

UNIVERSITÀ DEGLI STUDI DI GENOVA



DIPARTIMENTO DI MATEMATICA

CORSO DI LAUREA MAGISTRALE IN
MATEMATICA

Anno accademico 2025/2026

Tesi di Laurea
Curriculum in Matematica Generale

**Developing a Theoretical Framework for
Complex Network Analysis:
an Application to Criminology**

Candidata

Sara Buscaglia

Relatrice

Prof.ssa Emanuela De Negri

Relatore

Dr. Ulderico Fugacci

Correlatrice

Prof.ssa Elisabetta Robotti

Contents

Introduction	1
1 Basic Notions on Graphs	5
1.1 Basic definitions	6
1.2 Sociocentric and egocentric graph analysis	12
1.2.1 Graph structures: a sociocentric perspective	13
1.2.2 Centrality measures: an egocentric perspective	18
1.3 Analysing sets of vertices	24
1.3.1 Similarities between sets of vertices	24
1.3.2 Comparing collections of sets of vertices	30
2 Algorithms for Structure Identification and Graph Dismantling	35
2.1 Community detection algorithm	36
2.1.1 Algorithm description	36
2.1.2 A step by step example	37
2.2 Disrupting graph algorithm	40
2.2.1 Algorithm description	40
2.2.2 A step by step example	41
3 An Application to a Mafia Case	46
3.1 State of the art	46
3.2 Introduction	47
3.2.1 Dataset and previous studies	47
3.2.2 Problem formulation	48
3.3 Community detection algorithm on a Mafia case	49
3.3.1 Research hypothesis	49
3.3.2 Results	50
3.4 Disrupting graph algorithm on a Mafia case	54
3.4.1 Research hypothesis	54
3.4.2 Results	56
Conclusions	62
A Community Detection Algorithm: Python Implementation	65
B Disrupting Graph Algorithm: Python Implementation	74
C Additional Results and Plots	96

Abstract

Motivated by the work of Cavallaro et al. [5] on the disruption of Sicilian Mafia networks, and progressively enlarging the field of investigation to the general perspective of graph theory, the thesis aims at providing an answer to the question on whether it is possible to build a theoretical framework to understand the cohesion and the robustness of a graph. Starting from the idea that a graph already carries, within its own structure, the information needed to describe its organisation, the thesis introduces two main theoretical contributions. The first is a new family of centrality measures, the k -clique community centrality and its generalised version, together with an auxiliary centrality based on atomic communities; unlike classical measures such as degree or betweenness centrality, these quantities value the role of a vertex within the cohesive structure of the graph. The second is a general framework for comparing two finite collections of sets, extending classical similarity measures between individual sets, such as the Jaccard similarity and the overlap coefficient, to the comparison of entire community subdivisions. The introduced theoretical tools allow the development of (and subsequent implementation in Python of) two algorithms: a community detection algorithm, which identifies which graph structure best reproduces a given subdivision of the vertices, and a disrupting graph algorithm, which studies the robustness of a graph by progressively removing vertices according to a chosen centrality measure. The two algorithms are designed to work together, using the structure discovered by the first to guide the choice of centrality and robustness measures in the second. The framework is finally applied to the meetings network built from the judicial documents of the 2007 Montagna anti-Mafia operation, showing that 4-clique communities provide the best approximation of the original subdivision into Mafia clans, and that centrality measures based on this structure are particularly effective at disrupting the cohesive groups corresponding to the criminal clans themselves. Moreover, removing vertices according to these new centralities is shown to target, among the first individuals removed, several known Mafia bosses and other key figures of the organisation, confirming that the structural information captured by these measures aligns with the actual roles played by individuals within the criminal network. During the development of this work, I also had the opportunity to discuss the proposed approach with two of the authors of the paper that originally motivated this thesis, Dr. A. Ficara and Prof. G. Fiumara, whose interest in the framework and in its preliminary results provided further encouragement to pursue the addressed direction.

Introduction

The starting point for this thesis can be traced back to the final project of the course #DIMA (Divulgazione Matematica). The project consisted of preparing a podcast on a mathematical topic and presenting it in a way that was sufficiently accessible and engaging for the chosen audience. At a time when true-crime podcasts are becoming increasingly popular, I started wondering whether mathematics had ever played an important role in the investigation or analysis of criminal cases.

I found several interesting examples, but the studies that attracted my attention the most were those concerning organised crime. In these studies, the authors do not necessarily claim that the mathematical methods were directly used during the investigations. Rather, mathematical and computational techniques are applied to data collected during investigations in order to identify patterns, relationships, and structural properties that can help describe and understand organised criminal groups. In particular, the representation of criminal organisations as networks provides a natural connection between the analysis of organised crime and graph theory.

The paper that attracted my attention the most was “Disrupting resilient criminal networks through data analysis: The case of Sicilian Mafia” by Cavallaro et al. [5]. In this work, the authors analysed data obtained from judicial documents related to the 2007 Montagna anti-Mafia operation and constructed networks representing interactions between individuals. In particular, they considered two different networks: a calls network, based on wiretapping data, and a meetings network, based on information collected through surveillance activities. The nodes represent individuals, while the edges represent interactions between them. The authors then studied the resilience of these networks by simulating different node-removal strategies and analysing how their structure changed as individuals were progressively removed. Several centrality measures were considered in order to investigate whether highly central individuals were also associated with important positions within the criminal organisation.

Reading this paper led me to several questions. Could the organisation of Mafia clans be inferred from the network structure alone, using only information about calls or meetings? Are there individuals who are more important than others because of their position within the network? If so, could mathematical measures of centrality help identify them?

These questions became the starting point for my work. Rather than simply trying to reproduce the analysis carried out by Cavallaro et al., I decided to take a step backwards and investigate the mathematical structures underlying the problem. If Mafia clans can be represented as groups of vertices in a graph, then questions concerning the organisation of the clans, the importance of particular individuals, and the resilience of the network can be reformulated as more general questions in graph theory.

My original questions therefore evolved into the following:

- Are there graph structures that can naturally describe the points of aggregation of a graph?
- How can the robustness of a graph be measured?
- Which vertices are most important for maintaining the structure of a graph, and whose removal causes the graph to disrupt more rapidly?

At this point, I realised that the problem was no longer restricted to the analysis of a Mafia network. It had become a problem in graph theory, and I therefore started looking for mathematical tools that could provide answers to these more general questions.

I began this investigation in Chapter 1, where I first introduced the basic definitions of graph theory that are needed throughout the thesis. This preliminary step is particularly important because terminology and notation are not completely uniform across the literature. I therefore considered it necessary to explicitly state the definitions adopted in this work, so that all the concepts used in the following chapters have a precise and unambiguous meaning.

In the same chapter, I then investigated which mathematical structures could be used to identify meaningful groups of vertices within a graph. In other words, I wanted to understand whether, given a graph, it is possible to identify a natural way of grouping its vertices and whether the graph contains an intrinsic substructure that can provide information about its organisation. These ideas are formalised through the graph structures introduced in Section 1.2.1. In particular, I focused on clique-based structures and k -clique communities, which later proved to be particularly relevant for the application considered in this thesis.

Once I had identified possible structures within a graph, another question naturally arose: how can the importance of individual vertices be evaluated? This problem is already addressed in graph theory through the concept of centrality, discussed in Section 1.2.2. Classical measures such as degree centrality and betweenness centrality provide different ways of assigning importance to vertices according to their position in the graph.

However, I wanted to investigate whether the information provided by the identified graph structures could also be incorporated into the definition of centrality. For this reason, I introduced new centrality measures based on the community structures described in the previous section. The aim was to give greater importance to the structural role played by vertices within the communities of the graph and to investigate whether this information could provide a different perspective from that obtained through classical centrality measures. These definitions represent one of the original contributions developed in this thesis.

After dividing a graph into communities or other meaningful substructures, I also needed a way to determine how well one subdivision resembles another. This led me to the problem of comparing collections of sets. In Section 1.3.1, I therefore started by introducing some notions of similarity and distance between finite sets. Many of these measures have their origins in biological and botanical applications, where they were developed to compare the distribution of flora and fauna in different geographical areas.

Measures such as the Jaccard similarity and the overlap coefficient, however, compare two sets at a time. This is not sufficient when the object of comparison is a whole subdivision of the vertices into several communities. I therefore extended these ideas in Section 1.3.2 by introducing a framework for comparing collections of finite sets. This allowed me

to quantify how well a collection of communities obtained from a graph reproduces a given subdivision. This extension represents a further contribution of this thesis and provides the mathematical basis for the comparison between the graph structures identified by the algorithms and the original subdivision of the vertices.

In Chapter 2, I developed two algorithms, which I then implemented in Python, to investigate the questions introduced above on given graphs. The first algorithm, which I refer to as the community detection algorithm, is designed to determine which of the graph structures introduced in Section 1.2.1 provides the best approximation of a given subdivision. In the cases considered in this thesis, the graphs are associated with an initial subdivision of their vertices, and the aim is to determine which of the mathematically defined structures can best reproduce this given organisation. The code for this algorithm can be found in Appendix A.

The results obtained from the first algorithm then provide the basis for the second algorithm, namely the disrupting graph algorithm. In this case, I focus on the problem of graph robustness. The idea is to rank the vertices according to a selected centrality measure and progressively remove them from the graph. At each step, I keep track of the extent to which the graph has been disrupted through a parameter ρ_i . This parameter provides a quantitative description of the structural degradation of the graph and makes it possible to compare different vertex-removal strategies.

The two algorithms are therefore closely connected. The first algorithm allows me to investigate which mathematical structures best describe the organisation of a graph, while the second uses this structural information to investigate the importance of individual vertices and the robustness of the graph. In particular, once a relevant community structure has been identified, the corresponding community-based centrality measures can be used to determine which vertices should be removed first. And, moreover, the selected community structure could be useful to define the parameter ρ_i , giving importance to the dismantling of such communities. The code for the second algorithm is provided in Appendix B.

To make the functioning of the two algorithms clearer, I present two step-by-step examples in Sections 2.1.2 and 2.2.2. These examples allow the different stages of the algorithms to be followed in detail and show how the quantities introduced in the theoretical part of the thesis are used in practice.

During the development of this work, I also had the opportunity to discuss the original problem with two of the authors of the paper that had initially inspired my research, Dr. A. Ficara and Prof. G. Fiumara. We discussed the problem in greater detail, and I presented the mathematical framework that I had developed through the study of graph analysis. In particular, I explained my intention to investigate whether the structures and centrality measures introduced in this thesis could provide meaningful results when applied to the networks constructed in their study of the Sicilian Mafia. They showed considerable interest in this approach and expressed their curiosity about the results that could be obtained from the application of my framework to the original networks.

In Chapter 3, I finally apply the framework I developed to a real-world network, namely the meetings network considered in the work of Cavallaro et al. This represents an important step in the development of the thesis, since it allows the theoretical and computational framework to be tested on an actual network obtained from a real criminal organisation.

In Section 3.3, I first investigate which of the graph structures introduced in Chapter 1 best reproduces the original subdivision of the network into Mafia clans. The results show that the 4-clique communities provide the best approximation among the structures considered. This result is particularly relevant for the subsequent analysis because it gives a mathematical justification for focusing on 4-clique communities rather than selecting them arbitrarily.

This finding led me to apply the second algorithm using not only the two classical centrality measures, degree centrality and betweenness centrality, but also the centrality measures I had introduced on the basis of the 4-clique communities. In this way, I could investigate in Section 3.4 whether taking the internal community structure into account provides a different and potentially more effective way of identifying the vertices whose removal most strongly affects the network.

Finally, as a further contribution, I introduced several versions of the parameter ρ_i in order to evaluate the effectiveness of a disruption strategy from different structural perspectives. In particular, these definitions allow the effect of node removal to be evaluated with respect to the disintegration of the 4-clique communities and, more generally, of the k -clique communities of the graph. This makes it possible to move beyond a measure based only on the global connectivity of the network and to investigate how the specific structures identified within the graph are affected by the removal of important vertices.

The path followed in this thesis therefore started from a concrete question concerning the analysis of organised crime and progressively developed into a more general investigation in graph theory. The Sicilian Mafia network provided the original motivation and, ultimately, the real-world setting in which the proposed framework could be tested. The study of graph structures, similarities between collections of sets, centrality measures, and network robustness allowed me to reformulate the original questions in mathematical terms and to develop new tools for addressing them. The final application to the meetings network then provided the opportunity to bring these theoretical and computational developments back to the problem that initially inspired this work.

Chapter 1

Basic Notions on Graphs

The origins of graph theory, as specified in [2], are commonly associated with a problem posed in the city of Königsberg in 1736. At that time, the city was divided by the two branches of the Pregel River and connected by seven bridges. This particular configuration led to the question of whether it was possible to cross all seven bridges exactly once during a single walk.

The problem was studied by Leonhard Euler, who introduced an abstract representation of the city in order to analyse it mathematically. The different areas of land were represented by vertices, while the bridges connecting them were represented by edges. This abstraction allowed the original geographical problem to be reformulated as a problem concerning the structure of a graph. By considering the degree of the vertices, Euler established that no walk satisfying the required conditions could exist.

The significance of Euler's contribution lies not only in the result obtained, but also in the method used to obtain it. The geographical details of the city could be disregarded and replaced by a mathematical structure containing only the information relevant to the problem. This showed that properties of a graph can determine whether certain configurations or paths are possible, independently of the particular way in which the graph is represented.

The Königsberg bridges problem is consequently regarded as a foundational example in the development of graph theory. It illustrates the central idea underlying this mathematical framework: complex or apparently different problems can be represented through vertices and edges, allowing their structural properties to be studied using formal mathematical tools.

This chapter introduces some basic notions from graph theory, starting from the basic definitions on graph theory introduced in Section 1.1 and then developing an interesting and, in some cases, innovative framework for the graph analysis in Section 1.2. In particular, several graph structures and centrality measures will be introduced in 1.2.1 and in 1.2.2, most of them are already well established in the literature, while there will be a novel contribution of this regarding new centrality measures based on the well-known concept of k -clique communities. Moreover, in order to better comprehend the interactions between sets of vertices, some examples of similarities and a new strategy for comparing collections of sets will be introduced respectively in 1.3.1 and in 1.3.2.

The starting source for the notions introduced in this chapter is provided by the lecture notes collected during the course "Matematica Discreta e Applicazioni" held by Professor Aldo Conca at Università degli Studi di Genova. These notions have then been

reorganised and integrated by referring to the following books: [2], [11], [7], [14] and [5].

1.1 Basic definitions

In this section the focus is on some starting notions mainly concerning the local approach of graph theory in order to create a common ground.

Definition 1.1 (Simple graph). *A simple undirected graph G , also just called a graph, is a pair (V, E) , where $V = \{v_1, \dots, v_n\}$ is the set of vertices and $E \subseteq \{A \subseteq V : |A| = 2\}$ is the set of edges.*

Remark 1.2. *More generally, one can consider a multigraph, that is, a graph that allows multiple edges, namely distinct edges connecting the same two vertices, as well as loops, that is, edges connecting a vertex to itself.*

Example 1.3. *Graphs are generally represented, in the plane, by a finite set of points, corresponding to the vertices, and a finite set of arcs, corresponding to the edges, each of which starts at a vertex i and ends at a vertex j . Vertices are usually indexed by natural numbers and/or letters of the alphabet and, in the cases considered here, their ordering is not relevant. Mainly natural numbers will be used in this work. The regions bounded by edges, together with the unbounded outer region, are called faces. An example of a multigraph with **multiple edges** and a **loop** is shown below.*

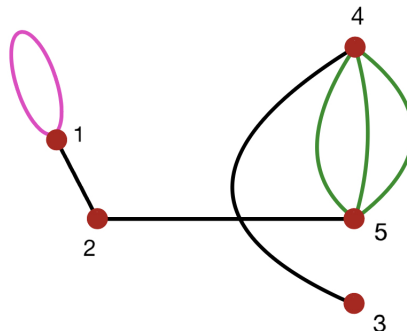


Figure 1.1

The following is an example of a simple graph.

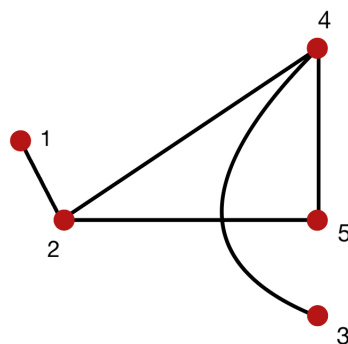


Figure 1.2

Sometimes, when the graphic representation or the definition of a graph may appear redundant, it can also be referred to as a square matrix.

Definition 1.4 (Adjacent matrix). Given a graph $G = (V, E)$ where $V = \{v_1, \dots, v_n\}$, the adjacent matrix of G is a square matrix $n \times n$, denoted as $A = (a_{i,j})$, where $a_{i,j} = 1$ if there exist an edge connecting the two vertices v_i and v_j , and 0 otherwise.

Example 1.5. The adjacent matrix of figure 1.2 is

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

Definition 1.6 (Relations among vertices and edges). Two vertices v, v' are said to be adjacent if they are endpoints of the same edge e . Two edges e, e' are said to be consecutive if they have a common vertex v . A vertex v and an edge e are said to be incident if $v \in e$. Finally, two faces are said to be adjacent if they share at least one edge.

Definition 1.7 (Degree). Let $G = (V, E)$ be a graph and let $v \in V$. The degree of v is defined as

$$\delta(v) = |\{e \in E : v \in e\}| \in \mathbb{N}.$$

That is, it is the number of edges incident to v , counting multiplicities.

Example 1.8. The degrees of the vertices in the graph shown in figure 1.3 are:

$$\delta(A) = 1, \quad \delta(B) = 5, \quad \delta(C) = 2, \quad \delta(D) = 4.$$

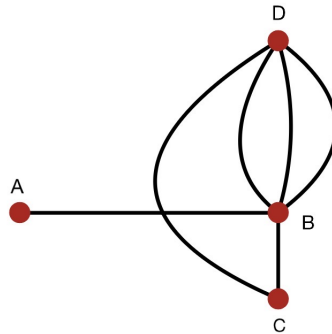


Figure 1.3

Definition 1.9 (Subgraph). Let $G = (V, E)$ be a graph and let $V_1 \subseteq V$ and $E_1 \subseteq E$. $G_1 = (V_1, E_1)$ is called a subgraph of G if, for every $e = v_i, v_j \in E_1$, one has $v_i, v_j \in V_1$.

Example 1.10. Given the graph G :

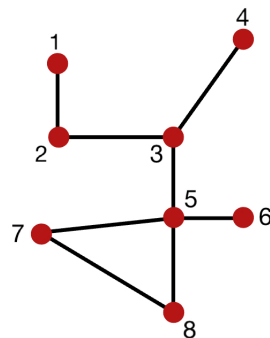


Figure 1.4

an example of one of its subgraphs is given by $G_1 = (V_1, E_1)$ where $V_1 = \{1, 2, 3, 5, 7\}$ and $E_1 = \{\{1, 2\}, \{2, 3\}, \{5, 7\}\}$

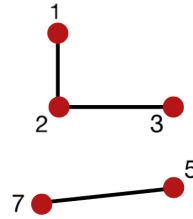


Figure 1.5

Definition 1.11 (Complete graph). A complete graph is a graph $G = (V, E)$ where $V = \{v_1, \dots, v_n\}$ and $E = \{\{v_i, v_j\} : v_1 \leq v_i < v_j \leq v_n\}$.

Definition 1.12 (Walk). Let $G = (V, E)$ be a graph. A walk in a graph is a sequence of adjacent vertices, not necessarily distinct. A walk is usually denoted as

$$(v_0, e_1, v_1, e_2, v_2, \dots, e_t, v_t)$$

where e_i is the edge $\{v_{i-1}, v_i\}$ for $i = 1, \dots, t$.

Remark 1.13. Note that in simple graphs, the edges of a walk are uniquely determined by the vertices, so the walk is often referred to as $(v_0, \dots, v_t)$, with the condition that v_{i-1} and v_i are adjacent for $i = 1, \dots, t$. In the following, this notation will be adopted.

Definition 1.14 (Path). Let $G = (V, E)$ be a graph. A path is a walk in which no vertex is repeated more than once.

Remark 1.15. Given a walk from vertex v to vertex w there is always a path connecting the two vertices.

Definition 1.16 (Shortest path). Let $G = (V, E)$ be a graph and let $v, w \in V$. A shortest path between v and w is a path between them having the minimum possible number of edges. It is often called the distance between vertex v and vertex w .

Definition 1.17 (Distance). Let $G = (V, E)$ be a graph and let $v, w \in V$. The distance between the two vertices v and w is the number of edges composing one of the shortest path among them.

Definition 1.18 (Cycle). A cycle is a walk in which no vertex is repeated, except for the first and last vertices, which coincide.

Example 1.19. Consider the following simple graph

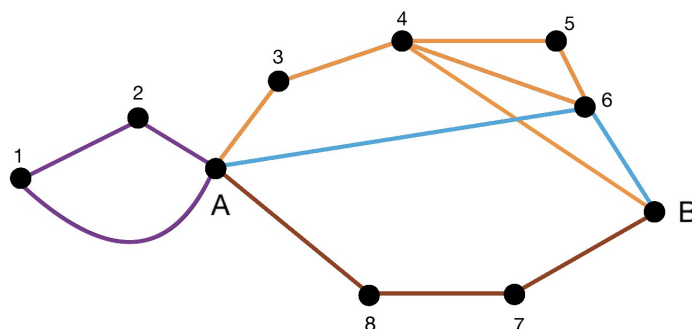


Figure 1.6

- $(A, 3, 4, 5, 6, 4, B)$ is an example of a *walk from A to B*;
- $(A, 8, 7, B)$ is an example of a *path from A to B*;
- $(A, 6, B)$ is the only *shortest path from A to B*;
- $(A, 2, 1, A)$ is an example of a *cycle from A to A*;
- the distance between A and B is 2.

Definition 1.20 (Connected or disconnected graph). A graph $G = (V, E)$ is *connected* if, for every $v, w \in V$, there exists a path from v to w . Otherwise it is called *disconnected*.

Definition 1.21 (Connected component). A *connected component* of a graph G is a maximal connected subgraph, i.e. there is a path between any two vertices that belong to the subgraph and no other vertex or edge can be added and still have the same property.

Example 1.22. For a better understanding of the connectivity concept, consider the following three graphs

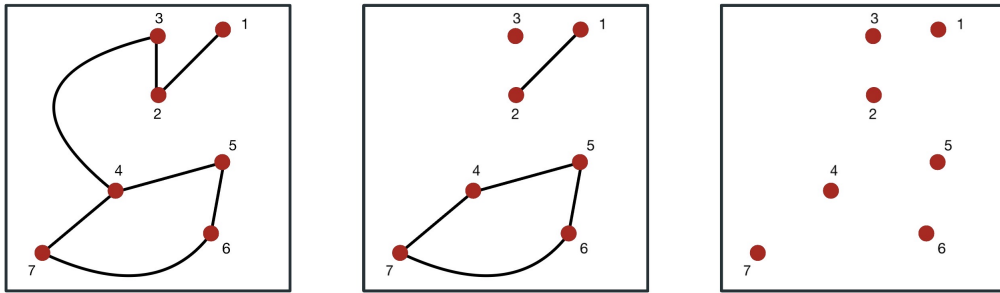


Figure 1.7

The first graph is a connected graph. The second one is disconnected and it is formed by three different connected components $C_1 = (E_1 = \{3\}, V_1 = \emptyset)$, $C_2 = (V_2 = \{1, 2\}, E_2 = \{\{1, 2\}\})$ and $C_3 = (V_3 = \{4, 5, 6, 7\}, E_3 = \{\{4, 5\}, \{5, 6\}, \{6, 7\}, \{4, 7\}\})$. The third graph is often called totally disconnected, meaning that there are no edges and every vertex form a single connected component.

Remark 1.23. The subdivision into connected components forms a partition of the vertices of the graph.

Definition 1.24 (Bridge). Given a graph $G = (V, E)$, an edge $e \in E$ is *bridge* if

$$|C(G)| \leq |C(G - e)|$$

where $C(G)$ is the set of all connected components of the graph G and $G - e$ is short for the graph $G = (V, E \setminus \{e\})$, meaning the graph G where the edge e is removed.

Example 1.25. In the following graph G , the edge $e = \{3, 6\}$ is a *bridge*,

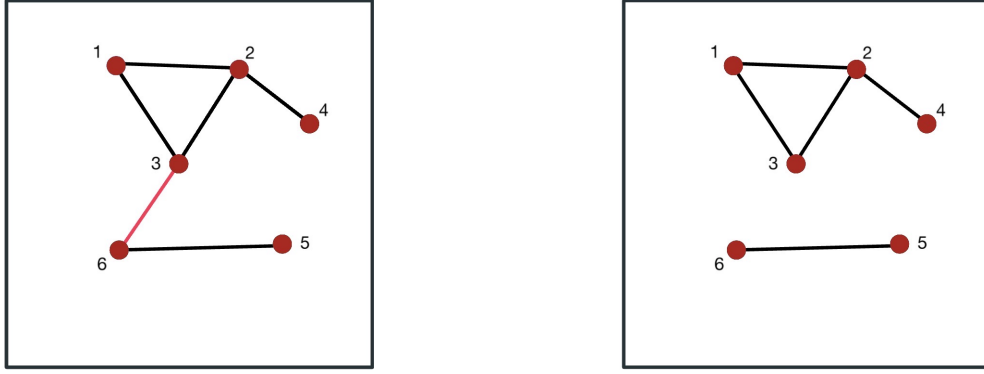


Figure 1.8

in fact, $|C(G)| = 1$ and $|C(G - e)| = 2$.

Sometimes, a graph can carry additional numerical information on the edges or on the vertices, so it is important to have a way of keeping track of this influence.

Definition 1.26 (Weight function). Let X be a set. A weight function is a map assigning each element of the set to a non-negative real number

$$\begin{aligned} w: X &\longrightarrow [0, +\infty) \\ x_i &\longmapsto y_i \end{aligned}$$

Since a graph is a couple of two sets, one of vertices and one of edges, the weight function can be associated with both of them. A vertex-weighted graph is not very common, but it is still worth mentioning it in this context.

Definition 1.27 (Edge-weighted graph). An edge-weighted graph is a triplet $G = (V, E, w_e)$ where $V = v_1, \dots, v_n$ is the set of vertices $E \subseteq \{A \subseteq V : |A| = 2\}$ is the set of edges and

$$\begin{aligned} w_e: E &\longrightarrow [0, +\infty) \\ e_i &\longmapsto y_i \end{aligned}$$

Definition 1.28 (Vertex-weighted graph). A vertex-weighted graph is a triplet $G = (V, E, w_v)$ where $V = v_1, \dots, v_n$ is the set of vertices $E \subseteq \{A \subseteq V : |A| = 2\}$ is the set of edges and

$$\begin{aligned} w_v: V &\longrightarrow [0, +\infty) \\ v_i &\longmapsto y_i \end{aligned}$$

Example 1.29. A conventional way of representing the two weighted graphs is the following:



Figure 1.9

The first one is a vertex-weighted graph and the second one an edge-weighted graph. A special attention is needed for the vertex-weighted graph, whenever the weights are all natural numbers, in order not to mix the numbers on the vertices with the usual vertex enumeration.

Remark 1.30. Conventionally, a simple graph can be seen as an edge-weighted graph with $w_e \equiv 1$. Moreover, just edge-weighted graphs will be considered and will be referred to as weighted graphs with the notation $G = (V, E, w)$.

A special note has still to be made, in fact, the definition of the degree of the vertices doesn't count the weights on each edge, so it is essential to define another quantity for this purpose.

Definition 1.31. Let $G = (V, E, w)$ be a weighted graph. For a vertex $v \in V$, the strength of v is defined as

$$s(v) = \sum_{\substack{e \in E \\ v \in e}} w(e).$$

Thus the strength takes into account the weights of edges incident to v .

Directed graphs provide a natural extension of the framework introduced so far by incorporating an orientation into the edges between vertices. Unlike the undirected case, the direction of an edge is relevant and may affect the structural properties of the graph.

Definition 1.32 (Directed graph). A directed graph, also called digraph for short, is a pair (V, E) of disjoint sets, where V is the set of vertices and E is the set of edges, together with two maps

$$\text{init} : E \longrightarrow V, \quad \text{ter} : E \longrightarrow V,$$

assigning to each edge $e \in E$ an initial vertex $\text{init}(e)$ and a terminal vertex $\text{ter}(e)$. The edge e is said to be directed from $\text{init}(e)$ to $\text{ter}(e)$.

Example 1.33. A common representation of a directed graph is

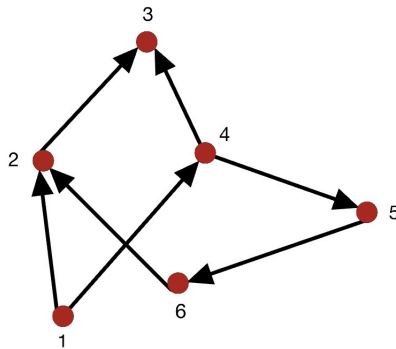


Figure 1.10

and it is obvious why it is important the order of vertices when talking about an edge, in fact there exist the edge $e = \{4, 3\}$ directed from $\text{init}(e) = 4$ to $\text{ter}(e) = 3$, but it does not exist the edge $e' = \{3, 4\}$ directed from $\text{init}(e') = 3$ to $\text{ter}(e') = 4$.

Remark 1.34. *In this case, contrary to what happens in undirected graphs, the order in which the vertices of an edge appear does matter. Moreover, if there exists an edge from vertex v to vertex w , it does not have to exist an edge in the opposite direction. Consequently, a walk from vertex v to vertex w does not necessarily imply the existence of a walk from w to v , in fact it is essential to follow the direction of the edges, meaning the order of the vertices.*

Definition 1.35 (In-degree and out-degree). *Let $G = (V, E)$ be a directed graph. For a vertex $v \in V$, the in-degree of v is defined as*

$$\delta^-(v) = |\{e \in E : \text{ter}(e) = v\}|,$$

while the out-degree of v is defined as

$$\delta^+(v) = |\{e \in E : \text{init}(e) = v\}|.$$

Example 1.36. *The degrees of the vertices in the digraph shown in figure 1.11 are*
 $\delta^-(A) = 0, \delta^+(A) = 2; \quad \delta^-(B) = 2, \delta^+(B) = 1; \quad \delta^-(C) = 2, \delta^+(C) = 0; \quad \delta^-(D) = 1, \delta^+(D) = 2;$
 $\delta^-(E) = 1, \delta^+(E) = 1; \quad \delta^-(F) = 1, \delta^+(F) = 1.$

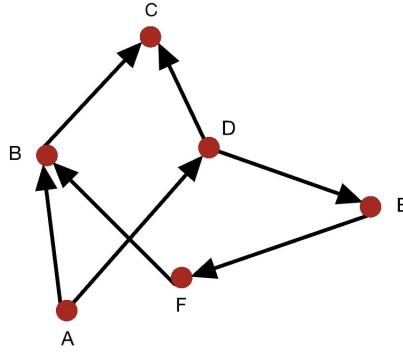


Figure 1.11

Note that, for example, there is a path from vertex 1 to vertex 3, but not vice versa.

1.2 Sociocentric and egocentric graph analysis

A graph can be analysed from both an egocentric and a sociocentric perspective: the egocentric perspective focuses on the individual vertices and their interactions with one another, such as the number of neighbours. The sociocentric one, on the other hand, describes the relationships among different collections of vertices. The properties of individual vertices and edges provide useful information about a graph, but they do not fully describe its overall structure. A global approach therefore considers the graph as a whole, focusing on the relationships among its different components and on the structural properties. This perspective makes it possible to characterize a graph and to identify patterns that cannot be captured by analysing individual vertices independently.

From now on, only undirected and unweighted simple graphs will be considered.

1.2.1 Graph structures: a sociocentric perspective

The aggregations of vertices can be divided into atomic communities and communities or clusters. Some examples of atomic communities are cliques, n -cliques, n -clans, n -plexes and n -cores. Gluing together atomic communities of the same kind, with the rules of the case, produces communities. In this section only k -clique communities will be analysed. Note that $n, k \in \mathbb{N}$; however for clarity the letter n will be used for the atomic communities whereas the letter k will be used for k -clique communities to avoid any potential confusion with n -cliques.

Definition 1.37 (Clique). *A complete subgraph, i.e. a subgraph with mutually adjacent nodes, is called clique. The cardinality of the set of vertices of the subgraph is called the size of the clique.*

Definition 1.38 (Maximal clique). *A maximal clique is a clique that is not properly contained in any larger clique of the graph.*

Example 1.39. *Consider the following representation of the graph $G = (V, E)$ with $V = \{1, 2, 3, 4\}$ and $E = \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}, \{3, 4\}\}$*

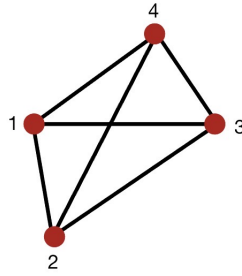


Figure 1.12

G is a clique of size 4 and it is also a maximal clique of size 4. But, for example, $G_1 = (V_1 = \{1, 3, 4\}, E_1 = \{\{1, 3\}, \{1, 4\}, \{3, 4\}\})$ is not a maximal clique, but it is just a clique

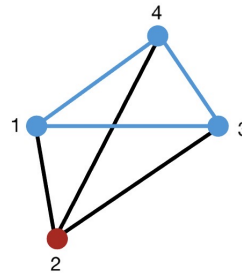


Figure 1.13

Consider the graph $G' = (V', E')$ with $V' = \{1, 2, 3, 4\}$ and $E' = \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}, \{3, 4\}\}$, depicted in the following figure

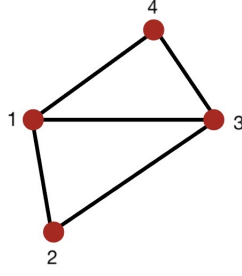


Figure 1.14

is neither a clique nor a maximal clique, but it is formed by two different maximal cliques $G'_1 = (V'_1 = \{1, 3, 4\}, E'_1 = \{\{1, 3\}, \{1, 4\}, \{3, 4\}\})$ and $G'_2 = (V'_2 = \{1, 2, 3\}, E'_2 = \{\{1, 2\}, \{1, 3\}, \{2, 3\}\})$

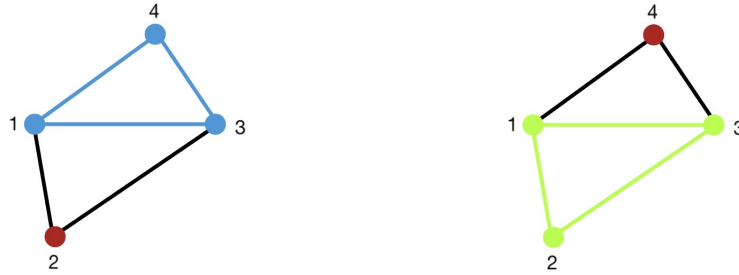


Figure 1.15

sharing the vertices 1 and 3 and the edge connecting them.

The concept of clique is quite selective, so more relaxed variations on the idea can be found, such as n -plexes and n -cores.

Definition 1.40 (n -plex). Let $n, m \in \mathbb{N}$. An n -plex of size m is a subgraph with m vertices such that each vertex is connected to at least $m - n$ of the others.

Example 1.41. Consider the following representations of three different graphs

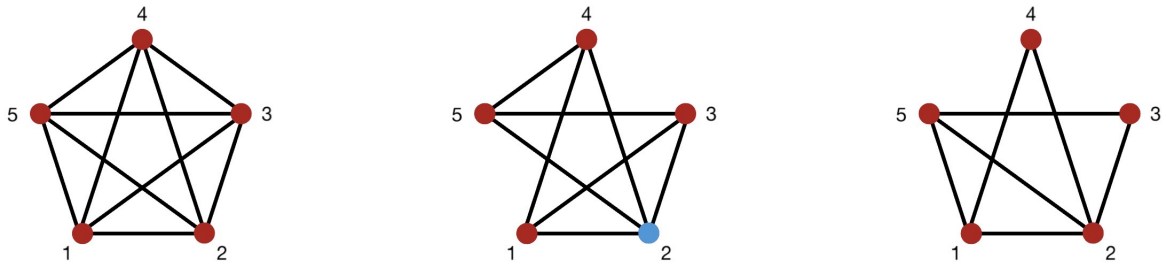


Figure 1.16

The first graph represents a 1-plex of size 5 that is also a clique. In fact, the definition itself says that a 1-plex of size m is a subgraph where every node is connected to at least $m - 1$ nodes, meaning that every node of the subgraph is connected to all the other nodes of the graph except for itself, since loops are not allowed in simple graphs. This definition is equivalent to the definition of cliques.

The second graph is a 2-plex of size 5 because each vertex has degree at least 3. To be more precise, all vertices of this graph have degree 3 except for vertex 2 which is connected

with 4 other vertices.

The third graph is an example of a 3-plex because each vertex has degree at least $m - n = 5 - 3 = 2$.

A peculiarity of cliques and n -plexes is that they don't form a partition of the vertices of the graph, in fact, overlapping is admitted and a vertex can be part of more atomic communities.

Definition 1.42 (n -core). Let $n \in \mathbf{N}$. An n -core is a subgraph in which each vertex is adjacent to at least n other vertices of the subgraph.

Example 1.43. Here is an example of a 3-core that is not a clique

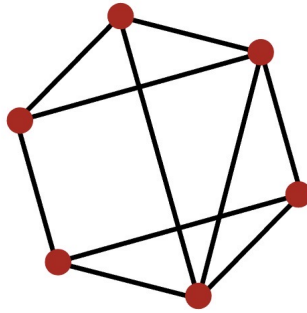


Figure 1.17

Remark 1.44. Note that an n -core with m vertices is also a $(m - n)$ -plex of size m , in fact, a $(m - n)$ -plex is a subgraph where every vertex is adjacent to at least $m - (m - n) = n$ other vertices of the subgraph. Also, in this case, a vertex cannot be part of more n -cores since, by definition, two overlapping n -cores would just form a single larger n -core. This does not happen in the case of n -plexes because of the variable m .

Two other generalizations of cliques are the n -cliques and the n -clans. The main difference between the two of them is in where the distance between two vertices is considered. In fact, in the n -cliques the distance between two nodes of the subgraph is calculated considering all the edges of the graph. In contrast, in the n -clans, only the edges within the subgraph can be considered.

Definition 1.45 (n -clique). Let $n \in \mathbf{N}$. An n -clique is a subgraph in which each vertex is no more than a distance n away from any other vertices of the subgraph via the edges of the graph.

Definition 1.46 (n -clan). Let $n \in \mathbf{N}$. An n -clan is a subgraph in which each vertex is no more than a distance n away from any other vertices of the subgraph via the edges of the subgraph itself.

Remark 1.47. By definition, a 1-clan is just a clique.

Example 1.48. Every n -clan is also an n -clique, but the opposite is false, as shown in the representation of the graph G

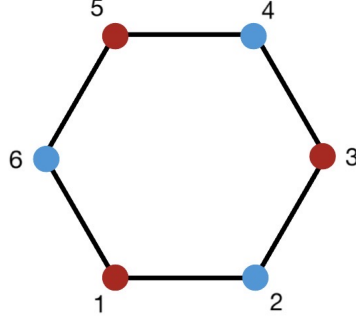


Figure 1.18

Here, the three vertices $\{2, 4, 6\}$ form the valid subgraph $G = (V = \{2, 4, 6\}, E = \emptyset)$ and they also form a 2-clique, but they are not connected via path of the 2-clique itself, so they are not a 2-clan. Another interesting case is shown in figure 1.19

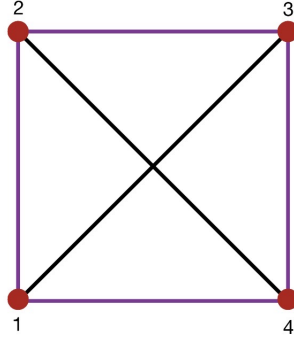


Figure 1.19

where the subgraph $G_1 = (V_1, E_1)$ where $V_1 = \{1, 2, 3, 4\}$ and $E_1 = \{\{1, 2\}, \{2, 3\}, \{3, 4\}, \{1, 4\}\}$ is a 1-clique but also a 2-clan.

Although there does not exist a concrete definition, a community is usually considered to be highly-connected group of vertices of the graph. A common approach for identifying communities in a graph uses the detection of k -clique communities.

Definition 1.49 (k -clique adjacency). Given $k \in \mathbb{N}$, $k \geq 2$. Two cliques σ and σ' of size k are k -adjacent if their intersection is a clique of size $k - 1$, i.e., if they share $k - 1$ vertices.

Example 1.50. In order for two 3-cliques, such as triangles, to be considered 3-connected, they must share a common edge, namely a 2-clique.

The adjacency definition allows to glue together different cliques.

Definition 1.51 (k -clique connectivity). Given $k \in \mathbb{N}$, $k \geq 2$. Two cliques σ and σ' of size k are k -connected if there exists a sequence of cliques such that any two consecutive cliques are k -adjacent.

Definition 1.52 (k -clique community). Let $k \in \mathbb{N}$, $k \geq 2$. A k -clique community of a graph G is a maximal union of cliques of size k that are pairwise k -connected.

Example 1.53. Here is an example of a graph G that has one 1-clique community, one 2-clique community, two 3-clique communities, one 4-clique community.

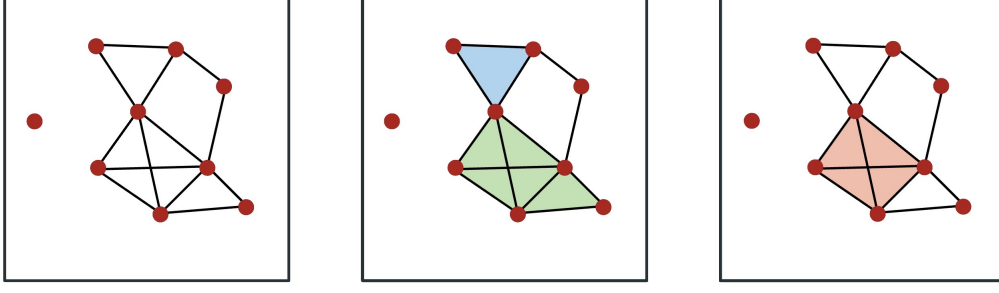


Figure 1.20

The identification of the k -clique communities for the graph $G = (V, E)$ proceeds as follows for different values of k :

- Case $k = 1$: the only 1-clique community is the isolated vertex.
- Case $k = 2$: the 2-clique community correspond to the standard connected components, except for isolated vertices. There is just one 2-clique community, that is the main connected component that contains 8 vertices.
- Case $k = 3$: Two distinct 3-clique communities emerge:
 - (a) The upper community C, formed by the single clique of size 3 (triangle) at the top, consisting of 3 vertices.
 - (b) The lower community C, formed by the four 2-adjacent cliques of size 3 and the adjacent bottom-right triangle.
- Case $k = 4$: There is a single 4-clique community corresponding to the complete subgraph in the lower-left region of the structure. This community consists of 4 vertices all connected with each other, so it is also a clique of size 4.

In the following part, some noteworthy properties of k -clique communities are presented.

Proposition 1.54. *Given a graph G , every $(k + 1)$ -clique community of G is contained in exactly one k -clique community of G .*

Proof. To prove the claim, two steps are needed.

- *Existence:* let C_{k+1} be a $(k + 1)$ -clique community of a graph G . By definition, C_{k+1} is a maximal union of cliques of size $k + 1$ that are pairwise $(k + 1)$ -connected, meaning that there exists a sequence of cliques of size $k + 1$ such that any two consecutive cliques are $(k + 1)$ -adjacent. This happens when their intersection is a clique of size k .

Consider now the cliques of size k contained in the cliques of size $k + 1$ forming C_{k+1} . Any two cliques of size k contained in the same clique of size $k + 1$ share at least $k - 1$ vertices and are therefore k -adjacent. Moreover, the sequence of cliques of size $k + 1$ connecting the elements of C_{k+1} induces a sequence of cliques of size k that are connected through a cliques of size $k - 1$. Hence, C_{k+1} is contained in a k -clique community of G .

Since the condition for k -adjacency is less restrictive than the condition for $(k + 1)$ -adjacency, the resulting k -clique community can contain additional vertices and cliques. By maximality, there exists a k -clique community C_k such that

$$C_{k+1} \subseteq C_k.$$

- *Uniqueness*: Suppose that there exist two distinct k -clique communities C_k and D_k such that

$$C_{k+1} \subseteq C_k \cap D_k.$$

Since C_{k+1} is a maximal sequence of $(k + 1)$ -adjacent cliques of size $k + 1$, it also contains cliques of size k . Thus, C_k and D_k share at least one clique of size k and, moreover, each one of the k -clique communities is a sequence of k -adjacent cliques of size k . Consequently, they are k -connected and, by maximality, must belong to the same k -clique community. This contradicts $C_k \neq D_k$. Therefore, C_{k+1} is contained in exactly one k -clique community. □

Proposition 1.55. *Given a graph G and H one of its subgraphs, every k -clique community of H is contained in exactly one k -clique community of G .*

Proof. Also for this claim, the same two steps are needed.

- *Existence*: it is an immediate consequence of the definition of subgraphs.
- *Uniqueness*: Suppose to have H a subgraph of the graph G and that there exist two distinct k -clique communities C_k and D_k of G such that

$$C_k^H \subseteq C_k \cap D_k.$$

where C_k^H is a k -clique community of H . For the definition of subgraph, $C_k^H \subseteq G$, so it is a sequence of k -adjacent cliques of size k , connecting C_k and D_k . So, for maximality, $C_k = D_k$. □

1.2.2 Centrality measures: an egocentric perspective

A crucial objective in graph theory is determining whether some nodes are more important than others, identifying some nodes playing a key role, or detecting nodes that may be discharged. To address these questions, it is necessary to quantify the centrality of the nodes, a mathematical framework that formalizes the importance of a node based on the influence that has on the other nodes.

Definition 1.56 (Centrality measure). *A centrality measure is a function*

$$\begin{aligned} F: \quad V &\longrightarrow \mathbb{R} \\ v_i &\longmapsto x_i \end{aligned}$$

assigning to each node a centrality value.

In this context, first some well known centrality measures, such as degree centrality and betweenness centrality, will be described; then some new centrality measures based on the global concept of clique communities are going to be introduced.

Perhaps the simplest centrality measure in a graph is just the degree of its vertices. It is also called *degree centrality* whenever it is more appropriate to emphasize its use as a centrality measure rather than a property of the node. In directed graphs, vertices have an in-degree and an out-degree, and both may be useful as measures of centrality.

Another centrality measure is *betweenness centrality*, which is based on shortest paths. In fact, it is often the case where the importance of a single vertex is based on the fact that it lies on the shortest path between other two.

Definition 1.57 (Betweenness centrality). Given $G = (V, E)$ a graph, the betweenness centrality of vertex $v_i \in V$ is

$$x_i = \sum_{\substack{v_j, v_k \in V \setminus \{v_i\} \\ v_j \neq v_k}} \frac{n_{j,k}^i}{g_{j,k}}.$$

Where $g_{j,k}$ is the number of shortest paths from vertex j to vertex k and $n_{j,k}^i$ is the number of those paths passing through v_i .

If two distinct vertices belong to two different connected components, the corresponding term of the sum evaluates to zero. As a result, the presence of multiple disconnected components naturally bounds the absolute value of this measure, since paths cannot cross between isolated components. Consequently, the measure evaluates the importance of a vertex strictly relative to the component to which it belongs. It is worth noting that, with this definition, every path connecting any pair of vertices in an undirected graph is counted twice: once from vertex v_j to vertex v_k , once in the reverse direction from v_k to v_j . While this double-counting could be solved by dividing the final score x_i by 2 for each vertex $v_i \in V$, this adjustment does not change the final ranking of the vertices. In fact, the importance of this measure is the result in relation with the other vertices, not the absolute result. Furthermore, by not dividing by 2, the same definition can also be extended to directed graphs. In this case, it is important to follow the directions of the edges in order to calculate the paths because there might exist a path from vertex v_j to vertex v_k but not in the other direction.

When a vertex lies on the shortest path of every distinct couple of vertices, it reaches the maximum value of the betweenness centrality it can get. A canonical representation of this situation is the star graph which consists of a central vertex v_i attached to $n - 1$ other vertices by single edges. In this configuration, the central vertex v_i lies on all $(n - 1)(n - 2)$ shortest paths.

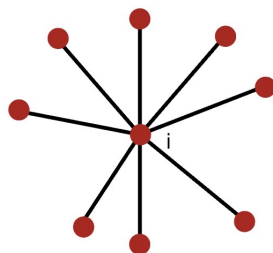


Figure 1.21

The graph G has $n = 10$ vertices and $x_i = (10 - 1)(10 - 2) = 72$. In fact, there is just one shortest path between any couple of vertices and it passes through v_i , so every possible couple of vertices formed by two different vertices that also differ from v_i contributes 1 to the sum of the betweenness centrality. The sum of all these possible couple of vertices is $(n - 1)(n - 2)$.

It is sometimes convenient to normalize the formula of betweenness centrality in order to obtain values in the range $[0,1]$.

Definition 1.58 (Normalized betweenness centrality). *Given a graph $G = (V, E)$, the normalised betweenness centrality of vertex $v_i \in V$ is*

$$x_i = \frac{1}{(n - 1)(n - 2)} \sum_{\substack{v_j, v_k \in V \setminus v_i \\ v_j \neq v_k}} \frac{n_{j,k}^i}{g_{j,k}}$$

Each centrality measure produces a ranking of the vertices of the graphs reflecting the specific features the measure is designed to capture. When a vertex exhibits high scores across multiple centrality measures, it indicates that the vertex has a position of structural importance. Conversely, if the scores of all measures are low, it could probably mean that this node has a small impact on the entire structure of the graph, suggesting it might be discharged depending on the analytical context. Some vertices might have high betweenness centrality and low degree centrality, or vice versa. Consequently, it is important to carefully consider whether the interest of the analysis is based on how well connected a vertex is or on how often a vertex falls “between” others.

Example 1.59. *The following graph is an example of measures discrepancies:*

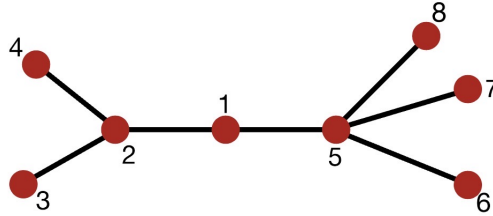


Figure 1.22

in fact, for example, vertex 1 has a low degree centrality, $\delta(1) = 2$, but a high betweenness

centrality:

$$\begin{aligned}
\frac{x_1}{2} &= \frac{1}{2} \sum_{\substack{v_j, v_k \in V \setminus \{1\} \\ v_j \neq v_k}} \frac{n_{j,k}^1}{g_{j,k}} \\
&= \frac{n_{2,3}^1}{g_{2,3}} + \frac{n_{2,4}^1}{g_{2,4}} + \frac{n_{2,5}^1}{g_{2,5}} + \frac{n_{2,6}^1}{g_{2,6}} + \frac{n_{2,7}^1}{g_{2,7}} + \frac{n_{2,8}^1}{g_{2,8}} + \frac{n_{3,4}^1}{g_{3,4}} + \frac{n_{3,5}^1}{g_{3,5}} + \frac{n_{3,6}^1}{g_{3,6}} \\
&\quad + \frac{n_{3,7}^1}{g_{3,7}} + \frac{n_{3,8}^1}{g_{3,8}} + \frac{n_{4,5}^1}{g_{4,5}} + \frac{n_{4,6}^1}{g_{4,6}} + \frac{n_{4,7}^1}{g_{4,7}} + \frac{n_{4,8}^1}{g_{4,8}} + \frac{n_{5,6}^1}{g_{5,6}} + \frac{n_{5,7}^1}{g_{5,7}} + \frac{n_{5,8}^1}{g_{5,8}} \\
&\quad + \frac{n_{6,7}^1}{g_{6,7}} + \frac{n_{6,8}^1}{g_{6,8}} + \frac{n_{7,8}^1}{g_{7,8}} \\
&= \frac{0}{1} + \frac{0}{1} + \frac{1}{1} + \frac{1}{1} + \frac{1}{1} + \frac{1}{1} + \frac{0}{1} + \frac{1}{1} + \frac{1}{1} + \frac{1}{1} + \frac{1}{1} + \frac{1}{1} + \frac{1}{1} + \frac{1}{1} + \frac{1}{1} + \frac{1}{1} \\
&\quad + \frac{0}{1} + \frac{0}{1} + \frac{0}{1} + \frac{0}{1} + \frac{0}{1} + \frac{0}{1} \\
&= 12.
\end{aligned}$$

Here is the graph with the degree centralities calculated for each vertex:

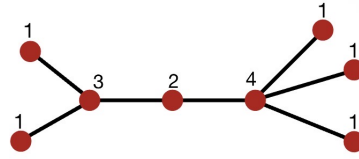


Figure 1.23

and the one with the betweenness centralities:

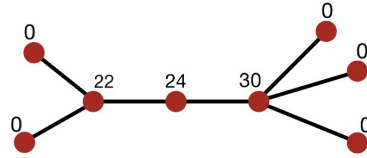


Figure 1.24

Before shifting the focus to a novel contribution to this work, two further centrality measures are introduced, despite being frequently overlooked. These are called Katz centrality and collective influence.

Katz centrality defines the centrality for a vertex based on the centrality of its neighbours

Definition 1.60 (Katz centrality). *Given a graph $G = (V, E)$, the Katz centrality of a vertex is defined as*

$$c_K(v_i) = \alpha \sum_{v_j \in V \setminus \{v_i\}} a_{i,j} C_K(j) + \beta$$

where α and β are positive constants, $A = (a_{i,j})$ is the adjacency matrix of the graph, whose eigenvalue are denoted by λ_i for $i = 1, \dots, n$.

Remark 1.61. The parameter β controls the initial centrality, while the parameter α satisfies the inequality:

$$\alpha < \frac{1}{\max\{\lambda_i : 1 \leq i \leq n\}}$$

Definition 1.62 (Collective influence). The collective influence computes the centrality of a vertex $v \in V$ of a graph $G = (V, E)$ according to the formula

$$CI_l(v) = (\delta(v) - 1) \sum_{w \in \delta B(v, l)} (\delta(w) - 1)$$

where $\delta(v)$ is the degree of the vertex v , $B(v, l)$ is the ball of radius l centered on the vertex v and $\delta B(v, l)$ is the frontier of the ball, that is, the set of vertices at distance l from v .

Having introduced k -clique communities as an effective way of clustering vertices of a graph in the previous subsection, it is natural to investigate whether this concept can also be used to define a new centrality measure and whether such a measure can provide additional information for the study of graph structures. Centrality measures incorporating k -clique community information have already been proposed in the literature [16], but the approach developed in this work introduces a different centrality measure based directly on the structure of these communities.

Let $\mathcal{C}_k(v)$ denote the set of all k -clique communities containing the vertex v , C one of its elements and $|C|$ the number of vertices contained in C . A first attempt to define a centrality measure is given below.

Definition 1.63. Given a graph $G = (V, E)$ and a vertex $v \in V$, define

$$c'(v) = \sum_{k=2}^{\infty} |\mathcal{C}_k(v)|$$

Remark 1.64. This quantity seems to be quite selective because it is relevant when a high number of k -clique communities is involved, a scenario that is relatively rare. However, a property worth noting is that, although the upper bound of the sum is ∞ , it reduces to a finite sum due to the finiteness of G .

To address the limitation of $c_o(i)$ a more informative measure can be defined.

Definition 1.65 (k -clique community centrality). Given a graph $G = (V, E)$ and a vertex $v_i \in V$, define

$$c_k(v) = \max_{C \in \mathcal{C}_k(v)} |C|$$

The k -clique community centrality of vertex v_i is

$$c(v) = \max_{k \geq 3} c_k(v)$$

Sometimes, it is beneficial to account for the evolution of the k -clique community to which a vertex belongs across different values of k . For this reason, consider the following generalised definition.

Definition 1.66 (Generalised k -clique community centrality). Given a graph $G = (V, E)$ and a vertex $v \in V$, define

$$\bar{c}_k(v) = \left| \bigcup_{C \in \mathcal{C}_k(v)} C \right|$$

The generalised k -clique community centrality of vertex v is

$$\bar{c}(v) = \left| \bigcup_{k=3}^{\infty} \bar{c}_k(v) \right|$$

Remark 1.67. Note that if a vertex of a given graph is part of more k -clique communities for the same k or for different k , the quantity $\bar{c}(v)$ considers it one time only, since it takes the union of all relevant communities.

Remark 1.68. Sometimes, it might be useful to calculate these centralities fixing a given k , especially if k -clique communities for that fixed k are particularly relevant for the structure of the graph.

Example 1.69. Consider the following graph and its vertex A

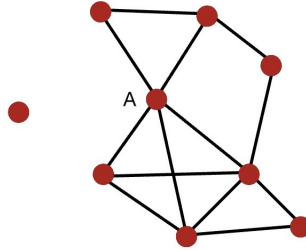


Figure 1.25

To compute its centralities $c'(A)$, $c(A)$, and $\bar{c}(A)$, it is necessary analyse the k -clique communities containing A across all relevant values of $k \geq 3$:

- For $k = 3$: vertex A acts as a bridge between the two distinct 3-clique communities that have 3 and 5 vertices respectively, with just one in common. Thus $|\mathcal{C}_3(A)| = 2$.
- For $k = 4$: vertex A is a member of the single 4-clique community that has 4 vertices, thus $|\mathcal{C}_4(A)| = 1$.

For $k \geq 5$, no cliques exist in G , so $|\mathcal{C}_k(A)| = 0$ for $k \geq 5$. Consequently, the centrality values for vertex A are:

$$c'(A) = \sum_{k=2}^{\infty} |\mathcal{C}_k(A)| = 1 + 2 + 1 = 4,$$

$$c(A) = \max_{k \geq 3} c_k(A) = \max_{k \geq 3} \max_{C \in \mathcal{C}_k(A)} |C| = \max\{5, 4\} = 5,$$

$$\bar{c}(A) = \left| \bigcup_{k=3}^{\infty} \bar{c}_k(v) \right| = \left| \bigcup_{k=3}^{\infty} \bigcup_{C \in \mathcal{C}_k(A)} C \right| = 7.$$

However, even without using more complex constructions such as k -clique communities, atomic communities can sometimes provide meaningful information about the structure of a graph. In such cases, they can be used to define a new centrality measure, such as the following.

Definition 1.70. Given a graph G with a subdivision into atomic communities $\mathcal{C} = \{C_1, \dots, C_r\}$, define

$$c_{\mathcal{C}}(v) = |\{C \in \mathcal{C} : v \in C\}|$$

1.3 Analysing sets of vertices

1.3.1 Similarities between sets of vertices

Comparing finite sets is a fundamental problem in many mathematical and applied contexts. Different notions can be introduced to quantify the relationship between two sets, depending on whether their separation, similarity, or structural properties are to be measured. In this section, three such approaches will be considered: *distance*, *similarity* and *metric*. These notions provide different perspectives for comparing sets and will be useful in the developments that follow. The first notion, which will be referred to simply as a distance in order to avoid cumbersome terminology, is formally a pseudo-semimetric. The second approach will include the definitions for classical Jaccard similarity and the overlap coefficient. Finally, the definition of metric will be introduced as a further way of quantifying the relationship between sets, satisfying stronger structural properties.

From now on, unless otherwise specified, X will denote a finite set, while A , B , and C will denote subsets of X .

Definition 1.71 (Distance). A function

$$\begin{aligned} d: \mathcal{P}(X) \times \mathcal{P}(X) &\longrightarrow \mathbb{R} \\ (A, B) &\longmapsto x_i \end{aligned}$$

is called a distance on X if, for all A, B , it holds:

- $d(A, B) \geq 0$ (non-negativity)
- $d(A, B) = d(B, A)$ (symmetry)
- $d(A, A) = 0$ (reflexivity)

Definition 1.72 (Similarity). A function

$$\begin{aligned} s: \mathcal{P}(X) \times \mathcal{P}(X) &\longrightarrow \mathbb{R} \\ (A, B) &\longmapsto x_i \end{aligned}$$

is called a similarity on X if, for all A, B , it holds:

- $s(A, B) \geq 0$ (non-negativity)
- $s(A, B) = s(B, A)$ (symmetry)
- $s(A, B) \leq s(A, A)$.

Proposition 1.73. Given a similarity s , if for every $A \subseteq X$ $s(A, A) = 1$, then the following transformations are distances:

- $d = 1 - s$,
- $d = \frac{1-s}{s}$ whenever $s(A, B) > 0$ for every $A, B \subseteq X$,
- $d = \sqrt{1-s}$,
- $d = \sqrt{2(1-s^2)}$,

- $d = -\ln(s)$.

Moreover, if $s(A, A) = \frac{\pi}{2}$, also

$$d = \arccos(s)$$

is a distance.

Two similarities measures are going to be introduced: Jaccard similarity and overlap coefficient similarity.

The formal origin of the Jaccard similarity index traces back to the work of Paul Jaccard in 1912 [10] on the distribution of alpine vegetation. Jaccard argued about the importance of botanical studies of common species, not only rare ones, for a more significant characterization of the underlying factors of biological distribution. In order to systematically analyze the presence of common flora, he divides the studied region into three parts and compares the composition on the flora by introducing the *coefficient of community*, now known as the Jaccard similarity, to rigorously quantify the floristic overlap between these regions. This coefficient is defined as the ratio of the number of species shared by two districts to the total number of all unique species present in both combined. Using this ratio, Jaccard showed that despite their geographical proximity and similar ecological environments, the plant communities of the three districts have remarkably distinct compositions.

The concept has been formalized as follows:

Definition 1.74 (Jaccard similarity). *The Jaccard similarity s_J is defined by*

$$s_J(A, B) = \begin{cases} 1 & \text{if } A = B = \emptyset, \\ \frac{|A \cap B|}{|A \cup B|} & \text{else.} \end{cases}$$

The overlap coefficient, also known as the Simpson coefficient or Szymkiewicz-Simpson coefficient, is a similarity measure that quantifies the degree of overlap between two finite sets. The historical origins can be traced back to Simpson's work on the quantitative comparison of faunal assemblages. In his 1936 study, Simpson used a measure of faunal resemblance to compare mammalian assemblages from different geographical regions [17]. The underlying idea was subsequently formalized in his 1947 work through a coefficient of faunal similarity based on the number of taxa shared by two faunas relative to the number of taxa in the smaller fauna [18]. Here, the term *taxa* denotes the units of biological classification, such as species or genus. In modern set-theoretic notation, this principle leads to the overlap coefficient defined below.

Definition 1.75 (Overlap coefficient). *The overlap coefficient is a similarity s_O defined by*

$$s_O(A, B) = \begin{cases} 1 & \text{if } A = B = \emptyset, \\ 0 & \text{if } A = \emptyset \text{ and } B \neq \emptyset, \text{ or } A \neq \emptyset \text{ and } B = \emptyset \\ \frac{|A \cap B|}{\min\{|A|, |B|\}} & \text{else.} \end{cases}$$

A straightforward inspection of the definition of similarity measure immediately verifies that these two expressions constitute valid similarities.

Even though these definitions are formulated in terms of general sets, the reader is encouraged to think about X as a set of vertices of a given graph G and as A , B and C as its subsets of vertices, as illustrated in the upcoming example. Adopting this perspective early on serves as a valuable exercise to better understand the concepts developed in the next part of this work, where these definitions will, indeed, be used to compare sets of vertices.

Example 1.76. *The following example shows how the above definitions can be applied on sets of vertices.*

If $A = \{1, 2, 3, 4\}$ and $B = \{1, 2, 3, 5, 6, 7, 8\}$

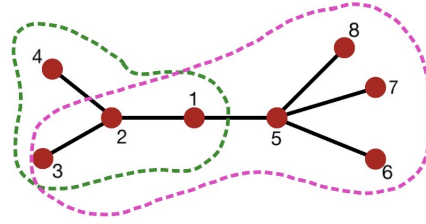


Figure 1.26

then $s_J(A, B) = \frac{|A \cap B|}{|A \cup B|} = \frac{3}{8}$ and $s_O(A, B) = \frac{|A \cap B|}{\min\{|A|, |B|\}} = \frac{3}{4}$.
And notice that if $A = \{1, 2, 3, 4\}$ and $B = \{1, 2, 3, 4, 5, 6, 7, 8\}$

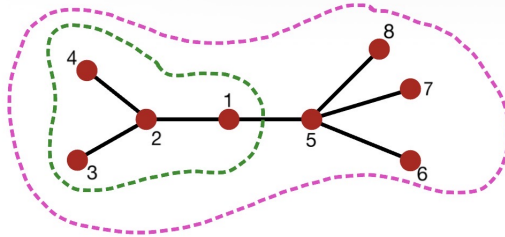


Figure 1.27

$s_J(A, B) = \frac{|A \cap B|}{|A \cup B|} = \frac{4}{8} = \frac{1}{2}$ and $s_O(A, B) = \frac{|A \cap B|}{\min\{|A|, |B|\}} = \frac{4}{4} = 1$.

It is readily observed that $s_J(A, A) = 1$ and $s_O(A, A) = 1$ and since the conditions $s(A, B) \leq s(A, A)$ and $s(A, B) \geq 0$ hold for any pair of sets A and B and any similarity s , consequently, for all A and B , the similarity score is bounded such that $0 \leq s_J(A, B) \leq 1$ and $0 \leq s_O(A, B) \leq 1$. These conditions fulfill the hypothesis of 1.73, yielding the following result:

Theorem 1.77. *In accordance with proposition 1.73, the following are distances:*

- $d_J = 1 - s_J$ is a distance called Jaccard distance;
- $d_O = 1 - s_O$ is a distance called overlap coefficient distance.

Furthermore, the same transformation used to obtain a distance can lead to a metric. This is the case for the Jaccard distance that produces the Jaccard metric. The same cannot be said for the overlap coefficient distance.

Definition 1.78 (Metric). A function

$$\begin{aligned} d: \mathcal{P}(X) \times \mathcal{P}(X) &\longrightarrow \mathbb{R} \\ (A, B) &\longmapsto x_i \end{aligned}$$

is called a metric on X if, for every A, B and C , there holds:

- $d(A, B) \geq 0$ (non-negativity)
- $d(A, B) = 0$ if and only if $A = B$
- $d(A, B) = d(B, A)$ (symmetry)
- $d(A, C) \leq d(A, B) + d(B, C)$ (triangle inequality).

Remark 1.79. A metric is a distance with the addition of the triangle inequality and the fact that if $d(A, B) = 0$ then $A = B$.

Theorem 1.80. $d_J = 1 - s_J$ is a metric.

Proof. It has already been proved that d_J is a distance, moreover it is obvious from the definition that if $d_J(A, B) = 0$ then $A = B$. So it is left to prove that the triangle inequality holds:

$$d_J(A, C) \leq d_J(A, B) + d_J(B, C) \quad (1.1)$$

There are three cases:

- if $A = C = \emptyset$,
- if $A = \emptyset$ and $C \neq \emptyset$ or vice versa but, thanks to the symmetry property of distances, it is enough to check one case,
- if $A \neq \emptyset$ and $C \neq \emptyset$.

For the first case, it suffices to show that $d_J(\emptyset, \emptyset) = 1 - s_J(\emptyset, \emptyset) = 1 - 1 = 0$. So, the inequality to prove is

$$0 \leq d_J(\emptyset, B) + d_J(B, \emptyset).$$

It comes directly from the definition of distances since $d(A, B) \geq 0$ for every A and B .

For the second case, the inequality to prove is the following:

$$d_J(\emptyset, C) \leq d_J(\emptyset, B) + d_J(B, C).$$

If $B = \emptyset$ then

$$1 - 0 \leq (1 - 1) + (1 - 0)$$

$$1 \leq 1$$

that is trivially true.

If $B \neq \emptyset$, then the validity of

$$1 \leq 1 + d_J(B, C)$$

comes directly from the non-negativity property of distances.

Another immediate case is the third one when A and C are non-empty sets, but $B = \emptyset$, in fact

$$d_J(A, C) \leq d_J(A, \emptyset) + d_J(\emptyset, C)$$

$$d_J(A, C) \leq 1 + 1$$

is verified thanks to the upper bound equal to 1 of the distances.

Finally, the core of the proof is when A , B and C are all non-empty sets. By substituting the definition $d_J(A, B) = 1 - \frac{|A \cap B|}{|A \cup B|}$, the triangular inequality 1.1 becomes:

$$1 - \frac{|A \cap C|}{|A \cup C|} \leq \left(1 - \frac{|A \cap B|}{|A \cup B|}\right) + \left(1 - \frac{|B \cap C|}{|B \cup C|}\right)$$

$$1 - \frac{|A \cap C|}{|A \cup C|} \leq 2 - \frac{|A \cap B|}{|A \cup B|} - \frac{|B \cap C|}{|B \cup C|}$$

Subtracting 1 from both sides and multiplying by -1 yields:

$$1 + \frac{|A \cap C|}{|A \cup C|} \geq \frac{|A \cap B|}{|A \cup B|} + \frac{|B \cap C|}{|B \cup C|}$$

which is equivalent to showing that:

$$\frac{|A \cap B|}{|A \cup B|} + \frac{|B \cap C|}{|B \cup C|} \leq \frac{|A \cap C|}{|A \cup C|} + 1. \quad (1.2)$$

To simplify inequality 1.2, it is necessary to break up $A \cup B \cup C$ into 7 mutually disjoint subsets $\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}, \mathcal{E}, \mathcal{F}, \mathcal{G}$, defined as follows:

$$\begin{aligned} \mathcal{A} &= A \setminus (B \cup C), & \mathcal{B} &= B \setminus (A \cup C), & \mathcal{C} &= C \setminus (A \cup B) \\ \mathcal{D} &= (A \cap B) \setminus C, & \mathcal{E} &= (B \cap C) \setminus A, & \mathcal{F} &= (A \cap C) \setminus B \\ \mathcal{G} &= A \cap B \cap C \end{aligned}$$

To better visualize the subdivision into subsets the following diagram can be drawn:

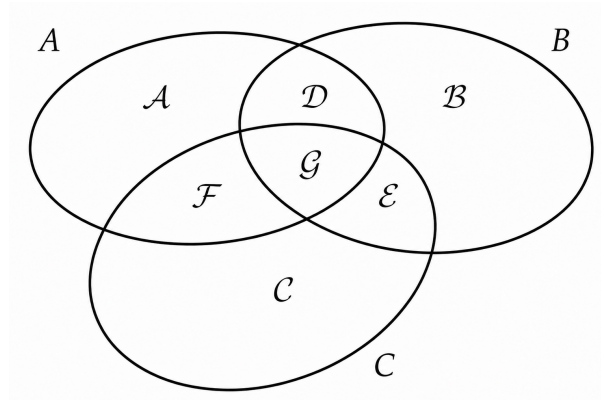


Figure 1.28

Let a, b, c, d, e, f, g denote the number of elements in the corresponding sets $\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}, \mathcal{E}, \mathcal{F}, \mathcal{G}$ and define the total number of elements in the union $A \cup B \cup C$ as:

$$v = |A \cup B \cup C| = a + b + c + d + e + f + g$$

Using these variables, the cardinalities of the relevant intersections and unions become:

- $A \cap B = \mathcal{D} \cup \mathcal{G} \implies |A \cap B| = d + g$
- $B \cap C = \mathcal{E} \cup \mathcal{G} \implies |B \cap C| = e + g$
- $A \cap C = \mathcal{F} \cup \mathcal{G} \implies |A \cap C| = f + g$
- $|A \cup B| = v - c$
- $|B \cup C| = v - a$
- $|A \cup C| = v - b$

Substituting these expressions into inequality 1.2 yields:

$$\frac{d+g}{v-c} + \frac{e+g}{v-a} \leq \frac{f+g}{v-b} + 1 \quad (1.3)$$

For shorthand, we adopt the symbolic notation:

$$\begin{aligned} c' &= v - c, & a' &= v - a, & b' &= v - b \\ d' &= d + g, & e' &= e + g, & f' &= f + g \end{aligned}$$

Inequality 1.3 then reads:

$$\frac{d'}{c'} + \frac{e'}{a'} \leq \frac{f'}{b'} + 1 \quad (1.4)$$

Multiplying both sides of the obtained inequality by the common positive denominator $a'b'c' > 0$:

$$a'b'd' + b'c'e' \leq a'c'f' + a'b'c'$$

Next, a', b', c' are expanded in terms of v :

- $a'b' = (v-a)(v-b) = v^2 - av - bv + ab$
- $b'c' = (v-b)(v-c) = v^2 - bv - cv + bc$
- $a'c' = (v-a)(v-c) = v^2 - av - cv + ac$

Substituting these expansions into the inequality:

$$(v^2 - av - bv + ab)d' + (v^2 - bv - cv + bc)e' \leq (v^2 - av - cv + ac)f' + (v^2 - av - bv + ab)c'$$

Substituting the terms d', e', f' and c' with their definitions, multiplying, collecting terms by powers of v and moving all terms to the right-hand side, the inequality takes the form:

$$\begin{aligned} 0 &\leq +v^3 \\ &\quad + v^2(-a - b - c - d - e + f - g) \\ &\quad + v(ab + ac + bc + ad + bd + be + ce + 2bg - af - cf) \\ &\quad + (-abc - abd - abg - bce - bcf + acg + acf). \end{aligned} \quad (1.5)$$

Define S_1 and S_2 by:

$$S_1 = ab + ac + bc + ad + bd + be + ce + 2bg - af - cf$$

$$S_2 = -abc - abd - abg - bce - bcg + acg + acf.$$

Notice that the coefficient of v^2 can be rewritten by adding and subtracting f :

$$-a - b - c - d - e + f - g = 2f - (a + b + c + d + e + f + g) = 2f - v$$

Therefore, the first two terms of inequality 1.5 simplify to:

$$v^3 + v^2(2f - v) = v^3 + 2fv^2 - v^3 = 2fv^2$$

The inequality now reduces to:

$$0 \leq v(2fv + S_1) + S_2 \tag{1.6}$$

Expanding $2fv = 2f(a + b + c + d + e + f + g)$ all negative terms in S_1 (namely $-af$ and $-cf$) are canceled. Then the remaining negative terms in S_2 (namely $-abc, -abd, -abg, -bce, -bcg$) are completely canceled by the positive terms arising from the expansion of the last v in the equation.

Since only non-negative terms remain after cancellation (as $a, b, c, d, e, f, g \geq 0$), the inequality:

$$0 \leq v(2fv + S_1) + S_2$$

is established. □

Example 1.81. *The overlap coefficient does not induce a metric in fact, if $A \subsetneq B$ then $s_O(A, B) = 1 - \frac{|A \cap B|}{\min\{|A|, |B|\}} = 1 - \frac{|A|}{|A|} = 0$ but $A \neq B$*

1.3.2 Comparing collections of sets of vertices

The following framework constitutes a contribution of the present work.

In the previous section, several ways of comparing two finite sets were introduced. However, in many applications, and in particular in the analysis of graphs, the objects of interest are not individual sets, but collections of sets. This is the case, for instance, when the vertices of a graph are divided into communities. Each community can be represented as a subset of the vertex set, and a community structure can be viewed as a finite collection of subsets of vertices. In such a setting, comparing two collections of sets cannot, in general, be reduced to comparing two sets at a time. If a graph can be associated with two different community structures, comparing the communities individually would require a suitable correspondence between the two collections. Different choices of such a correspondence may lead to different similarity scores, and therefore the result of the comparison may depend on how the communities are paired. To address this problem, a framework is proposed for comparing two finite collections of sets by considering all possible correspondences between their elements. The proposed approach consists in considering all possible bijections between the two set collections and a similarity measure to evaluate the compatibility of each pair is introduced. Each similarity values are summed together in order to have a single value for each possible bijection and the maximum score over all is selected.

More precisely, suppose to have a finite set V and let $\mathcal{A} = \{A_1, \dots, A_m\}$ and $\mathcal{B} = \{B_1, \dots, B_n\}$ where $\mathcal{A}, \mathcal{B} \subseteq \mathcal{P}(V)$ and $A_1, \dots, A_m, B_1, \dots, B_n \in \mathcal{P}(V)$.

Example 1.82. Let, for example, V be the set of vertices of the graph $G = (V, E)$ and $\mathcal{A}$ and $\mathcal{B}$ be the two collections of the set of vertices induced by the two partitions. Note that the subdivision of the vertices into communities does not have to be a partition, for example when k -cliques communities, cliques or n -plexes are used, but for simplicity that will be the hypothesis for this example.

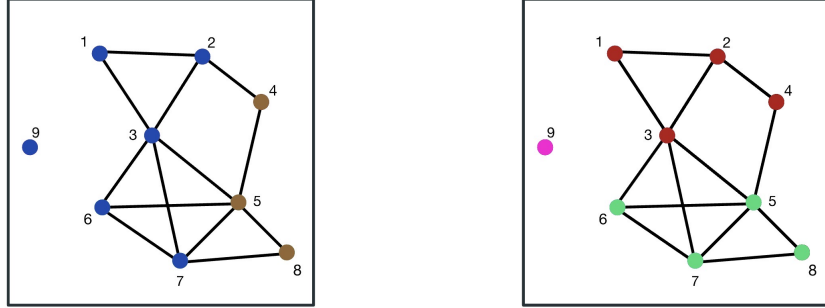


Figure 1.29

So $\mathcal{A} = \{A_1, A_2\}$ with $A_1 = \{1, 2, 3, 6, 7, 9\}$, $A_2 = \{4, 5, 8\}$ and $\mathcal{B} = \{B_1, B_2, B_3\}$ with $B_1 = \{1, 2, 3, 4\}$, $B_2 = \{5, 6, 7, 8\}$, $B_3 = \{9\}$.

Since $\mathcal{A}$ and $\mathcal{B}$ might have two different cardinalities, in order to compare their elements through a bijection, the two collections can be extended by introducing empty sets. Since several copies of the empty set may be required, the resulting objects are formally multisets of subsets of V . Define two new subsets of $\mathcal{P}(V)$ as follows:

$$\bar{\mathcal{A}} = \{A_1, \dots, A_m, \emptyset, \dots, \emptyset\} \text{ where } \emptyset \text{ appears in } n \text{ copies}$$

$$\bar{\mathcal{B}} = \{B_1, \dots, B_n, \emptyset, \dots, \emptyset\} \text{ where } \emptyset \text{ appears in } m \text{ copies.}$$

From now on, the cardinality will count all multiplicity of the multiset. So, at this point $|\bar{\mathcal{A}}| = |\bar{\mathcal{B}}| = m + n$, so it is possible to pair the elements of the two new collections by building all the $(m + n)!$ possible bijections $\phi : \bar{\mathcal{A}} \rightarrow \bar{\mathcal{B}}$.

Proceeding further, consider $\Phi_{\bar{\mathcal{A}}, \bar{\mathcal{B}}}$ the set of all bijections between $\bar{\mathcal{A}}$ and $\bar{\mathcal{B}}$. Then, the similarity between the two new collections, $\bar{\mathcal{A}}$ and $\bar{\mathcal{B}}$, can be evaluated by summing the similarity scores for each pair induced by a fixed bijection of $\Phi_{\bar{\mathcal{A}}, \bar{\mathcal{B}}}$.

Definition 1.83 (Matching similarity score). Given a similarity $s : \mathcal{P}(V) \times \mathcal{P}(V) \rightarrow [0, 1]$ and a bijection $\phi : \bar{\mathcal{A}} \rightarrow \bar{\mathcal{B}}$ between two finite collections $\mathcal{A}$ and $\mathcal{B}$ of sets of V , the matching similarity score for each ϕ is

$$m_s(\phi) = \sum_{A \in \bar{\mathcal{A}}} s(A, \phi(A))$$

The value of $c_s(\phi)$ depends on the particular matching induced by the bijection ϕ . Therefore, different bijections may produce different scores. Since the purpose is to quantify the maximum possible similarity between the two collections, the highest matching similarity score is considered.

Definition 1.84 (Similarity score between set collections). Let $\mathcal{A}$ and $\mathcal{B}$ be two finite collections of sets with cardinality m and n respectively, and let $\bar{\mathcal{A}}$ and $\bar{\mathcal{B}}$ be the two finite extensions with cardinality $m + n$. Let s be a similarity measure and $\phi : \bar{\mathcal{A}} \rightarrow \bar{\mathcal{B}}$ be a bijection. The similarity score between $\mathcal{A}$ and $\mathcal{B}$ is defined as

$$\text{score}_s(\mathcal{A}, \mathcal{B}) = \max_{\phi \in \Phi_{\bar{\mathcal{A}}, \bar{\mathcal{B}}}} m_s(\phi).$$

Example 1.85. Here is an example to better visualize the construction of this new quantities.

Suppose to have the following graph G with two different vertices partitions: $\mathcal{A} = \{A_1\}$ with $A_1 = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ and $\mathcal{B} = \{B_1, B_2\}$ with $B_1 = \{1, 2, 4\}$ and $B_2 = \{3, 5, 6, 7, 8, 9\}$:

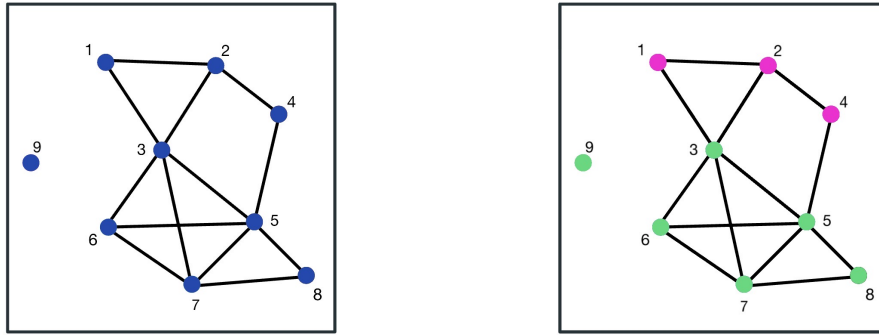


Figure 1.30

The two partition have different numbers of sets, so define the two new collections:

$$\bar{\mathcal{A}} = \{A_1, \emptyset_1, \emptyset_2\}$$

$$\bar{\mathcal{B}} = \{B_1, B_2, \emptyset_1\}.$$

At this point $|\bar{\mathcal{A}}| = |\bar{\mathcal{B}}| = 3$. Now consider all the possible bijections between this two

collections $\Phi_{\bar{\mathcal{A}}, \bar{\mathcal{B}}} = \{\phi_1, \phi_2, \phi_3, \phi_4, \phi_5, \phi_6\}$:

$$\begin{array}{lcl} \phi_1: & \bar{\mathcal{A}} & \longrightarrow \bar{\mathcal{B}} \\ & A_1 & \longmapsto B_1 \\ & \emptyset_1 & \longmapsto B_2 \\ & \emptyset_2 & \longmapsto \emptyset_1 \end{array}$$

$$\begin{array}{lcl} \phi_4: & \bar{\mathcal{A}} & \longrightarrow \bar{\mathcal{B}} \\ & A_1 & \longmapsto B_1 \\ & \emptyset_1 & \longmapsto \emptyset_1 \\ & \emptyset_2 & \longmapsto B_2 \end{array}$$

$$\begin{array}{lcl} \phi_2: & \bar{\mathcal{A}} & \longrightarrow \bar{\mathcal{B}} \\ & A_1 & \longmapsto \emptyset_1 \\ & \emptyset_1 & \longmapsto B_1 \\ & \emptyset_2 & \longmapsto B_2 \end{array}$$

$$\begin{array}{lcl} \phi_5: & \bar{\mathcal{A}} & \longrightarrow \bar{\mathcal{B}} \\ & A_1 & \longmapsto \emptyset_1 \\ & \emptyset_1 & \longmapsto B_2 \\ & \emptyset_2 & \longmapsto B_1 \end{array}$$

$$\begin{array}{lcl} \phi_3: & \bar{\mathcal{A}} & \longrightarrow \bar{\mathcal{B}} \\ & A_1 & \longmapsto B_2 \\ & \emptyset_1 & \longmapsto \emptyset_1 \\ & \emptyset_2 & \longmapsto B_1 \end{array}$$

$$\begin{array}{lcl} \phi_6: & \bar{\mathcal{A}} & \longrightarrow \bar{\mathcal{B}} \\ & A_1 & \longmapsto B_2 \\ & \phi_1 & \longmapsto B_1 \\ & \phi_2 & \longmapsto \emptyset_1 \end{array}$$

Then, decide a similarity to calculate the matching similarity scores for every bijections. In this example suppose to take the Jaccard similarity

$$s_J(A, B) = \begin{cases} 1 & \text{if } A = B = \emptyset, \\ \frac{|A \cap B|}{|A \cup B|} & \text{else.} \end{cases}$$

Take ϕ_1 and calculate $m_s(\phi_1) = \sum_{A \in \bar{\mathcal{A}}} s_J(A, \phi_1(A))$ as follow. The first step is to calculate all the three Jaccard similities:

- $s_J(A_1, \phi_1(A_1)) = s_J(A_1, B_1) = \frac{|A_1 \cap B_1|}{|A_1 \cup B_1|} = \frac{3}{9} = \frac{1}{3},$
- $s_J(\emptyset_1, \phi_1(\emptyset_1)) = s_J(\emptyset_1, B_2) = \frac{|\emptyset_1 \cap B_1|}{|\emptyset_1 \cup B_1|} = 0,$
- $s_J(\emptyset_2, \phi_1(\emptyset_2)) = s_J(\emptyset_2, \emptyset_1) = 0.$

So

$$m_s(\phi_1) = \sum_{A \in \bar{\mathcal{A}}} s_J(A, \phi_1(A)) = s_J(A_1, \phi_1(A_1)) + s_J(\emptyset_1, \phi_1(\emptyset_1)) + s_J(\emptyset_2, \phi_1(\emptyset_2)) = \frac{1}{3}.$$

Do the same for all the other bijections and obtain:

$$m_s(\phi_1) = \frac{1}{3}$$

$$m_s(\phi_4) = \frac{1}{3}$$

$$m_s(\phi_2) = 0$$

$$m_s(\phi_5) = 0$$

$$m_s(\phi_3) = \frac{2}{3}$$

$$m_s(\phi_6) = \frac{2}{3}$$

And finally, the similarity score between $\mathcal{A}$ and $\mathcal{B}$ is:

$$score_{s_J}(\mathcal{A}, \mathcal{B}) = \max_{\phi_i \in \Phi_{\bar{\mathcal{A}}, \bar{\mathcal{B}}}} m_{s_J}(\phi_i) = \frac{2}{3}.$$

The following proposition provides immediate bounds for the two new definitions.

Proposition 1.86. *Let $\mathcal{A}$ and $\mathcal{B}$ be two finite collections of sets with cardinality m and n respectively. Let s be a similarity measure and $\phi \in \Phi_{\bar{\mathcal{A}}, \bar{\mathcal{B}}}$. It follows that*

- $0 \leq m_s(\phi) \leq |\mathcal{A}| + |\mathcal{B}|$ for every $\phi \in \Phi_{\bar{\mathcal{A}}, \bar{\mathcal{B}}}$;
- $0 \leq score_s(\mathcal{A}, \mathcal{B}) \leq |\mathcal{A}| + |\mathcal{B}|$.

Proof. Since $s(A, B) \geq 0$ for all $A, B \in \mathcal{P}(V)$, any matching similarity score $c_s(\phi)$ is defined as a sum of non-negative real numbers:

$$m_s(\phi) = \sum_{A \in \bar{\mathcal{A}}} s(A, \phi(A)) \geq 0 \quad \text{for all } \phi \in \Phi_{\bar{\mathcal{A}}, \bar{\mathcal{B}}}.$$

Consequently, also $score_s(\mathcal{A}, \mathcal{B}) \geq 0$.

To establish the upper bound $m_s(\phi) \leq |\mathcal{A}| + |\mathcal{B}|$, it is necessary to use the definition of matching similarity itself, in particular when is declared the hypothesis that is needed a similarity $s : \mathcal{P}(V) \times \mathcal{P}(V) \rightarrow [0, 1]$. In fact, this proves that $m_s(\phi)$ is a sum of $|\bar{\mathcal{A}}|$ quantities that are between 0 and 1. Since $|\bar{\mathcal{A}}| = |\mathcal{A}| + |\mathcal{B}|$, the inequality is proved. This also prove that $score_s(\mathcal{A}, \mathcal{B}) \leq |\mathcal{A}| + |\mathcal{B}|$. $\square$

Chapter 2

Algorithms for Structure Identification and Graph Dismantling

The aim of this chapter is to apply the tools described in the previous chapter to the analysis of graphs. This represents an essential theoretical step for achieving a more complete understanding of graph structures, since it makes it possible to relate abstract concepts, such as community structures or measures, with given graphs and their properties. In particular, the focus is on understanding how the structural properties of a graph can describe the organisation of its vertices and can be the cause of a quick disruption of its connectivity. To this purpose, two new algorithms have been developed and implemented in Python.

Suppose that a graph is given together with a subdivision of its vertices. A natural question is whether one of the community structures described in Section 1.2.1 is able to reproduce, or at least approximate, the original subdivision of the vertices. This question is particularly relevant when the initial clustering depends on information external to the graph structure. The first algorithm has been, indeed, developed to consider a subdivision of the nodes of the graphs into communities and then to compare the obtained groups with the original division. The comparison between the sets of vertices is performed using the two already introduced similarity, meaning the Jaccard similarity and the overlap coefficient.

The second algorithm addresses a different characteristic of the graph, namely its robustness to the removal of vertices. In particular, it has to answer the questions on how well connected the graph is and on how strong those connections are when vertices are progressively removed according to their importance within the graph. In the literature, there is no universally accepted way of measuring these parameters. For this reason in the algorithm are involved more centrality measures and different quantities in order to determine an index of connectivity of the graph.

The disruption process is carried out by ranking the vertices according to a given centrality measure and removing them sequentially, or five at the time if the number of vertices is sufficiently large, adopting the centrality score as a priority criterion. After each removal, the chosen disruption condition is updated in order to monitor the evolution of the graph and its decreasing curve is shown at the end of the algorithm. The number

of steps needed for obtaining a graph where all remaining vertices are isolated, as well as the speed at which the connectivity index curve declines, are used as indicator of the robustness of the graph.

2.1 Community detection algorithm

To better contextualise this section, it is useful to recall the discussion on graph structures and the similarity measures, namely the Jaccard similarity and the overlap coefficient, presented in Sections 1.2.1 and 1.3.1, respectively. It is also necessary to recall the similarity score definition given in Section 1.3.2.

A simplification is required for the implementation of this algorithm: both atomic communities and communities have to be represented as sets of vertices rather than as subgraphs. As a consequence, the subtle distinction between n -clans and n -cliques is not taken into account in the algorithm. Moreover, the input graph is considered to be an undirected simple graph. It may also be an edge-weighted simple graph; however, the edge weights do not affect the results of the algorithm, since neither the identification nor the comparison of the communities depends on the weights assigned to the edges.

The vertices of the graph need to have a given subdivision.

2.1.1 Algorithm description

The steps of the introduced algorithm are described here.

The code has been implemented in Python (Appendix A) using the library package `networkx` for the study of structures, dynamics and functions of graphs, the library `numpy` for matrix operations and the library package `matplotlib` for the visualisation part. Note that at the beginning of each item in the bulleted list, there are numbers referring to the corresponding Python function definition number. In the algorithm some examples of atomic communities, communities and similarities have been considered.

- (a) [Python function definition: 1] The algorithm starts by reading an input file that represents an undirected weighted simple graph $G = (V, E)$. The file has to be structured in three columns of numbers: the first two describe the endpoints of the edges of the graph, and the third one refers to the weight of that edge. To explicitly include isolated vertices, lines of the form $(v, v, 0)$ can be provided in the input file, so the vertex v will be registered while the artificial loop of degree 0 will be discharged. In the “main” of the algorithm the original given subdivisions $\mathcal{A} = \{A_1, \dots, A_n\}$ has to be written.
- (b) [Python function definition: 2] After having decided which atomic community or community of interest one wants to check on the graph, depending on its structure, the algorithm can compute:
 - Atomic communities, as defined in Section 1.2.1;
 - k -clique communities for each $k \geq 2$. In this case, the algorithm automatically iterates over all valid integer values $k \in [2, k_{\max}]$.

Then the collection $\mathcal{B} = \{B_1, \dots, B_n\}$ is created. Note that both functions for cliques and n -cliques have been written, even though cliques are just 1-cliques.

Moreover, in this case and in most of the applications, the atomic communities are all considered maximal.

- (c) [Python function definition: 3] The algorithm prints the graph colouring each vertex according to the belonged community. It divides the vertex into more colours if it is part of more communities and it colours it black if it does not belong to any.
- (d) Now there are two different subdivisions of the vertices of the graph: $\mathcal{A}$ is the original one and $\mathcal{B}$ is the one generated by the function.
- (e) [Python function definitions: 4, 5, 6] The algorithm compares the two collections of sets using $score_s(\mathcal{A}, \mathcal{B})$ with $s = s_J$ and $s = s_O$.

2.1.2 A step by step example

The algorithm will now be examined in detail by following its steps on an example.

- (a) Suppose to have the following graph G together with the following subdivision of the vertices:
 - $A_1 = \{1, 2, 3, 4\}$;
 - $A_2 = \{5, 6, 7, 8\}$;
 - $A_3 = \{9\}$.

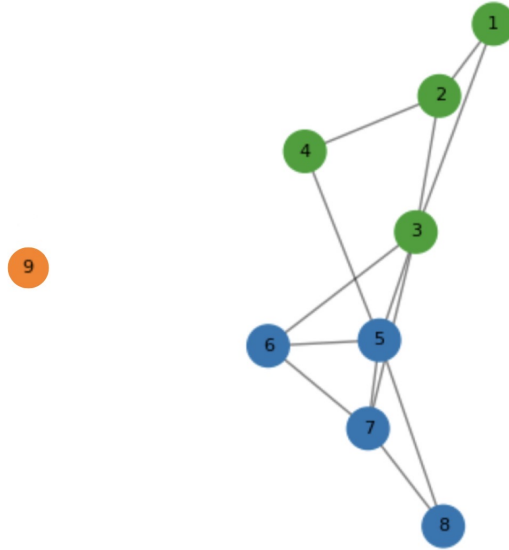


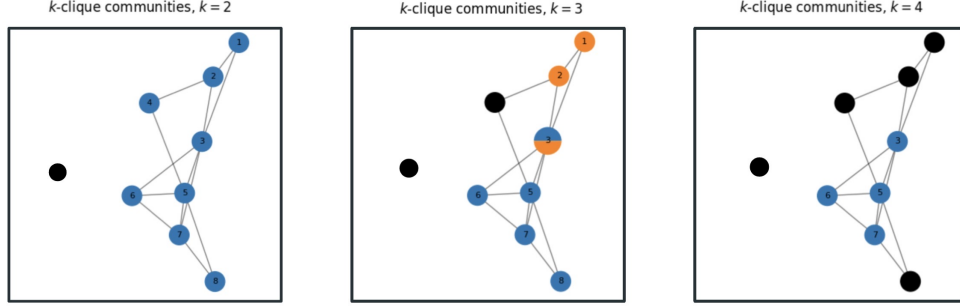
Figure 2.1

Note that this graph is just another way of representing the second graph in figure 1.82.

- (b) The algorithm allows for determining which one of the known graph structures is the most suitable for representing the subdivision $\mathcal{A}$. In the specific example, just by looking at the graph, one may think that the subdivision into k -clique communities, already calculated in Example 1.53, could probably best reproduce the original subdivision. If the choice is not obvious from the beginning, then it

would be wise to check more subdivisions $\mathcal{B}$ originated from different communities and atomic communities. For this example, just k -clique communities for various k will be controlled, for the sake of brevity.

(c) For each $k \geq 2$, the algorithm provided the following graphs:



There are not any k -clique communities for $k \geq 5$.

(d) In this case, there are not just two different subdivisions of the set of the vertices, but there is a new couple for each fixed k for $2 \leq k \leq 4$ where $\mathcal{A}$ remains the same and $\mathcal{B}$ changes every time:

- $k = 2$: there is just one 2-clique communities $B_1 = \{1, 2, 3, 4, 5, 6, 7, 8\}$, so $\mathcal{B} = \{B_1, \}$ and $\mathcal{A} = \{A_1, A_2, A_3\}$.
- $k = 3$: there are two different 3-clique communities $B_1 = \{1, 2, 3\}$ and $B_2 = \{3, 5, 6, 7, 8\}$, so $\mathcal{B} = \{B_1, B_2\}$ and $\mathcal{A} = \{A_1, A_2, A_3\}$.
- $k = 4$: there is just one 4-clique community $B_1 = \{3, 5, 6, 7\}$, so $\mathcal{B} = \{B_1\}$ and $\mathcal{A} = \{A_1, A_2, A_3\}$.

(e) Now the algorithm calculates the similarity scores between each couple of $\mathcal{A}$ and $\mathcal{B}$ both for the Jaccard similarity and the overlap coefficient.

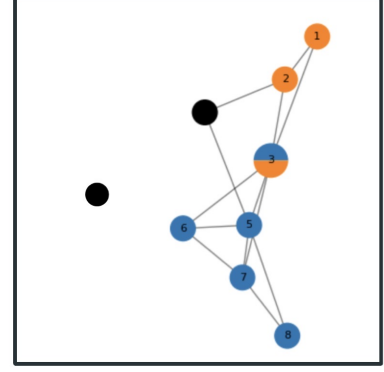
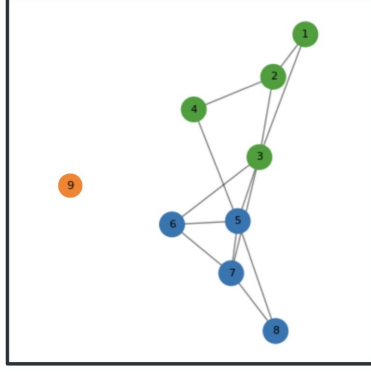
- $k = 2$:



$$- \text{score}_J(\mathcal{A}, \mathcal{B}) = \max_{\phi \in \Phi_{\mathcal{A}, \mathcal{B}}} c_J(\phi) = 1, 5$$

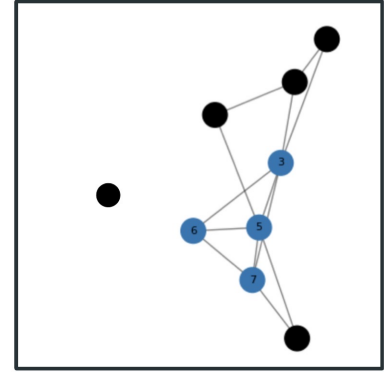
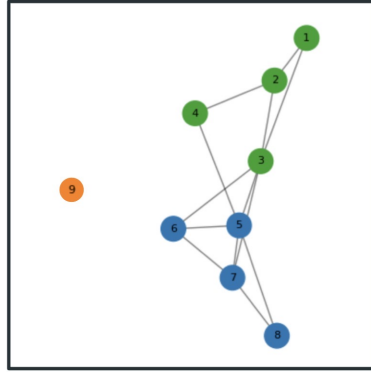
$$- \text{score}_O(\mathcal{A}, \mathcal{B}) = \max_{\phi \in \Phi_{\mathcal{A}, \mathcal{B}}} c_O(\phi) = 2$$

- $k = 3$:



$$\begin{aligned}
 - \text{score}_J(\mathcal{A}, \mathcal{B}) &= \max_{\phi \in \Phi_{\mathcal{A}, \mathcal{B}}} c_J(\phi) = 3,55 \\
 - \text{score}_O(\mathcal{A}, \mathcal{B}) &= \max_{\phi \in \Phi_{\mathcal{A}, \mathcal{B}}} c_O(\phi) = 4
 \end{aligned}$$

- $k = 4$:



$$\begin{aligned}
 - \text{score}_J(\mathcal{A}, \mathcal{B}) &= \max_{\phi \in \Phi_{\mathcal{A}, \mathcal{B}}} c_J(\phi) = 1,6 \\
 - \text{score}_O(\mathcal{A}, \mathcal{B}) &= \max_{\phi \in \Phi_{\mathcal{A}, \mathcal{B}}} c_O(\phi) = 1,75
 \end{aligned}$$

As a conclusion, the algorithm provided the results to complete the following table

$\mathcal{B}$	2-clique communities	3-clique communities	4-clique communities
Score			
Score_J	1, 5	3,5	1, 6
Score_0	2	4	1, 74

Table 2.1

showing that 3-clique communities are the better representing the original subdivision $\mathcal{A}$ of the vertices of the graph. To better visualise this result recall from Proposition 1.86 that the upper bound of $\text{score}_s(\mathcal{A}, \mathcal{B})$ is $|\mathcal{A}| + |\mathcal{B}|$ for every similarity s and in this example:

- 2-clique communities: $\text{score}_s(\mathcal{A}, \mathcal{B}) \leq 4$ since $\mathcal{B} = \{B_1\}$ and $\mathcal{A} = \{A_1, A_2, A_3\}$;
- 3-clique communities: $\text{score}_s(\mathcal{A}, \mathcal{B}) \leq 5$ since $\mathcal{B} = \{B_1, B_2\}$ and $\mathcal{A} = \{A_1, A_2, A_3\}$;

- 4-clique communities: $score_s(\mathcal{A}, \mathcal{B}) \leq 4$ since $\mathcal{B} = \{B_1\}$ and $\mathcal{A} = \{A_1, A_2, A_3\}$.

. Knowing that, the results from Table 2.2 can be normalised producing another table showing the percentages

$\mathcal{B}$ Score	2-clique communities	3-clique communities	4-clique communities
$Score_J$	37,5%	70%	40%
$Score_O$	20%	80%	$\approx 40\%$

Table 2.2

2.2 Disrupting graph algorithm

This algorithms aims to proposing a disrupting procedure tracking the decay of structural connectivity.

In this algorithm the symbol ρ_i will denote a value measuring the connectivity of the graph at the i -th step of the removal procedure. To establish the connectivity measure to be adopted the community detection algorithm is used as a guide. If, for example, the subdivision of the vertices of the graph into cliques appears to be the best at representing the original one then it could be smart to introduce a quantity ρ_i that prioritizes the strengths of cliques rather than other characteristics of the structure of the graph. If, however, one does not wish to use the previous algorithm to describe this quantity, other indicators such as connected components defined in Definition 1.21 can be used to introduce, for example, the largest connected components (LCC) of a graph G , that is the connected component of the graph G containing the highest number of vertices. Then, introduce $LCC(G)$ for the size of the largest connected component of G .

As for the centrality measures, if the community detection algorithm reveals an high compatibility with some of the communities or the atomic communities, as well as before, the suggestion is to use the centrality measures 1.63,1.65,1.66 and 1.70, or new ones based on those to compute the ranking. Otherwise also well known centrality measures such as the degree centrality or the betweenness centrality are strong choices. A good way to operate is also to select different centrality measures to compute and compare them.

The aim of this algorithm is to remove nodes, so at each iteration i a new graph G_i is created and it will have less vertices than the graph G_{i-1} obtained after the previous iteration $i - 1$. After every i -th iteration, the chosen $\rho_i(G_i)$ is calculated.

In this situation, as well as in the first algorithm, even though the function that is in charge of reading the input graph G can read weighted simple graphs, this will not influence the results of the algorithm since graph structures do not depend on weighted edges. On the contrary to what happened in the first algorithm, even if the input graph has isolated vertices, the reading function will not see them since they do not influence the disgregation process.

2.2.1 Algorithm description

The proposed algorithm operates through the following steps.

The code has been implemented in Python using the library package `networkx` for the study of structures, dynamics and functions of graphs and the library package `matplotlib` for the visualisation part. Note that at the beginning of each item in the bulleted list, there are numbers referring to the corresponding Python function definition number. Some of them, like the function definitions implementing ρ_i or the centrality measures, are just some examples of what can be calculated, but they can be changed if other quantities are better suiting the graph structure.

- (a) [Python function definition: 1] An input graph G_0 is given. As in the first algorithm, the file has to be structured in three columns of numbers: the first two describe the endpoints of the edges of the graph, and the third one refers to the weight of that edge.
- (b) [Python function definitions: 3, 4, 5, 6] The centrality measures are calculated for each vertex of G_0 .
- (c) [Python function definitions: 2, 8, 9] The algorithm computes $\rho_i(G_0)$ according to the selected connectivity measure.
- (d) [Python function definitions: 11, 12] This step depends on the removal strategy: either one vertex or five vertices are removed from the graph. The order of the removal depends on the centrality measure decided. At every iteration, a new graph G_i is obtained.
- (e) [Python function definitions: 2, 8, 9] $\rho_i(G_i)$ is computed.
- (f) [Python function definitions: 3, 4, 5, 6] The algorithm calculates the centrality measures for each remaining vertex.
- (g) Steps (d), (e) and (f) are repeated until the obtained graph consists of only isolated vertices.

2.2.2 A step by step example

The algorithm will now be examined in detail by following its steps on an example.

- (a) Suppose that, as input, the same graph 2.1 used in the previous step by step example is given.

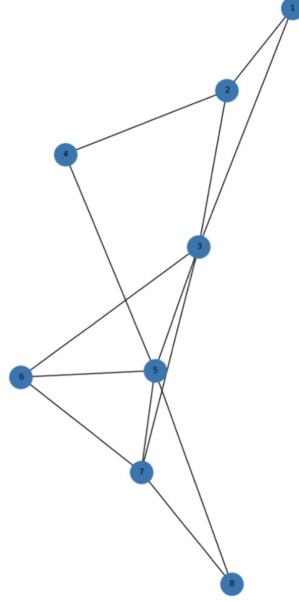


Figure 2.2

Note that vertex $\{9\}$ is missing, since it is an isolated vertex and the algorithm does not read those.

- (b) Since the community detection algorithm suggested that 4-clique communities are quite good at reproducing the original vertex subdivision, it has been decided to use the generalised k -clique community centrality for $k = 4$, as suggested in 1.68. Moreover, the algorithm also considers the degree of each node in order to handle cases in which two or more nodes have the same generalised k -clique community centrality value. The algorithm produced the following results for G_0 :

- Vertex 3: generalized k-clique centrality = 4; degree = 5
- Vertex 5: generalized k-clique centrality = 4; degree = 5
- Vertex 7: generalized k-clique centrality = 4; degree = 4
- Vertex 6: generalized k-clique centrality = 4; degree = 3
- Vertex 2: generalized k-clique centrality = 0; degree = 3
- Vertex 1: generalized k-clique centrality = 0; degree = 2
- Vertex 4: generalized k-clique centrality = 0; degree = 2
- Vertex 8: generalized k-clique centrality = 0; degree = 2

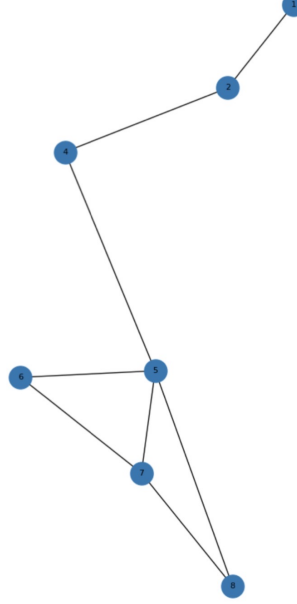
- (c) For the same reasons as in the previous step, it has been decided for a measure of connectivity that would represent the importance of k -clique communities for this graph:

$$\rho_i(G_i) = \sum_{k=2}^{\infty} k |C_k|$$

where C_k denotes the largest k -clique community of G_i . So in this example $\rho_0(G_0) = (2 \times 8) + (3 \times 5) + (4 \times 4) = 47$ (check Example 1.53 as reference for the calculation of k -clique communities).

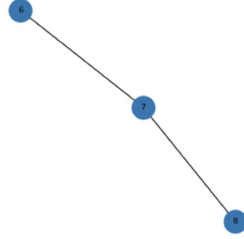
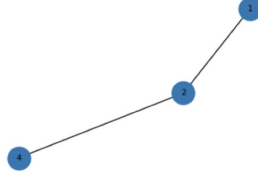
(d,e,f,g) Here, the removal strategy is to proceed one vertex at the time.

- The first vertex to be removed is vertex {3}. Since it has the same centrality measure and the same degree as vertex {5}, the algorithm uses the order of the vertex names to break the tie. The resulting graph G_1 is:



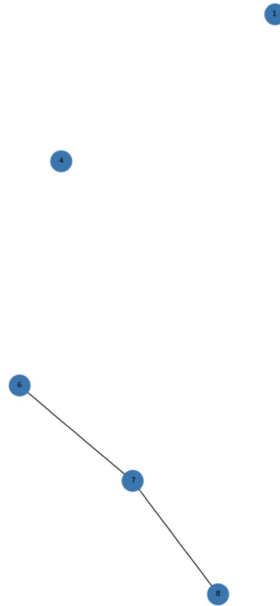
- The algorithm calculates the ranking of the vertices of G_1 , but since there are not 4-clique communities anymore, all the centralities of the remaining vertices would be 0, so the algorithm calculates $\bar{c}_3(v)$ for all vertices:
 - Vertex 5: generalized k-clique centrality = 4; degree = 4
 - Vertex 7: generalized k-clique centrality = 4; degree = 3
 - Vertex 6: generalized k-clique centrality = 4; degree = 2
 - Vertex 8: generalized k-clique centrality = 0; degree = 2
 - Vertex 2: generalized k-clique centrality = 0; degree = 2
 - Vertex 4: generalized k-clique centrality = 0; degree = 2
 - Vertex 1: generalized k-clique centrality = 0; degree = 1

and computes $\rho_1(G_1) = 26$. Note that it is necessary to calculate the ranking each time, since the order on which vertices appear might be different every time. Graph G_2 is computed by removing node {5}:



- All generalized k -clique centralities are zero for $k = 4$, so centrality is recalculated with $k = 3$.
 - Vertex 2: generalized k -clique centrality = 3; degree = 2
 - Vertex 7: generalized k -clique centrality = 3; degree = 2
 - Vertex 1: generalized k -clique centrality = 3; degree = 1
 - Vertex 4: generalized k -clique centrality = 3; degree = 1
 - Vertex 6: generalized k -clique centrality = 3; degree = 1
 - Vertex 8: generalized k -clique centrality = 3; degree = 1

and computes $\rho_2(G_2) = 6$. Graph G_3 is computed by removing node $\{2\}$:



- All generalized k -clique centralities are zero also for $k = 3$, so centrality is recalculated with $k = 2$.
 - Vertex 7: generalized k -clique centrality = 3; degree = 2

- Vertex 6: generalized k-clique centrality = 3; degree = 1
- Vertex 8: generalized k-clique centrality = 3; degree = 1
- Vertex 1: generalized k-clique centrality = 0; degree = 0
- Vertex 4: generalized k-clique centrality = 0; degree = 0

and computes $\rho_3(G_3) = 6$. Graph G_4 is computed by removing node $\{7\}$:



- All vertices are isolated.
- The evolution of the value of the connectivity measure across the iterations is shown in the following graph

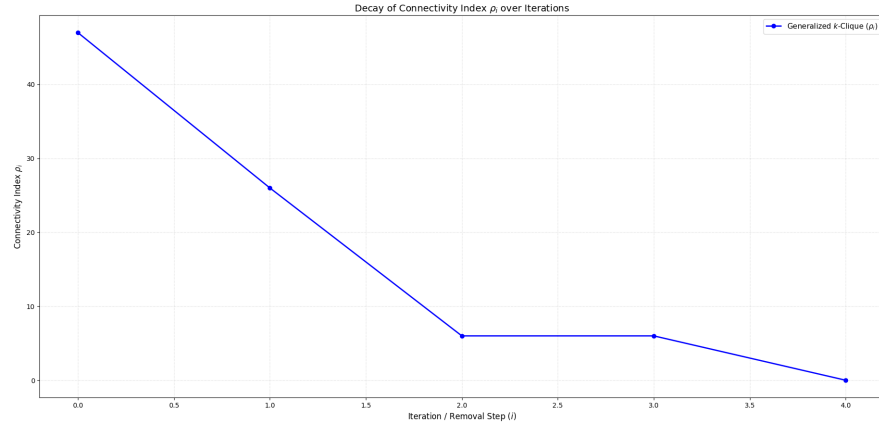


Figure 2.3

From these results, it is clear that four iterations are necessary to achieve a completely disconnected graph. Another fact is that, after only two iterations, the graph is already divided into smaller connected components, as the connectivity index drops significantly from 26 to 6 between the second and third iterations. Considering that the initial graph contains 9 nodes (including the isolated vertex at the start), reaching the desired outcome in just four steps represents a successful result.

Chapter 3

An Application to a Mafia Case

3.1 State of the art

Graph theory is a well-established branch of discrete mathematics providing a formal framework for modelling apparently complex problems. A graph can be used to represent complex networks, which are systems composed of a large number of entities connected by specific ties. So, network science has its roots in graph theory: it borrows its theoretical framework and applies it to empirical data, testing the value of each tool by the insights it offers about a system's structure or evolution. Moreover, network science offers a language through which different disciplines can interact with each other.

The modern study of complex networks is generally associated with the work of Watts and Strogatz [20] and, one year later, with that of Barabási and Albert [1]. The outcome of the papers is that real-world networks differ substantially from both regular networks and random networks and they can have high clustering and short path lengths at the same time. In particular, they highlighted the importance of network structure for the dynamics of a complex system especially in pointing out some universal structural properties.

However, only in the past decade the theoretical tools of complex analysis have been applied to social network analysis (SNA), even though SNA itself has a much longer history and predates the emergence of complex network theory by several decades. This combined study has proven particularly fascinating in criminology, where graph theory and complex analysis were able to represent social interactions within criminal organizations. In this framework, individuals can be modelled as vertices in a graph, and the relationships between them, such as if they met or called each other or if they organized crimes together, are represented as edges connecting those vertices. In SNA literature and, more generally, where real systems are involved, graphs are often called networks and they are composed by nodes and links instead of vertices and edges. From now on, this terminology will be adopted.

A foundational contribution to this line of research is the work of Sparrow [19], who highlighted the importance to apply network analysis to criminal intelligence, even before the emergence of modern complex network theory. In fact, social network analysis originated in the early 1920s for studying and describing relational structures and Sparrow thought of combining the tools of this discipline, such as notions of centrality and equivalence and the concept of weak ties, to criminal networks. He formalized the invitation to network theorists and law enforcement agencies to embrace their skills and work together.

From here, Cavallaro, Bagdasar, De Meo, Fiumara and Liotta [6] conducted more

recent research building the bridge between graph theory and social network analysis. An overview of the key methods and tools available for the analysis of criminal networks has been provided, starting from a real-world case study in which data on interactions among suspects within two Sicilian Mafia clans were extracted from juridical documents, producing two weighted undirected graphs. The authors analyse the two graphs by considering weight distribution, shortest path length and by comparing their structures with well-known artificial graphs.

The study of criminal networks through SNA has been approached from different perspectives. One prominent line of research focuses on the identification of key actors, such as brokers and bosses [3]. Brokers can often be identified through betweenness centrality, a measure that captures how often an individual is positioned along the shortest path between two other nodes of the network. The importance of betweenness centrality is also crucial if the aim of the research is to understand the quickest way to disrupt a criminal network. In fact, systematically removing nodes in the order established by this measure of centrality represents the most effective strategy to reduce the largest connected component of the network [5]. The complexity of criminal interactions can also be approached by a multilayer study which can help for a better understanding of the key leaders [9]. Another important issue concerns the reliability and incompleteness of the available data. This problem can be investigated, for example, by analysing the robustness of link prediction methods applied to criminal networks [4].

The Sicilian Mafia represents a particularly compelling case study, in fact, the Sicilian Mafia originated in Sicily has spread worldwide [13], so the study of this kind of criminal organizations is far from selective and it might offer a practical value in the fight against organized crime.

3.2 Introduction

3.2.1 Dataset and previous studies

The starting point of this part of the thesis is the data from the datasets constructed from judicial documents associated with the 2007 Montagna anti-Mafia operation and analysed by Ficara, Fiumara and colleagues in their previously cited study: “Disrupting resilient criminal networks through data analysis: The case of Sicilian Mafia” [5], published in PLOS ONE. From this dataset two networks have been created. The first one, the phone calls network, was derived from police wiretapping records; the second one, the meetings network, was derived from police surveillance and stakeout data. In both cases, nodes represent individuals and links capture their interactions. Both weighted and unweighted versions of the networks were considered, with links weights representing the frequency of interactions. The datasets are fully anonymised and publicly available on GitHub repository (<https://github.com/lcucav/criminal-nets/tree/master/disruption>). In their study, the authors performed a network disruption analysis based on two intervention scenarios. In the first, individuals were removed sequentially, simulating arrests carried out one at a time. In the second, groups of individuals were removed simultaneously, simulating police raids. To determine which individuals should be targeted first, several centrality measures were compared: degree centrality, betweenness centrality, collective influence centrality and Katz centrality. The effectiveness of each strategy was evaluated by measuring the reduction in the size of the largest connected component (LCC) of the network, measured in terms of the number of vertices in the

component. The results indicate that betweenness centrality is particularly effective for identifying critical individuals. Although it requires more iterations to achieve a totally disconnected network, its curve exhibits the steepest initial drop. This rapid early decline is a critical factor, as the primary goal is to achieve substantial disruption in the shortest time possible. Similarly, degree centrality performs well, displaying comparable initial behaviour while reaching a totally disconnected network in fewer steps. This results, referring to weighted networks, are observed both when individuals are removed sequentially (Figure 3.1 A) and when groups of individuals are removed simultaneously (Figure 3.1 B)

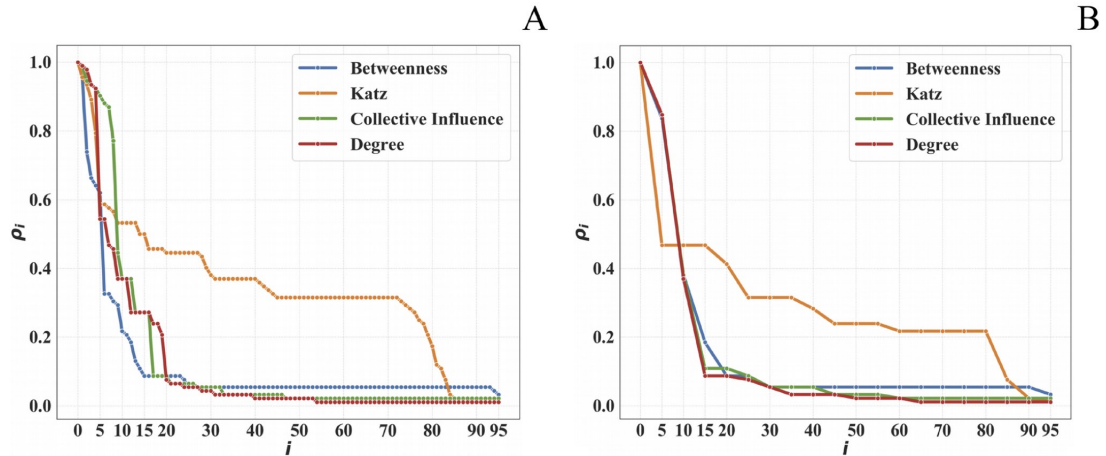


Figure 3.1: Images courtesy of [5].

In this case the connectivity measure used is

$$\rho_i = 1 - \left| \frac{LCC(G_i) - LCC(G_0)}{LCC(G_0)} \right|$$

where $LCC(G_i)$ is the size of the largest connected component of graph G_i obtained after the i -th iteration and G_0 the input graph. A further argument for selecting betweenness centrality as the optimal choice is that the first ten removals successfully target three bosses and this is the highest number achieved among all tested centrality measures. However, the authors themselves note that this result could be further improved by incorporating a mechanism to better account for human capital, namely the skills and resources possessed by specific individuals within the organization that play a crucial role in its structure. Another interesting finding is that the difference between weighted and unweighted graph analysis, in the majority of cases, do not pinpoint to major difference. This is due to the fact that weights are concentrated in very few individuals, with most of the others links having $w(e) = 1$.

3.2.2 Problem formulation

The aim of this part of the thesis is to reproduce the disruption analysis using new centrality measures introduced in the first chapter: the k -clique community centrality and the generalised k -clique community centrality.

I had the opportunity to present my approach to Dr. A. Ficara and Prof. G. Fiumara, two of the author of the paper [5], receiving their interest and appreciation.

The analysis will be developed just on the meetings network, not considering the weights, given the considerations made by the authors of the original paper, and it will proceed in two main steps. First, the network will be analysed with the community detection algorithm, to validate the choice of subdividing the nodes of the network obtained from the dataset into k -clique communities. This algorithm identifies the nodes of the network into communities, such as cliques, n -cliques and k -clique communities, and then it compares the resulting subdivision with the original partition into different Mafia families. Note that, from now on, all the atomic communities taken into consideration are to be considered maximal. The similarity between the resulting community structures and the original family-based partition will be evaluated using the similarity scores between set collections based on Jaccard similarity and the overlap coefficient, as shown in the previous chapter.

The network will then be analysed using the second algorithm, the disrupting graph algorithm. Here, different choices for the value of ρ_i have been made and they are all based on k -clique communities, accordingly to the results obtained with the first algorithm. As in the original paper, the disruption of the network will proceed by removing both one node and five nodes at the time. The resulting disruption process will be analysed considering the number of steps needed to obtain a network composed only by isolated nodes and the comparison of the decreasing curves produced by all the ρ_i . The meeting network is an undirected simple graph formed by 101 nodes and 256 edges. Most of the information from judicial documents were about the Mistretta and the Batanesi families, that form the main clans, but members of other district or close enough to the families were intercepted as well.

3.3 Community detection algorithm on a Mafia case

3.3.1 Research hypothesis

The objective of this part of the analysis is to investigate whether the original organization of the criminal network can be recovered from the structure of the graph. This constitutes a preliminary step of the analysis and, more specifically, an original development of this thesis, since the algorithm used for this purpose was not present in the paper that motivated the work [5]. In particular, the aim is to identify groups of individuals that may correspond to the original clans. However, this cannot be considered only as a standard graph partitioning problem. Although the identification of the clans requires the detection of cohesive groups, the network may also contain individuals who have strong relationships with members of different groups. These individuals can play an important role in connecting different parts of the network and, consequently, in maintaining its overall connectivity.

This aspect is particularly relevant in the analysis of criminal networks. Previous studies have highlighted the importance of individuals acting as bridges between otherwise separate groups. In the article considered here, particular attention is given to nodes that connect different regions of the network and to the role of brokers in linking distinct criminal groups. Such individuals can occupy strategic positions because their relationships connect different parts of the organization. Therefore, identifying only disjoint groups could provide an incomplete representation of the network structure and could hide some of the relationships between different clans.

For this reason, the considered structures are: cliques, k -clique communities, and

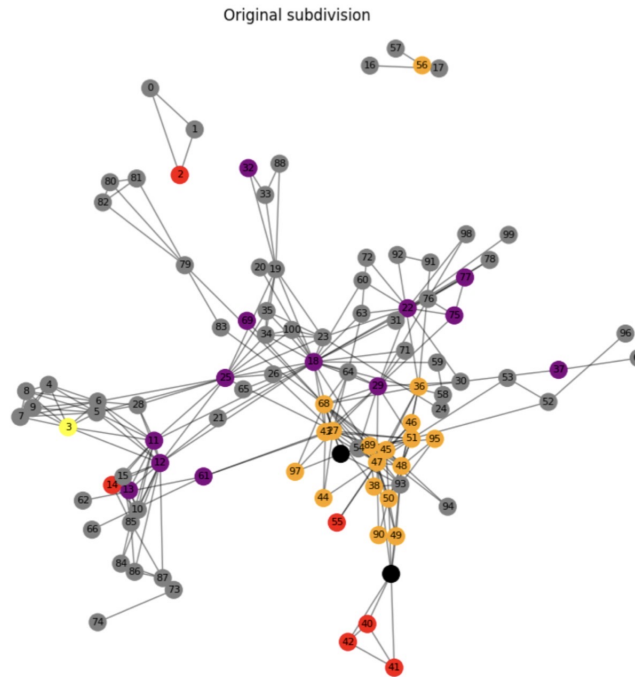
n -clique. Also in this situation, cliques and n -cliques are considered maximal. They are suitable for the present purpose since they do not necessarily form partitions of the nodes of the network, but the same individual may belong to more than one community. Remember that a clique represents a maximally connected group in which every pair of vertices is directly connected. A k -clique community extends this idea by combining cliques that overlap sufficiently, thus allowing larger cohesive groups to be identified. An n -clan, instead, identifies a group in which the members remain within a specified distance from one another. So, even though they are similar structures, they all highlight slightly different characteristics that may be crucial for the subdivision into original clans. For the formal definitions refer to Section 1.2.1.

Consequently, the analysis is not limited to determining which individuals belong to which clan, it also aims to understand how the detected groups are connected and which individuals may link different groups.

3.3.2 Results

In this representation of the the meeting network, the colours refer to the following groups of people:

- A_1 purple: Mistretta family;
- A_2 orange: Batanesi family;
- A_3 red: other Mafia family;
- A_4 yellow: other Mafia family;
- A_5 grey: close to the family, but not classifiable in any of the previous categories.



So the comparison made by the community detection algorithm between the input $\mathcal{A} = \{A_1, A_2, A_3, A_4, A_5\}$ and the various subdivisions $\mathcal{B}$ returned some similarity scores. The results have been normalised and collected in the table:

$\mathcal{B}$ Score \	cliques	2-cliques	3-cliques	4-cliques	5-cliques	6-cliques	7-cliques	8-cliques
$Score_J$	7%	13%	23%	36%	43%	53%	58%	58%
$Score_0$	12%	19%	34%	56%	69%	80%	78%	78%

Table 3.1

The results obtained show that the low value of n for n -cliques do not reproduce the original subdivision of the network particularly well. In particular, the results for the cliques and for $n = 2$ and $n = 3$ provide a poor correspondence with the five original groups. More interesting values of the similarity scores start to appear for $n \geq 4$. However, these numerical results should be interpreted together with the definition of the n -cliques.

In fact, a superficial analysis could lead to the conclusion that the results obtained for $n = 6$ are satisfactory, but this conclusion would be misleading. Nevertheless, the structure of the corresponding 6-cliques shows that the original subdivision is not actually recovered. The algorithm identifies six 6-cliques, but four of them are very small and correspond to the sets $\{0, 1, 2\}$, $\{73, 74\}$, $\{16, 17\}$, and $\{56, 57\}$, but the remaining two 6-cliques are extremely large: the second contains 91 vertices and the third contains 90 vertices. Moreover, these two sets overlap almost completely.

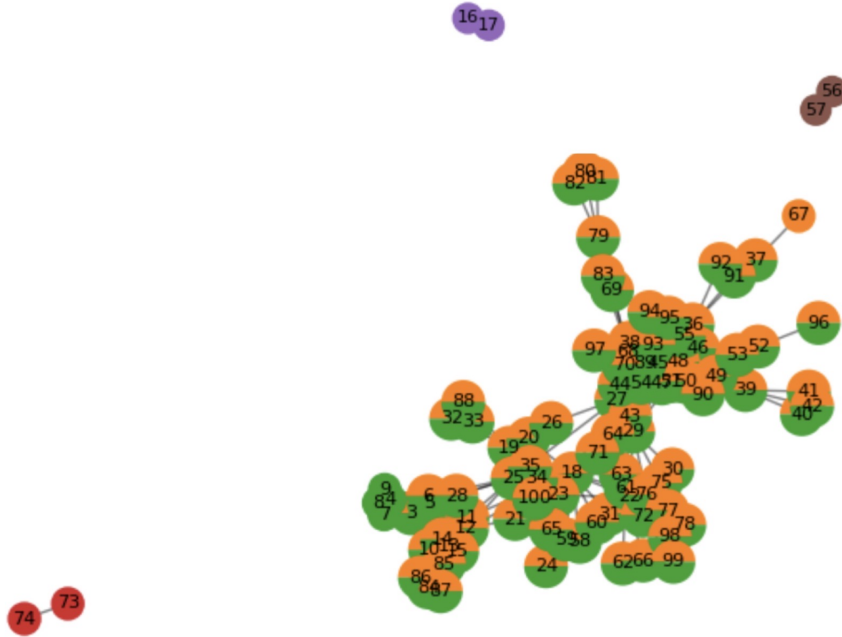


Figure 3.2

This observation is particularly important for the interpretation of the resulting network because it is obvious that 6-clique subdivision does not provide a meaningful separation of the original groups. The graphical representation makes this phenomenon immediately visible: most of the central part of the network is covered by the two large and strongly overlapping 6-cliques, resulting in a dense central region with multiple colours.

This behaviour suggests that, for $n = 6$, the parameter is too large to preserve the distinction between the original groups. In fact, increasing n makes the notion of this

atomic community too wide, so that vertices belonging to different original clans can be included in the same n -clique. Consequently, a high similarity score in this case does not necessarily indicate a good reconstruction of the original subdivision, but it may stand for the ability of a large n -clique to contain vertices from several different groups simultaneously.

The results for larger values of n reinforce this interpretation. For $n = 7$, only five n -cliques are identified, and one of them already contains almost the entire central part of the network. For $n \geq 8$, exactly the same five n -cliques are obtained as for $n = 7$, and therefore this indicates that, beyond a certain value of n , increasing the parameter no longer produces a new subdivision of the network.

Overall, the results suggest that the most interesting range is around $n = 4$ and $n = 5$, particularly when the overlap coefficient is considered. However, even these values do not provide a fully satisfactory reconstruction of the original clan subdivision.

This analysis suggests that it is reasonable to explore other types of communities. As an example, it can be considered what happens when k -clique communities are identified.

$\mathcal{B}$ Score	2-clique communities	3-clique communities	4-clique communities	5-clique communities	6-clique communities	7-clique communities
$Score_J$	57%	42%	50%	46%	58%	23%
$Score_O$	78%	60%	72%	64%	83%	33%

Table 3.2

Also in this case, the results need to be interpreted carefully. From the table, it is immediately clear that the 7-clique communities cannot provide a good approximation of the original subdivision. However, the values of $score_O$ are generally quite high, and this should not be misleading. It is important to recall the particular behaviour of the overlap coefficient discussed in Example 1.76, where it is shown that if one set is entirely contained in the other, their similarity is equal to 1. This characteristic suggests that particularly high values of $score_O$ may not always indicate a genuinely good correspondence between the two subdivisions. Moreover, just thinking about the definition and the results obtained before, one can also discharge 2-cliques communities because it allows for too big communities. One could also ask why the original number of clans is not taken into account when comparing it with the number of communities obtained. This choice is mainly related to the structural properties of the graph. In fact, the graph contains some small groups of vertices that are almost completely separated from the rest of the network. For sufficiently small values of k , these groups are likely to form independent k -clique communities. Therefore, some of the communities identified by the algorithm may correspond only to these small groups and do not provide meaningful information about the original subdivision into clans.

For example, if two or three of the detected communities are formed by these isolated groups, only the remaining k -clique communities are available to represent the rest of the network. Consequently, even if the original number of clans is relatively close to the total number of communities detected, the effective number of communities available for comparison with the clans may be considerably smaller. This makes a direct comparison based only on the number of communities potentially misleading. This is precisely what happens, for example, for $k = 2$ and $k = 3$, where some of the communities are

associated with these small groups of vertices, leaving too few communities to adequately reproduce the original subdivision into clans. These considerations also suggest that the original clans are more likely to be represented by the union of different k -clique communities, particularly for larger values of k , rather than by single k -clique communities. In other words, a clan may contain several highly connected subgroups that are identified separately by the k -clique community detection algorithm.

Therefore, k -clique communities $k = 4, 5, 6$ remain as plausible candidates. In particular, the 4-clique communities can reasonably be considered the ones that provide the best approximation of the original subdivision into clans. This hypothesis is further confirmed by the visual comparison of the graphs corresponding to each subdivision.

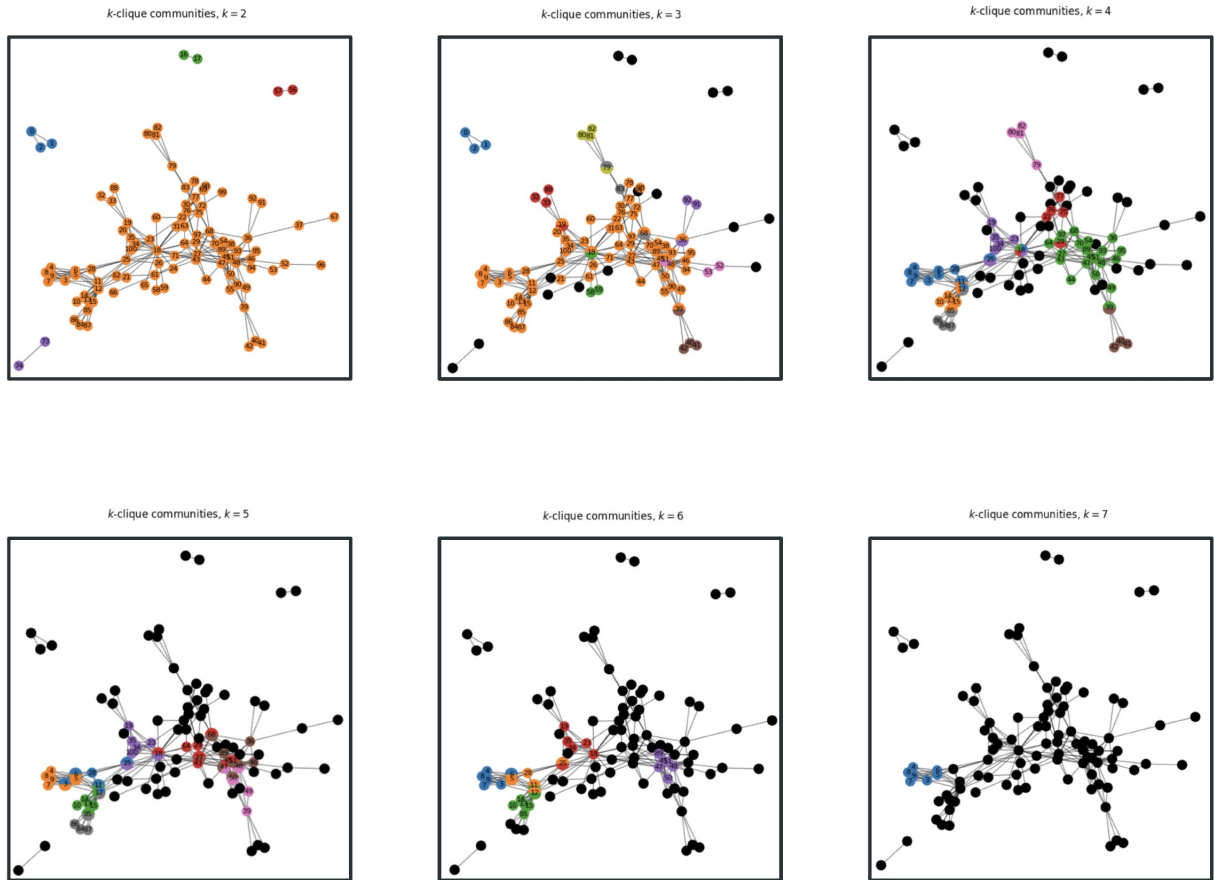
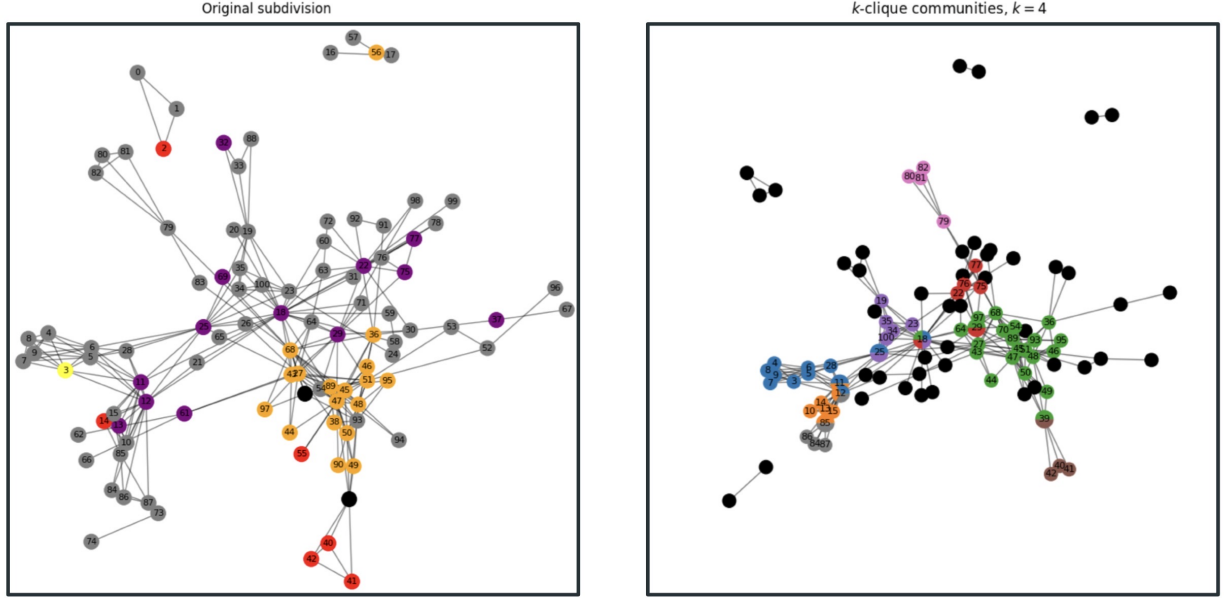


Figure 3.3

Another particularly interesting aspect is to investigate the nodes that belong to more than one 4-clique community. Identifying these nodes could make it possible to understand who they might represent in the original network and whether their position could provide additional information about the structure of the criminal organization. In fact, if the 4-clique subdivision were to provide the best approximation of the original clans, the nodes shared by several communities could play a particularly important role, as they might connect different highly cohesive groups.



The nodes that emerge as particularly interesting from the subdivision into 4-clique communities are $V_1 = \{11, 12, 18, 25, 29, 39, 85\}$. The first important observation is that, according to the available data, three of these nodes, namely $V_2 = \{11, 25, 18\}$, are known to be Mafia bosses. This already suggests that belonging to multiple 4-clique communities may be related to a particularly relevant position within the network.

Moreover, node 29 deserves particular attention. When considering the original subdivision into clans, this node is precisely the individual who maintains most of the contacts between the two main clans investigated in the case study. Therefore, its presence among the nodes belonging to multiple 4-clique communities is particularly meaningful.

In conclusion, this analysis proved that the subdivision of the nodes of the network into 4-clique communities, beyond reproducing the original clan structure, also reveals additional structural information about single nodes and the relationships between different clans. Furthermore, this result could prove useful in defining connectivity and centrality measures to be used in the graph disruption algorithm.

3.4 Disrupting graph algorithm on a Mafia case

3.4.1 Research hypothesis

The aim of this part of the analysis is to reproduce, as far as possible, the approach proposed by Cavallaro et al. and to investigate the resilience of the Mafia network to attacks, where an attack is modelled as the progressive removal of nodes from the network. In particular, the objective is to analyse how the structure of the network changes as increasingly important nodes are removed, and to determine which centrality measures are most effective in identifying nodes whose removal produces a significant disruption of the network.

However, the assumptions for the present analysis are different from those adopted in the original approach. Connected components are certainly considered an important aspect of the network structure, since they provide information about the global connectivity of the graph. Nevertheless, the community detection analysis presented in the

previous part of this work showed that the k -clique communities obtained for $k = 4$ provide a reasonably good approximation of the initial subdivision of the nodes into Mafia clans. Therefore, it was decided that these communities should remain central also in the definition of the quantities used in this second algorithm.

The first choice concerns the centrality measures used to determine the order in which nodes are removed. As in the previous analysis, four different centrality measures were selected in order to allow a comparison: degree centrality, betweenness centrality, k -clique community centrality, and generalized k -clique community centrality.

The last two measures do not correspond exactly to the definitions introduced in Definition 1.65 and Definition 1.66. Nevertheless, the same terminology will be used from this point onwards, since they are particular instances of those definitions obtained by fixing the value of k , as suggested in Remark 1.68. The two centralities are therefore defined as

- $c_4(v) = \max_{C \in \mathcal{C}_4(v)} |C|$
- $\bar{c}_4(v) = \left| \bigcup_{C \in \mathcal{C}_4(v)} C \right|$

where $\mathcal{C}_4(v)$ denotes the collection of all 4-clique communities containing the node v . As already suggested in the example shown in Section 2.2.2, also in this situation if two nodes have the same centrality, first the algorithm checks the centralities for lower values of k . If this comparison still produces a tie, the algorithm checks the degrees of the nodes in question.

The community detection analysis also provided a basis for defining the connectivity measures used to evaluate the effect of node removal. This choice requires particular attention, since a connectivity measure determines which structural features of the network are considered most relevant when assessing its robustness. In other words, different connectivity measures may lead to different conclusions about whether a network has been significantly disrupted.

In the present analysis, greater importance is assigned to the k -clique community structure than to the size of the largest connected component. The reason for this choice is that from the perspective of the community detection algorithm, k -clique communities are central, therefore the disruption of this cohesive subgroup may represent a more relevant structural change than the subsequent fragmentation of the remaining chain. This means that, if the following graph is given:

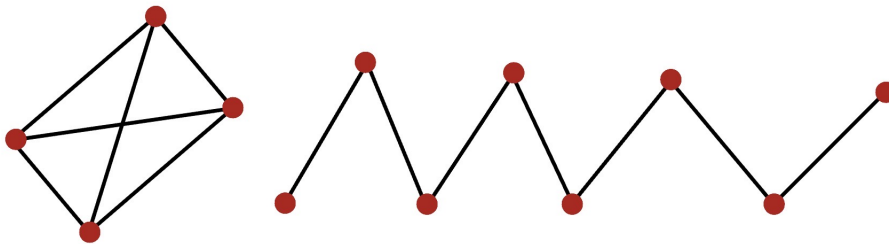


Figure 3.4

This second algorithm will start by disrupting the 4-clique community and, by then, the connectivity index would have dropped consistently, even though there is still a quite big largest connected component alive formed by the sequence of vertices. On the other hand, if one wants to consider the connectivity index, then the curve representing it will drop only when the big connected component is disrupted.

Accordingly, the following connectivity measures were considered for the analysis:

- $\rho_i = \sum_{k=2}^{\infty} k \sum_{C \in \mathcal{C}_k(G_i)} |C|.$
- $\rho_i = \sum_{k=2}^{\infty} k \cdot \max\{|C| \text{ such that } C \in \mathcal{C}_k(G_i)\}.$
- $\rho_i = \begin{cases} \max\{k \geq 2 \mid \mathcal{C}_k(G_i) \neq \emptyset\} & \text{if } \exists k \geq 2 \text{ such that } \mathcal{C}_k(G_i) \neq \emptyset, \\ 0 & \text{otherwise.} \end{cases}$
- $\rho_{i,k} = \max_{C \in \mathcal{C}_k(G_i)} |C|, \quad \forall k \geq 2 \text{ such that } \mathcal{C}_k(G_i) \neq \emptyset$

where ρ_i refers to the connectivity index calculated on the G_i obtained after the i -th iteration and $\mathcal{C}_k(G_i)$ is the set of all k -clique communities in the graph G_i .

The precise definition of these measures is therefore based on the structural features that are considered relevant for the robustness analysis.

It should be emphasised that this choice is not intended to imply that these connectivity measures are necessarily the only possible ones, nor that they are universally better than measures based on connected components. Rather, the choice reflects the specific objectives of the present analysis and the structural information obtained from the previous community detection algorithm.

3.4.2 Results

In this section, for readability reasons, only some of the results will be discussed in detail. Readers interested in the complete set of results can find all the other plots in Appendix C

Figure 3.5 and 3.6 report the evolution of the structural measure ρ_i during the sequential and the block removal of nodes, respectively, when the nodes are selected according to the four different centrality measures: degree, betweenness, c_4 and $\bar{c}_4$. The measure ρ_i is defined in terms of the largest k -clique community of the graph obtained after the i -th removal, namely

$$\rho_i = \sum_{k=2}^{\infty} k \cdot \max\{|C| \text{ such that } C \in \mathcal{C}_k(G_i)\}.$$

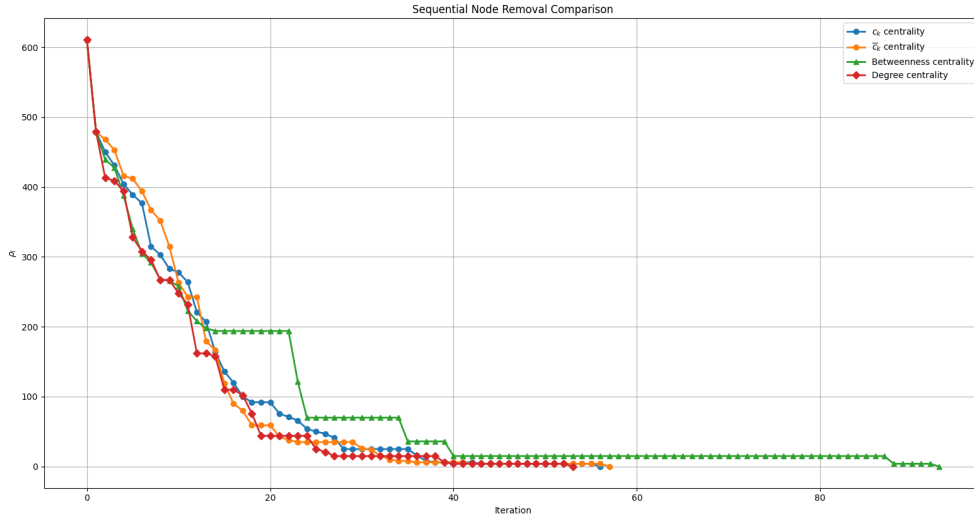


Figure 3.5

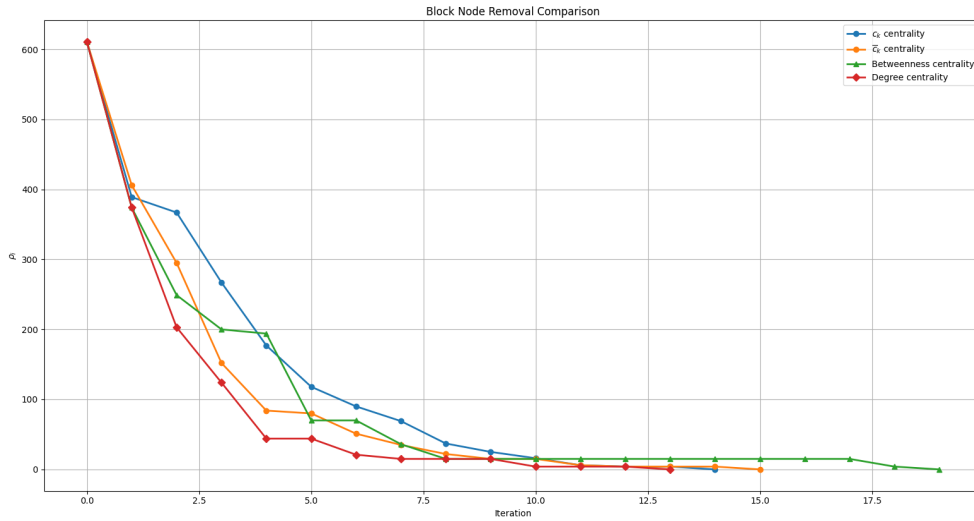


Figure 3.6

The first characteristic that stands out in this graph is the number of iterations after which the curves reach zero, that is, after how many iterations no edges remain in the graph. It can be immediately noticed that the node-by-node disintegration of the graph based on betweenness centrality requires 93 iterations, which is much more than the other measures, which instead require 57 iterations for both c_4 and $\bar{c}_4$, and 53 iterations for degree centrality.

The number of iterations necessary to obtain a totally disconnected graph naturally decreases when the nodes are removed 5 at a time, but the ranking of the centralities remains the same. However, in this case, the difference between the number of iterations required by betweenness centrality and those required by the other centralities is substantially smaller.

In Figure 3.5, the four curves all start at the same level because ρ_0 , that is, the level of connectivity of the initial graph, is the same for all the centralities. After that, however, degree centrality is the one that decreases least rapidly. Although ρ_i initially decreases, the curve subsequently exhibits several extended plateaus. This behaviour indicates that the removal of nodes with high betweenness centrality does not continuously reduce the size of the largest 4-clique community. Instead, several consecutive removals may leave the largest community substantially unchanged.

Degree centrality, c_4 and $\bar{c}_4$ centralities produce the steepest decline, driving ρ_i toward zero within a relatively small number of iterations. This initial steep decrease suggests that these centralities are effective at disrupting the cohesive groups identified by the 4-clique communities. Looking more closely, it can be noticed that the three curves decrease rapidly during the first part of the simulation, with several marked drops. Subsequently, the decrease becomes progressively slower. This indicates that also high-degree nodes are often involved in the densely connected portion of the network, so their removal can initially affect the largest 4-clique community.

As for the graph in Figure 3.6, the block-removal strategy reinforces the observations obtained from sequential removal. In addition, in this case, some further differences can be observed between the three curves that perform better: the two clique-based metrics, c_4 and $\bar{c}_4$, follow a very similar trajectory to one another, decreasing slightly more gradually than degree centrality. Nevertheless, all three centralities are particularly effective when the objective is to disrupt the densely interconnected structures represented by 4-clique communities.

Another very important indicator to keep in mind when analysing these centralities is the number of bosses removed during the first iterations. For example, if the first 10 iterations of the sequential node removal are considered, the number of bosses from the Mistretta and Batanesi families removed by each centrality is the following:

- c_4 centrality: 4
- $\bar{c}_4$ centrality: 5
- betweenness centrality: 4
- degree centrality: 5

Considering that the total number of bosses available for the two families is 6, these results can be considered satisfactory. Moreover, among the other nodes removed during these iterations are, for example, nodes 29, 39, 12, which are also the same nodes identified in Figure 3.4 as important because they belong to multiple 4-clique communities simultaneously.

Another case in which these characteristics are clearly highlighted is when using:

$$\rho_{i,k} = \max_{C \in \mathcal{C}_k(G_i)} |C|, \quad \text{for } k = 4.$$

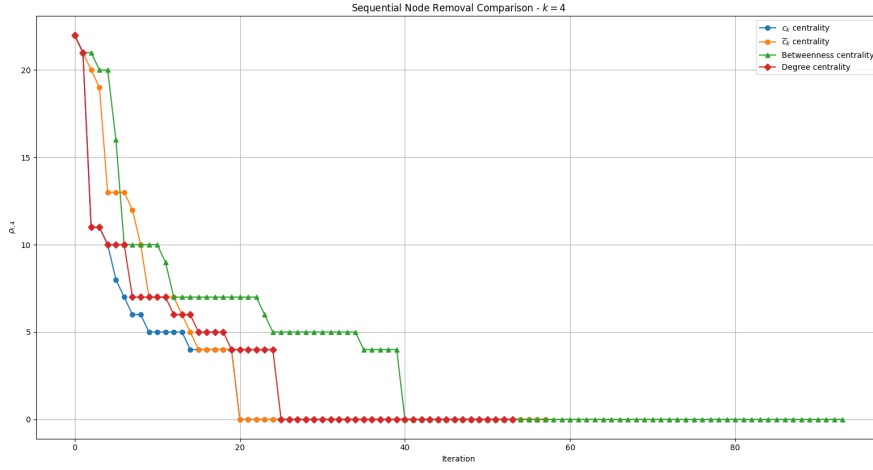


Figure 3.7

This is the measure that appears most natural to use after observing the results of the community detection algorithm. In fact, this measure places particular emphasis on the disruption of the 4-clique communities, which are the structures that best represent the original clans. In this case, degree centrality and c_4 centrality show the best performance and, in fact, they initially follow exactly the same behaviour. However, after the first iterations, c_4 centrality appears to reach lower values of the measure in fewer iterations, suggesting that it may be slightly more effective in disrupting the 4-clique community structure. Although, few iterations before reaching the value zero, its curve completely overlaps with the curve of the $\bar{c}_4$ centrality.

Here, the parameter $\rho_{i,4}$ reaches zero after only a few iterations, while more iterations are required to obtain a completely disconnected graph. This means that, after only a few node removals, all the 4-clique communities have already been disrupted, even though some edges and, consequently, other less cohesive structures are still present in the network.

The characteristics described above can also be observed in the block removal strategy:

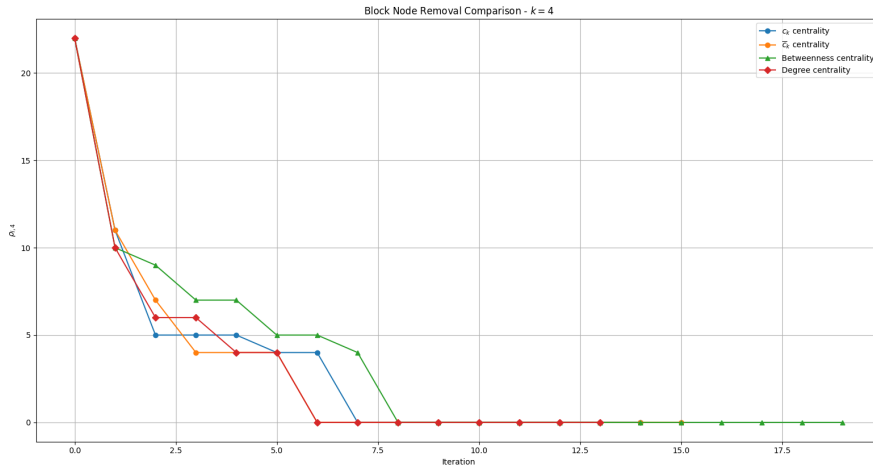


Figure 3.8

Another situation that has been taken into consideration is when

$$\rho_i = \max \{k \geq 2 \mid \mathcal{C}_k(G_i) \neq \emptyset\},$$

so when it simply reports the highest order k for which at least one k -clique community still exists in the graph after the i -th removal.

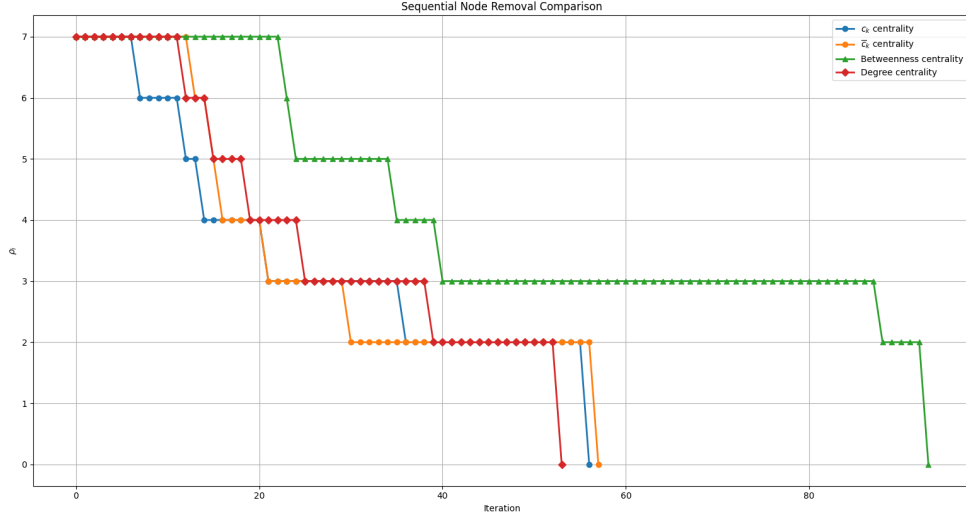


Figure 3.9

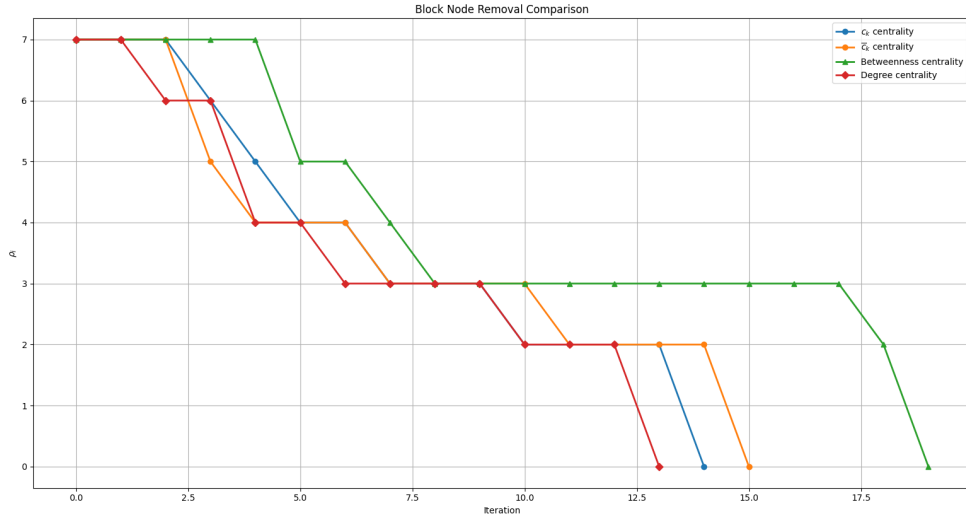


Figure 3.10

As a consequence, ρ_i is now an integer-valued, strictly decreasing step function in the sequential node removal strategy, as shown in Figure 3.9.

This change in definition has a direct effect on the shape of the curves: rather than showing a continuous decline, each centrality now produces a staircase pattern, with long horizontal plateaus.

Looking at the relative behaviour of the four metrics in the sequential removal strategy, Betweenness centrality clearly maintains the highest value of ρ_i for the longest stretch of iterations and only reaches 0 around iteration 93. So, also in this situation it is the one most ineffective at eliminating the k -clique communities.

Degree centrality, c_4 and $\bar{c}_4$ centralities show a similar overall trend, all bringing ρ_i down considerably faster than betweenness centrality, and eventually reaching zero around iteration 53 for the degree centrality and around iteration 57 for the $\bar{c}_4$ centrality (note that towards the end the two curves overlap completely and the one representing c_4 disappears behind the one representing $\bar{c}_4$). But, even though degree centrality is few iterations faster than the other two, c_4 and $\bar{c}_4$ decrease faster for the most of the iterations, making them more interesting for the scope of this analysis.

Taken together, all the results shown here indicate that degree-based and clique-based centralities are more effective than betweenness centrality at disrupting the k -clique community structure of the network, regardless of whether nodes are removed sequentially or in blocks, or of which formulation of ρ_i is adopted. Betweenness centrality, by contrast, preserves higher-order cliques for a considerably longer time, suggesting that this centrality is better suited to disrupting global connectivity than k -clique communities.

Conclusions

This thesis started from a real problem concerning the analysis of organised crime, in particular a Mafia scenario, and progressively developed into a more general investigation in graph theory. The initial motivation came from the study of criminal networks and, in particular, from the work of Cavallaro et al. [5] on the disruption of Sicilian Mafia networks. Rather than simply reproducing the analysis carried out in that work, I decided to take a step backwards and investigate the mathematical structures underlying the problem. The original curiosity about a specific criminal case was progressively reformulated into a set of general questions in graph theory: which structures best describe the points of aggregation of a graph, how can the robustness of a graph be measured, and which vertices are the ones whose removal disrupts the graph fastest. Answering these questions required building a new mathematical framework.

Chapter 1 was devoted to laying the theoretical foundations on which the rest of the thesis is built. Besides recalling the classical notions of graph theory and the well-established atomic communities and centrality measures, such as degree centrality, betweenness centrality, Katz centrality and collective influence, this chapter introduced two lines of original contribution. The first is a family of centrality measures, namely the k -clique community centrality $c(v)$ and its generalised version $\bar{c}(v)$ (Definitions 1.65 and 1.66), together with the auxiliary measure $c_{\mathcal{C}}(v)$ based on a subdivision into atomic communities (Definition 1.70). Unlike degree or betweenness centrality, which look at a vertex in isolation or in relation to shortest paths, these measures value the role played by a vertex within the cohesive structure of the graph, and they proved, in Chapter 3, to be interesting parameters for the analysis. The second contribution is the framework introduced in Section 1.3.2 for comparing two finite collections of sets. Starting from classical similarity measures between finite sets, such as the Jaccard similarity and the overlap coefficient recalled in Section 1.3.1, I extended their use from the comparison of two individual sets to the comparison of collections of finite sets (Definition 1.84). The proposed framework considers possible correspondences between the sets belonging to two subdivisions and assigns a score to each correspondence according to the similarity between the matched sets. This makes it possible to compare a subdivision obtained from a theoretical method with a given subdivision.

The two algorithms of Chapter 2, at first sight, might just appear as the implementation of the tools from Chapter 1: the community detection algorithm checks which known graph structure best reproduces a given subdivision of the vertices, using the similarity score of Definition 1.84; the disrupting graph algorithm removes vertices, following a chosen centrality measure, and keeps track of a parameter ρ_i that measures how much the graph has broken down. But what I consider the real contribution here is not the computation itself, nor the specific numbers it produces, but the idea behind building

these two algorithms in the first place. Instead of choosing from the outside what should count as "robust" or "important" in a graph, the two algorithms start from the idea that a graph already carries, inside its own structure, the information needed to answer these questions. If a graph has a natural way of grouping its vertices into communities, then this same grouping should be used to decide which vertices matter most and which removals hurt the graph the most. This is why the two algorithms were built to work together. The first one is used to find out whether a graph has a community structure that fits it particularly well. The second one takes this finding and uses it to build new centrality measures and a new notion of robustness that are specific to that structure.

Chapter 3 brought this framework back to the real case that had started the whole thesis: the meetings network built by Cavallaro et al. from the judicial documents of the 2007 Montagna anti-Mafia operation [5]. Several structures such as cliques, n -cliques and k -clique communities were considered for the community detection algorithm. The comparison of these new communities with the original subdivision showed that numerical similarity scores alone are not sufficient to identify the most meaningful representation or, at least the chosen similarities are not enough. Some high-order n -cliques, for example, produced very large groups containing most of the central part of the network. Although these structures obtained relatively high similarity scores, they did not provide a satisfactory reconstruction of the original clans. This observation motivated a closer investigation of k -clique communities, whose overlapping nature makes it possible to describe groups while also preserving information about vertices shared by different communities. The analysis showed that 4-clique communities are the ones that better represent the original subdivision into Mafia families. Moreover, the vertices that belong to more than one 4-clique community turned out to include three known Mafia bosses and the individual who, in the original data, keeps most of the contacts between the two main clans.

Building on this, the disrupting graph algorithm was then run on the same network, using both classical centrality measures and the new k -clique community centralities, together with several versions of ρ_i built around the 4-clique community structure rather than around the size of the largest connected component. The results in Section 3.4.2 show a consistent pattern, whether nodes are removed one at a time or in blocks of five: degree centrality and the two clique-based centralities, c_4 and $\bar{c}_4$, break down the 4-clique community structure much faster than betweenness centrality. This suggests that betweenness centrality remains a strong choice when the goal is to disrupt the overall connectivity of a network, as already found by Cavallaro et al., but this thesis proposes a different perspective, that is to aim at breaking apart the cohesive groups that, in this case, correspond closely to the criminal clans themselves. It is also worth remembering that, in the first ten removals of the sequential strategy, both $\bar{c}_4$ and degree centrality managed to remove five out of the six known bosses of the two main families.

An important part of this work was also the opportunity to discuss the proposed approach with Dr. A. Ficara and Prof. G. Fiumara, two of the authors of the paper that originally motivated this thesis. During these discussions, I presented the mathematical framework developed in this work and the intention to apply it to the original networks. Their interest and appreciation of the proposed direction and of the preliminary results were particularly encouraging. This feedback does not constitute a formal validation of the mathematical results, but it provided an important opportunity to discuss the approach with researchers directly connected to the original study and confirmed

the potential interest of developing the problem from a different mathematical perspective.

This framework has some clear limits and they all point to natural directions for future work. The disrupting graph algorithm was only tested on one real network. Applying it to the calls network from the same case study, or to networks from other investigations, would help to understand how much of what was observed here is specific to these Mafia clans and how much is a more general property. Also, it could be interesting to extend this framework beyond the analysis of criminal networks, for instance to biological structures, where the detection of similar cohesive substructures and their robustness might prove equally enlightening.

Another limitation concerns the fact that edge weights were not taken into account in this analysis. As already observed by Cavallaro et al. [5], weighted and unweighted versions of the meetings network gave, in the majority of cases, quite similar results, since the weights turn out to be concentrated on very few edges. For this reason, in this thesis I chose to work only with the unweighted version of the network. This does not mean, however, that edge weights are irrelevant in general.

A related but distinct direction concerns the vertices rather than the edges. All the centrality measures considered in this thesis treat every vertex as equivalent only taking into account its position in the graph. Yet, as already remarked by Cavallaro et al. [5], the effectiveness of a disruption strategy could be improved by incorporating some notion of human capital, that is, the specific skills, resources or role that an individual brings to the organisation. A natural way to include this aspect within the present framework would be to introduce a vertex-weighted graph, as recalled in Definition 1.28, and to let the weight $w_v(v)$ represent an estimate of this human capital.

Appendix A

Community Detection Algorithm: Python Implementation

This section provides the Python implementation for the community detection algorithm.

```
import csv
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np

from scipy.optimize import linear_sum_assignment
from networkx.algorithms.community import k_clique_communities

# =====
# 1. READ THE GRAPH
# =====

def leggi_grafo_da_csv(percorso_file):

    #Reads a weighted undirected graph from a tab-separated file.
    #Each row must contain: node_1 node_2 weight

    G = nx.Graph()

    with open(percorso_file, 'r', newline='') as file:
        reader = csv.reader(file, delimiter='\\t')

        for riga in reader:
            # if len(riga) != 3:
            # continue
            # print(riga)

            nodo1 = int(riga[0])
            nodo2 = int(riga[1])
            peso = float(riga[2])

            if nodo1 == nodo2:
                G.add_node(nodo1)
            else:
                G.add_edge(nodo1, nodo2, weight=peso)

    return G

# =====
# 2. ATOMIC COMMUNITIES AND COMMUNITIES
# =====

# -----
# K-CLIQUE COMMUNITIES
# -----

def calcola_k_clique_communities(G, k):
    # Calculates the k-clique communities of G.
    # Returns a list of sets, where each set contains the vertices belonging to one k-clique community.
```

```

    communities = list(k_clique_communities(G, k))

    return [set(community) for community in communities]

def calcola_tutte_le_communities(G):
    """
    Calculates all k-clique communities for k >= 2.

    Returns:
        dictionary {k: list of communities}
    """

    # Find the maximum clique size
    maximal_cliques = list(nx.find_cliques(G))

    if not maximal_cliques:
        return {}

    max_k = max(len(clique) for clique in maximal_cliques)

    tutte_le_communities = {}

    for k in range(2, max_k + 1):

        communities = calcola_k_clique_communities(G, k)

        if communities:
            tutte_le_communities[k] = communities

    return tutte_le_communities

# -----
# CLIQUES
# -----

def find_cliques(G):
    """
    Finds the maximal cliques of G.

    Returns:
        list of maximal cliques.
    """

    # Find all maximal cliques of the graph
    cliques = list(nx.find_cliques(G))

    return cliques

# -----
# N-CLIQUES
# -----

def find_n_cliques(G, n):
    """
    Finds the n-cliques of G.

    Returns:
        list of n-cliques.
    """

    # Calculate the n-th power of the graph.
    # This connects all nodes whose distance is <= n.
    Gn = nx.power(G, n)

    # Find the maximal cliques in the power graph.
    # These correspond to the n-cliques of the original graph.
    n_cliques = list(nx.find_cliques(Gn))

    return n_cliques

# =====
# 3. COLOUR THE GRAPH
# =====

def disegna_communities(G, communities, k):

```

```

#Draws the graph colouring each vertex according to the k-clique communities to which it belongs.
#A vertex belonging to several communities is divided into several coloured sectors.
#Vertices belonging to no community are coloured black.

```

```

pos = nx.spring_layout(G, k=0.2, seed=42)

# -----
# Assign a colour to each community
# -----

cmap = plt.cm.get_cmap("tab10")

community_colours = {
    i: cmap(i % 10)
    for i in range(len(communities))
}

# -----
# Create node -> communities dictionary
# -----

node_communities = {
    node: []
    for node in G.nodes()
}

for i, community in enumerate(communities):
    for node in community:
        node_communities[node].append(i)

# -----
# Draw edges
# -----

plt.figure(figsize=(10, 8))

nx.draw_networkx_edges(
    G,
    pos,
    alpha=0.5
)

# -----
# Draw nodes
# -----

for node, (x, y) in pos.items():

    belonging = node_communities[node]

    # Vertex does not belong to any community
    if not belonging:

        circle = plt.Circle(
            (x, y),
            0.025,
            color="black"
        )

        plt.gca().add_patch(circle)

    else:

        n = len(belonging)

        # One colour if the vertex belongs to one community
        if n == 1:

            circle = plt.Circle(
                (x, y),
                0.025,
                color=community_colours[belonging[0]]
            )

            plt.gca().add_patch(circle)

        # Several colours if the vertex belongs
        # to several communities
        else:

            colours = [
                community_colours[i]

```

```

        for i in belonging
    ]

    sizes = [1] * n

    plt.pie(
        sizes,
        colors=colours,
        center=(x, y),
        radius=0.035
    )

# -----
# Labels
# -----

nx.draw_networkx_labels(
    G,
    pos,
    font_size=8
)

plt.title(f"$k$-clique communities, $k={k}$")

plt.axis("equal")
plt.axis("off")

plt.show()

# =====
# 3b. DRAW ORIGINAL SUBDIVISION
# =====

def disegna_suddivisione_A(G, A):
    """
    Draws the graph colouring each vertex according to
    the subdivision A defined in the main program.
    """

    pos = nx.spring_layout(G, k=0.3, seed=42)

    # -----
    # Assign a colour to each set of the subdivision
    # -----

    colori = {
        0: "orange",
        1: "purple",
        2: "red",
        3: "grey",
        4: "yellow"
    }

    # -----
    # Create node -> set dictionary
    # -----

    node_set = {
        node: None
        for node in G.nodes()
    }

    for i, insieme in enumerate(A):
        for node in insieme:
            node_set[node] = i

    # -----
    # Draw graph
    # -----

    plt.figure(figsize=(10, 8))

    nx.draw_networkx_edges(
        G,
        pos,
        alpha=0.5
    )

    # -----
    # Draw nodes

```

```

# -----
for node, (x, y) in pos.items():
    if node_set[node] is None:
        circle = plt.Circle(
            (x, y),
            0.025,
            color="black"
        )
    else:
        circle = plt.Circle(
            (x, y),
            0.025,
            color=colori[node_set[node]]
        )

    plt.gca().add_patch(circle)

# -----
# Labels
# -----

nx.draw_networkx_labels(
    G,
    pos,
    font_size=8
)

plt.title("Original subdivision")

plt.axis("off")
plt.gca().set_aspect("equal", adjustable="datalim")

plt.show()

# =====
# 4. SIMILARITY MEASURES
# =====

def jaccard_similarity(A, B):
    A = set(A)
    B = set(B)

    if len(A) == 0 and len(B) == 0:
        return 1.0

    return len(A & B) / len(A | B)

def overlap_coefficient(A, B):
    A = set(A)
    B = set(B)

    if len(A) == 0 and len(B) == 0:
        return 1.0

    if len(A) == 0 or len(B) == 0:
        return 0.0

    return len(A & B) / min(len(A), len(B))

# =====
# 5. MATCHING SIMILARITY SCORE
# =====

def matching_similarity_score(A, B, similarity_function):
    A = [set(X) for X in A]
    B = [set(X) for X in B]

    m = len(A)
    n = len(B)

# -----

```

```

# Extend the two collections with empty sets
# -----

A_extended = A + [set() for _ in range(n)]
B_extended = B + [set() for _ in range(m)]

N = m + n

# -----
# Build similarity matrix
# -----

similarity_matrix = np.zeros((N, N))

for i in range(N):
    for j in range(N):
        similarity_matrix[i, j] = similarity_function(
            A_extended[i],
            B_extended[j]
        )

# -----
# Find the bijection maximizing the total similarity
# -----

rows, columns = linear_sum_assignment(
    similarity_matrix,
    maximize=True
)

# -----
# Compute the matching similarity score
# -----

score = similarity_matrix[rows, columns].sum()

return score

# =====
# 6. SCORES(A,B)
# =====

def scores(A, B, similarity_function):
    return matching_similarity_score(
        A,
        B,
        similarity_function
    )

# =====
# 7. MAIN
# =====

if __name__ == "__main__":
    # -----
    # Input graph
    # -----

    grafo = leggi_grafo_da_csv(
        "Montagna_meetings_edgelist.csv"
    )

    # -----
    # Original subdivision A
    # -----

    arancioni = [
        46, 44, 51, 90, 43, 50, 47, 45,
        89, 95, 27, 49, 48, 68, 38, 97,
        36, 56
    ]

    viola = [
        77, 75, 22, 61, 29, 69, 37,
        32, 18, 12, 25, 11, 13
    ]

    rossi = [

```

```

    2, 55, 14, 41, 40, 42
]

bianchi = [
    57, 16, 17, 73, 74, 0, 1, 4, 7,
    9, 8, 6, 5, 28, 15, 10, 85, 87,
    86, 84, 21, 100, 34, 35, 23, 20,
    19, 33, 88, 26, 24, 65, 58, 59,
    60, 31, 72, 99, 98, 78, 76, 30,
    63, 71, 64, 54, 62, 66, 53, 52,
    96, 94, 93, 83, 79, 80, 81, 82,
    92, 91, 67
]

gialli = [3]

A = [
    set(arancioni),
    set(viola),
    set(rossi),
    set(bianchi),
    set(gialli)
]

disegna_suddivisione_A(grafo, A)

# =====
# CHOOSE WHAT TO CALCULATE
# =====

# -----
# OPTION 1: k-KLIQUE COMMUNITIES
# -----

tutte_le_communities = calcola_tutte_le_communities(
    grafo
)

# -----
# Analyse each k separately
# -----

for k, communities in tutte_le_communities.items():
    print("\n" + "=" * 60)
    print(f"k = {k}")
    print("=" * 60)

    print(f"Number of communities: {len(communities)}")

    for i, community in enumerate(communities):
        print(
            f"Community {i + 1}: "
            f"{sorted(community)}"
        )

    # -----
    # Draw graph
    # -----

    disegna_communities(
        grafo,
        communities,
        k
    )

    # -----
    # B = subdivision generated by the algorithm
    # -----

    B = communities

    # -----
    # Jaccard score
    # -----

    score_jaccard = scores(
        A,
        B,
        jaccard_similarity
    )

```

```

# -----
# Overlap score
# -----

score_overlap = scores(
    A,
    B,
    overlap_coefficient
)

# -----
# Print results
# -----

print(
    f"Score using Jaccard similarity: "
    f"{score_jaccard:.4f}"
)

print(
    f"Score using overlap coefficient: "
    f"{score_overlap:.4f}"
)

# =====
# OPTION 2: CLIQUES
# =====
"""
cliques = find_cliques(grafo)

print("\n" + "=" * 60)
print("MAXIMAL CLIQUES")
print("=" * 60)

print(f"Number of cliques: {len(cliques)}")

for i, clique in enumerate(cliques):

    print(
        f"Clique {i + 1}: "
        f"{sorted(clique)}"
    )

B = [set(clique) for clique in cliques]

score_jaccard = scores(
    A,
    B,
    jaccard_similarity
)

score_overlap = scores(
    A,
    B,
    overlap_coefficient
)

print(
    f"Score using Jaccard similarity: "
    f"{score_jaccard:.4f}"
)

print(
    f"Score using overlap coefficient: "
    f"{score_overlap:.4f}"
)
"""

# =====
# OPTION 3: N-CLIQUES
# =====
"""
n = 6

n_cliques = find_n_cliques(
    grafo,
    n
)

print("\n" + "=" * 60)
print(f"{n}-CLIQUES")

```

```

print("=" * 60)

print(f"Number of n-cliques: {len(n_cliques)}")

for i, clique in enumerate(n_cliques):

    print(
        f"n-clique {i + 1}: "
        f"{sorted(clique)}"
    )

B = [set(clique) for clique in n_cliques]

score_jaccard = scores(
    A,
    B,
    jaccard_similarity
)

score_overlap = scores(
    A,
    B,
    overlap_coefficient
)

print(
    f"Score using Jaccard similarity: "
    f"{score_jaccard:.4f}"
)

print(
    f"Score using overlap coefficient: "
    f"{score_overlap:.4f}"
)

# -----
# Draw graph with n-clique subdivision
# -----

disegna_communities(
    grafo,
    n_cliques,
    n
)
"""

```

Appendix B

Disrupting Graph Algorithm: Python Implementation

This section provides the Python implementation for the disrupting graph algorithm.

```
import csv
import networkx as nx
import matplotlib.pyplot as plt

# =====
# 1. READ THE GRAPH
# =====

def leggi_grafo_da_csv(percorso_file):

    # Reads a weighted undirected graph from a tab-separated file.
    # Each row must contain:
    # node_1    node_2    weight

    G = nx.Graph()

    with open(percorso_file, 'r', newline='') as file:
        reader = csv.reader(file, delimiter='\t')

        for riga in reader:

            nodo1 = int(riga[0])
            nodo2 = int(riga[1])
            peso = float(riga[2])

            G.add_edge(
                nodo1,
                nodo2,
                weight=peso
            )

    return G

# =====
# 2. LARGEST CONNECTED COMPONENT
# =====

def lcc_size(G):

    """
    Returns the size of the largest connected component (LCC).
    """

    if len(G) == 0:
        return 0

    return len(
        max(
            nx.connected_components(G),
            key=len
        )
    )
```

```

# =====
# 3. K-CLIQUE COMMUNITY CENTRALITY
# =====

def k_clique_centrality(G, k):
    """
    Computes the k-clique community centrality for a fixed k:

         $c_k(v) = \max_{\{C \in C_k(v)\}} |C|$ 

    The value of k is specified manually.

    Returns:
        dictionary {node: centrality score}
    """

    centrality = {
        v: 0
        for v in G.nodes()
    }

    # Compute k-clique communities for the selected k
    communities = list(
        nx.community.k_clique_communities(
            G,
            k
        )
    )

    for C in communities:
        community_size = len(C)

        for v in C:
            centrality[v] = max(
                centrality[v],
                community_size
            )

    return centrality

# =====
# 4. GENERALIZED K-CLIQUE COMMUNITY CENTRALITY
# =====

def generalized_k_clique_centrality(G, k):
    """
    Computes the generalized k-clique community centrality
    for a fixed k:

         $c_{\text{bar}}_k(v) = \text{union}_{\{C \in C_k(v)\}} C$ 

    and its value is the cardinality of this union.

    The value of k is specified manually.

    Returns:
        dictionary {node: centrality score}
    """

    communities_containing_node = {
        v: set()
        for v in G.nodes()
    }

    # Compute k-clique communities for the selected k
    communities = list(
        nx.community.k_clique_communities(
            G,
            k
        )
    )

    for C in communities:
        for v in C:
            communities_containing_node[v].update(C)

```

```

        centrality = {
            v: len(nodes)
            for v, nodes in communities_containing_node.items()
        }

    return centrality

# =====
# 5. BETWEENNESS CENTRALITY
# =====

def betweenness centrality(G):
    """
    Computes the betweenness centrality.

    Returns:
        dictionary {node: centrality score}
    """

    return nx.betweenness centrality(G)

# =====
# 6. DEGREE CENTRALITY
# =====

def degree centrality(G):
    """
    Computes the degree centrality.

    Returns:
        dictionary {node: centrality score}
    """

    return nx.degree centrality(G)

# =====
# 7. RANK NODES
# =====

def rank_nodes(G, centrality_type, k=None):
    """
    Computes the selected centrality and ranks the nodes.

    centrality_type:
        1 -> k-clique community centrality (c_k)
        2 -> generalized k-clique community centrality (c_bar_k)
        3 -> betweenness centrality
        4 -> degree centrality

    k:
        current value of k for centrality types 1 and 2.

    Returns:
        ranked_nodes -> nodes ordered from highest to lowest score
        scores         -> dictionary containing the scores for the current k
    """

    if centrality_type == 1:
        scores = k_clique centrality(
            G,
            k
        )

    elif centrality_type == 2:
        scores = generalized_k_clique centrality(
            G,
            k
        )

    elif centrality_type == 3:
        scores = betweenness centrality(
            G

```

```

    )

elif centrality_type == 4:

    scores = degree_centrality(
        G
    )

else:

    raise ValueError(
        "centrality_type must be 1, 2, 3 or 4"
    )

# -----
# Tie-breaking multi-livello per k-clique centralities
# -----

if (
    (centrality_type == 1 or centrality_type == 2)
    and k is not None
    and k > 2
):

    # Calcola le centralit per tutti i k inferiori
    # (da k fino a 2)

    all_scores = {}

    for k_sub in range(k, 1, -1):

        if centrality_type == 1:

            all_scores[k_sub] = k_clique_centrality(
                G,
                k_sub
            )

        else:

            all_scores[k_sub] = generalized_k_clique_centrality(
                G,
                k_sub
            )

    # Calcola il grado di ciascun nodo.
    # Il grado viene utilizzato SOLO dopo aver
    # confrontato tutte le centralit per
    # k, k-1, ..., 2.

    degrees = dict(
        G.degree()
    )

    # La chiave di ordinamento confronta:
    #
    # 1. c_k
    # 2. c_{k-1}, c_{k-2}, ..., c_2
    # 3. il grado del nodo
    # 4. l'ID del nodo

    def sorting_key(v):

        lower_k_scores = tuple(
            -all_scores[k_sub][v]
            for k_sub in range(k - 1, 1, -1)
        )

        return (
            -all_scores[k][v],
            lower_k_scores,
            -degrees[v],
            v
        )

    ranked_nodes = sorted(
        G.nodes(),
        key=sorting_key
    )

else:

```

```

        # Ordinamento standard per betweenness/grado
        # o quando k <= 2

        ranked_nodes = sorted(
            scores,
            key=lambda v: (-scores[v], v)
        )

    return ranked_nodes, scores

# =====
# 8. rho_i Option A
# =====

def rho_i(G):

    # Computes
    #
    # rho_i = sum_{k=2}^{infinity} k
    # sum_{C in C_k(G)} |C|
    #
    # Only values of k for which k-clique communities exist
    # are considered.
    #
    # Returns:
    # scalar value of rho_i

    if len(G) == 0:
        return 0

    total = 0

    # Find the maximum clique size of the graph.
    # No k larger than this value can generate a
    # k-clique community.

    cliques = list(
        nx.find_cliques(G)
    )

    if len(cliques) == 0:
        return 0

    largest_clique_size = max(
        len(clique)
        for clique in cliques
    )

    # Compute the contribution for every k >= 2

    for k in range(
        2,
        largest_clique_size + 1
    ):

        communities = list(
            nx.community.k_clique_communities(
                G,
                k
            )
        )

        for C in communities:

            total += k * len(C)

    return total

# =====
# 8. rho_i Option B
# =====
"""

def rho_i(G):

    # Computes
    #
    # rho_i = sum_{k=2}^{infinity} k |C_k|
    #

```

```

# where C_k is the largest k-clique community.
#
# Only values of k for which k-clique communities exist
# are considered.
#
# Returns:
#     scalar value of rho_i

if len(G) == 0:
    return 0

total = 0

# Find the maximum clique size of the graph.

cliques = list(
    nx.find_cliques(G)
)

if len(cliques) == 0:
    return 0

largest_clique_size = max(
    len(clique)
    for clique in cliques
)

# Compute the contribution for every k >= 2

for k in range(
    2,
    largest_clique_size + 1
):

    communities = list(
        nx.community.k_clique_communities(
            G,
            k
        )
    )

    if len(communities) > 0:

        # Select the largest k-clique community

        C = max(
            communities,
            key=len
        )

        total += k * len(C)

return total

"""

# =====
# 8. rho_i Option C
# =====
"""

def rho_i(G):

    # Computes
    #
    #     rho_i = max { k >= 2 : C_k(G) is not empty }
    #
    # where C_k(G) is the set of k-clique communities.
    #
    # Returns:
    #     scalar value of rho_i

    if len(G) == 0:
        return 0

    # Find the maximum clique size of the graph

    cliques = list(
        nx.find_cliques(G)
    )

```

```

    if len(cliques) == 0:
        return 0

    largest_clique_size = max(
        len(clique)
        for clique in cliques
    )

    # Find the largest k for which at least one
    # k-clique community exists

    for k in range(
        largest_clique_size,
        1,
        -1
    ):

        communities = list(
            nx.community.k_clique_communities(
                G,
                k
            )
        )

        if len(communities) > 0:
            return k

    return 0

"""
# =====
# 9. ALL rho_{i,k}
# =====

def rho_i_k_all(G, k_values=None):

    """
    Computes all values

        rho_{i,k} = max_{C in C_k(G)} |C|

    for every k >= 2 in k_values.

    If no k-clique community exists, rho_{i,k} is set to 0.

    IMPORTANT:
    The k used here is completely independent from the
    current k used by the node-removal algorithm.

    Returns:
        dictionary

        {
            k1: rho_{i,k1},
            k2: rho_{i,k2},
            ...
        }

    Example:

        {
            2: 150,
            3: 72,
            4: 31,
            5: 12
        }
    """

    rho_values = {}

    # If no fixed set of k values is supplied, determine it from the
    # current graph. In the removal algorithm, however, k_values is
    # fixed from the initial graph so that the same rho_{i,k} values
    # are followed throughout the whole simulation.
    if k_values is None:
        if len(G) == 0:
            return rho_values

        cliques = list(nx.find_cliques(G))

```

```

        if len(cliques) == 0:
            return rho_values

        largest_clique_size = max(
            len(clique)
            for clique in cliques
        )

        k_values = range(2, largest_clique_size + 1)

        # Compute rho_{i,k} for every fixed k.
        for k in k_values:

            communities = list(
                nx.community.k_clique_communities(
                    G,
                    k
                )
            )

            # If at least one community exists, take the largest one.
            # Otherwise rho_{i,k} is defined to be 0.
            if len(communities) > 0:

                rho_values[k] = max(
                    len(C)
                    for C in communities
                )
            else:
                rho_values[k] = 0

        return rho_values

# =====
# 10. COMPUTE SELECTED RHO
# =====

def compute_rho(G, rho_type, k_values=None):

    """
    Computes the selected rho measure.

    rho_type:
    "rho_i"
        returns the scalar rho_i

    "rho_i_k"
        returns a dictionary containing all rho_{i,k}
        values for the current graph.
    """

    if rho_type == "rho_i":

        return rho_i(G)

    elif rho_type == "rho_i_k":

        return rho_i_k_all(G, k_values=k_values)

    else:

        raise ValueError(
            "rho_type must be 'rho_i' or 'rho_i_k'"
        )

# =====
# FIXED k VALUES FOR rho_i,k
# =====

def get_rho_k_values(G):

    """
    Returns the k values to be tracked during the whole simulation.

    The range is determined from the initial graph and remains fixed
    after nodes are removed.
    """
    if len(G) == 0:
        return []

```

```

    cliques = list(nx.find_cliques(G))

    if len(cliques) == 0:
        return []

    largest_clique_size = max(
        len(clique)
        for clique in cliques
    )

    return list(range(2, largest_clique_size + 1))

# =====
# 11. SEQUENTIAL NODE REMOVAL
# =====

def sequential_removal(
    G,
    centrality_type,
    initial_k=None,
    rho_type="rho_i"
):
    """
    Sequential node removal.

    For k-clique centralities:
        1. Compute centrality scores using the current k.
        2. Remove the highest-ranked node.
        3. If no node has positive centrality, decrease k by 1.
        4. Continue until k = 2 and no node has positive score,
           or until no edges remain.

    For betweenness and degree centrality:
        1. Compute centrality scores on the current graph.
        2. Remove the highest-ranked node.
        3. Recalculate the centrality on the new graph.
        4. Continue until no edges remain.

    rho_type:
        "rho_i"
            computes the scalar rho_i.

        "rho_i_k"
            computes ALL rho_{i,k} values at every iteration.

    IMPORTANT:
    These k values are independent of the k used
    for the node-removal centrality.

    Isolated nodes are not removed.
    """

    # Make a copy so that the original graph is not modified.
    G_current = G.copy()

    # Fix the k values for rho_{i,k} using the initial graph.
    rho_k_values = get_rho_k_values(G_current)

    # Current value of k.
    k = initial_k

    results = []

# -----
# Initial state G_0
# -----

    initial_rho = compute_rho(
        G_current,
        rho_type,
        k_values=rho_k_values
    )

    results.append({

```

```

    "iteration": 0,
    "removed_nodes": [],
    "number_of_nodes": len(G_current),
    "number_of_edges": G_current.number_of_edges(),
    "LCC": lcc_size(G_current),
    "rho": initial_rho
})

iteration = 0

# -----
# Main loop
# -----

while G_current.number_of_edges() > 0:

    # =====
    # K-CLIQUE CENTRALITIES
    # =====

    if centrality_type == 1 or centrality_type == 2:

        # Compute centrality using the current k.

        ranked_nodes, scores = rank_nodes(
            G_current,
            centrality_type,
            k
        )

        # Check whether at least one node has
        # positive centrality.

        positive_nodes = [
            v
            for v in G_current.nodes()
            if scores[v] > 0
        ]

        # If no node has positive centrality,
        # decrease k.

        while len(positive_nodes) == 0 and k > 2:

            k -= 1

            print(
                f"\n--- Changing k: now k = {k} ---\n"
            )

            ranked_nodes, scores = rank_nodes(
                G_current,
                centrality_type,
                k
            )

            positive_nodes = [
                v
                for v in G_current.nodes()
                if scores[v] > 0
            ]

            # If no positive k-clique centrality remains but edges are
            # still present, use degree to continue removing nodes.
            # The simulation must stop only when no edges remain.
            if len(positive_nodes) == 0:
                ranked_nodes, scores = rank_nodes(
                    G_current,
                    4
                )

        # =====
        # BETWEENNESS / DEGREE CENTRALITY
        # =====

    else:

        ranked_nodes, scores = rank_nodes(
            G_current,
            centrality_type

```

```

    )

    iteration += 1

    # Select the highest-ranked node.

    node_to_remove = ranked_nodes[0]

    if centrality_type == 1 or centrality_type == 2:

        print(
            f"Iteration {iteration}: "
            f"k = {k}, "
            f"removed node {node_to_remove} "
            f"(score = {scores[node_to_remove]})"
        )

    elif centrality_type == 3:

        print(
            f"Iteration {iteration}: "
            f"removed node {node_to_remove} "
            f"(betweenness = {scores[node_to_remove]})"
        )

    elif centrality_type == 4:

        print(
            f"Iteration {iteration}: "
            f"removed node {node_to_remove} "
            f"(degree = {scores[node_to_remove]})"
        )

    # Remove the node.

    G_current.remove_node(
        node_to_remove
    )

    # Compute the new LCC.

    current_lcc = lcc_size(
        G_current
    )

    # -----
    # Compute rho
    # -----

    rho = compute_rho(
        G_current,
        rho_type,
        k_values=rho_k_values
    )

    # -----
    # Save results
    # -----

    results.append({
        "iteration": iteration,
        "removed_nodes": [node_to_remove],
        "number_of_nodes": len(G_current),
        "number_of_edges": G_current.number_of_edges(),
        "LCC": current_lcc,
        "rho": rho,
        "score": scores[node_to_remove],
        "k": k
    })

    return results

# =====
# 12. BLOCK NODE REMOVAL
# =====

def block_removal(
    G,
    centrality_type,
    initial_k=None,
    block_size=5,

```

```

    rho_type="rho_i"
):
    """
    Block node removal.

    For k-clique centralities:

        At each iteration:

        1. Compute centrality scores using the current k
           on the current graph.

        2. If at least one node has positive centrality,
           select the five highest-ranked nodes.

        3. If no node has positive centrality, decrease k by 1
           and recompute the centrality on the SAME graph
           and in the SAME iteration.

        4. Remove the selected nodes simultaneously.

        5. Move to the next iteration.

    For betweenness and degree centrality:

        1. Compute centrality scores on the current graph.
        2. Select the five highest-ranked nodes.
        3. Remove them simultaneously.
        4. Recalculate the centrality on the new graph.

    rho_type:

        "rho_i"
            computes the scalar rho_i.

        "rho_i_k"
            computes ALL rho_{i,k} values at every iteration.

        IMPORTANT:
        These k values are independent of the k used
        for the node-removal centrality.

    Isolated nodes are not removed.
    """

    # Make a copy so that the original graph is not modified.
    G_current = G.copy()

    # Fix the k values for rho_{i,k} using the initial graph.
    rho_k_values = get_rho_k_values(G_current)

    # Current value of k.
    k = initial_k

    results = []

# -----
# Initial state G_0
# -----

    initial_rho = compute_rho(
        G_current,
        rho_type,
        k_values=rho_k_values
    )

    results.append({
        "iteration": 0,
        "removed_nodes": [],
        "number_of_nodes": len(G_current),
        "number_of_edges": G_current.number_of_edges(),
        "LCC": lcc_size(G_current),
        "rho": initial_rho
    })

    iteration = 0

```

```

# -----
# Main loop
# -----

while G_current.number_of_edges() > 0:

    # =====
    # K-CLIQUE CENTRALITIES
    # =====

    if centrality_type == 1 or centrality_type == 2:

        # -----
        # Search for the first k with at least one
        # node having positive centrality.
        #
        # IMPORTANT:
        # The graph does NOT change during this process.
        # The iteration remains the same.
        # -----

        while True:

            # Calculate the centrality on the CURRENT graph.

            ranked_nodes, scores = rank_nodes(
                G_current,
                centrality_type,
                k
            )

            # Check whether at least one node has
            # positive centrality.

            positive_nodes = [
                v
                for v in G_current.nodes()
                if scores[v] > 0
            ]

            # If at least one node has positive centrality,
            # use this k for the current iteration.

            if len(positive_nodes) > 0:
                break

            # If k = 2 and no node has positive centrality,
            # there is no smaller k to consider.

            if k == 2:
                # No positive 2-clique centrality means that the
                # current graph has no edges. The main while loop
                # will therefore terminate after this state is
                # recorded.
                break

            # Otherwise decrease k and repeat the calculation
            # on THE SAME GRAPH and THE SAME ITERATION.

            k -= 1

            print(
                f"\n--- Changing k: now k = {k} "
                f"(same iteration {iteration + 1}) ---\n"
            )

        # -----
        # Select the highest-ranked nodes.
        # -----

        iteration += 1

        nodes_to_remove = ranked_nodes[:block_size]

        print(
            f"Iteration {iteration}: "
            f"k = {k}, "
            f"removed nodes = {nodes_to_remove}"
        )

        # Stampa dettagliata con visualizzazione
        # dei punteggi di tie-break.

```

```

if k > 2 and (
    centrality_type == 1
    or centrality_type == 2
):
    if centrality_type == 1:
        sub_scores = k_clique Centrality(
            G_current,
            k - 1
        )

    else:
        sub_scores = generalized_k_clique Centrality(
            G_current,
            k - 1
        )

    for node in nodes_to_remove:
        print(
            f"    node {node}: "
            f"score(k={k}) = {scores[node]}, "
            f"score(k={k-1}) = {sub_scores[node]}"
        )

    else:
        for node in nodes_to_remove:
            print(
                f"    node {node}: "
                f"score = {scores[node]}"
            )

# =====
# BETWEENNESS / DEGREE CENTRALITY
# =====

else:
    ranked_nodes, scores = rank_nodes(
        G_current,
        centrality_type
    )

    iteration += 1

    nodes_to_remove = ranked_nodes[:block_size]

    if centrality_type == 3:
        print(
            f"Iteration {iteration}: "
            f"removed nodes = {nodes_to_remove}"
        )

        for node in nodes_to_remove:
            print(
                f"    node {node}: "
                f"betweenness = {scores[node]}"
            )

    elif centrality_type == 4:
        print(
            f"Iteration {iteration}: "
            f"removed nodes = {nodes_to_remove}"
        )

        for node in nodes_to_remove:
            print(
                f"    node {node}: "
                f"degree = {scores[node]}"
            )

```

```

# -----
# Remove all selected nodes simultaneously.
# -----

G_current.remove_nodes_from(
    nodes_to_remove
)

# -----
# Compute the new LCC.
# -----

current_lcc = lcc_size(
    G_current
)

# -----
# Compute rho
# -----

rho = compute_rho(
    G_current,
    rho_type,
    k_values=rho_k_values
)

# -----
# Save results.
# -----

results.append({
    "iteration": iteration,
    "removed_nodes": nodes_to_remove,
    "number_of_nodes": len(G_current),
    "number_of_edges": G_current.number_of_edges(),
    "LCC": current_lcc,
    "rho": rho,
    "k": k
})

return results

# =====
# 13. PLOT RHO_i
# =====

def plot_rho(results, title, rho_type):
    """
    Plots the selected rho measure.

    If rho_type == "rho_i":

        one curve representing rho_i.

    If rho_type == "rho_i_k":

        one curve for every k appearing in the results.
    """

    iterations = [
        r["iteration"]
        for r in results
    ]

    plt.figure(
        figsize=(9, 6)
    )

    # =====
    # rho_i
    # =====

    if rho_type == "rho_i":

        rho_values = [
            r["rho"]
            for r in results
        ]

        plt.plot(

```

```

        iterations,
        rho_values,
        marker="o"
    )

    plt.xlabel("Iteration")
    plt.ylabel(r"$\rho_{i,k}$")
    plt.title(title)

# =====
# rho_i,k
# =====

elif rho_type == "rho_i_k":

    # Find all k values appearing in the results.

    all_k_values = sorted({
        k
        for r in results
        for k in r["rho"].keys()
    })

    for k_value in all_k_values:

        rho_values = []

        for r in results:

            # Every tracked k must be present at every iteration.
            # If it is missing for any reason, rho_{i,k}=0.
            rho_values.append(
                r["rho"].get(k_value, 0)
            )

        plt.plot(
            iterations,
            rho_values,
            marker="o",
            label=r"$\rho_{i,{k_value}}$"
        )

        plt.xlabel("Iteration")
        plt.ylabel(r"$\rho_{i,k}$")
        plt.title(title)
        plt.legend(
            loc="best"
        )

    else:

        raise ValueError(
            "rho_type must be 'rho_i' or 'rho_i_k'"
        )

    plt.grid(True)
    plt.show()

# =====
# 14. COMPARATIVE PLOT
# =====

def plot_comparison_rho(
    results_dict,
    title,
    rho_type
):

    """
    Comparative plot.

    results_dict:

        dictionary of the form

        {
            "Label": results_list
        }

    For rho_i:

```

```

        one curve per centrality.

For rho_i,k:

    one curve for every combination of
    centrality and k.
"""

plt.figure(
    figsize=(10, 7)
)

colors = {
    "c1": "#1f77b4",
    "c2": "#ff7f0e",
    "betweenness": "#2ca02c",
    "degree": "#d62728"
}

markers = {
    "c1": "o",
    "c2": "s",
    "betweenness": "^",
    "degree": "D"
}

# =====
# rho_i
# =====

if rho_type == "rho_i":

    plt.figure(figsize=(10, 7))

    for label, results in results_dict.items():

        iterations = [
            r["iteration"]
            for r in results
        ]

        rho_values = [
            r["rho"]
            for r in results
        ]

        color_key = next(
            (
                key
                for key in colors
                if key in label.lower()
            ),
            None
        )

        c = colors.get(
            color_key,
            None
        )

        m = markers.get(
            color_key,
            "o"
        )

        plt.plot(
            iterations,
            rho_values,
            marker=m,
            color=c,
            linewidth=2,
            label=label
        )

        plt.xlabel("Iteration")
        plt.ylabel(r"$\rho_i$")
        plt.title(title)
        plt.grid(True)
        plt.legend(loc="best")
        plt.show()

# =====

```

```

# rho_i,k
# =====

elif rho_type == "rho_i_k":

    # Find all k values appearing in any centrality result.
    all_k_values = sorted({
        k
        for results in results_dict.values()
        for r in results
        for k in r["rho"].keys()
    })

    # Create one graph for each fixed k.
    for k_value in all_k_values:

        plt.figure(figsize=(10, 7))

        for label, results in results_dict.items():

            color_key = next(
                (
                    key
                    for key in colors
                    if key in label.lower()
                ),
                None
            )

            c = colors.get(
                color_key,
                None
            )

            m = markers.get(
                color_key,
                "o"
            )

            iterations = [
                r["iteration"]
                for r in results
            ]

            rho_values = []

            for r in results:

                # Every tracked k must be present at every iteration.
                # If it is missing for any reason, rho_{i,k}=0.
                rho_values.append(
                    r["rho"].get(k_value, 0)
                )

            plt.plot(
                iterations,
                rho_values,
                marker=m,
                color=c,
                linewidth=1.5,
                label=label
            )

            plt.xlabel("Iteration")
            plt.ylabel(rf"$\rho_{\{i,\{k\_value\}\}}$")
            plt.title(
                f"{title} - $k=\{k\_value\}$"
            )
            plt.grid(True)
            plt.legend(loc="best")
            plt.show()

        else:

            raise ValueError(
                "rho_type must be 'rho_i' or 'rho_i_k'"
            )

# =====
# 15. MAIN
# =====

```

```

# Path of the input file
percorso_file = "Montagna_meetings_edgelist.csv"

# Read the graph
grafo = leggi_grafo_da_csv(
    percorso_file
)

# -----
# SELECT INITIAL k
# -----

initial_k = int(
    input(
        "Enter the initial value of k: "
    )
)

print(
    f"\nInitial value of k = {initial_k}"
)

# -----
# SELECT RHO
# -----

print("\nChoose the rho measure:")

print("1 -> rho_i")
print("2 -> all rho_i,k")

rho_choice = input(
    "Enter 1 or 2: "
)

if rho_choice == "1":
    rho_type = "rho_i"
elif rho_choice == "2":
    rho_type = "rho_i,k"
else:
    raise ValueError(
        "Invalid choice. Enter 1 or 2."
    )

print(
    f"\nSelected rho = {rho_type}\n"
)

# =====
# 16. SEQUENTIAL REMOVAL
# =====

# -----
# Sequential removal - k-clique community centrality
# -----

results_seq_c1 = sequential_removal(
    grafo,
    centrality_type=1,
    initial_k=initial_k,
    rho_type=rho_type
)

# -----
# Sequential removal - generalized k-clique community centrality
# -----

results_seq_c2 = sequential_removal(
    grafo,
    centrality_type=2,

```

```

        initial_k=initial_k,
        rho_type=rho_type
    )

# -----
# Sequential removal - betweenness centrality
# -----

results_seq_betweenness = sequential_removal(
    grafo,
    centrality_type=3,
    rho_type=rho_type
)

# -----
# Sequential removal - degree centrality
# -----

results_seq_degree = sequential_removal(
    grafo,
    centrality_type=4,
    rho_type=rho_type
)

# =====
# 17. BLOCK REMOVAL
# =====

# -----
# Block removal - k-clique community centrality
# -----

results_block_c1 = block_removal(
    grafo,
    centrality_type=1,
    initial_k=initial_k,
    block_size=5,
    rho_type=rho_type
)

# -----
# Block removal - generalized k-clique community centrality
# -----

results_block_c2 = block_removal(
    grafo,
    centrality_type=2,
    initial_k=initial_k,
    block_size=5,
    rho_type=rho_type
)

# -----
# Block removal - betweenness centrality
# -----

results_block_betweenness = block_removal(
    grafo,
    centrality_type=3,
    block_size=5,
    rho_type=rho_type
)

# -----
# Block removal - degree centrality
# -----

results_block_degree = block_removal(
    grafo,
    centrality_type=4,
    block_size=5,
    rho_type=rho_type
)

# =====

```

```

# 18. PRINT NUMBER OF ITERATIONS
# =====

print(
    "Sequential removal - c1:",
    len(results_seq_c1) - 1,
    "iterations"
)

print(
    "Sequential removal - c2:",
    len(results_seq_c2) - 1,
    "iterations"
)

print(
    "Sequential removal - betweenness:",
    len(results_seq_betweenness) - 1,
    "iterations"
)

print(
    "Sequential removal - degree:",
    len(results_seq_degree) - 1,
    "iterations"
)

print(
    "Block removal - c1:",
    len(results_block_c1) - 1,
    "iterations"
)

print(
    "Block removal - c2:",
    len(results_block_c2) - 1,
    "iterations"
)

print(
    "Block removal - betweenness:",
    len(results_block_betweenness) - 1,
    "iterations"
)

print(
    "Block removal - degree:",
    len(results_block_degree) - 1,
    "iterations"
)

# =====
# 19. PLOT RESULTS - SEQUENTIAL REMOVAL
# =====

plot_rho(
    results_seq_c1,
    "Sequential Node Removal - K-Clique Community Centrality",
    rho_type
)

plot_rho(
    results_seq_c2,
    "Sequential Node Removal - Generalized K-Clique Community Centrality",
    rho_type
)

plot_rho(
    results_seq_betweenness,
    "Sequential Node Removal - Betweenness Centrality",
    rho_type
)

plot_rho(
    results_seq_degree,
    "Sequential Node Removal - Degree Centrality",
    rho_type
)

# =====

```

```

# 20. PLOT RESULTS - BLOCK REMOVAL
# =====

plot_rho(
  results_block_c1,
  "Block Node Removal - K-Clique Community Centrality",
  rho_type
)

plot_rho(
  results_block_c2,
  "Block Node Removal - Generalized K-Clique Community Centrality",
  rho_type
)

plot_rho(
  results_block_betweenness,
  "Block Node Removal - Betweenness Centrality",
  rho_type
)

plot_rho(
  results_block_degree,
  "Block Node Removal - Degree Centrality",
  rho_type
)

# =====
# 21. COMPARATIVE PLOT - SEQUENTIAL REMOVAL
# =====

seq_comparison = {
  "$c_k$ centrality":
    results_seq_c1,

  "$\\overline{c}_k$ centrality":
    results_seq_c2,

  "Betweenness centrality":
    results_seq_betweenness,

  "Degree centrality":
    results_seq_degree
}

plot_comparison_rho(
  seq_comparison,
  "Sequential Node Removal Comparison",
  rho_type
)

# =====
# 22. COMPARATIVE PLOT - BLOCK REMOVAL
# =====

block_comparison = {
  "$c_k$ centrality":
    results_block_c1,

  "$\\overline{c}_k$ centrality":
    results_block_c2,

  "Betweenness centrality":
    results_block_betweenness,

  "Degree centrality":
    results_block_degree
}

plot_comparison_rho(
  block_comparison,
  "Block Node Removal Comparison",
  rho_type
)

```

Appendix C

Additional Results and Plots

For the sake of completeness, this appendix presents the additional plots that complement the findings discussed in Chapter 3.

- If $\rho_i = \sum_{k=2}^{\infty} k \sum_{C \in \mathcal{C}_k(G_i)} |C|$:

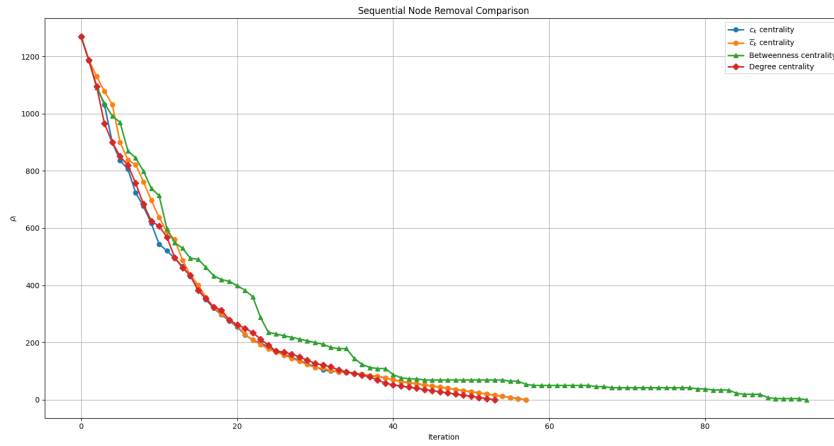


Figure C.1

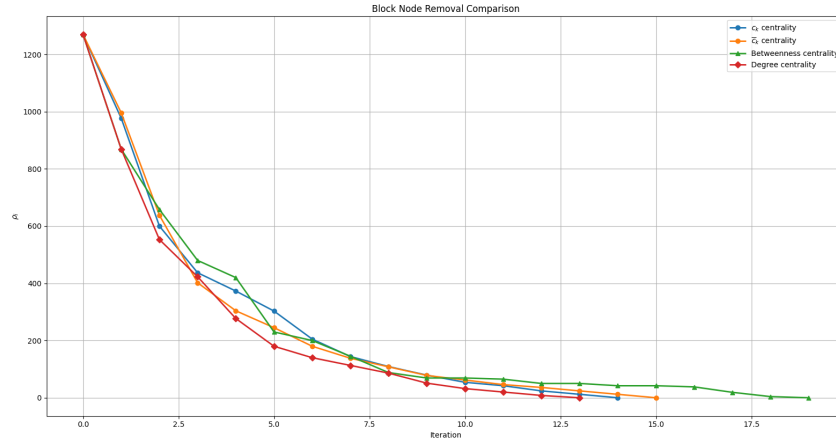


Figure C.2

- If $\rho_{i,k} = \max_{C \in \mathcal{C}_k(G_i)} |C|$, $\forall k \geq 2$ such that $\mathcal{C}_k(G_i) \neq \emptyset$:

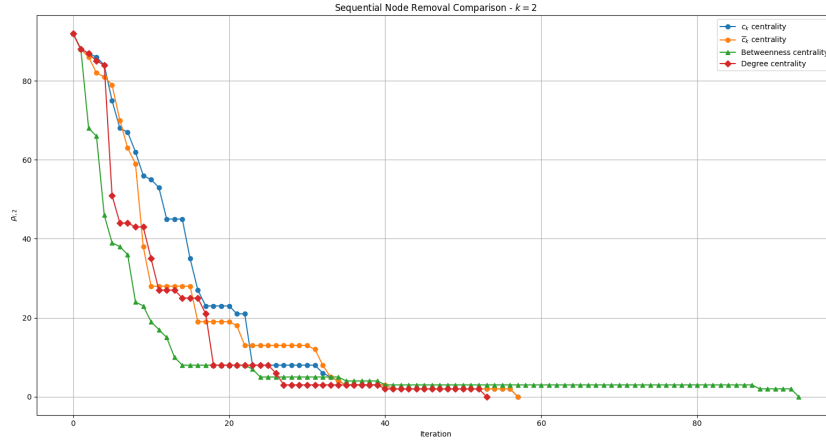


Figure C.3

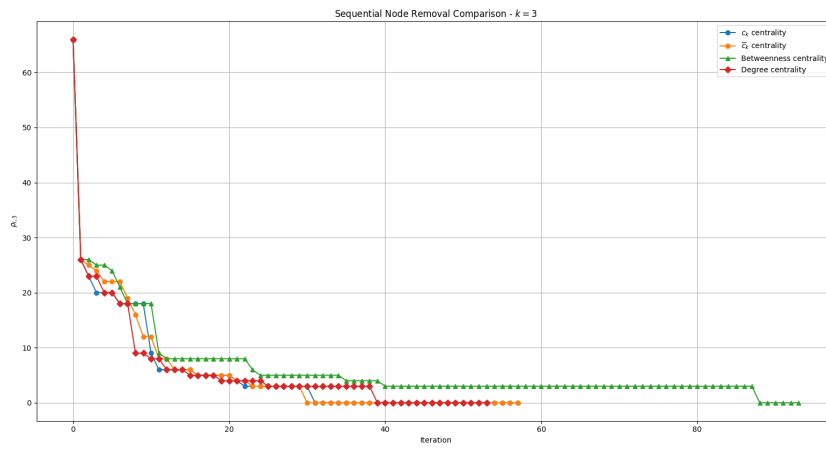


Figure C.4

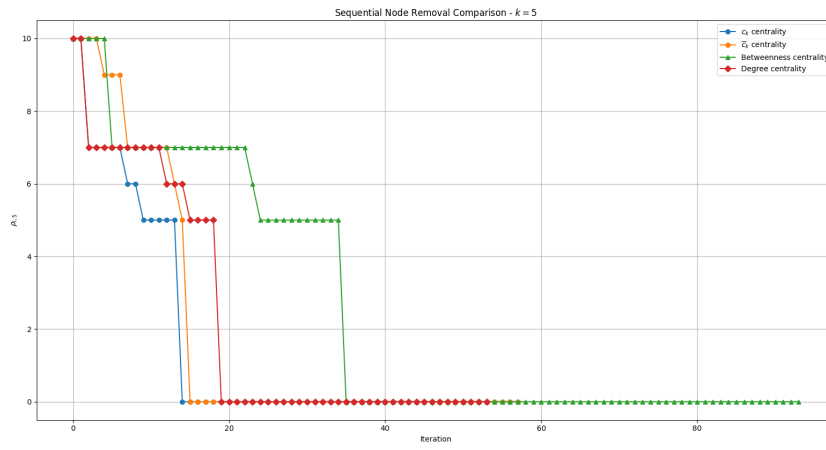


Figure C.5

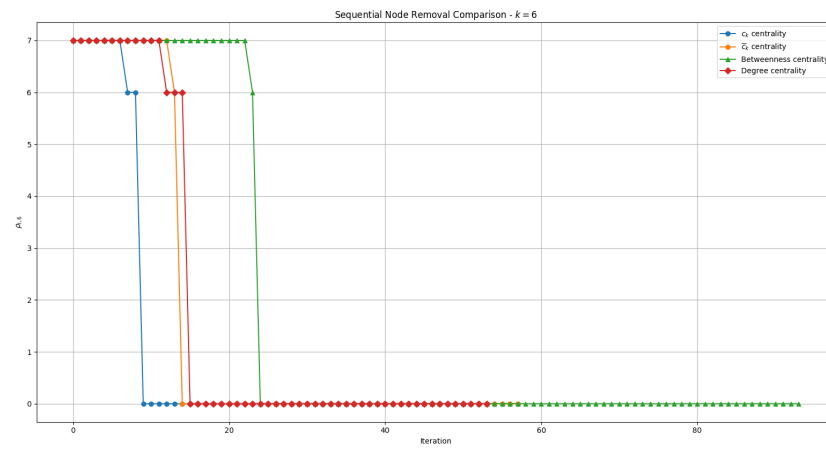


Figure C.6

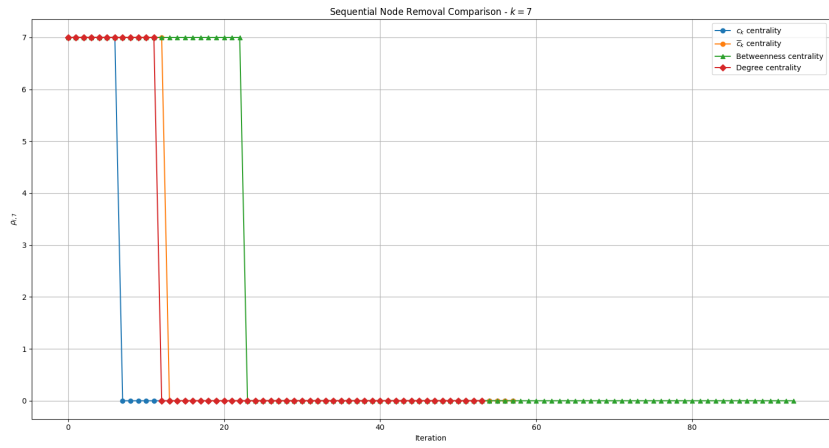


Figure C.7

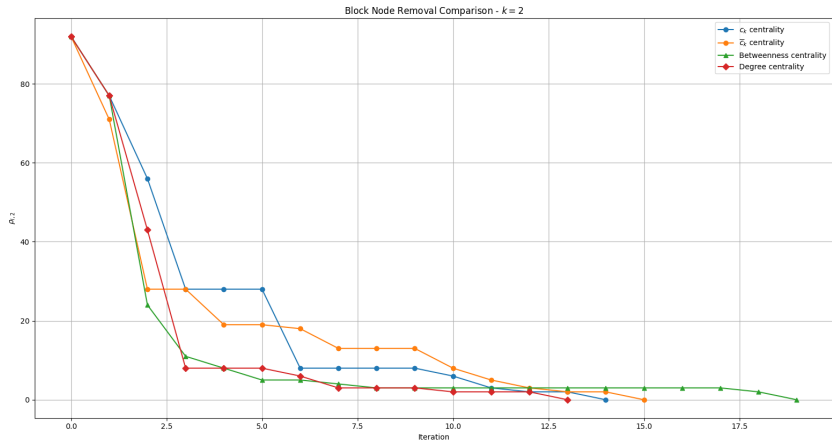


Figure C.8

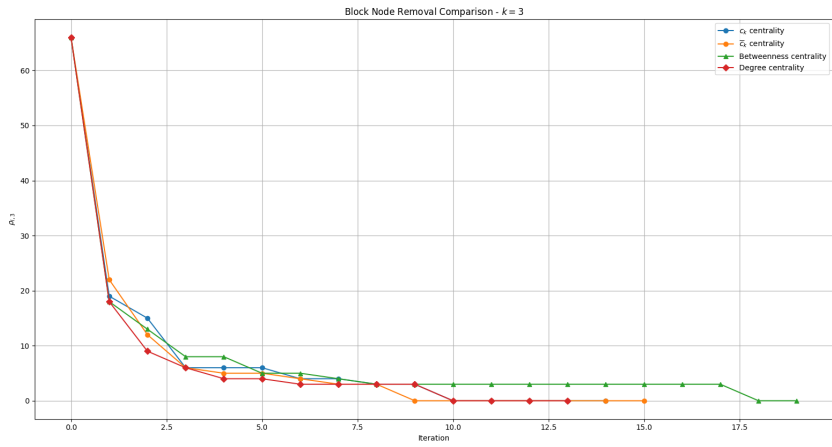


Figure C.9

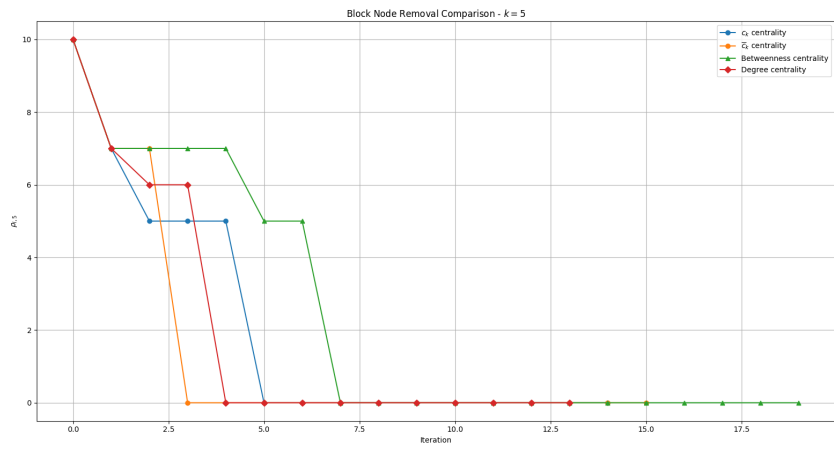


Figure C.10

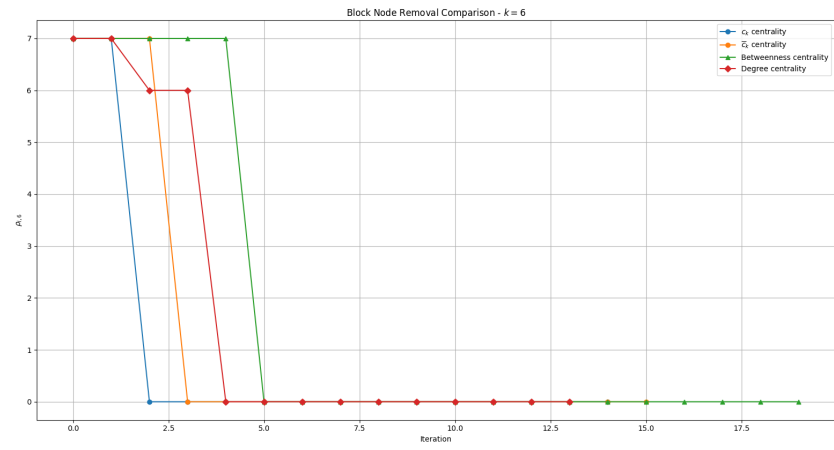


Figure C.11

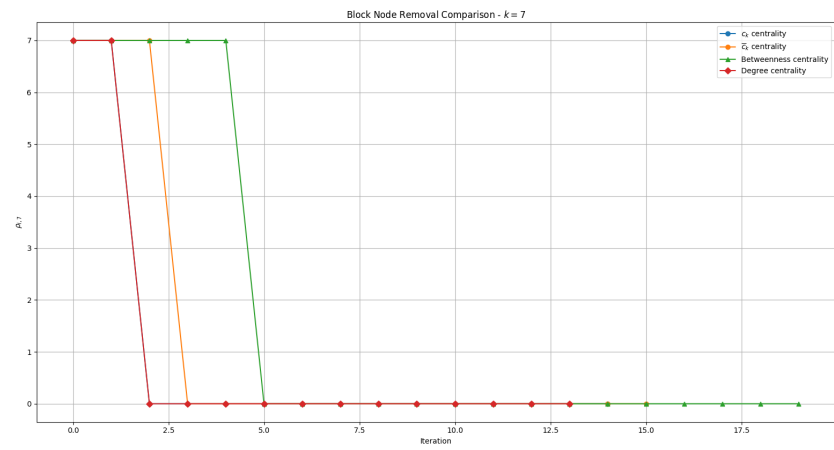


Figure C.12

Bibliography

- [1] Albert-László Barabási and Réka Albert. “Emergence of scaling in random networks”. In: *Science* 286.5439 (1999), pp. 509–512.
- [2] Albert-László Barabási and Márton Pósfai. *Network science*. Vol. 1. Cambridge university press Cambridge, 2016.
- [3] Francesco Calderoni and Elisa Superchi. “The nature of organized crime leadership: Criminal leaders in meeting and wiretap networks”. In: *Crime, Law and Social Change* 72.4 (2019), pp. 419–444.
- [4] Francesco Calderoni et al. “Robust link prediction in criminal networks: A case study of the Sicilian Mafia”. In: *Expert Systems with Applications* 161 (2020), p. 113666.
- [5] Lucia Cavallaro et al. “Disrupting resilient criminal networks through data analysis: The case of Sicilian Mafia”. In: *Plos one* 15.8 (2020), e0236476.
- [6] Lucia Cavallaro et al. “Graph and network theory for the analysis of criminal networks”. In: *Data Science and Internet of Things: Research and Applications at the Intersection of DS and IoT*. Springer, 2021, pp. 139–156.
- [7] Michel Marie Deza and Elena Deza. “Encyclopedia of Distances”. In: *Encyclopedia of Distances*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 1–583. ISBN: 978-3-642-00234-2. DOI: 10.1007/978-3-642-00234-2_1. URL: https://doi.org/10.1007/978-3-642-00234-2_1.
- [8] Reinhard Diestel. *Graph Theory*. 6th ed. Vol. 173. Graduate Texts in Mathematics. Springer, 2025. ISBN: 978-3-662-70106-5. DOI: 10.1007/978-3-662-70107-2.
- [9] Annamaria Ficara et al. “The whole is greater than the sum of the parts: A multi-layer approach on criminal networks”. In: *Future Internet* 14.5 (2022), p. 123.
- [10] Paul Jaccard. “The Distribution of the Flora in the Alpine Zone.” In: *New Phytologist* 11.2 (1912), pp. 37–50. DOI: <https://doi.org/10.1111/j.1469-8137.1912.tb05611.x>. URL: <https://nph.onlinelibrary.wiley.com/doi/abs/10.1111/j.1469-8137.1912.tb05611.x>.
- [11] Vito Latora, Vincenzo Nicosia, and Giovanni Russo. *Complex Networks: Principles, Methods and Applications*. Cambridge: Cambridge University Press, 2017. ISBN: 978-1-107-10318-4.
- [12] Michael Levandowsky and David K. Winter. “Distance between Sets”. In: *Nature* 234 (1971), pp. 34–35. URL: <https://api.semanticscholar.org/CorpusID:4283015>.
- [13] Carlo Morselli. *Inside criminal networks*. Springer Science & Business Media, 2008.
- [14] Mark Newman. *Networks*. Oxford university press, 2018.

- [15] Gergely Palla et al. “Uncovering the overlapping community structure of complex networks in nature and society”. In: *Nature* 435.7043 (2005), pp. 814–818.
- [16] Bastian Rieck et al. “Clique Community Persistence: A Topological Visual Analysis Approach for Complex Networks”. In: *IEEE Transactions on Visualization and Computer Graphics* 24.1 (2018), pp. 822–831. DOI: 10.1109/TVCG.2017.2744321.
- [17] George Gaylord Simpson. “Data on the Relationships of Local and Continental Mammalian Faunas”. In: *Journal of Paleontology* 10 (1936), pp. 410–414.
- [18] George Gaylord Simpson. “Evolution, Interchange, and Resemblance of the North American and Eurasian Cenozoic Mammalian Faunas”. In: *Geological Society of America Bulletin* 58.7 (1947), pp. 613–688.
- [19] Malcolm K. Sparrow. “The application of network analysis to criminal intelligence: An assessment of the prospects”. In: *Social Networks* 13.3 (1991), pp. 251–274. ISSN: 0378-8733. DOI: [https://doi.org/10.1016/0378-8733\(91\)90008-H](https://doi.org/10.1016/0378-8733(91)90008-H). URL: <https://www.sciencedirect.com/science/article/pii/037887339190008H>.
- [20] Duncan J. Watts and Steven H. Strogatz. “Collective dynamics of ‘small-world’ networks”. In: *Nature* 393 (1998), pp. 440–442. URL: <https://api.semanticscholar.org/CorpusID:3034643>.

Ringraziamenti

Anche per questa laurea non aspettatevi un papiro di ringraziamenti, sono passati anni vero, ma sono pur sempre io. In questo caso però, qualche parolina in più è doverosa visto che si è concluso un capitolo decisamente ingombrante della mia vita durante il quale ho conosciuto persone fantastiche che porterò per sempre nel cuore.

Un ringraziamento veramente speciale va ai miei relatori e alla mia correlatrice, perchè ognuno di loro ha reso questa tesi, alla quale sono molto affezionata, possibile. Tutto è partito dalla mia correlatrice, la Professoressa Robotti, la quale, dopo aver sentito il mio podcast, mi ha spronata a non lasciare cadere il progetto nel nulla, anche se ancora non sapevo quale direzione avrebbe preso. E' qui che è entrata in gioco la Professoressa De Negri. Lo sapevamo già entrambe che ad un certo punto avrei bussato alla porta del suo ufficio per chiederle la tesi, ma il suo entusiasmo per questo nuovo argomento e la sua voglia di mettersi in gioco per aiutarmi, mi hanno comunque sorpresa perchè non è un approccio scontato. E' stata proprio lei ad indirizzarmi verso Ulderico. Ad Ulderico devo un ringraziamento veramente enorme. Non pensavo di poter trovare una persona che avesse così tanta voglia di seguirmi e che potesse mettere così tanto impegno, tempo e passione in questo lavoro di tesi. E' proprio la sua passione che mi affascina ogni volta e che mi ha spronata a dare il massimo, anche se lui, per qualche motivo a me ancora ignoto, ha creduto fin da subito in me e in questa tesi (o ha finto molto bene). E' stata veramente una spinta enorme sentirmi sostenuta e appoggiata da tre persone delle quali ho immensa stima.

Grazie alla mia famiglia perchè da dietro le quinte mi ha sempre sostenuta in tutto quello che ho voluto fare, non sempre magari comprendendo fino in fondo, ma senza mai farmi pesare le scelte che ho preso. Sono veramente fortunata ad avere una sorella come Viola e dei genitori come la mia mamma e il mio papà che sono sempre presenti ma non invadenti.

Grazie ad Elia perchè mi stai sempre accanto, anche se gli occhi ti si chiudono durante le mie notti insonni a preparare esami o a scrivere la tesi. Grazie perchè mi fai sempre sentire la tua priorità, che mi appoggi e che ti lasci sempre trascinare in quello che faccio, ma sempre prendendoti cura di me. Grazie.

Grazie alle mie adorato Winx: Cecilia, Chiara, Elena, Gaia e Sofia. Senza di voi non sarei mai arrivata qui, lo sapete bene. Un po' perchè senza di voi non avrei ancora dato la maggior parte degli esami (sia per il sostegno morale costante, sia perchè con alcune di voi ho preparato la maggior parte dei miei esami), ma soprattutto perchè trovare delle amiche come voi durante l'università mi ha permesso di sentirmi sempre accompagnata durante questo percorso e di sapere di avere delle persone su cui contare per sfogarmi o semplicemente distrarmi. Probabilmente l'amicizia che ci lega è così forte perchè abbiamo affrontato gli stessi traumi, ma comunque, me la tengo stretta.

Grazie anche ad Andre Minna, Luca e Samolino perchè non mi hanno mai negato una spiegazione quando non ci stavo più capendo nulla o una parola di conforto nei momenti

di disperazione, ma soprattutto perchè abbiamo potuto passare sempre dei bei momenti insieme.

Ringrazio anche di cuore tutti gli altri miei compagni di università perchè ho trovato delle persone meravigliose con cui condividere la preparazione di esami o semplicemente un pranzo in piccionaia.

Grazie alle mie amiche d'infanzia Bibi e Giuly. Lo so che non capiranno mai la mia scelta di studiare matematica ma, nonostante ciò, sono sempre state curiose e disposte a provare a capire cosa stessi facendo. Si sono addirittura impegnate tantissimo a capire di cosa parlava la mia tesi e l'ho apprezzato molto. Lo so che è una cosa un po' materialista, ma avere tutti i giorni al polso, da ormai quasi tre anni, il bracciale che ci siamo fatte insieme, mi ricorda che voi ci siete sempre. Grazie di cuore.

Grazie ai miei amici scout perchè, per quanto mi facciate impazzire molte volte, mi avete aiutata a crescere e sicuramente non avrei affrontato questo percorso allo stesso modo senza di voi. E questo vale per tutti i miei compagni di Co.Ca. di questi anni, ma anche per tutti i ragazzi del clan con i quali ho camminato insieme nell'ultimo anno. Un grazie speciale va alla Bea perchè questo anno universitario è stato impegnativo per entrambe, ma durante il nostro servizio insieme siamo sempre state capaci di aiutarci. Ad Ake perchè è sempre pronto a parlare di qualsiasi cosa e di essere un punto d'appoggio nella vita e negli scout, soprattutto davanti ad un aperitivo. Alla Moon perchè il suo entusiasmo non finisce mai, come le nostre chiacchierate sotto casa. Grazie ragazzi.

Grazie a Carlo, Tacco, Ceci, Edo, Fabio, Gine e Lalla perchè mi avete accolta nel vostro gruppo e in questi anni abbiamo sempre trovato il modo di fare vacanzette stupende, nonostante le nostre vite estremamente impegnate.

Grazie ai mie amici dell'Elba. Vi lamentate sempre che sto troppo poco all'Elba e avete ragione. Ma la verità è che siete sempre stati il mio spazio per staccare da tutto durante anni e sessioni impegnativi e per questo ve ne sarò sempre grata. Vi aspetto a Genova.

Grazie a tutte le persone che ho incontrato in questo lungo cammino e a tutte quelle che, con tanta pazienza, mi hanno accompagnata. E' stato un onore!