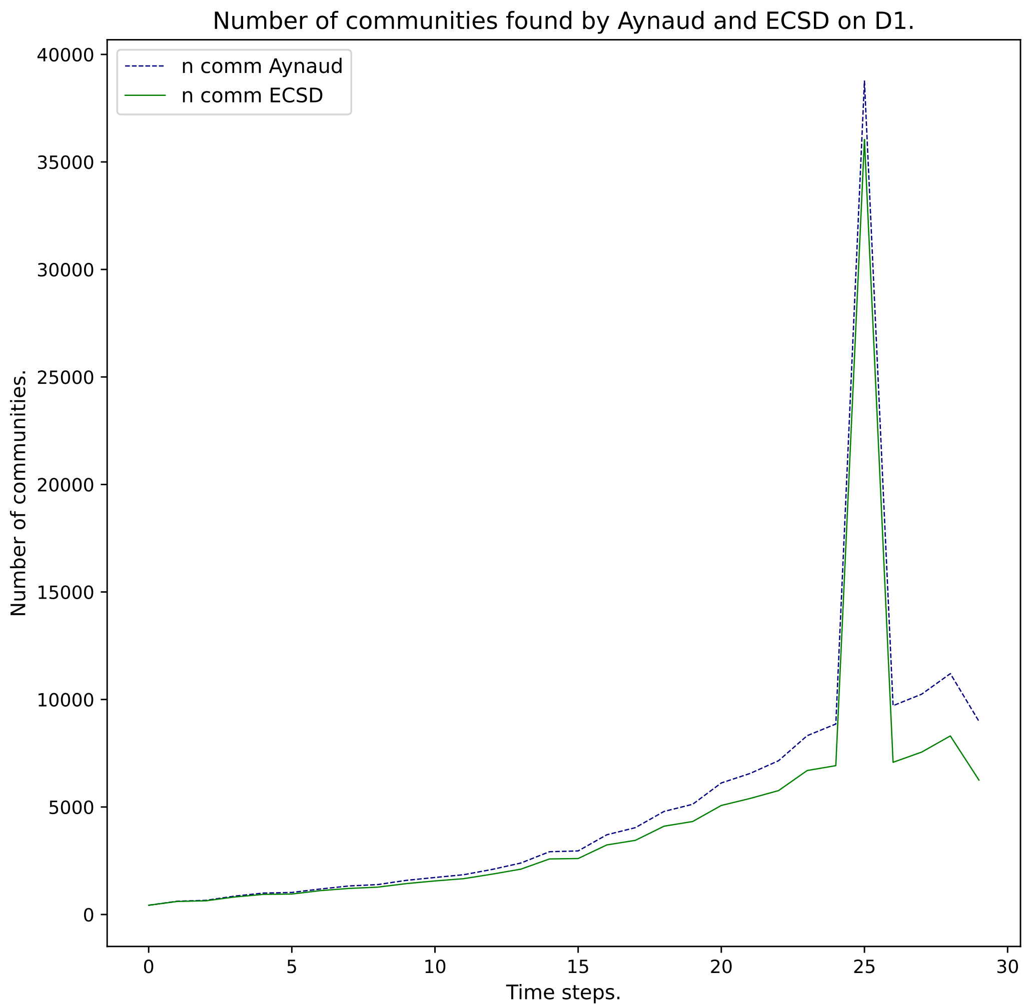

Figure 3.12: Number of communities on D1.

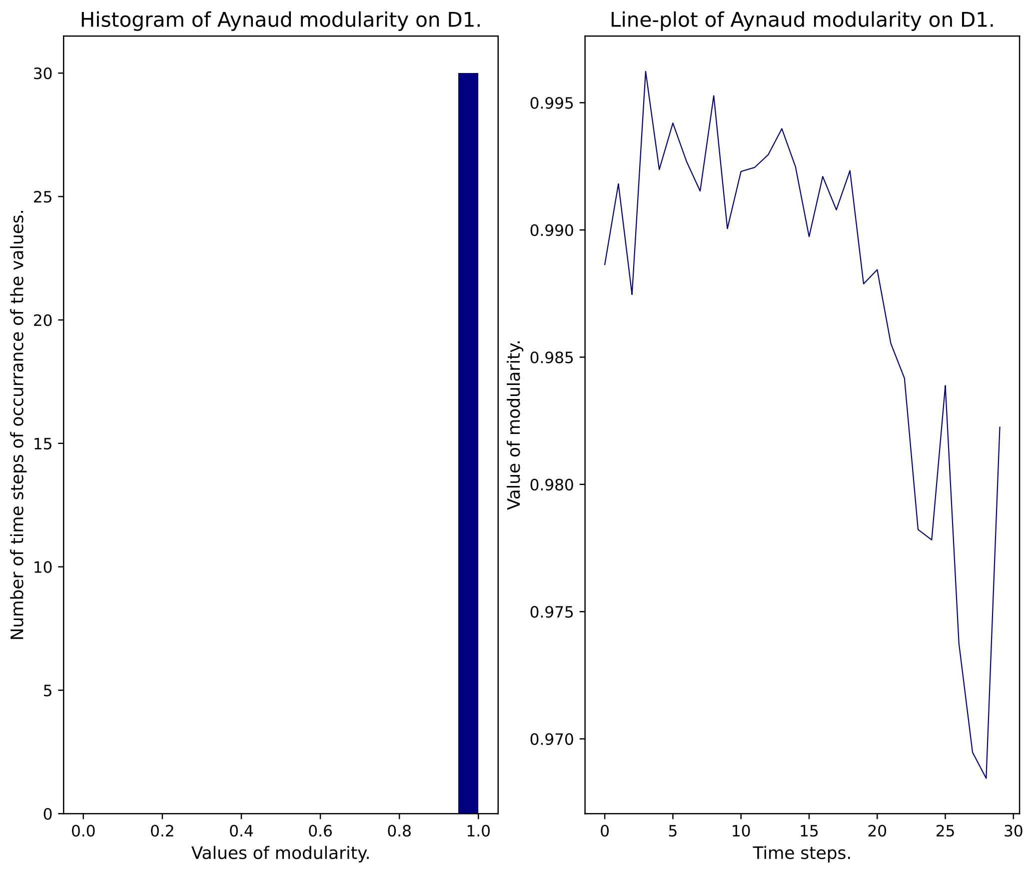

Figure 3.13: Aynaud modularity on D1.

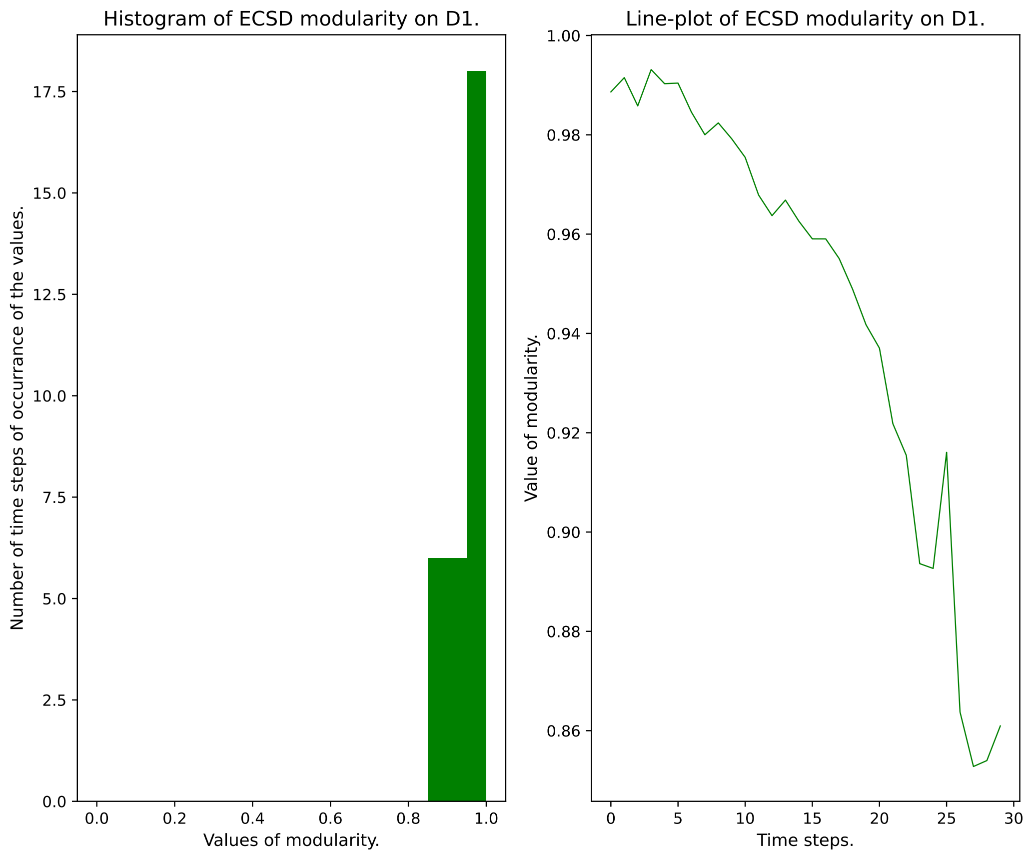

Figure 3.14: ECSD modularity on D1.

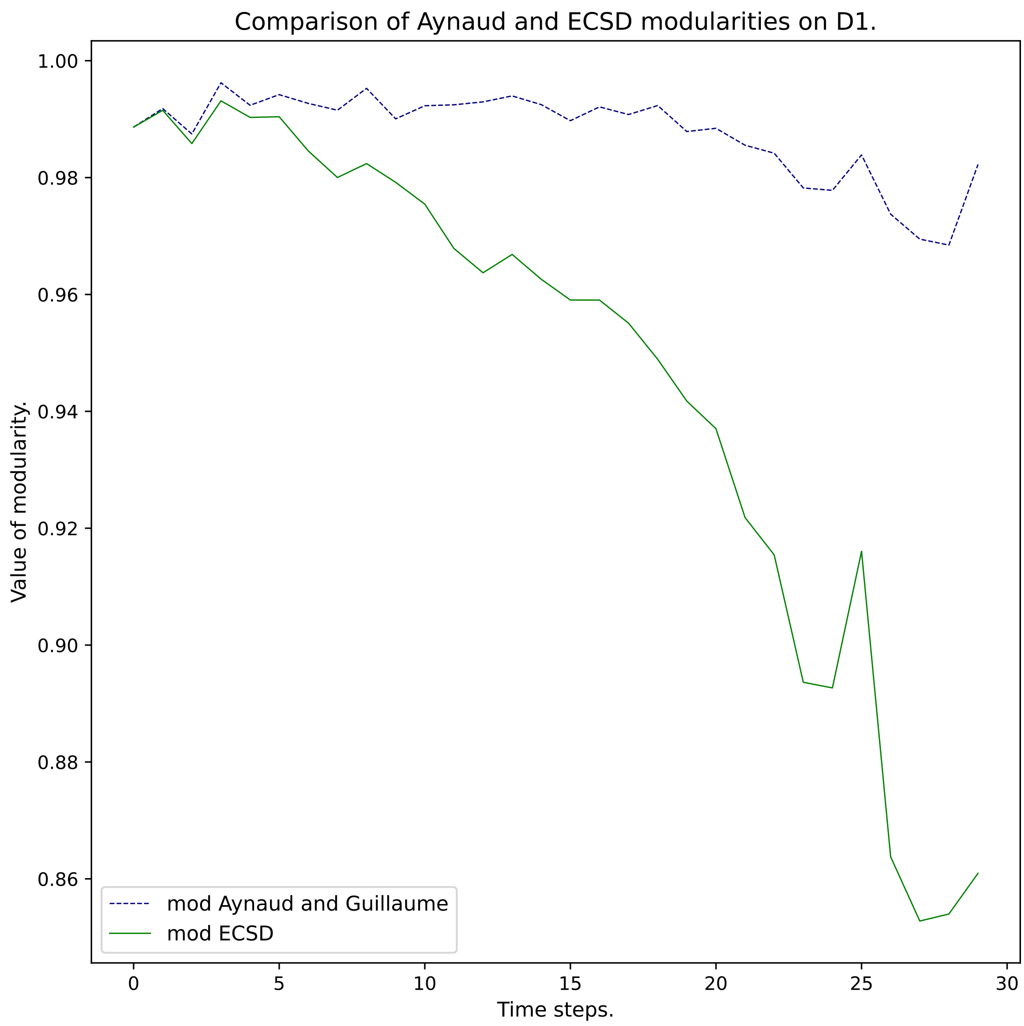

Figure 3.15: Comparison between Aynaud and ECSD modularity on D1.

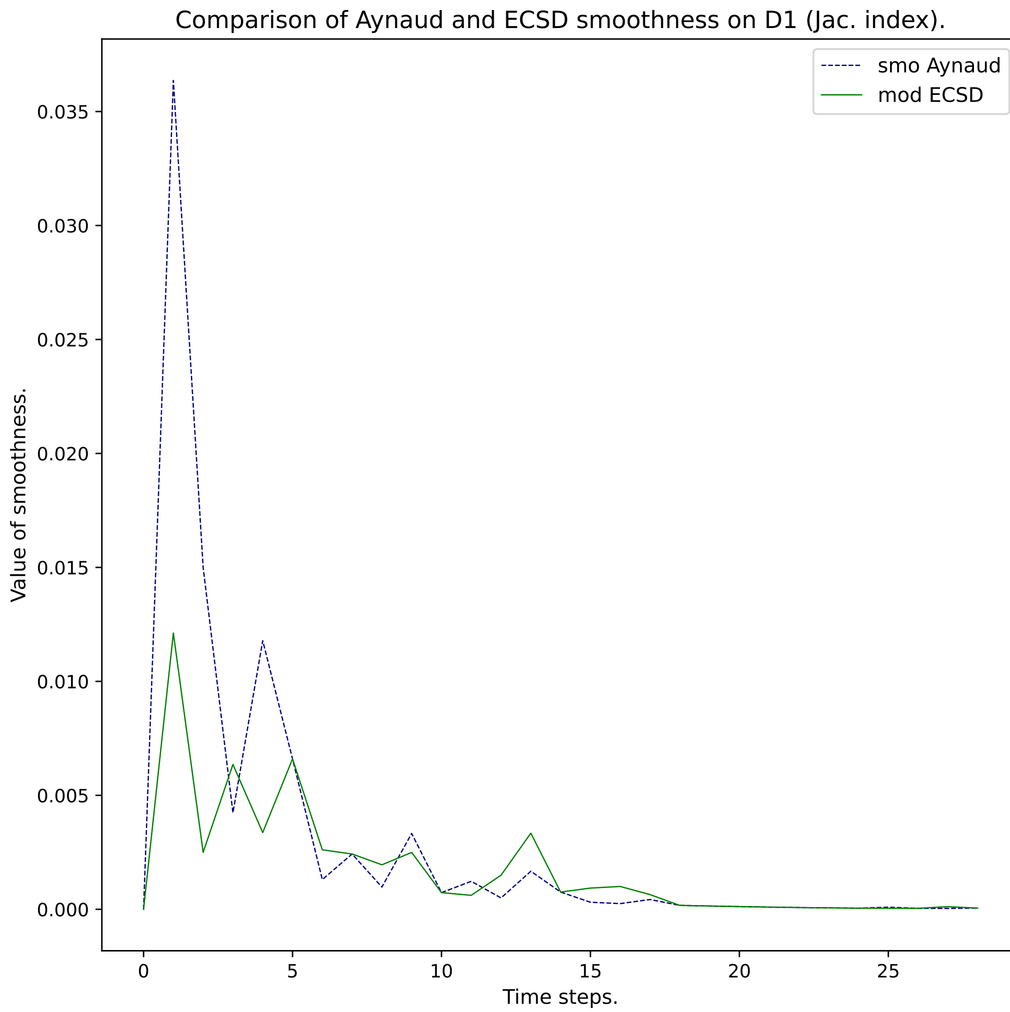

Figure 3.16: Comparison between Aynaud and ECSD smoothness (Jaccard index) on D1.

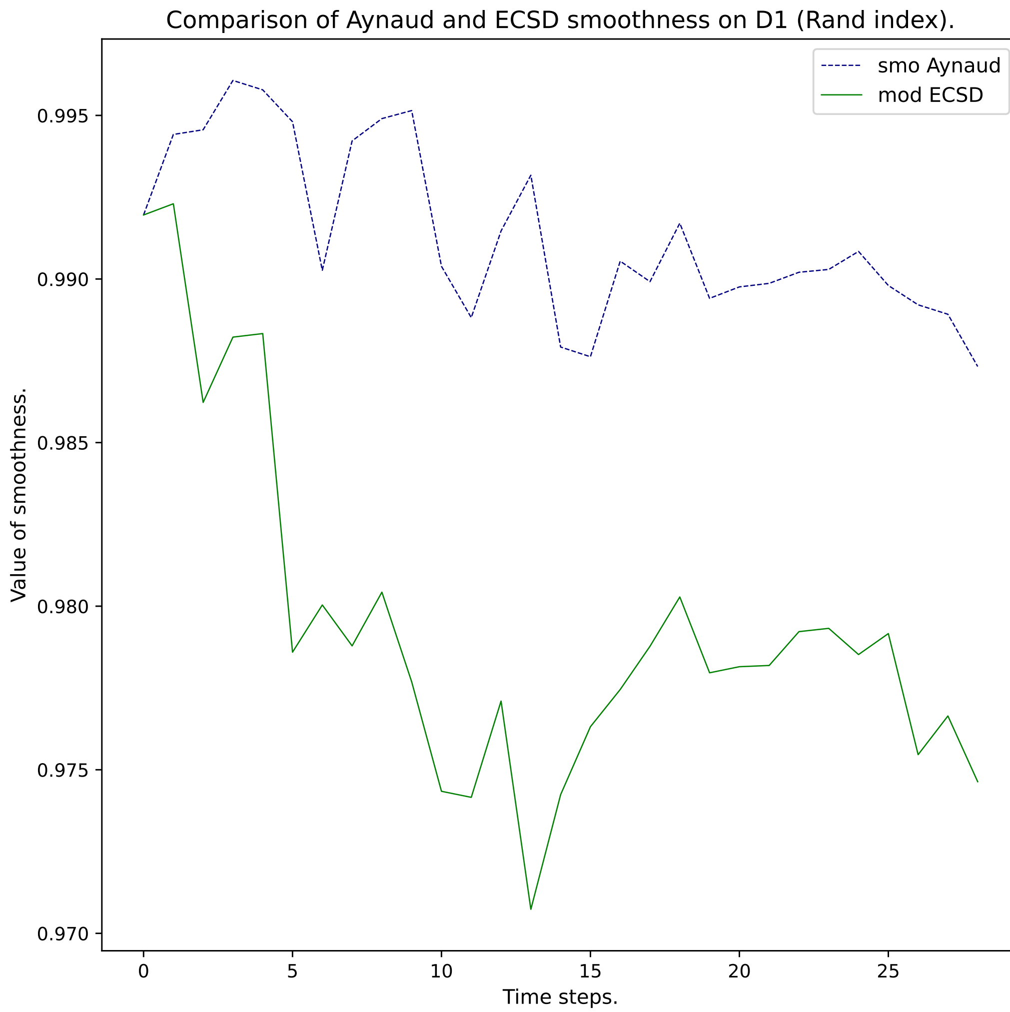

Figure 3.17: Comparison between Aynaud and ECSD smoothness (Rand index) on D1.

Conclusions and future work

As we said Chapters 1 and 2, the field of community detection is very vast. In the literature a huge variety of algorithms to detect communities has been proposed, whereas with respect to their comparison or classification the literature is still scarce.

In this thesis we decided to analyse two algorithms, the Aynaud and Guillaume's algorithm and the ECSD algorithm and we compared them on three different networks. The problem of validation and comparison of this kind of algorithms is really hard. One of the main reasons is the lack of datasets with ground truth communities, that could confirm if the communities detected by the algorithms are the real ones or not.
In order to overcome this issue we decided to base our comparison on two aspects: the modularity and the Jaccard/Rand smoothness. With respect to the computational time, the two algorithms are analogous for the networks we used.

We tested the two algorithms on three networks that are different in nature and size: the Sociopatterns network, an artificial random generated network and the DBLP co-authorship network.
According to our experiments presented in Chapter 3, we can conclude that both the algorithms are valid on networks that do not change dramatically from one time step to the next one.
We underline that in some cases, like in the artificial network, the ECSD algorithm seems to get better results. We think that this happens mainly when the communities split from one time step to the next one, supposing that the constraint of the Aynaud's algorithm to start the iteration of the Louvain algorithm by placing the nodes in the communities found previously might be too strict.
On other cases instead, like in the DBLP network, the algorithm of Aynaud seems to be more performant both in terms of modularity and smoothness. Moreover, in the case of the DBLP network, we noticed that both the algo-

rithms achieve very high values of modularity and we assume that the reason for this is that the network is very sparse and the communities are easily detected by taking each disconnected component of the network. Further studies and comparisons with more dense and connected network have still to be done.
Notice then that in other cases yet, like in the Sociopatterns network, the modularity and smoothness vectors present various oscillation. We do not know if they are due to an instability of the algorithms or if they are their attempts to adjust to the changes that happened.

Finally, regarding the smoothness, we noticed that the Rand and Jaccard indices might not be indicative when the set of the nodes that belong to two consecutive snapshots is too small. Indeed in this case, if the number of pair of nodes that are placed into different communities at the two snapshots is high, the Rand index is almost optimal and the Jaccard index is almost null, but this gives not much information about the smoothness of the algorithms. In fact it does not make much sense to compare the similarity of two partitions if they are relative to widely different set of nodes but this, more than a drawback of the algorithms, is an issue of the data we are working with.

Appendix A

In the following pages we report the code that we used for the experimental evaluation of Chapter 3.
Notice that in the code we are using the Sociopatterns network S1. If one wants to use an artificial generated random network G1 or a network D1 that is built by reading the interactions of a text file, it is sufficient to:

- uncomment the part relative to G1 or D1 in the step 1 "Building the network";
- comment the Sociopatterns lines;
- replace the name S1 with the name of the new network.

We can visualize the evolution of communities also by plotting the graph at some snapshots. By calling the function `plot_as_graph` with several time steps we can plot the network at that time steps ensuring that the positions of the nodes remain the same between snapshot and that different colours are used for different communities. We used it for the Sociopatterns network but, since it was time consuming and the images were not clear, we did not use it for bigger datasets.

```
#Import what we need.
import networkx as nx
import tnetwork as tn
import matplotlib.pyplot as plt
import statistics
import math
import numpy as np
import sklearn
path = "Where we want to save the results"
resultsS1 = open(path + "results_S1.txt", "w")

#_____________________________________________________________________________

#1 BUILDING THE NETWORK.

#Sociopatterns network.
sociopatterns = tn.graph_socioPatterns2012(tn.DynGraphSN)
S1 = sociopatterns.aggregate_time_period("hour")
#Artificial network.
#(gen_net, gen_comm) = tn.generate_simple_random_graph(nb_com=5,min_size=3,
#                        max_size=20,operations=8,mu=0.03,mu_noise=0.08)
#G1 = gen_net.to_DynGraphSN(slices=2)
#DBLP network.
#D1 = tn.read_interactions(path + "filewithinteractions.txt",
format=tn.DynGraphSN, time_first_column=False, sep='\t')

timesteps = S1.snapshots_timesteps()
resultsS1.write("\nThe network S1 has " + str(len(timesteps)) + "
snapshots.")
first_time = S1.start()
last_time = S1.end()
resultsS1.write("\nThe first snapshot is at time " + str(first_time))
resultsS1.write("\nThe last snapshot is at time " + str(last_time))
nn = []
mm = []
for i in timesteps:
      g = S1.graph_at_time(i)
      n = g.number_of_nodes()
      m = g.number_of_edges()
      nn.append(n)
      mm.append(m)
resultsS1.write("\nThe mean number of nodes per timestep is " +
                str(round(sum(nn)/len(nn),5)))
resultsS1.write("\nThe max number of nodes at timestep is " + str(max(nn)))
resultsS1.write("\nThe min number of nodes at timestep is " + str(min(nn)))
resultsS1.write("\nThe mean number of edges per timestep is " +
                str(round(sum(mm)/len(mm),5)))
resultsS1.write("\nThe max number of edges at timestep is " + str(max(mm)))
resultsS1.write("\nThe min number of edges at timestep is " + str(min(mm)))

#_____________________________________________________________________________

#2 DETECTING THE COMMUNITIES.

#Aynaud's algorithm.
comm_ayn_S1 = tn.DCD.smoothed_louvain(S1)
n_comm_ayn_S1 = []
mean_sizes_ayn = []
max_sizes_ayn = []
```

```
min_sizes_ayn = []
for t in timesteps:
      n = len(comm_ayn_S1.communities(t))
      n_comm_ayn_S1.append(n)
      sizes = []
      for i in comm_ayn_S1.communities(t).keys():
            sizes.append(len(comm_ayn_S1.communities(t)[i]))
      mean_sizes_ayn.append(round(sum(sizes)/n,5))
      max_sizes_ayn.append(max(sizes))
      min_sizes_ayn.append(min(sizes))
resultsS1.write("\n\nSizes Aynaud Communities")
resultsS1.write("\nThe mean number of communities per timestep is " +
str(round(sum(n_comm_ayn_S1)/len(timesteps),5)))
resultsS1.write("\nThe mean size per community per timestep is " +
str(round(sum(mean_sizes_ayn)/len(timesteps),5)))
resultsS1.write("\nThe max size of a community among all time steps is " +
str(max(max_sizes_ayn)))
resultsS1.write("\nThe min size of a community among all time steps is " +
str(min(min_sizes_ayn)))

#ECSD algorithm.
comm_ecsd_S1 = tn.DCD.smoothed_graph(S1)
n_comm_ecsd_S1 = []
mean_sizes_ecsd = []
max_sizes_ecsd = []
min_sizes_ecsd = []
for t in timesteps:
      n = len(comm_ecsd_S1.communities(t))
      n_comm_ecsd_S1.append(n)
      sizes = []
      for i in comm_ecsd_S1.communities(t).keys():
            sizes.append(len(comm_ecsd_S1.communities(t)[i]))
      mean_sizes_ecsd.append(round(sum(sizes)/n,5))
      max_sizes_ecsd.append(max(sizes))
      min_sizes_ecsd.append(min(sizes))
resultsS1.write("\n\nSizes ECSD Communities")
resultsS1.write("\nThe mean number of communities per timestep is " +
str(round(sum(n_comm_ecsd_S1)/len(timesteps),5)))
resultsS1.write("\nThe mean size per community per timestep is " +
str(round(sum(mean_sizes_ecsd)/len(timesteps),5)))
resultsS1.write("\nThe max size of a community among all time steps is " +
str(max(max_sizes_ecsd)))
resultsS1.write("\nThe min size of a community among all time steps is " +
str(min(min_sizes_ecsd)))

#Plotting some charachteristics of the communities detected.
step = int(len(timesteps)/3)
times_to_plot = [first_time,timesteps[step],timesteps[step*2],last_time]
plt.figure(figsize=(9,9))
plt.title("Aynaud's communities of S1.")
tn.plot_as_graph(S1, comm_ayn_S1, ts=times_to_plot, width=500, height=500)
plt.savefig(path + "comm_ayn_S1.png")
plt.close()

plt.figure(figsize=(9,9))
plt.title("ECSD communities of S1.")
tn.plot_as_graph(S1, comm_ecsd_S1, ts=times_to_plot, width=500, height=500)
plt.savefig(path + "comm_ecsd_S1.png")
plt.close()
```

```
plt.figure(figsize=(9,9))
plt.title("Mean sizes of the communities found by Aynaud and ECSD on S1.",
fontsize='13')
plt.plot(mean_sizes_ayn, linestyle='--', linewidth=0.7, color ='navy',
label='sizes Aynaud')
plt.plot(mean_sizes_ecsd, linewidth=0.7, color='green', label='sizes ECSD')
plt.legend(fontsize='11')
plt.xlabel("Time steps.", fontsize='11')
plt.ylabel("Mean size of the communities.", fontsize='11')
plt.savefig(path + "comparison_sizes_AynEcsd_S1.pdf", bbox_inches='tight')
plt.close()

plt.figure(figsize=(9,9))
plt.title("Number of communities found by Aynaud and ECSD on S1.",
fontsize='13')
plt.plot(n_comm_ayn_S1, linestyle='--', linewidth=0.7, color ='navy',
label='n comm Aynaud')
plt.plot(n_comm_ecsd_S1, linewidth=0.7, color='green', label='n comm ECSD')
plt.legend(fontsize='11')
plt.xlabel("Time steps.", fontsize='11')
plt.ylabel("Number of communities.", fontsize='11')
plt.savefig(path + "n_comm_comparison_AynEcsd_S1.pdf", bbox_inches='tight')
plt.close()

#______________________________________________________________________________

#3 MODULARITY ANALYSIS.

def mystatistical(vector, sizes):
      resultsS1.write("\nmean = " + str(round(statistics.mean(vector),5)))
      resultsS1.write("\nweighted mean = " + str(round(np.average(vector,
weights=sizes),5)))
      resultsS1.write("\nvariance = " +
str(round(statistics.variance(vector),5)))
      resultsS1.write("\nstdev = " + str(round(statistics.stdev(vector),-
5)))
      resultsS1.write("\nmedian = " + str(round(statistics.median(vector),-
5)))
      resultsS1.write("\nmin = " + str(round(min(vector),5)))
      resultsS1.write("\nmax = " + str(round(max(vector),5)))

resultsS1.write("\n\n~~Modularity with Aynaud and Guillaume's
algorithm.~~")
mod_ayn_S1, size_mod_ayn_S1 = tn.DCD.quality_at_each_step(comm_ayn_S1, S1)
mystatistical(mod_ayn_S1, size_mod_ayn_S1)

resultsS1.write("\n\n~~Modularity with ECSD algorithm.~~")
mod_ecsd_S1, size_mod_ecsd_S1 = tn.DCD.quality_at_each_step(comm_ecsd_S1,
S1)
mystatistical(mod_ecsd_S1, size_mod_ecsd_S1)

#Plotting mod_ayn_S1.
plt.figure(figsize=(11,9))
plt.subplot(121)
plt.title("Histogram of Aynaud modularity on S1.", fontsize='13')
plt.hist(mod_ayn_S1, color='navy', bins=20, range=(0,1))
plt.xlabel("Values of modularity.", fontsize='11')
plt.ylabel("Number of time steps of occurrance of the values.",
```

```
fontsize='11')
plt.subplot(122)
plt.title("Line-plot of Aynaud modularity on S1.", fontsize='13')
plt.plot(mod_ayn_S1, linewidth=0.7, color='navy')
plt.xlabel("Time steps.", fontsize='11')
plt.ylabel("Value of modularity.", fontsize='11')
plt.savefig(path + "mod_ayn_S1.pdf", bbox_inches='tight')
plt.close()

#Plotting mod_ecsd_S1.
plt.figure(figsize=(11,9))
plt.subplot(121)
plt.title("Histogram of ECSD modularity on S1.", fontsize='13')
plt.hist(mod_ecsd_S1, color='green', bins=20, range=(0,1))
plt.xlabel("Values of modularity.", fontsize='11')
plt.ylabel("Number of time steps of occurrance of the values.",
fontsize='11')
plt.subplot(122)
plt.title("Line-plot of ECSD modularity on S1.", fontsize='13')
plt.plot(mod_ecsd_S1, linewidth=0.8, color='green')
plt.xlabel("Time steps.", fontsize='11')
plt.ylabel("Value of modularity.", fontsize='11')
plt.savefig(path + "mod_ecsd_S1.pdf", bbox_inches='tight')
plt.close()

#Comparing modularities of the two algorithms.
plt.figure(figsize=(9,9))
plt.title("Comparison of Aynaud and ECSD modularities on S1.",
fontsize='13')
plt.plot(mod_ayn_S1, linestyle='--', linewidth=0.7, color ='navy',
label='mod Aynaud')
plt.plot(mod_ecsd_S1, linewidth=0.7, color='green', label='mod ECSD')
plt.legend(fontsize='11')
plt.xlabel("Time steps.", fontsize='11')
plt.ylabel("Value of modularity.", fontsize='11')
plt.savefig(path + "mod_comparison_AynEcsd_S1.pdf", bbox_inches='tight')
plt.close()

#Infinity norm of the difference.
d = []
for i in range(len(mod_ayn_S1)):
      d.append(abs(mod_ayn_S1[i]-mod_ecsd_S1[i]))
infnorm = max(d)
resultsS1.write("\nThe infnorm of the difference between Aynaud and ecsd
modularity vectors is " + str(round(infnorm,5)))

#______________________________________________________________________________

#4 SMOOTHNESS ANALYSIS.

def mystatistical_smo(vector, sizes):
      resultsS1.write("\nmean = " + str(round(statistics.mean(vector),5)))
      resultsS1.write("\nweighted mean = " + str(round(np.average(vector,
weights=sizes),5)))
      resultsS1.write("\nvariance = " +
str(round(statistics.variance(vector),5)))

def myjaccard(x,y):
    setx = set()
```

```
    for i in x:
        setx = setx.union(i)
    sety = set()
    for i in y:
        sety = sety.union(i)
    I = set.intersection(setx,sety)
    lx = []
    ly = []
    labelx = 0
    for i in x:
        i = i.intersection(I)
        for n in range(len(i)):
            lx.append(labelx)
        labelx += 1
    labely = 0
    for i in y:
        i = i.intersection(I)
        for n in range(len(i)):
            ly.append(labely)
            labely += 1
    if lx == [] or ly == []:
        return 0
    else:
        return sklearn.metrics.jaccard_score(lx,ly,average='weighted')

def myrand(x,y):
    setx = set()
    for i in x:
        setx = setx.union(i)
    sety = set()
    for i in y:
        sety = sety.union(i)
    I = set.intersection(setx,sety)
    lx = []
    ly = []
    labelx = 0
    for i in x:
        i = i.intersection(I)
        for n in range(len(i)):
            lx.append(labelx)
        labelx += 1
    labely = 0
    for i in y:
        i = i.intersection(I)
        for n in range(len(i)):
            ly.append(labely)
            labely += 1
    if lx == [] or ly == []:
        return 0
    else:
        return sklearn.metrics.rand_score(lx,ly)

myscorejaccard = lambda x,y : myjaccard(x,y)
myscorerand = lambda x,y : myrand(x,y)

smo_jac_ayn_S1, size_smo_jac_ayn_S1 =
tn.DCD.consecutive_sn_similarity(comm_ayn_S1, score=myscorejaccard)
smo_ran_ayn_S1, size_smo_ran_ayn_S1 =
tn.DCD.consecutive_sn_similarity(comm_ayn_S1, score=myscorerand)
```

```
smo_jac_ecsd_S1, size_smo_jac_ecsd_S1 =
tn.DCD.consecutive_sn_similarity(comm_ecsd_S1, score=myscorejaccard)
smo_ran_ecsd_S1, size_smo_ran_ecsd_S1 =
tn.DCD.consecutive_sn_similarity(comm_ecsd_S1, score=myscorerand)

resultsS1.write("\n\n~~Aynaud smoothness (Jaccard-coefficient).~~")
mystatistical_smo(smo_jac_ayn_S1,size_smo_jac_ayn_S1)
resultsS1.write("\n\n~~Aynaud smoothness (Rand-coefficient).~~")
mystatistical_smo(smo_ran_ayn_S1,size_smo_ran_ayn_S1)
resultsS1.write("\n\n~~ECSD smoothness (Jaccard-coefficient).~~")
mystatistical_smo(smo_jac_ecsd_S1,size_smo_jac_ecsd_S1)
resultsS1.write("\n\n~~ECSD smoothness (Rand-coefficient).~~")
mystatistical_smo(smo_ran_ecsd_S1,size_smo_ran_ecsd_S1)

#Comparing smoothness of the two algorithms.
plt.figure(figsize=(9,9))
plt.title("Comparison of Aynaud and ECSD smoothness on S1 (Jac. index).",
fontsize='13')
plt.plot(smo_jac_ayn_S1, linestyle='--', linewidth=0.7, color ='navy',
label='smo Aynaud')
plt.plot(smo_jac_ecsd_S1, linewidth=0.7, color='green', label='mod ECSD')
plt.legend(fontsize='11')
plt.xlabel("Time steps.", fontsize='11')
plt.ylabel("Value of smoothness.", fontsize='11')
plt.savefig(path + "smoJac_comparison_AynEcsd_S1.pdf", bbox_inches='tight')
plt.close()

plt.figure(figsize=(9,9))
plt.title("Comparison of Aynaud and ECSD smoothness on S1 (Rand index).",
fontsize='13')
plt.plot(smo_ran_ayn_S1, linestyle='--', linewidth=0.7, color ='navy',
label='smo Aynaud')
plt.plot(smo_ran_ecsd_S1, linewidth=0.7, color='green', label='mod ECSD')
plt.legend(fontsize='11')
plt.xlabel("Time steps.", fontsize='11')
plt.ylabel("Value of smoothness.", fontsize='11')
plt.savefig(path + "smoRan_comparison_AynEcsd_S1.pdf", bbox_inches='tight')
plt.close()

resultsS1.close()
```

List of Figures

1.1 An example of a static network. 7
1.2 A partition into three communities of the static network in Figure 1.1. 13
1.3 Tracking the communities across three time steps. 19
1.4 Community events. 22

2.1 Louvain algorithm. 32

3.1 Number of communities on S1. 58
3.2 Aynaud modularity on S1. 59
3.3 ECSD modularity on S1. 60
3.4 Comparison between Aynaud and ECSD modularity on S1. . . 61
3.5 Comparison between Aynaud and ECSD smoothness (Jaccard index) on S1. 62
3.6 Comparison between Aynaud and ECSD smoothness (Rand index) on S1. 63
3.7 Number of communities on G1. 67
3.8 Aynaud modularity on G1. 68
3.9 ECSD modularity on G1. 69
3.10 Comparison between Aynaud and ECSD modularity on G1. . 70
3.11 Comparison between Aynaud and ECSD smoothness (Rand index) on G1. 71
3.12 Number of communities on D1. 75
3.13 Aynaud modularity on D1. 76
3.14 ECSD modularity on D1. 77
3.15 Comparison between Aynaud and ECSD modularity on D1. . 78
3.16 Comparison between Aynaud and ECSD smoothness (Jaccard index) on D1. 79
3.17 Comparison between Aynaud and ECSD smoothness (Rand index) on D1. 80

Bibliography

[1] G. Rossetti and R. Cazabet, *Community discovery in dynamic networks: a survey*, arXiv:1707.03186v3, 2019.

[2] P. Holme and J. Saramäki, *Temporal networks*, Physics Reports, vol. 519, pp. 97-125, 2012.

[3] C. Staudt, *Experimental Evaluation of Dynamic Graph Clustering Algorithms*, master thesis, 2010.

[4] F. Menczer, S. Fortunato and C. A. Davis, *A first course in network science*, Cambridge University Press, 2020.

[5] D. Weber and F. Neumann, *Who's in the gang? Revealing coordinating communities in social media*, arXiv:2010.08180v1, 2020.

[6] D. Weber and F. Neumann, *Amplifying influence through coordinated behaviour in social networks*, Social Network Analysis and Mining, 11:111, Springer, 2021.

[7] A. Condon and R. M. Karp, *Algorithms for graph partitioning on the planted partition model*, Lecture Notes in Computer Science, vol. 1671, Springer, 2000.

[8] M. El-Telbany, S. Refat, H. Hefny, A. Abdelwhab and A. Dakroury, *From evolution to swarm intelligence: data clustering survey*, International Journal of Intelligent Computing and Information Sciences (IJICIS), 2008.

[9] S. Fortunato, *Community detection in graphs*, arXiv:0906.0612v2, 2010.

[10] S. Fortunato and D. Hric, *Community detection in networks: a user guide*, Physics Reports 659, pp. 1-44, 2016.

[11] N. Dakiche, F. Benbouzid-Si Tayeb, Y. Slimani and K. Benatchba, *Tracking community evolution in social networks: a survey*, Information Processing and Management, vol. 56, pp. 1084-1102, 2019.

[12] D. Greene, D. Doyle and P. Cunningham, *Tracking the evolution of communities in dynamic social networks*, Proceedings - 2010 International Conference on Advances in Social Network Analysis and Mining, 2010.

[13] D. Delling, M. Gaertler, R. Görke and D. Wagner, *Engineering comparators for graph clustering*, Preoceedings of the 4th International Conference on on Algorithmic Aspects in Information and Managemenr, pp. 131-142, Springer, 2008.

[14] R. Görke, P. Maillard, A. Schumm, C. Staudt and D. Wagner, *Dynamic graph clustering combining modularity and smoothness*, ACM Journal of Experimental Algorithmics, vol. 18, no. 1, Article 1.5, 2013.

[15] G. Palla, A. Barabási and T. Vicsek, *Quantifying social group evolution*, Nature 446, pp. 664–667, 2007.

[16] T. van Laarhoven and E. Marchiori, *Local network community detection with continuous optimization of conductance and weighted kernel k-means*, Journal of Machine Learning Research 17, pp. 1-28, 2016.

[17] J. Leskovec, K. J. Lang and M. W. Mahoney, *Empirical comparison of algorithms for network community detection*, arXiv:1004.3539, 2010.

[18] M. E. J. Newman and M. Girvan, *Finding and evaluating community structure in networks*, Physical Review E 69, 026113, 2004.

[19] A. Clauset, M. E. J. Newman and C. Moore, *Finding community structure in very large networks*, Physical Review E 70, 066111, 2004.

[20] M. E. J. Newman, *Fast algorithm for detecting community structure in networks*, Physical Review E 69, 066133, 2004.

[21] M. E. J. Newman, *Mixing patterns in networks*, Physical Review E 67, 026126, 2003.

[22] M. E. J. Newman, *Analysis of weighted networks*, arXiv:cond-mat/0407503v1, 2004.

[23] U. Brandes, D. Delling, M. Gaertler, R. Görke, M. Hoefer, Z. Nikolski and D. Wagner, *Maximizing Modularity is hard*, arXiv:physics/0608255v2, 2006.

[24] S. Fortunato and M. Barthélemy, *Resolution limit in community detection*, PNAS, vol. 104, no.1, pp. 36-41, 2007.

[25] P. Pons and M. Latapy, *Computing communities in large networks using random walks*, Computer and Information Sciences, v. 3733, pp. 284-293, Springer, 2005.

[26] , M. Rosvall and C. T. Bergstrom, *Maps of random walks on complex networks reveal community structure*, PNAS, v. 105, pp. 1118-1123, 2008.

[27] U. von Luxburg, *Tutorial on spectral clustering*, arXiv:0711.0189, 2007.

[28] S. Harenberg, G. Bello, L. Gjeltema, S. Ranshous, J. Harlalka, R. Seay, K. Padmanabhan and N. Samatova, *Community detection in large-scale networks: a survey and empirical evaluation*, WIREs Comput Stat, vol. 6, pp. 426-439, 2014.

[29] S. E. Schaeffer, *Graph clustering*, Computer Science Review, vol. 1, pp. 27-64, 2007.

[30] T. Aynaud, E. Fleury, J.-L. Guillaume and Q. Wang, *Communities in evolving networks: definitions, de- tection, and analysis techniques*, In Dynamics On and Of Complex Networks, v. 2, pp. 159–200. Springer, 2013.

[31] V. D. Blondel, J. - L. Guillaume, R. Lambiotte and E. Lefebvre, *Fast unfolding of communities in large networks*, arXiv:0803.0476, 2008.

[32] N. Alotaibi and D. Rhouma, *A review on community structures detection in time evolving social networks*, Journal of King Saud University - Computer and information Sciences, https://doi.org/10.1016/j.jksuci.2021.08.016, 2021.

[33] X. Su, S. Xue, F. Liu, J. Wu, C. Zhou, W. Hu, C. Paris, S. Nepal, D. Jin, Q. Sheng and P. S. Yu, *A comprehensive survey on community detection with deep learning*, IEEE Transactions on Neural Networks and Learning Systems, pp. 1-21, 2022

[34] M. Coscia, F. Giannotti and D. Pedreschi, *A classification for community discovery methods in complex networks*, arXiv:1206.3552v1, 2012.

[35] S. Bansal, S. Bhowmick and P. Paymal, *Fast Community Detection for Dynamic Complex Networks*, Communications in Computer and Information Science, v. 116, Springer, 2011.

[36] T. Aynaud, J. - L. Guillaume, *Static community detection algorithms for evolving networks*, WiOpt'10: Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks, pp.508-514, 2010.

[37] C. Guo, J. Wang and Z. Zhang, *Evolutionary community structure discovery in dynamic weighted networks*, Physica A 413, pp. 565-576, 2014.

[38] D. Chen, M. Shang, Z. Lv and Y. Fu, *Detecting overlapping communities of weighted networks via a local algorithm*, Physica A: Statistical Mechanics and its Applications, v. 389, pp. 4177-4187, 2010.

[39] https://networkx.org/

[40] https://tnetwork.readthedocs.io/en/latest/

[41] https://docs.python.org/3/library/statistics.html

[42] https://matplotlib.org/

[43] http://www.sociopatterns.org/

[44] http://konect.cc/

Ringraziamenti

Ringrazio il mio relatore Giovanni da San Martino e la mia correlatrice Giorgia Callegaro, con cui è stato possibile realizzare questo lavoro, che è stato per me molto stimolante. In particolare vi ringrazio per la disponibilità e per l'allegria con cui mi avete sempre accolta a ricevimento.

Ringrazio tutti i miei compagni di corso, in particolare Pippo e la Vale. Poter condividere lo studio con voi lo ha sempre reso estremamente appassionante e ci ha portato ad avvincenti discussioni.

Ringrazio tutti i miei coinquilini di questi anni, grazie ai quali mi sono sempre sentita a casa. Grazie in particolare alle mie care amiche Terry e la Berta.

Ringrazio tutti gli amici che ho incontrato in Università, da quelli che mi hanno accolta quando sono arrivata a quelli con cui ho condiviso questi ultimi mesi di tesi. Grazie soprattutto a tutti gli amici del Clu e ai Boombers per la compagnia che ci siamo fatti e per la provocazione di vivere le cose con intensità.

Ringrazio i fratelli Niero, siete stati per me una costante in tutti questi anni a Padova e nutro per voi una profonda stima e simpatia.

Ringrazio la mia Big Family: mamma, papà, Merion, Marghe, Jackie, Verò, Cate e Ans. Siete una buona famiglia.

Ringrazio Lorenzo per il sostegno e per l'affetto con cui continuiamo a crescere insieme.