



# Graph-based contributions to machine-learning

Quentin Lutz

## ► To cite this version:

Quentin Lutz. Graph-based contributions to machine-learning. Data Structures and Algorithms [cs.DS]. Institut Polytechnique de Paris, 2022. English. NNT : 2022IPPAT010 . tel-03634148

**HAL Id: tel-03634148**

**<https://theses.hal.science/tel-03634148>**

Submitted on 7 Apr 2022

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



INSTITUT  
POLYTECHNIQUE  
DE PARIS



## Graph-based contributions to machine learning

Thèse de doctorat de l'Institut Polytechnique de Paris  
préparée à Télécom Paris

École doctorale n°626 École doctorale de l'Institut Polytechnique de Paris (EDIPP)  
Spécialité de doctorat : Informatique, Données et Intelligence Artificielle

Thèse présentée et soutenue à Paris, le 9 février 2022, par

**QUENTIN LUTZ**

Composition du Jury :

Jean-Loup Guillaume Professeur, Université de la Rochelle	Président
Matthieu Latapy Directeur de recherche, CNRS et Sorbonne Université	Rapporteur
Conrado Martínez Professeur, Universitat Politècnica de Catalunya	Rapporteur
Cécile Mailler Maîtresse de conférence, University of Bath	Examinatrice
Thomas Bonald Professeur, Télécom Paris	Directeur de thèse
Élie de Panafieu Chercheur, Nokia Bell Labs	Co-encadrant
Gérard Burnside Chef de département, Nokia Bell Labs	Invité

## Abstract

Un graphe est un objet mathématique permettant de représenter des relations entre des entités (appelées nœuds) sous forme d'arêtes. Les graphes sont depuis longtemps un objet d'étude pour différents problèmes allant d'Euler au PageRank en passant par les problèmes de plus courts chemins. Les graphes ont plus récemment trouvé des usages pour l'apprentissage automatique. Avec l'avènement des réseaux sociaux et du web, de plus en plus de données sont représentées sous forme de graphes. Ces graphes sont toujours plus gros, pouvant contenir des milliards de nœuds et arêtes. La conception d'algorithmes efficaces s'avère nécessaire pour permettre l'analyse de ces données.

Cette thèse étudie l'état de l'art et propose de nouveaux algorithmes pour la recherche de communautés et le plongement de nœuds dans des données massives. Par ailleurs, pour faciliter la manipulation de grands graphes et leur appliquer les techniques étudiées, nous proposons Scikit-network, une librairie libre développée en Python dans le cadre de la thèse. Nous nous intéressons également au problème d'annotation de données. Les techniques supervisées d'apprentissage automatique nécessitent des données annotées pour leur entraînement. La qualité de ces données influence directement la qualité des prédictions de ces techniques une fois entraînées. Cependant, obtenir ces données ne peut pas se faire uniquement à l'aide de machines et requiert une intervention humaine que nous cherchons à minimiser.

Les contributions décrites dans le manuscrit ont toutes recours à des graphes. Pour la fouille de graphes et le développement de Scikit-network, les données étudiées sont représentées sous forme de graphes. Pour le problème d'annotation des données, les graphes servent d'outil formel à un raisonnement mathématique. Toutes ces contributions ont des applications dans le domaine de l'apprentissage automatique.

Les contributions logicielles de la thèse sont rassemblées dans Scikit-network. Scikit-network est inspirée de la librairie Scikit-learn, qui propose un large éventail d'algorithmes pour l'apprentissage automatique sur des données vectorielles. Par analogie, nous rassemblons dans Scikit-network diverses méthodes pour la fouille de données et l'apprentissage automatique sur des graphes. Cette librairie libre fait usage de nombreux outils présents dans l'écosystème du langage Python pour permettre à ses utilisateurs d'analyser des jeux de données massifs tout en offrant une bonne ergonomie. De nombreuses tâches, telles que le calcul de centralités et la classification de nœuds, peuvent être accomplies à l'aide de Scikit-network.

Le manuscrit fait l'état de l'art des techniques de recherche de communautés dans des graphes. Nous distinguons les techniques selon qu'elles sont hiérarchiques ou non. Dans le second cas, nous étudions tout particulièrement la méthode de Louvain et certaines de ses variantes. Nous comparons qualitativement les performances de ces variantes. Nous proposons également une nouvelle variante de la méthode de Louvain visant à en accélérer l'exécution.

Nous nous appuyons sur les techniques de recherche de communautés ainsi décrites pour proposer une nouvelle méthode pour le plongement des nœuds d'un graphe. Nous

mesurons la performance de cette méthode en la comparant à d'autres algorithmes issus de l'état de l'art. Nous illustrons également certaines de ses propriétés, notamment en expliquant comment les résultats peuvent être interprétés en fonction des communautés trouvées par l'algorithme de Louvain.

Une autre contribution de la thèse consiste en l'étude d'un problème particulier d'annotation de jeux de données motivé par un cas d'usage industriel. Après avoir introduit le problème en question, sous un formalisme utilisant des graphes, nous caractérisons simplement les solutions optimales. Nous prouvons ensuite que cette famille de solutions optimales partage la même distribution de complexité dont nous étudions le comportement. Nous décrivons également plusieurs variantes possibles du problème étudié et leurs applications.

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Motivation . . . . .	3
1.2	Graphs . . . . .	3
1.3	Graph analysis . . . . .	5
1.3.1	Graph theory . . . . .	5
1.3.2	Graph mining . . . . .	7
1.4	Outline . . . . .	9
1.5	Publications . . . . .	9
<b>2</b>	<b>Scikit-network</b>	<b>11</b>
2.1	Motivation . . . . .	11
2.2	Software Features . . . . .	12
2.3	Practical considerations . . . . .	13
2.3.1	Efficient graph representation . . . . .	13
2.3.2	Guidelines for the package . . . . .	17
2.3.3	Performance . . . . .	19
2.4	Conclusion . . . . .	20
<b>3</b>	<b>Node clustering</b>	<b>21</b>
3.1	Introduction . . . . .	21
3.1.1	Node sampling . . . . .	21
3.2	Flat clustering . . . . .	22
3.2.1	Similarity measures . . . . .	22
3.2.2	Modularity functions . . . . .	24
3.2.3	Louvain method . . . . .	28
3.2.4	Leiden refinements . . . . .	31
3.3	Hierarchical clustering . . . . .	35
3.3.1	Dendrograms . . . . .	35
3.3.2	Agglomerative approach . . . . .	36
3.3.3	Divisive approach . . . . .	37
3.3.4	Experiments . . . . .	37
3.4	Conclusion . . . . .	39

<b>4</b>	<b>Node embedding</b>	<b>40</b>
4.1	Introduction . . . . .	40
4.1.1	Motivation . . . . .	40
4.1.2	Related work . . . . .	41
4.2	Embedding method . . . . .	41
4.2.1	Algorithm . . . . .	41
4.2.2	Link with soft clustering . . . . .	43
4.3	Results . . . . .	43
4.3.1	Link prediction . . . . .	44
4.3.2	Node classification . . . . .	45
4.3.3	Time performance . . . . .	45
4.4	Properties . . . . .	46
4.4.1	Interpretability . . . . .	46
4.4.2	Sparsity of the embedding . . . . .	47
4.5	Conclusion . . . . .	48
<b>5</b>	<b>Dataset labeling</b>	<b>49</b>
5.1	Introduction . . . . .	49
5.1.1	Motivation and related work . . . . .	49
5.1.2	Setting . . . . .	51
5.2	Chordal algorithms . . . . .	52
5.2.1	Chordal graphs . . . . .	53
5.2.2	Optimality of chordal algorithms . . . . .	54
5.2.3	Cost equivalence of chordal algorithms . . . . .	57
5.2.4	Complexity estimates . . . . .	61
5.3	Practical use . . . . .	72
5.3.1	Considerations for implementations . . . . .	73
5.3.2	Assistance by data-specific techniques . . . . .	73
5.3.3	Imperfect oracle . . . . .	74
5.4	Conclusion . . . . .	78
<b>6</b>	<b>Conclusion and perspectives</b>	<b>79</b>
<b>7</b>	<b>Bibliography</b>	<b>81</b>

# Chapter 1

## Introduction

### 1.1 Motivation

Machine learning gathers algorithms that are designed to "learn" from data, i.e. algorithms that can improve their performance for a given task by themselves. It may be seen as a subfield of artificial intelligence in that it allows computers to emulate some form of reasoning. Machine learning algorithms aim at predicting information about the data they are fed. This data may come in a number of shapes and formats: it can be images of animals, law bills gathered in text corpora or molecule interactions represented by a graph. The applications for machine learning algorithms are equally varied and range from agriculture and medicine to banking and linguistics.

In this thesis, we tackle selected problems in machine learning pertaining to graphs (which are a mathematical representation of interactions between several entities). Graphs have been used in many different ways in the field of machine learning. Among recent developments are graph kernels (Vishwanathan et al., 2010), which aim at measuring the similarity between pairs of graphs, and graph neural networks (Scarselli et al., 2008) that make it possible to use neural networks on data represented by graphs. Throughout this thesis, we distinguish cases where graphs are used to model relational data (as seen in Chapters 2, 3 and 4) as opposed to settings where graphs are used as a formal tool regardless of the actual data (as is the case in Chapter 5). In both cases, however, we seek to harness graphs and their properties to improve the performance of machine learning algorithms.

### 1.2 Graphs

A *graph* is a mathematical object used to represent relations between entities called *nodes* or *vertices*. The relations between nodes are represented by pairwise connections between them called *edges* or *links* depending on the context. Two nodes linked by an edge are *neighbors* of one another and referred to as *adjacent*. The *degree* of a node is its number of neighbors. Let  $G = (V, E)$  denote a graph, where  $V$  is the node set and  $E \subset V \times V$  is the

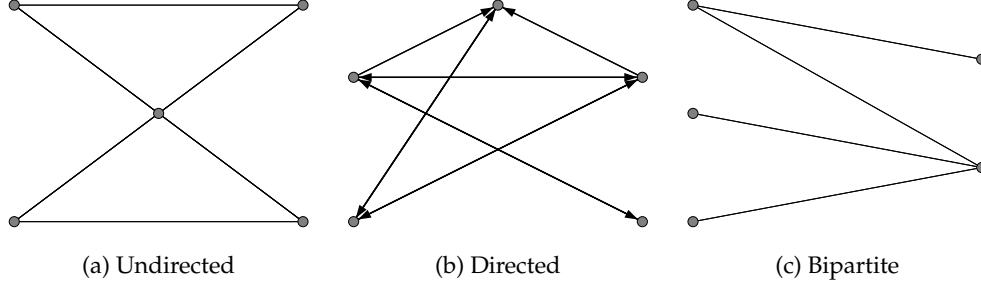


Figure 1.1: Various graphs

edge set.

Graphs can be found in a wide variety of fields ranging from bioinformatics to social sciences. Depending on the domain under study, graphs may sometimes be called *networks*. They can be used to represent many different datasets including web graphs where each node is a website and links represent the hyperlinks between them.

There are a number of families of graphs. We describe some of those that are found in this work and illustrate them in Figure 1.1:

- **Undirected graphs** describe symmetric relations between the nodes. That is  $\forall i, j \in V$ , if  $(i, j) \in E$  then  $(j, i) \in E$ . This is the case of users on Facebook as the friendship link is symmetric.
- Conversely, **directed graphs** describe relations where the previous implication does not hold for all node pairs. This corresponds to the follower/followee relation on Twitter or hyperlinks on the web. In directed graphs, edges are sometimes called *arcs*. We distinguish the *source* node of an arc from its *target* node for clarity. Considering an edge  $(i, j) \in E$ , we say that  $i$  is a *predecessor* of  $j$  and that  $j$  is a *successor* of  $i$ .
- **Bipartite graphs** are a subclass of undirected graphs where the node set  $V$  is the disjoint union of two sets  $V_1$  and  $V_2$  such that all edges in  $E$  have one end in one set and the other end in the other set. Bipartite graphs are used to represent relations between two distinct types of entities such as diseases and their corresponding symptoms or Netflix users and the movies they rate.

In this work, we also consider one natural variant: weighted graphs. In a weighted graph, each edge is assigned a non-negative real number called weight. This allows to further characterize the relative strengths of the connections between the nodes.

We fix some notations that can be found throughout this thesis:

- Unless specified, we note  $n = |V|$ , the number of nodes and  $m = |E|$ , the number of edges.
- If the graph is unweighted and undirected,  $d_i$  is the degree of node  $i \in V$ . If the graph is directed,  $d_j^+$  (resp.  $d_i^-$ ) is the out-degree (resp. in-degree), i.e. the number of edges



whose source (resp.target) is  $i$ . Then,  $d_j^+$  (resp. $d_i^-$ ) just corresponds to the number of successors (resp.predecessors) of  $i$ . If the graph is weighted,  $d_j^+$  (resp. $d_i^-$ ) is the sum of the weights of edges whose source (resp.target) is  $i$ .

- We note  $w$ , the total weight of the graph, that is the sum of the weights of all the edges.
- For readability, we sometimes abbreviate the edge  $(i, j)$  into  $ij$ .

## 1.3 Graph analysis

In this section, we introduce two research fields centered around the analysis of graphs. While graph theory seeks to describe the properties of graphs as a mathematical object, graph mining is rather aimed at extracting information from datasets that can be represented as a graph. Both fields differ in the methods they use but they do overlap as illustrated by the shortest-path problem which is an historically significant problem from a theoretical standpoint (Ahuja et al., 1990) that has also seen some use in real-world applications (Zhang and Tu, 2009) such as social sciences.

### 1.3.1 Graph theory

Graph theory is concerned with the mathematical properties of graphs. Here, we give a list of some common graph-theoretic problems:

- **Graph classes** There exist many families of graphs defined by some of their properties. For instance, among those families are planar graphs, which can be drawn without any of their edges intersecting. We give an example of a graph having this property in Figure 1.2. For many of those families, characterizing them (Lovász, 1972; Farber, 1983) and being able to identify if a graph belongs to some family (Tarjan and Yannakakis, 1984; Hopcroft and Tarjan, 1974) is of particular interest. One such class is a focal point of Chapter 5.

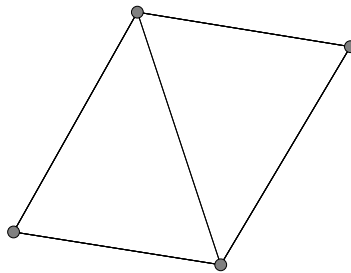


Figure 1.2: A planar graph.

- **Graph coloring** One is interested in finding a coloring of the nodes (i.e. assigning a color to each node of the graph) so that no adjacent nodes have the same color (Jensen and Toft, 2011). While there exists a number of variations of this problem, one of the most well-known results is the four-color theorem (Gonthier et al., 2008) which states that on loopless planar graphs, no more than four colors are required to solve the problem. One simple application of coloring problems occurs when determining if a graph is bipartite or not: a graph is bipartite if, and only if, it has a 2-coloring, each node being colored depending on the node subset it is in (as illustrated in Figure 1.3).

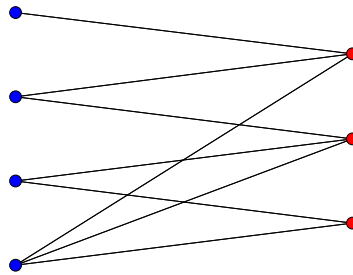


Figure 1.3: 2-coloring of a bipartite graph.

- **Route problems** Route problems are those that aim at finding a path (sometimes a tree) within a graph that satisfies some conditions such as going through each node exactly once or finding the shortest path between a pair of nodes as illustrated in Figure 1.4. They gather famous problems including that of the seven bridges of Königsberg (Euler, 1956) and the Hamiltonian path problem (Gurevich and Shelah, 1987). Route problems have many applications in fields such as computer science (Li et al., 2013) and transportation (Ford Jr, 1956; Zhan and Noon, 1998).

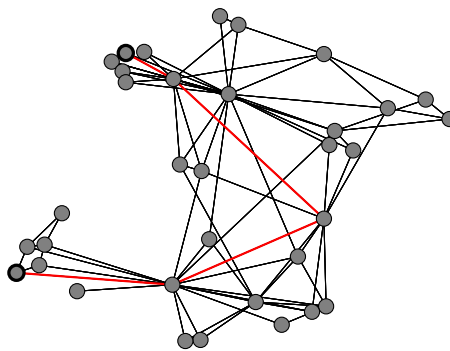


Figure 1.4: Shortest path (in red) between two nodes.

We now give some graph-theoretical definitions that we will use throughout this work

(Golumbic, 2004).

- A graph is *connected* if there is a path between every pair of nodes and *disconnected* otherwise. A *connected component* of a graph is a connected subgraph, maximal for the inclusion.
- A  *$u, v$ -separator* is a subset of the node set that, once removed, disconnects the graph so that  $u$  and  $v$  are in distinct connected components.
- A *cycle*  $\mathcal{C}$  is a path  $\mathcal{C} = (u, v, w, \dots, u)$  from one node to itself of size at least three. No node other than the first and last can be repeated. A *chord* of a cycle is an edge connecting two non-consecutive nodes of a cycle. A cycle without a chord is *induced* or *chordless*.
- A *tree* is a connected graph with no cycles.
- A *clique* of a graph is one of its complete subgraphs.

### 1.3.2 Graph mining

Graph mining is a collection of techniques designed to find the properties of real-world graphs. It consists of data mining techniques used on graphs (Rehman et al., 2012). While this definition hints at some overlap with graph theory, a number of techniques are encountered almost exclusively in a graph mining context. Here, we provide a list of some common graph-mining tasks:

- **Node ranking** Ranking consists in assigning a score to each node of the graph denoting its importance or centrality. There are many ranking methods (Bloch et al., 2019; Landherr et al., 2010) defining a wide range of measures, some based on random walks in the graph, as illustrated by the PageRank ranking (Page et al., 1998) originally designed for web pages; some using solutions to the shortest path problems (Okamoto et al., 2008). An example is given in Figure 1.5. We do not tackle this task in the present work.

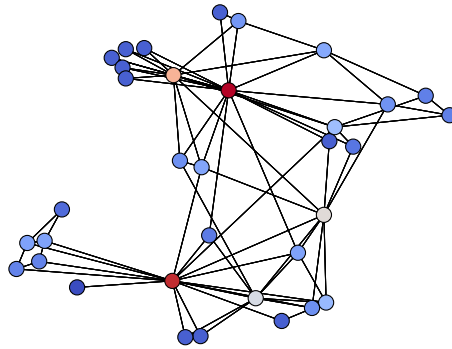


Figure 1.5: Ranking of the nodes using PageRank.

- **Node clustering** Node clustering, also called community detection, gathers methods that look for significant communities in a graph based on its structure. In particular, a good clustering should group nodes so that they are more connected to their own cluster than the others as seen in Figure 1.6. Node clustering has applications in the Internet of Things (Mitra et al., 2012) and for recommendation systems (Fortunato, 2010). Chapter 3 reviews some clustering methods.

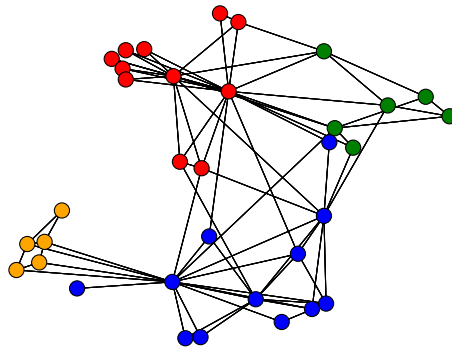


Figure 1.6: Clustering of the nodes using the Louvain method.

- **Node embedding** While the graph formalism is very well suited for representing relational data, the need may arise to represent each node as a vector, this operation is called embedding the nodes of the graph. For instance, the dimension of the embedding space can be set to a low value compared to the number of nodes for dimensionality reduction. It is also a way to apply machine learning techniques that take vectors as input. We illustrate the embedding task in Figure 1.7. Chapter 4 describes an embedding method.

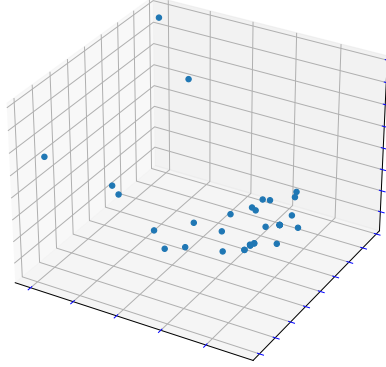


Figure 1.7: 3-dimensional embedding of the nodes using the singular values decomposition.

## 1.4 Outline

This thesis is organized as follows:

- Chapter 2 introduces Scikit-network, a graph analysis library used in our experiments. It was extensively developed in the framework of this thesis and offers a simple API along with competitive performance at scale.
- Chapter 3 reviews some selected node clustering methods. We describe the flat clustering Louvain method and some of its refinements and some hierarchical clustering algorithms.
- Chapter 4 describes a novel node embedding method. It is based on the Louvain clustering method of the previous chapter to tackle large-scale datasets and offer interpretable results.
- Chapter 5 deals with the problem of dataset labeling. The particular setting under study is derived from a real-world problem. We describe the optimal solution for this setting.
- Chapter 6 concludes this thesis.

## 1.5 Publications

Two articles have been accepted during this thesis:

- Bonald, T., de Lara, N., Lutz, Q., & Charpentier, B. (2020). Scikit-network: Graph Analysis in Python. *J. Mach. Learn. Res.*, 21, 185-1. Related to Chapter 2

- Lutz, Q., De Panafieu, E., Stein, M., & Scott, A. (2021). Active clustering for labeling training data. *Advances in Neural Information Processing Systems*, 34. Related to Chapter 5

## Chapter 2

# Scikit-network

The Scikit-network package was created in 2018 by Thomas Bonald and Bertrand Charpentier. The work described in this chapter has been carried out jointly with Nathan de Lara and Thomas Bonald. We refer the reader to the package’s GitHub insights<sup>1</sup> for a rough overview of the respective contributions of the developers. It should be noted however, that a significant portion of the package was developed by several contributors at once (in pair programming fashion) while just one gets to commit to the Git repository. Furthermore, part of the work on the package is not accounted for when looking at the repository only: trying out different features, libraries or continuous integration tools; designing efficient code through careful understanding of scientific articles and learning new skills in software development are all demanding tasks. Overall, it is safe to assume that no one contributor among the core developers had a disproportionate role over the others.

### 2.1 Motivation

Scikit-learn (Pedregosa et al., 2011) is a machine learning package based on the popular Python language. This package is well-established in today’s machine learning community thanks to its versatility, performance and ease of use, making it suitable for both researchers, data scientists and data engineers. Its main assets are the variety of algorithms, the performance of their implementation and their common API.

Scikit-network is a Python package inspired by scikit-learn for graph analysis. The sparse nature of real graphs, with up to millions of nodes, prevents their representation as dense matrices and rules out most algorithms of scikit-learn. Scikit-network takes as input a sparse matrix in the CSR format of SciPy and provides state-of-the-art algorithms for ranking, clustering, classifying, embedding and visualizing the nodes of a graph.

The design objectives of Scikit-network are the same as those having made scikit-learn a success: versatility, performance and ease of use. The result is a Python-native package, like NetworkX (Hagberg et al., 2008), that achieves the state-of-the-art performance of iGraph

---

<sup>1</sup>See <https://github.com/sknetwork-team/scikit-network/graphs/contributors>.

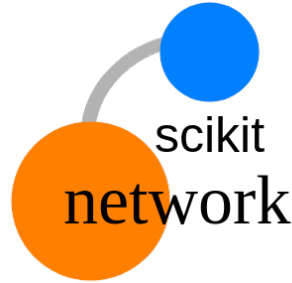


Figure 2.1: Scikit-network logo

(Csardi and Nepusz, 2006), graph-tool (Peixoto, 2014) and NetworKit (Staudt et al., 2016) (see the benchmark in section 2.3.3). Scikit-network uses the same API as Scikit-learn, with algorithms available as classes with the same methods (e.g., `fit`). It is distributed with the BSD license, with dependencies limited to NumPy (Walt et al., 2011) and SciPy (Virtanen et al., 2020).

## 2.2 Software Features

The package is organized in modules with consistent API, covering various tasks:

- **Data.** Module for loading graphs from distant repositories, including Konect (Kunegis, 2013), parsing `tsv` files into graphs, and generating graphs from standard models, like the stochastic block model (Airoldi et al., 2008).
- **Clustering.** Module for clustering graphs, including a soft version that returns a node-cluster membership matrix.
- **Hierarchy.** Module for the hierarchical clustering of graphs, returning dendrograms in the standard format of SciPy. The module also provides various post-processing algorithms for cutting and compressing dendrograms.
- **Embedding.** Module for embedding graphs in a space of low dimension. This includes spectral embedding and standard dimension reduction techniques like SVD and GSVD, with key features like regularization.
- **Ranking.** Module for ranking the nodes of the graph by order of importance. This includes PageRank (Page et al., 1998) and various centrality scores.
- **Classification.** Module for classifying the nodes of the graph based on the labels of a few nodes (semi-supervised learning).
- **Topology.** Module for exploring the structure of the graph: graph traversals,  $k$ -core decomposition, etc.



Modules	NetworkX	iGraph	graph-tool	NetworkKit
Data	✓	✗	✓	✓
Clustering	✓	✓	✗	✓
Hierarchy	✗	✓	✓	✗
Embedding	✓	✗	✓	✗
Ranking	✓	✓	✓	✓
Classification	✓	✗	✗	✗
Topology	✓	✓	✓	✓
Path	✓	✓	✓	✓
Link prediction	✓	✓	✓	✓
Visualization	✓	✓	✓	✓

Table 2.1: Overview of graph software features. ✓: Available. ✓: Partially available or slow implementation. ✗: Not available.

- **Path.** Module relying on SciPy for shortest-paths problems.
- **Link prediction.** Module for assigning a probability of existence of an edge to a node pair based on a partial graph.
- **Visualization.** Module for visualizing graphs and dendrograms in SVG (Scalable Vector Graphics) format. Examples are displayed in Figure 2.2.

These modules are only partially covered by existing graph softwares (see Table 2.1). Another interesting feature of Scikit-network is its ability to work directly on bipartite graphs, represented by their biadjacency matrix.

## 2.3 Practical considerations

### 2.3.1 Efficient graph representation

There exists multiple ways to represent a graph formally. We compare three of the most common ones:

- **Adjacency matrix**, a square matrix  $A_{ij}$  of size  $n$  where  $A_{ij}$  is the weight of the edge  $(i, j)$  (or 1 for unweighted graphs) if it exists and 0 otherwise.
- **Adjacency list**, a list of lists where the  $i$ -th list contains the neighbors of node  $i$ .
- **Incidence matrix**, a rectangular matrix  $B_{ik}$  of shape  $n \times m$  such that  $B_{ik} = 1$  if, and only if, node  $i$  is incident to the  $k$ -th edge.

In a more practical sense, the choice of the representation alone is not enough to assess the performance of some common operations such as finding the weight associated with a particular edge or removing a vertex from the graph. The implementation of any of those representations matters. In particular, the chosen data structure determines how well the

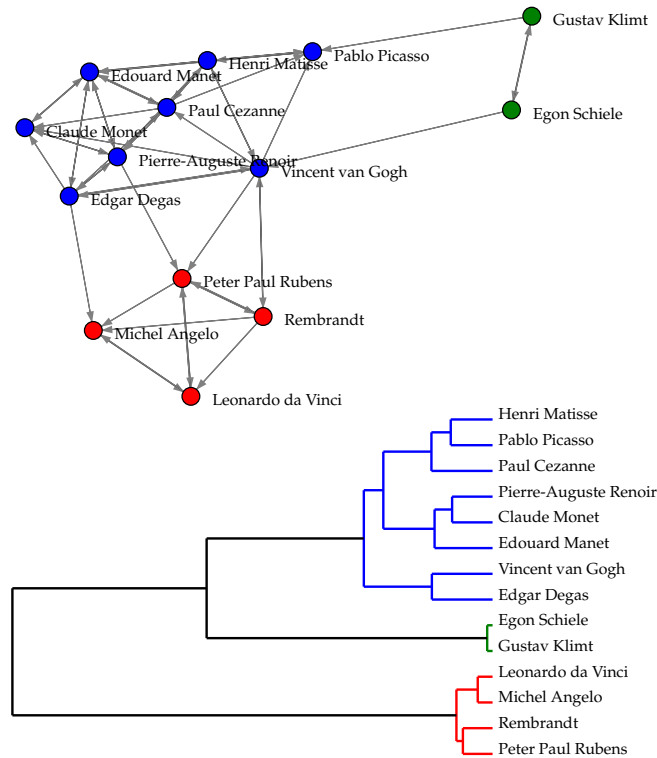


Figure 2.2: Visualization of a graph and a dendrogram as SVG images.

graph can be manipulated. For example, if one makes a naive implementation of the three representations mentioned above using dense C-like (i.e. contiguous in memory so that indexing is implicit) arrays of size  $n^2$  and  $nm$  for the adjacency and incidence matrices respectively, and a list of lists for the adjacency list, their performances are as described in Table 2.2.

Notice that the incidence matrix performs worse than both other representations in all cases, it is thus seldom used in practice. However, there is no absolute winner between the adjacency matrix and adjacency list.

To further explore the available data structures, we consider actual graph datasets. One useful feature of the majority of real-world graphs is their sparsity. Indeed, the number of edges  $m$  is usually very low compared to the number of possible connections  $n^2$ . Given the results displayed in table 2.2, this would favor adjacency lists as a preferred data structure as they are naturally sparse. Nevertheless, there also exists several sparse matrix formats that one can use to represent adjacency matrices. We compare formats that are available in SciPy's sparse module<sup>2</sup> and compare them to an adjacency list implementation using nested Python dictionaries:

<sup>2</sup><https://docs.scipy.org/doc/scipy/reference/sparse.html>

Task	Adjacency matrix	Adjacency list	Incidence matrix
Memory space	$O(n^2)$	$O(n + m)$	$O(nm)$
Check $(u, v) \in E$	$O(1)$	$O(d_u)$	$O(m)$
Add node $u$	$O(n^2)$	$O(1)$	$O(nm)$
Delete node $u$	$O(n^2)$	$O(d_u d_{\max})$	$O(nm)$
Add edge $(u, v)$	$O(1)$	$O(1)$	$O(nm)$
Delete edge $(u, v)$	$O(1)$	$O(d_{\max})$	$O(nm)$
List neighbors of $u$	$O(n)$	$O(1)$	$O(m + nd_u)$

Table 2.2: Complexities of different representations of a graph.  $d_{\max}$  is the maximum degree in the graph.

- The **coordinates (COO)** format consists of three arrays `row`, `col` and `data` of size  $m$  such that for any  $1 \leq i \leq m$ ,  $A_{\text{row}(i), \text{col}(i)} = \text{data}(i)$  and  $A_{jk} = 0$  otherwise. As with all array-based formats, the addition or deletion of edges is costly as it requires to resize the arrays.
- The **dictionary of keys (DOK)** format uses a dictionary where the keys are the edges given as tuples and the associated values are the corresponding weights. Unlike array-based formats, it allows for dynamic changes of the matrix while offering constant average access times to its values. Its downside is the memory space required by its hash table.
- The **compressed sparse row (CSR)** (respectively **compressed sparse column (CSC)**) format implicitly indexes the source (resp. target) nodes of every edge but explicitly indexes their target (resp. source) nodes. It consists of three arrays. In the case of the CSR format, two arrays `indices` and `data` of size  $m$  contain the target nodes and the weights of the edges respectively. The third array `indptr` has size  $n + 1$  and delimits the successors of each source node. For instance, the neighbors of node  $i$  are found in the `indices` array at positions `indptr(i)` to `indptr(i + 1)` which can be computed in constant time. Conversely, in the CSC format, `indptr` delimits the predecessors of each target node. Additionally, algebraic operations such as matrix/vector dot products are very efficient when using the CSR/CSC format.
- The **dictionary of dictionaries (DOD)** format is not found in SciPy. It is based on nested Python dictionaries: a main dictionary has nodes as keys and dictionaries as values. Additionally, an index is maintained to allow for the addition and deletion of nodes. Each nested dictionary has the neighbors of the corresponding node as keys and the weights of the associated edges as values. Unlike the DOK format, this makes it possible to have constant-time access to the neighborhood of a node by looking at the keys of the corresponding nested dictionary. This comes at the expense of increased memory requirements due to the numerous hash tables of the nested dictionaries. One notable downside is that this format is ill-suited for dot products.

Each format is illustrated on a toy graph depicted in Figure 2.3. In order to compare all the formats described above, we use each format to store various graphs and gather

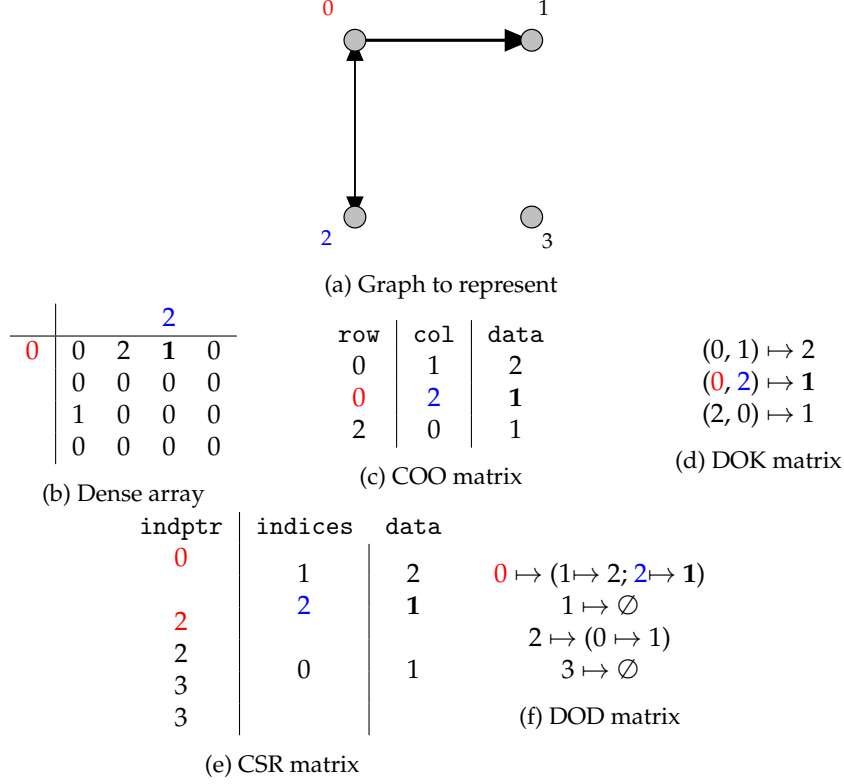


Figure 2.3: A graph and its different representations.  $\mapsto$  denotes the association of a key to a value via a hash function. Edge (0, 2) and its associated weight are also highlighted.

the amount of memory space they require in Table 2.3. We also report the execution times for some common graph operations in Table 2.4 using the WikiVitals graph. In the latter table, the COO format is absent as it does not allow most of the listed tasks to be performed (including slicing). It should also be noted that the tasks under study are by no means exhaustive. In particular, a common downside of all the formats at hand is that they are not suited for extracting subgraphs like dense matrices are.

Graph	$n$	$m$	CSR	COO	DOK	DOD
WikiSchools	$4.10^3$	$1.10^5$	<b><math>6.10^2</math></b>	$1.10^3$	$4.10^3$	$2.10^3$
WikiVitals	$1.10^4$	$8.10^5$	<b><math>4.10^3</math></b>	$7.10^3$	$3.10^4$	$1.10^4$
WikiLinks	$3.10^6$	$7.10^7$	<b><math>3.10^5</math></b>	$6.10^5$	$2.10^6$	$1.10^6$

Table 2.3: Memory usage (in kB) of different formats for various graphs.

In Table 2.4, notice how the DOD format benefits from the efficiency of Python dictionaries and is the overall fastest option for many tasks under study. The CSR format has a similar albeit slightly worse performance, except for listing neighbors and matrix-

Task	CSR	DOK	DOD
Check $(u, v) \in E$	$3.10^{-5}$	$2.10^{-5}$	<b><math>6.10^{-6}</math></b>
Add node $u$	$5.10^{-5}$	$2.10^{-4}$	<b><math>2.10^{-6}</math></b>
Delete node $u$	<b><math>3.10^{-3}</math></b>	$7.10^1$	$5.10^{-3}$
Add edge $(u, v)$	$5.10^{-3}$	$2.10^{-5}$	<b><math>6.10^{-6}</math></b>
Delete edge $(u, v)$	$1.10^{-3}$	$3.10^{-5}$	<b><math>1.10^{-5}</math></b>
List neighbors of $u$	<b><math>2.10^{-6}</math></b>	$4.10^{-3}$	$1.10^{-4}$
Matrix-vector product	<b><math>2.10^{-3}</math></b>	$2.10^0$	$1.10^0$

Table 2.4: Execution times (in seconds) of different formats on the WikiVitals graph.

vector products, while the DOK format performs significantly worse than either on some tasks. The DOD format is especially well-suited for adding and deleting edges and nodes alike. On these tasks it is about one order of magnitude faster than the CSR and DOK formats except for the deletion of nodes. As found in Table 2.3, the CSR format is the most memory-efficient format especially when compared to its dictionary-based counterparts. In choosing a format for our experiments, the CSR format is our preferred option because it offers the best memory footprint and is very well-suited for algebraic operations while also having a good performance for selected graph primitives. The CSR format formally represents the adjacency matrix of the graph, however, it also retains a number of characteristics expected from adjacency list and incidence matrix representations. Indeed, like adjacency lists, it makes it possible to access the neighborhood of a node in constant time. On the other hand, like incidence matrices, it is easy to find if node  $i$  is adjacent to the  $j$ -th edge provided that the ordering of the edges is that of the `indices` array.

### 2.3.2 Guidelines for the package

In designing Scikit-network, we set out a number of goals and features we sought to achieve. We mention them below and describe how they impact day-to-day work on the package.

**Open-source software.** The package is hosted on GitHub<sup>3</sup> and is part of SciPy kits aimed at creating open-source scientific software. Its BSD license enables maximum interoperability with other software. Contributions are encouraged and guidelines for contributing are described in the documentation of the package<sup>4</sup> and guidance is provided by the GitHub-hosted Wiki.

**Sustainable performance.** Scikit-network relies on a very limited number of external dependencies for ease of installation and maintenance. Only SciPy and NumPy are required on the user side.

<sup>3</sup>See <https://github.com/sknetwork-team/scikit-network>.

<sup>4</sup>See <https://scikit-network.readthedocs.io/en/latest/>.

- Many elements from **SciPy** are used for both high performance and simple code. The sparse matrix representations allow for efficient manipulation of large graphs as discussed in Chapter 2 while linear algebra solvers are used in many algorithms. Scikit-network also relies on the *LinearOperator* class for efficient implementation of certain algorithms.
- **NumPy** arrays are used through SciPy’s sparse matrices for memory-efficient computations. NumPy is used throughout the package for the manipulation of arrays. Some inputs and most of the outputs are given in the NumPy array format.
- In order to speed up execution times, **Cython** (Behnel et al., 2011) generates C++ files automatically using a Python-like syntax. Thanks to the Python wheel system, no compilation is required from the user on most platforms. Note that Cython has a built-in module for parallel computing on which Scikit-network relies for some algorithms. Otherwise, it uses Python’s native multiprocessing.

Note that all three packages are well-established within the Python ecosystem. Their mostly stable APIs make it possible to avoid unnecessary changes to the code.

**Code quality and availability.** To ensure that the library is bug-free, a number of tests are defined for each new algorithm and feature. Code quality is then assessed by standard code coverage metrics (Bacchelli and Bird, 2013): the coverage is the proportion of code lines that are executed when running the tests. Today’s coverage is at 99% for the whole package. Note that this coverage metric is a rough one (in particular, it does not denote how relevant tests are). In practice, special attention is given to identifying and testing pathological cases when implementing new features.

As a general rule of thumb, the package should work with the latest versions of its dependencies in order to avoid the use of obsolete features. Requirements are thus kept up to date thanks to IDE assistance. Scikit-network relies on GitHub Actions for continuous integration (i.e. testing and deploying code) and cibuildwheel for deploying on common platforms (i.e. building wheels for each platform so that no compilation is required on the user side). OSX, Windows and most Linux distributions (McGibbon and Smith, 2016) are supported for Python versions 3.7 and newer.

The amount of work spent ensuring that coverage remains high and that the package is available on all platforms cannot be understated.

**Documentation.** Scikit-network is provided with a complete documentation<sup>4</sup>. The API reference presents the syntax while the tutorials present applications on real graphs. Algorithms are documented with relevant formulas, specifications, examples and references, when relevant. This documentation aims at enabling users to have a quick grasp of the usage of each function. The quality of the documentation is easily as important as that of the code itself as the former is the main enabler for a proper use of the latter.

The documentation is built from structured comments in the code itself and files describing how the documentation is organized. Tutorials are built from Jupyter notebooks which are executed when the documentation is being built. All functions and classes of

Scikit-network are documented, most appear in the documentation and all methods have a tutorial attached.

**Code readability.** The source code follows the stringent PEP8 guidelines. Explicit variable naming and type hints make the code easy to read. The number of object types is kept to a minimum. Those readability guidelines aim at facilitating the review and addition of code to the package. Indeed, it makes it easier to fix faulty code or to repurpose it. In particular, when new contributions are made, it provides a common set of conventions for newcomers while allowing reviewers to provide feedback faster.

**Data collection.** The package offers multiple ways to fetch data. Some small graphs are embedded in the package itself for testing or teaching purposes. Part of the API makes it possible to fetch data from selected graph databases easily. Both of these options are of particular interest when implementing or benchmarking new algorithms or showcasing existing ones. Parsers are also present to enable users to import their own data and save it in a convenient format for later reuse. Those parsers make it possible to apply the algorithms of Scikit-network on proprietary data.

With this goal of broad data collection in mind, we also introduce the NetSet repository. Although not part of the Scikit-network package, the NetSet<sup>5</sup> repository is one of the two databases with a dedicated import function available (the other one being Konect). It is also maintained by the Scikit-network team. It gathers some common networks of the literature such as 20NewsGroup but also some original datasets such as the US Senate, US House of Representatives and French National Assembly graphs. Unlike imports from Konect that require at least an initial parsing of TSV files, importing datasets from the NetSet repository benefits from downloading files in the dedicated NumPy, SciPy and Pickle formats which allows for gains in both memory (and thus download size) and overall runtimes for the import command.

### 2.3.3 Performance

To show the performance of Scikit-network, we compare the implementation of some representative algorithms with those of the graph softwares of Table 2.1:

- the Louvain clustering algorithm (Blondel et al., 2008), with the tolerance parameter set at  $10^{-3}$  when available
- PageRank (Page et al., 1998), the number of iterations is set to 100 when possible (that is for all packages except iGraph)
- HITS (Kleinberg, 1999)
- the spectral embedding (Belkin and Niyogi, 2003) where dimension of the embedding space is set to 16

---

<sup>5</sup>See <https://netset.telecom-paris.fr/>

	scikit-network	NetworkX	iGraph	graph-tool
Louvain	771	✗	1,978	✗
PageRank	48	🔥	236	45
HITS	109	🔥	80	144
Spectral	534	🔥	✗	✗

Table 2.5: Execution times (in seconds). ✗: Not available. 🔥: Memory overflow.

	scikit-network	NetworkX	iGraph	graph-tool
RAM usage	1,222	🔥	17,765	10,366

Table 2.6: Memory usage (in MB). 🔥: Memory overflow.

Table 2.5 gives the running times of these algorithms on the *Orkut* graph of Konect (Kunegis, 2013). The graph has 3,072,441 nodes and 117,184,899 edges. The computer has a Debian 10 OS and is equipped with an AMD Ryzen Threadripper 1950X 16-Core Processor and 32 GB of RAM. Note that NetworkKit is absent as we could not get it to work on our test machine. As we can see, Scikit-network is highly competitive.

We also give in Table 2.6 the memory usage of each package when loading the graph. Thanks to the CSR format, Scikit-network has a minimal footprint.

## 2.4 Conclusion

We introduced Scikit-network, a Python library for graph analysis. We illustrated how competitive it was compared to other libraries without sacrificing its ease-of-use and described the main guidelines we set when developing it. It should be noted that the performance of Scikit-network does come with one notable string attached: all operations are done in RAM and it is expected that RAM is the limiting factor to what can be achieved with the package. Nevertheless, thanks to the efficient CSR format, even a common laptop can handle large graphs as seen in the experiments. Machines with hundreds of gigabytes of RAM (like the one used in the experiments of Chapter 4), while uncommon, are not unheard of and make it possible to tackle real-world graphs with billions of edges.

In the future, the library could benefit from a new graph format to either overcome some of the downsides of the CSR format mentioned in Chapters 3 and 4 or tackle new problems. For instance, the topic of dynamic graphs (i.e. graphs whose nodes and edges may appear and disappear over time) is of particular interest. Results displayed in Section 2.3.1 then suggest that formats such as the DOD one may be better suited.



## Chapter 3

# Node clustering

### 3.1 Introduction

Node clustering is a common task in graph mining. It consists in partitioning the node set  $V$  of a graph  $G = (V, E)$  in such a way that nodes in the same group are closer in some sense than to those in other groups. Each such group is called a cluster or community and clustering is also known as community detection. In this chapter we review different forms of clustering, defined by their outputs:

- *flat* clustering, where each node is assigned exactly one cluster. In this case, we review selected variations of the Louvain method (Blondel et al., 2008), a well-known algorithm for flat node clustering. We compare those and introduce a novel variant.
- *hierarchical* clustering, where each node is a leaf in a tree designed so that close leaves correspond to close nodes in the original graph.

#### 3.1.1 Node sampling

A natural notion of proximity between nodes is through node sampling. Let  $A$  be the adjacency matrix of an undirected graph. Consider the sampling of node pairs through the edges. Each (ordered) node pair  $(i, j)$  is then sampled with probability:

$$s(i, j) = \frac{A_{ij}}{2w}$$

where  $w$  is the total weight of the graph. This is a symmetric joint distribution with marginal distribution:

$$s(i) = \sum_{j \in V} s(i, j) = \frac{d_i}{2w}$$

Then, the probability of sampling  $j$  given that  $i$  is sampled is:

$$s(j|i) = \frac{s(i,j)}{s(i)}$$

We say that node  $j$  is close to node  $i$  if the probability of sampling node  $j$  given the sampling of node  $i$  is much higher than the probability of sampling node  $j$ . This yields the following similarity for nodes  $i$  and  $j$ :

$$\sigma(i,j) = \frac{s(j|i)}{s(j)} = \frac{s(i,j)}{s(i)s(j)}$$

Note that  $\sigma$  is symmetric. This sampling distribution and the derived similarity are used in the following sections.

## 3.2 Flat clustering

The flat clustering of a graph  $G$  is given as a partition of its node set  $V$ . Formally, the clustering of  $G$  is  $(\mathcal{C}_k)_{k \in C}$  such that,  $\forall k \neq l, \mathcal{C}_k \cap \mathcal{C}_l = \emptyset$  and  $\bigcup_k \mathcal{C}_k = V$ . Conversely, we define the cluster  $c_i$  of any node  $i \in V$  as  $c_i = k$  if  $i \in \mathcal{C}_k$ .

In this section, we will focus on the Louvain method, the modularity functions it optimizes and its variants.

### 3.2.1 Similarity measures

In order to assess the quality of a clustering, it is useful to compare it to the ground truth of the dataset under study whenever available. This requires to be able to compare any pair of clusterings. While it is trivial to check if two clusterings are equal, quantifying how different they are is more difficult.

To that end, we will go over:

- the Rand index measure defined by Rand (1971)
- the homogeneity and completeness scores defined by Rosenberg and Hirschberg (2007)

Note that the choice of those two similarity measures is far from exhaustive (Vinh et al., 2010; Yang et al., 2016).

We denote the ground truth by  $\mathcal{G}$  and a clustering to which it should be compared by  $\mathcal{C}$ . We say that  $i$  is in *class* (resp. *cluster*)  $k$  if  $i \in \mathcal{G}_k$  (resp.  $i \in \mathcal{C}_k$ ). Let  $n$  be the number of nodes, we pick a random node  $X$  and consider the random variables  $g_X$  and  $c_X$ , the class and the cluster of  $X$ .

**Homogeneity and completeness** One way to assess how close  $\mathcal{C}$  is to  $\mathcal{G}$  is to check that any cluster corresponds to at most one class. Whenever this is the case, we say that  $\mathcal{C}$  is homogeneous with respect to  $\mathcal{G}$ .

**Definition 1.**  $\mathcal{C}$  is homogeneous with respect to  $\mathcal{G}$ , if  $\forall l, \exists! k, \mathcal{C}_l \subset \mathcal{G}_k$ .

Conversely, another way to assess how close  $\mathcal{C}$  is to  $\mathcal{G}$  is to check that any class corresponds to at most one cluster. This property is called completeness and is immediately linked with homogeneity by switching the roles of  $\mathcal{C}$  and  $\mathcal{G}$ .

**Definition 2.**  $\mathcal{C}$  is complete with respect to  $\mathcal{G}$ , if  $\mathcal{G}$  is homogeneous with respect to  $\mathcal{C}$ .

Note that if  $\mathcal{C}$  is both complete and homogeneous with respect to  $\mathcal{G}$ , then  $\mathcal{C}$  and  $\mathcal{G}$  are identical (up to their numbering).

**Rand index** The Rand index is defined as the proportion of node pairs  $(i, j)$  whose classes and clusters are either both equal, i.e.  $c_i = c_j$  and  $g_i = g_j$  or different, i.e.  $c_i \neq c_j$  and  $g_i \neq g_j$ . By definition, the Rand index is thus bounded by 0 (when  $\mathcal{C}$  and  $\mathcal{G}$  do not agree on a single node pair) and 1 (when the two clusterings are identical).

We give a closed formula for the Rand index  $RI$ :

**Proposition 1.**

$$\binom{n}{2} RI = \binom{n}{2} + \sum_{k=1}^{|\mathcal{G}|} \sum_{l=1}^{|\mathcal{C}|} |\mathcal{G}_k \cap \mathcal{C}_l|^2 - \frac{1}{2} \left( \sum_{k=1}^{|\mathcal{G}|} |\mathcal{G}_k|^2 + \sum_{l=1}^{|\mathcal{C}|} |\mathcal{C}_l|^2 \right)$$

*Proof.* We use Kroenecker symbols to count pairs where  $\mathcal{C}$  and  $\mathcal{G}$  agree:

$$\begin{aligned} \binom{n}{2} RI &= \sum_{1 \leq i < j \leq n} \left( \delta_{g_i g_j} \delta_{c_i c_j} + (1 - \delta_{g_i g_j})(1 - \delta_{c_i c_j}) \right) \\ 2 \binom{n}{2} RI + n &= \sum_{1 \leq i, j \leq n} \left( 1 + 2\delta_{g_i g_j} \delta_{c_i c_j} - \delta_{g_i g_j} - \delta_{c_i c_j} \right) \end{aligned}$$

Note that  $\sum_{1 \leq i, j \leq n} \delta_{g_i g_j} = \sum_{1 \leq i, j \leq n} \sum_{k=1}^{|\mathcal{G}|} \delta_{g_i k} \delta_{g_j k} = \sum_{k=1}^{|\mathcal{G}|} |\mathcal{G}_k|^2$ . Using similar reasoning for the two other sums yields:

$$2 \binom{n}{2} RI + n = n^2 + 2 \sum_{k=1}^{|\mathcal{G}|} \sum_{l=1}^{|\mathcal{C}|} |\mathcal{G}_k \cap \mathcal{C}_l|^2 - \sum_{k=1}^{|\mathcal{G}|} |\mathcal{G}_k|^2 - \sum_{l=1}^{|\mathcal{C}|} |\mathcal{C}_l|^2$$

Hence the result.  $\square$

The Rand index gives an easily interpretable result thanks to a very simple definition. One noticeable downside of the Rand index is that two random clusterings may have a high score. If their number of clusters is high for instance, all but the  $\binom{n}{2}$  term in Proposition 1 vanish. To correct this issue, one can account for chance by subtracting the expected value of the Rand index (Hubert and Arabie, 1985). This measure is called the adjusted Rand index (ARI).

**V-measure** Homogeneity translates nicely in terms of entropy as  $\mathcal{G}$  is then entirely determined by  $\mathcal{C}$ . Let:

$$H(g_X|c_X) = - \sum_{k=1}^{|\mathcal{G}|} \sum_{l=1}^{|\mathcal{C}|} \frac{|\mathcal{G}_k \cap \mathcal{C}_l|}{n} \log \frac{|\mathcal{G}_k \cap \mathcal{C}_l|}{|\mathcal{C}_l|}$$

When  $\mathcal{C}$  is homogeneous with respect to  $\mathcal{G}$ ,  $H(g_X|c_X) = 0$ . This is normalized so that we have a score between 0 and 1 with 1 being reached whenever there is homogeneity. As  $H(g_X|c_X)$  is bounded by 0 and  $H(g_X)$  (whenever  $g_X$  and  $c_X$  are independent), we define the homogeneity score of  $\mathcal{C}$  with respect to  $\mathcal{G}$  as  $h_{\mathcal{G}}(\mathcal{C}) = 1 - \frac{H(g_X|c_X)}{H(g_X)}$ . Note that the trivial partition where each node is in its own cluster is homogeneous with respect to any partition.

Conversely, we define the completeness score  $m_{\mathcal{G}}(\mathcal{C})$  of  $\mathcal{C}$  with respect to  $\mathcal{G}$  as  $m_{\mathcal{G}}(\mathcal{C}) = h_{\mathcal{C}}(\mathcal{G})$ . Note that the trivial partition where all nodes are in the same cluster is complete with respect to any partition.

The V-measure score is defined as the harmonic mean of the homogeneity and the completeness scores.

### 3.2.2 Modularity functions

In this section, we define the different modularity functions used for graph clustering and present some of their properties.

Consider a graph  $G = (V, E)$ , a probability distribution  $p$  over  $V \times V$  and two probability distributions  $q^r$  and  $q^c$  over  $V$  (for the rows and columns). A modularity function, given a resolution parameter  $\gamma$ , can be written in its general form as:

$$Q = \sum_{i,j \in V} \left( p_{ij} - \gamma q_i^r q_j^c \right) \delta_{c_i c_j} \quad (3.1)$$

The modularity can then be interpreted as the difference between the probability of sampling a node pair inside a cluster using an edge sampling distribution and the probability of sampling the same pair using two node samplings. Maximizing the first term has a trivial solution in which all nodes are in one cluster. On the other hand, minimizing the second term has a trivial solution in which there are as many clusters as there are nodes (i.e. one node per cluster). The modularity is thus a balance between those two competing terms. In the literature, the resolution parameter may be absent, this is equivalent to having  $\gamma = 1$ . Equation (3.1) can be rewritten into an aggregated form by summing cluster by cluster:

$$Q = \sum_{k \in \mathcal{C}} (P_k - \gamma Q_k) \quad (3.2)$$

where  $P_k = \sum_{i,j \in \mathcal{C}_k} p_{ij}$  is the probability of sampling a pair of nodes in cluster  $k$  according to distribution  $p$ . Likewise, we define  $Q_k = \sum_{i,j \in \mathcal{C}_k} q_i^r q_j^c$ .

Note that there exists other modularity functions than the ones we list in this section. In particular, some may not be written in the form of Equation (3.1) (Campigotto et al., 2014).

**Undirected graphs.** One of the most simple choices of  $p, q^r$  and  $q^c$  for undirected graphs is  $p_{ij} \propto \mathbb{1}_{\{ij \in E\}}$  and  $q_i^r, q_i^c \propto 1$ , which gives the following aggregated form:

$$Q_P = \sum_{k \in C} \left( \frac{|E_k|}{m} - \gamma \frac{|C_k|^2}{n^2} \right) \quad (3.3)$$

where  $E_k$  is the set of edges between nodes of  $C_k$ .

We call this function the Potts modularity because of its link to the constant Potts model. Note that maximizing  $Q_P$  is equivalent to maximizing  $\sum_{k \in C} \left( |E_k| - \gamma \frac{m}{n^2} |C_k|^2 \right)$ . Traag et al. (2011) suggests using  $\gamma' = \gamma \frac{m}{n^2}$ . They describe the following aggregated form:

$$Q_T = \sum_{k \in C} \left( |E_k| - \gamma' |C_k|^2 \right) \quad (3.4)$$

This can be interpreted depending on  $\gamma'$ .  $|C_k|^2$  is the maximum number of edges that can be found within cluster  $k$ , for  $0 \leq \gamma' \leq 1$ ,  $\gamma' |C_k|^2$  is the expected number of edges within that cluster and  $\gamma'$  is the target density inside each cluster (whereas  $\frac{m}{n^2}$  is the mean density for the whole graph).  $\gamma$  denotes how much bigger the intra-cluster density should be compared to the mean density. We typically need  $\gamma > 1$ .

The most common choice of  $p, q^r$  and  $q^c$  for undirected graphs is found in Newman (2006). They are set so that  $p_{ij} \propto A_{ij}$  and  $q_i^r, q_i^c \propto d_i$  (recall that whenever the graph is weighted,  $d_i$  denotes the sum of the weights of edges incident to node  $i$  rather than its degree):

$$Q_N = \sum_{i,j \in V} \left( \frac{A_{ij}}{2w} - \gamma \frac{d_i d_j}{(2w)^2} \right) \delta_{c_i c_j} \quad (3.5)$$

Note that the latter modularity and that of Equation (3.3) coincide on unweighted regular graphs (i.e. unweighted graphs where all nodes have the same degree). In light of the sampling distribution defined in Section 3.1.1, we rewrite  $Q_N$  as:

$$Q_N = \sum_{i,j \in V} (s(i, j) - \gamma s(i) s(j)) \delta_{c_i c_j}$$

Intuitively, the modularity is high whenever the probability of sampling an edge whose ends are both within a cluster is higher than picking two nodes within that cluster independently. In particular, this is expected when there are many edges within clusters compared to the number of edges between distinct clusters. Alternatively, the aggregate form of Equation (3.5) is:

$$Q_N = \sum_{k \in C} \left( \frac{w_k}{2w} - \gamma \left( \frac{v_k}{2w} \right)^2 \right)$$

with  $w_k = \sum_{i,j \in C_k} A_{ij}$  and  $v_k = \sum_{i \in C_k} d_i$ , the volume of cluster  $k$ . Notice that the first term  $\sum_{k \in C} \frac{w_k}{2w}$  is exactly the proportion of the edge weights that is within the clusters. The second term  $\sum_{k \in C} \left( \frac{v_k}{2w} \right)^2$  is minimal when all clusters have the same volume (it is then equal to the inverse of the number of clusters) and maximal when there is only one cluster which

has volume  $v_k = 2w$  (it is then equal to 1).

**Directed graphs.** The modularity defined in Equation (3.5) is naturally extended to directed graphs. To do so, Dugué and Perez (2015) suggests  $p_{ij} \propto A_{ij}$ ,  $q_i^r \propto d_i^+$  and  $q_i^c \propto d_i^-$ :

$$Q_D = \sum_{i,j \in V} \left( \frac{A_{ij}}{2w} - \gamma \frac{d_i^+ d_j^-}{(2w)^2} \right) \delta_{c_i c_j} \quad (3.6)$$

**Bipartite graphs.** Likewise, in the case of bipartite graphs, Barber (2007) proposed to use  $p_{ij} \propto B_{ij}$ ,  $q_i^r \propto d_i^r$  and  $q_i^c \propto d_i^c$  where  $(i, j) \in V_1 \times V_2$  and  $d^r$  and  $d^c$  are the degree vectors for the rows and the columns of the biadjacency respectively. This leads to the bimodularity function:

$$Q_B = \sum_{i \in V_1, j \in V_2} \left( \frac{B_{ij}}{2w} - \gamma \frac{d_i^r d_j^c}{(2w)^2} \right) \delta_{c_i c_j} \quad (3.7)$$

Let

$$A = \begin{pmatrix} 0 & B \\ 0 & 0 \end{pmatrix} \in \mathcal{M}_{n_1+n_2}(\mathbb{R})$$

be the adjacency matrix of the graph, seen as a directed one. The bimodularity on  $B$  coincides with the directed modularity defined in Equation (3.6) on  $A$  as the in- and out-degree  $d^-$  and  $d^+$  of  $A$  can be written respectively as  $\begin{pmatrix} 0 \\ d^c \end{pmatrix}$  and  $\begin{pmatrix} d^r \\ 0 \end{pmatrix}$ .

The Newman modularity of Equation 3.5 can also be used on bipartite graphs (without taking the particular structure of the graph into account). The corresponding undirected graph has the following adjacency matrix:

$$A = \begin{pmatrix} 0 & B \\ B^T & 0 \end{pmatrix} \in \mathcal{M}_{n_1+n_2}(\mathbb{R})$$

**Aggregation properties.** We now define the aggregation of a graph  $G = (V, E)$  along a clustering  $\mathcal{C}$ . This results in a graph  $\hat{G} = (\hat{V}, \hat{E})$  where  $\hat{V} = \{1, \dots, |\mathcal{C}|\}$  (i.e., there is one node in the aggregated graph for each cluster in  $\mathcal{C}$ ) and the weight of edge  $(i, j)$  is  $\hat{A}_{kl} = \sum_{i,j \in \mathcal{C}_k \times \mathcal{C}_l} A_{ij}$ . Modularity functions are all compatible with aggregation based on a clustering of the graph. The corresponding conservation property is key to the efficiency of the Louvain method described in Section 3.2.3 (general conditions for a good integration with Louvain are discussed in Campigotto et al. (2014)). Here, we give a generalization of that property. Recall the homogeneity property defined in Definition 1. Let  $\mathcal{C}$  and  $\mathcal{C}'$  be two clusterings of a graph, if  $\mathcal{C}'$  is homogeneous with respect to  $\mathcal{C}$ ,  $\mathcal{C}$  can also be aggregated along  $\mathcal{C}'$ . This results in a partition  $\hat{\mathcal{C}}$  of  $|\mathcal{C}'|$ -sets such that each element in  $\hat{\mathcal{C}}_k$  corresponds to a cluster of  $\mathcal{C}'$  that is included in  $\mathcal{C}_k$ .

**Proposition 2.** *Given a graph  $G = (V, E)$  and two clusterings  $\mathcal{C}$  and  $\mathcal{C}'$  of  $G$  such that  $\mathcal{C}'$  is homogeneous with respect to  $\mathcal{C}$ , consider the graph  $G' = (V', E')$  and the clustering  $\hat{\mathcal{C}}$  of  $G'$  obtained*

by aggregating  $G$  and  $C$  along  $C'$ . The modularity of clustering  $C$  on  $G$  is equal to the modularity of  $\hat{C}$  on  $G'$ .

*Proof.* Let  $Y' \in \mathbb{R}^{n \times |C'|}$  (resp.  $\hat{Y} \in \mathbb{R}^{|C'| \times |\hat{C}|}$ ) be the membership matrix for the clustering  $C'$  (resp.  $\hat{C}$ ), i.e.  $Y'_{ik} = 1$  if node  $i \in V$  is in cluster  $C'_k$  (resp.  $\hat{C}_k$ ) and 0 otherwise. Note that the membership matrix  $Y$  of  $C$  satisfies  $Y = Y' \hat{Y}$ . Let  $A' = Y'^T A Y'$  be the adjacency matrix of  $G'$ . To simplify notations, let  $d$  (resp.  $f$ ) denote the out-weight (resp. in-weight) vector of  $A$  and let  $w$  denote the total weight of the graph. We have:

$$\begin{aligned} d' &= A' \vec{1} = Y'^T A Y' \vec{1} = Y'^T d \\ f' &= A'^T \vec{1} = Y'^T A^T Y' \vec{1} = Y'^T f \\ w' &= \vec{1}^T A' \vec{1} = \vec{1}^T Y'^T A Y' \vec{1} = \vec{1}^T A \vec{1} = w \end{aligned}$$

Hence,  $p'_{ij} = (Y'^T p Y')_{ij}$  and  $q'^r_i q'^c_j = (Y'^T q^r q^{cT} Y')_{ij}$  hold for all the probability distributions listed above:

$$\begin{aligned} Q(C) &= \sum_{i,j \in V} (p_{ij} - \gamma q^r_i q^c_j) \delta_{c_i c_j} \\ &= \sum_{i,j \in V} (p_{ij} - \gamma q^r_i q^c_j) Y_i^T Y_j \\ &= \text{Tr} \left( Y^T (p - \gamma q^r q^{cT}) Y \right) \\ &= \text{Tr} \left( \hat{Y}^T Y'^T (p - \gamma q^r q^{cT}) Y' \hat{Y} \right) \\ &= \text{Tr} \left( \hat{Y}^T (Y'^T p Y' - \gamma Y'^T q^r q^{cT} Y') \hat{Y} \right) \\ &= \sum_{i,j \in V'} (p'_{ij} - \gamma q'^r_i q'^c_j) \hat{Y}_i^T \hat{Y}_j \\ &= \sum_{i,j \in V'} (p'_{ij} - \gamma q'^r_i q'^c_j) \delta_{\hat{c}_i \hat{c}_j} \end{aligned}$$

□

Property 2 is used in Section 3.2.4. In particular, Figure 3.1 illustrates two partitions of the nodes such that one (in 3.1b) is homogeneous to the other (in 3.1a) and how the graph and the initial partition are partially aggregated (in 3.1d) along a refined partition.

The following property is an immediate consequence of the former for  $C = C'$ :

**Proposition 3.** *Given a graph  $G = (V, E)$  and a clustering  $C$  of  $G$ , consider the graph  $\hat{G} = (\hat{V}, \hat{E})$  obtained by aggregating along  $C$ . The modularity of clustering  $C$  on  $G$  is equal to the modularity of the trivial partition where each node is in its own cluster on  $\hat{G}$ .*

Proposition 3 is the commonly given conservation property for the Louvain method.

### 3.2.3 Louvain method

We now describe the Louvain method (Blondel et al., 2008) and how the modularity functions described above allow it to be a scalable approach to clustering. When it was first introduced, only the modularity for undirected networks from Equation (3.5) was mentioned. However, the general idea can be applied to any type of objective function that satisfies the aggregation property as described in Proposition 3 and that allows for the variation of its value due to a cluster change to be computed easily.

The Louvain method consists of two phases: optimization and aggregation (described respectively in Algorithms 1 and 2). In Algorithm 1,  $\text{MOD}$  denotes the modularity function and  $\text{MOVE}(G, \mathcal{C}, u, k)$  denotes the partition of  $G$  obtained from  $\mathcal{C}$  by setting  $c_u$  to  $k$ . In Algorithm 2,  $\text{MEMBERSHIP}(\mathcal{C})$  is the membership matrix  $Y$  for some partition  $\mathcal{C}$ , i.e.  $Y_{iu} = 1$  if node  $i$  is in cluster  $u$  under  $\mathcal{C}$  and 0 otherwise.

The optimization phase consists in visiting all the nodes of the graph repeatedly and moving them to a neighboring community whenever it induces an increase in modularity greater than some tolerance  $\epsilon$ . The tolerance parameter  $\epsilon$  is the only one that needs to be set. Empirically, we find that  $\epsilon = 10^{-3}$  offers a reasonable trade-off between the quality of the results and time performance. It is the value that is used in the experiments. It should be noted that the optimization is deterministic up to the order in which the nodes are visited.

After optimization, a partition  $\mathcal{C}$  which locally optimizes the modularity is obtained. The next step is to aggregate  $G$  with respect to  $\mathcal{C}$ . Once the graph is aggregated, another optimization phase begins and the two phases are repeated until aggregation does not change the graph.

The modularity increases with each local update performed in the optimization phase and remains the same during the aggregation phase according to Proposition 3. As the modularity is upper-bounded, the Louvain method converges towards a local maximum of the modularity.

The efficiency of the Louvain method relies on easily finding the best local cluster for each node (i.e. the cluster that induces the largest increase in modularity). This, in turn, depends on having a simple local update formula for the modularity when node  $i$  is moved from cluster  $c_i$  to  $c_j$ . The corresponding variation in modularity  $\Delta Q$  satisfies:

$$\begin{aligned}\Delta Q &= \sum_k (\Delta P_k - \gamma \Delta Q_k) \\ &= (\Delta P_{c_i} - \gamma \Delta Q_{c_i}) + (\Delta P_{c_j} - \gamma \Delta Q_{c_j})\end{aligned}$$

Only clusters  $c_i$  and  $c_j$  are affected. Choosing the best target community thus depends on  $(\Delta P_{c_j} - \gamma \Delta Q_{c_j})$  whenever at least one community yields an increase in modularity.



---

**Algorithm 1: OPTIMIZATION**

---

**Input:** A graph  $G = (V, E)$ , a tolerance  $\epsilon$

**Output:** A partition of  $V$

**for**  $u \in V$  **do**

$\mathcal{C}_u \leftarrow \{u\}$

**end**

**repeat**

$\mathcal{C}' \leftarrow \mathcal{C}$

**for**  $u \in V$  **do**

$Q = \text{MOD}(G, \mathcal{C})$

$l = \text{argmax}_{v \in \mathcal{N}(u)} \text{MOD}(G, \text{MOVE}(G, \mathcal{C}, u, c_v))$

$\Delta Q \leftarrow \text{MOD}(G, \text{MOVE}(G, \mathcal{C}, u, c_l)) - Q$

**if**  $\Delta Q > \epsilon$  **then**  $\mathcal{C} \leftarrow \text{MOVE}(G, \mathcal{C}, u, c_l)$

**end**

**until**  $\mathcal{C} = \mathcal{C}'$

**return**  $\mathcal{C}$

---

---

**Algorithm 2: AGGREGATION**

---

**Input:** The adjacency  $A$  of a graph  $G$ , a partition  $\mathcal{C}$  of  $G$

**Output:** The adjacency  $A'$  of the aggregated graph  $G'$

$Y \leftarrow \text{MEMBERSHIP}(\mathcal{C})$

$A' \leftarrow Y^T A Y$

**return**  $A'$

---

In the case of Equation (3.1):

$$\begin{aligned}\Delta P_{c_i} &= 2p_{ii} - \sum_{j \in \mathcal{C}_{c_i} \cap \mathcal{N}(i)} (p_{ij} + p_{ji}) \\ \Delta P_{c_j} &= \sum_{j' \in \mathcal{C}_{c_j} \cap \mathcal{N}(i)} (p_{ij'} + p_{j'i})\end{aligned}$$

where  $\mathcal{N}(i)$  is the neighborhood of node  $i$ . Furthermore, we write  $Q_k = Q_k^r Q_k^c$  where  $Q_k^r = \sum_{i \in \mathcal{C}_k} q_i^r$  is the probability of sampling a node in cluster  $k$  under  $q^r$  and similarly for  $Q^c$ . Then:

$$\begin{aligned}\Delta Q_{c_i} &= q_i^c (q_i^r - Q_{c_i}^r) + q_i^r (q_i^c - Q_{c_i}^c) \\ \Delta Q_{c_j} &= q_i^c Q_{c_j}^r + q_i^r Q_{c_j}^c\end{aligned}$$

To illustrate how the different modularity functions differ, we run the Louvain method using the modularity functions described in Section 3.2.2 on the WikiVitals dataset. It is built from an extraction of Wikipedia and comprises the adjacency matrix of the hyperlink graph of the extracted articles and a bag-of-words biadjacency of the text in the articles (one node set is the articles, the other is the words they contain). For the biadjacency, a source article and a target word have an edge between them if the former contains the latter, then the weight of the edge is the number of occurrences of the word in the article.

We use the V-measure and adjusted Rand scores (Rosenberg and Hirschberg, 2007; Hubert and Arabie, 1985) described in Section 3.2.1. We report the results for the hyperlinks adjacency in Table 3.1 and those for the bag-of-words biadjacency in Table 3.2. We remove the somewhat ambiguous "People" class of the dataset, leaving 10 classes in the ground truth. We find that the Newman, Dugué and Barber modularities offer very similar performance. The Potts modularity does not use the weights of the graph and is expectedly the worst option. This is especially true in the case of the bag-of-words biadjacency where both scores are low.

	Newman	Dugué	Barber	Potts
V-measure	0.49	0.47	0.44	0.41
ARI	0.45	0.44	0.45	0.23

Table 3.1: Clustering scores on the hyperlinks adjacency.

	Newman	Dugué/Barber	Potts
V-measure	0.48	0.48	0.05
ARI	0.41	0.41	0.00

Table 3.2: Clustering scores on the bag-of-words biadjacency.

### 3.2.4 Leiden refinements

Many variants of the Louvain method have been proposed in recent years (De Meo et al., 2011; Bhowmick and Srinivasan, 2013; Que et al., 2015). Among those, one notable variant called the Leiden algorithm has been proposed (Traag et al., 2019). It consists in a collection of three separate improvements of the original Louvain method:

- The **smart local move** (SLM) (Waltman and Van Eck, 2013) alters the aggregation phase. After every optimization phase, the Louvain method is run recursively on each subgraph corresponding to a cluster. This yields a new "refined" clustering for each subgraph which is concatenated into a clustering for the whole graph. This clustering is the one that is used to aggregate the graph whereas the original clustering is used to initialize the value of the partition at the start of the next optimization phase. We illustrate this process in Figure 3.1. Note that thanks to Proposition 2, the monotony of the modularity is still guaranteed as the refined clustering is homogeneous with respect to the original clustering. It is clear however that using a singleton partition (i.e. that of step 3.1c) with each node in its own cluster in the aggregated graph would lower the modularity (compared to that of step 3.1d).
- The **random local move** (RLM) (Traag, 2015) reduces the number of computations for modularity updates. To do so, for each node visited during the optimization phase, it picks a random neighbor instead of looking at the whole neighborhood. If moving the node under study to this neighbor's community increases the modularity, then it is done. Otherwise, another node is visited.
- The **fast local move** (FLM) or **Louvain Prune** (Ozaki et al., 2016) reduces the number of nodes visited during the optimization phase. Instead of iteratively considering all the nodes of the graph until the modularity does not increase any more, it maintains a queue to which only the neighbors of moved nodes that are in a different community are added.

It should be noted that the Leiden algorithm does not use those improvements in the way they are described by their respective authors.

Additionally, we propose a novel refinement for the Louvain algorithm which we name **chained local move** (CLM). It aims at speeding up the first optimization phases by reducing the number of modularity increase computations. In order to do so, instead of finding the best possible increase for each node, this is done for one initial node. If the initial node is moved to another community, a neighbor whose community is different from this new community is picked at random. If moving this neighbor to the initial node's new community increases the modularity, it is done and a neighbor of the node moved last is picked. A chain of neighbors is then built from the initial node. This chain ends when moving a node does not increase the modularity or no neighbor can be picked. We give the pseudo-code for the optimization phase of the CLM in Algorithm 3 where  $\text{RANDOMPOP}_S$  picks a node uniformly at random from the set passed as its input and removes it from  $S$ .

We compare the benefits of each of the variants in terms of modularity (using the one defined in Equation (3.6)) and time performance. We report the performance of the standard Louvain (as "Baseline") and the variants under study in Tables 3.4, 3.5, 3.6 and 3.7.

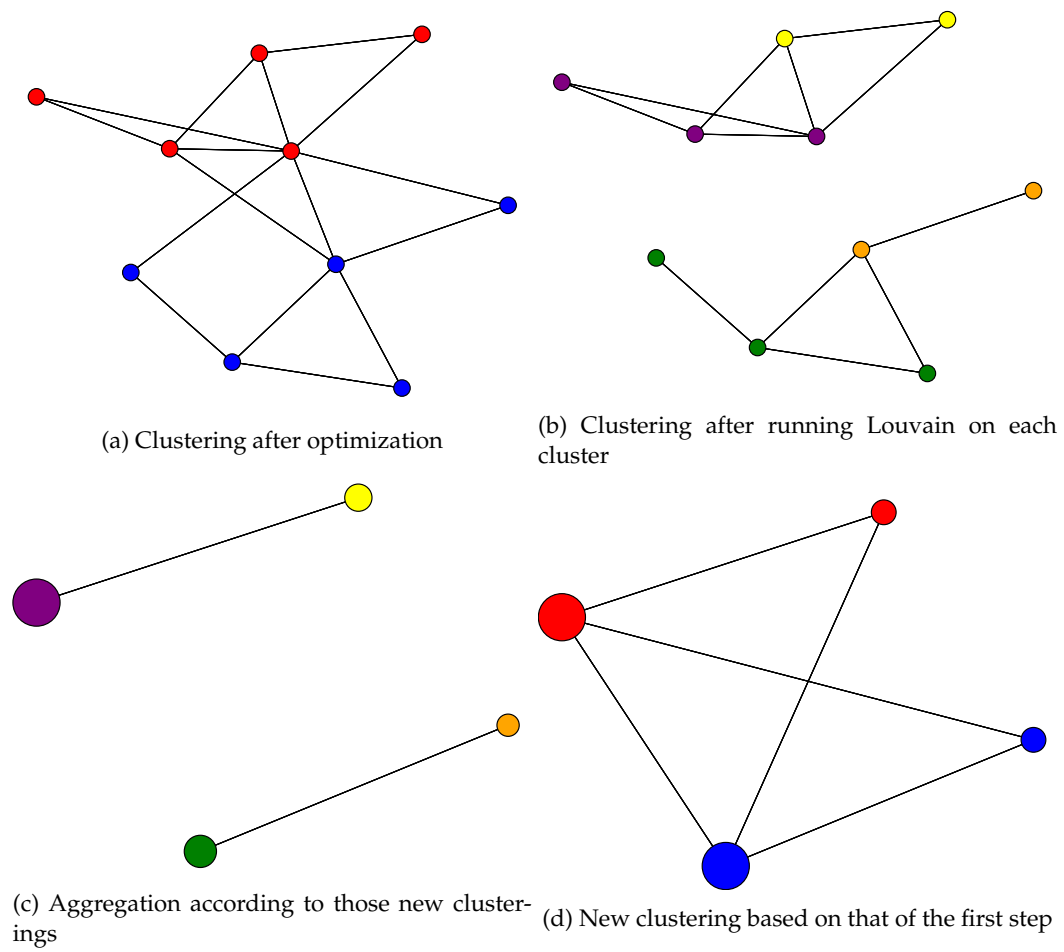


Figure 3.1: Refinement process for the Smart Local Move.

---

**Algorithm 3:** Optimization phase for the Chained Local Move

---

**Input:** A graph  $G = (V, E)$ , a tolerance  $\epsilon$   
**Output:** A partition of  $V$   
**for**  $u \in V$  **do**  
     $\mathcal{C}_u \leftarrow \{u\}$   
**end**  
**repeat**  
     $\mathcal{C}' \leftarrow \mathcal{C}$   
     $S \leftarrow \{1, \dots, |V|\}$   
    **while**  $S \neq \emptyset$  **do**  
         $u \leftarrow \text{POP}(S)$   
         $l = \text{argmax}_{v \in \mathcal{N}(u)} \text{MOD}(G, \text{MOVE}(G, \mathcal{C}, u, c_v))$   
         $\Delta Q \leftarrow \text{MOD}(G, \text{MOVE}(G, \mathcal{C}, u, c_l)) - \text{MOD}(G, \mathcal{C})$   
        **while**  $\Delta Q > \epsilon$  **do**  
             $\mathcal{C} \leftarrow \text{MOVE}(G, \mathcal{C}, u, c_l)$   
             $u \leftarrow \text{RANDOMPOP}_S((\mathcal{N}(u) \cap S) \setminus \mathcal{C}_{c_l})$   
             $\Delta Q \leftarrow \text{MOD}(G, \text{MOVE}(G, \mathcal{C}, u, c_l)) - \text{MOD}(G, \mathcal{C})$   
        **end**  
    **end**  
**until**  $\mathcal{C} = \mathcal{C}'$   
**return**  $\mathcal{C}$

---

In each table, we report the best absolute value in bold and otherwise display the relative errors to that value. The experiments are run on a computer running Debian 10 OS and equipped with an AMD Ryzen Threadripper 1950X 16-Core Processor and 128 GB of RAM. We run the experiments on the graphs described in Table 3.3 where the number of classes in the ground truth is given if available (in the results, the ‘Wiki-’ prefix is dropped for readability).

Considering the SLM algorithm, we notice that it is consistently the slowest option, sometimes up to three times as slow as the standard method. This poor time performance is due to the costly extraction of many subgraphs for the partition refinement. This operation is slow because of the CSR format described in Chapter 2. It should be noted however, that the SLM algorithm makes up for its speed by returning partitions that have a higher modularity on large datasets. It also offers a good performance in terms of V-measure on smaller ones although not in terms of ARI score.

Looking at the RLM variant, note that it is almost always the fastest option at the cost of decreased modularity values. The time gains are not consistent as illustrated by the Orkut graph where it is noticeably slower than the baseline while giving the worst clustering compared to the other options. Nevertheless, it sometimes offers interesting time gains as seen on the UK Domain dataset where its modularity is close to the baseline. It also performs fairly well in terms of V-measure on smaller datasets but this is not the case in terms of ARI score.

The FLM variant offers more modest time gains as it is generally placed between the

	$n$	$m$	# Labels
WikiSchools	$4.10^3$	$1.10^5$	16
WikiVitals	$8.10^3$	$6.10^5$	10
WikiVitals+	$3.10^4$	$2.10^6$	10
WikiLinks	$3.10^6$	$7.10^7$	-
Orkut	$3.10^6$	$2.10^8$	-
UK Domain	$2.10^7$	$5.10^8$	-
Twitter	$4.10^7$	$1.10^9$	-

Table 3.3: Dataset statistics.

	Schools	Vitals	Vitals+	Links	Orkut	UK Domain	Twitter
Baseline	2	1.3	1.6	1.1	1.1	1.2	1.1
SLM	3.7	2.1	2.7	3.8	1.7		
RLM	<b>0.03</b>	<b>0.15</b>	<b>1.02</b>	<b>44.79</b>	1.4	1.0	<b>1860.45</b>
FLM	1.3	1.2	1.6	1.0	1.1	1.1	1.1
CLM	1.7	1.7	1.4	1.3	<b>122.06</b>	<b>72.15</b>	1.1

Table 3.4: Execution times (in s). : Timeout (> 7200s)

	Schools	Vitals	Vitals+	Links	Orkut	UK Domain	Twitter
Baseline	<b>0.396</b>	<b>0.468</b>	<b>0.522</b>	0.98	0.99	<b>0.985</b>	<b>0.481</b>
SLM	0.99	0.97	0.96	<b>0.651</b>	<b>0.670</b>		
RLM	0.93	0.90	0.90	0.91	0.90	0.99	0.96
FLM	1.0	1.0	<b>0.522</b>	0.98	0.99	<b>0.985</b>	<b>0.481</b>
CLM	0.97	0.99	<b>0.522</b>	0.93	0.94	0.99	<b>0.481</b>

Table 3.5: Modularity of the result. : Timeout (> 7200s)

	Schools	Vitals	Vitals+
Baseline	0.91	0.98	0.90
SLM	<b>0.32</b>	0.94	<b>0.42</b>
RLM	<b>0.32</b>	0.88	0.98
FLM	0.91	0.98	0.90
CLM	0.94	<b>0.52</b>	0.90

Table 3.6: V-measure score of the result and the ground truth.

	Schools	Vitals	Vitals+
Baseline	0.86	<b>0.46</b>	0.93
SLM	0.90	0.72	0.63
RLM	<b>0.21</b>	0.50	0.63
FLM	0.86	<b>0.46</b>	0.93
CLM	<b>0.21</b>	<b>0.46</b>	<b>0.27</b>

Table 3.7: Adjusted Rand score (ARI) of the result and the ground truth.

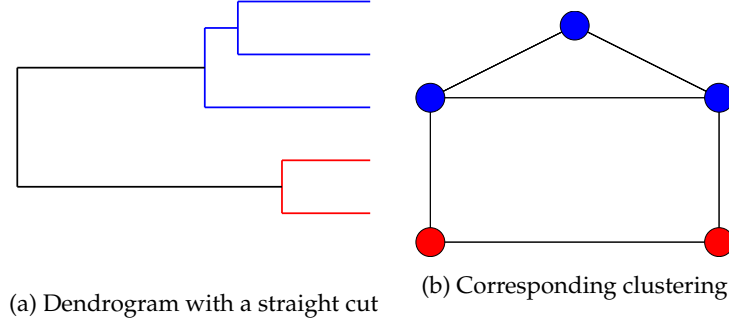


Figure 3.2: Hierarchical clustering of a graph

baseline and the RLM variant in terms of execution times. One exception in our benchmark is the Orkut graph where the FLM variant is even faster than the RLM variant. Unlike the RLM variant, the modularity, V-measure and ARI reached by the FLM algorithm is consistently equal or almost equal to that of the baseline. It is thus a simple way to speed up the Louvain method without noticeable downsides.

The CLM variant appears as an intermediate option between the baseline and the RLM variant in terms of time/modularity trade-off. The modularity of the clusterings it returns are consistently higher than that of the RLM while still lower than that of the baseline. As regards execution times, depending on the dataset at hand, it can be either faster or slower than both the RLM variant and the baseline. It is notably the fastest option on the Orkut and UK Domain datasets and the slowest on the Twitter graph. On smaller datasets, it is close to the baseline in terms of V-measure and the best performer in terms of adjusted Rand score.

### 3.3 Hierarchical clustering

Many graph datasets have a multi-layered community structure. Hierarchical clustering is a set of cluster analysis methods which seeks to build a hierarchy of clusters. Methods for hierarchical clustering generally fall into two categories:

- *agglomerative* approaches, where each node starts in its own cluster and pairs of clusters are successively merged to obtain just one cluster
- conversely, *divisive* approaches consist in successive splits of an initial cluster containing all nodes

#### 3.3.1 Dendrograms

Hierarchical clusterings are presented in the form of a tree structure called a dendrogram. In addition to the tree structure, nodes in the tree have a height attribute (with the leaves being at height 0 and the root having the highest height).

To obtain a flat clustering from a dendrogram, we make a *cut* of it. For instance, a straight cut with  $k$  clusters consists in looking at the graph after the  $n - k$  first merges (starting from the leaves, in increasing order of height) have been made. An example is given in Figure 3.2 where the cut of the dendrogram in Figure 3.2a corresponds to the flat clustering seen in Figure 3.2b.

### 3.3.2 Agglomerative approach

We describe the hierarchical clustering method proposed in Bonald et al. (2018a). Consider the similarity defined in Section 3.1.1 on an undirected graph  $G$ , a simple agglomerative approach is to find the pair of nodes with the highest similarity and merge them. Iterating this process suffices to build a hierarchical clustering of a graph. We note  $i \cup j$  the result of merging nodes  $i$  and  $j$ . In order to iterate the merges, we look at how the sampling distribution  $s$  and the similarity  $\sigma$  are affected. Let  $i, j, k \in V$  be distinct:

$$s(i \cup j, k) = \frac{A_{ik} + A_{jk}}{2w} = s(i, k) + s(j, k)$$

and  $v$  remains unchanged, so that:

$$s(i \cup j) = s(i) + s(j)$$

Then:

$$\sigma(i \cup j, k) = \frac{s(i, k) + s(j, k)}{(s(i) + s(j))s(k)} = \frac{s(i)}{s(i) + s(j)}\sigma(i, k) + \frac{s(j)}{s(i) + s(j)}\sigma(j, k) \quad (3.8)$$

This equation makes it possible to update the values of  $\sigma$  each time two nodes are merged.

In order to have a complete clustering of the graph,  $n - 1$  merge have to be performed. For each merge, the node pair with the highest similarity has to be found, this requires  $O(m)$  operations if done naively. However using a walk on the graph to find reciprocating nearest-neighbors suffices to merge two nodes even if this pair does not reach the global maximum in similarity (Murtagh and Contreras, 2012). This is due to one property of the similarity  $\sigma$  whose inverse defines a *reducible* distance on the node set:

**Proposition 4.**

$$\forall k \neq i, j, \sigma(i \cup j, k)^{-1} \geq \min(\sigma(i, k)^{-1}, \sigma(j, k)^{-1})$$

The walk (starting from a random node) then consists in successively visiting the neighbor of the current node that is nearest in terms of similarity. Two nodes can be merged whenever the successor of a node in the walk is also its predecessor. After each merge, the similarity is updated using Equation (3.8). When the whole graph has been merged, the obtained dendrogram is binary and the heights of the merges are defined by the distance  $\sigma(i, j)^{-1}$ .

In the case of directed graph, the sampling distribution  $s(i, j)$  is no longer symmetric. We thus define  $s^+(i) = \sum_{j \in V} s(i, j)$  and  $s^-(i) = \sum_{j \in V} s(j, i)$ , two marginal distributions



corresponding to the normalized in and out degrees the nodes. This enables us to define a new symmetric distribution:

$$t(i, j) = s^+(i)s^-(j) + s^-(i)s^+(j)$$

and a new similarity:

$$\tau(i, j) = \frac{s(i, j) + s(j, i)}{t(i, j)}$$

An equivalent of Equation (3.8) is:

$$\tau(i \cup j, k) = \frac{t(i, k)}{t(i, k) + t(j, k)} \tau(i, k) + \frac{t(j, k)}{t(i, k) + t(j, k)} \tau(j, k)$$

The distance defined by the inverse of  $\tau(i, j)$  is also reducible.

### 3.3.3 Divisive approach

A simple divisive approach for hierarchical clustering consists in recursively applying a flat clustering method. After each time such a method is applied, one can extract subgraphs corresponding to each cluster and start again until each node is in its own cluster. Here, we will consider the Louvain method described in Section 3.2.3. This recursive method is the one described by Bhowmick et al. (2020) where the same hierarchy to build an embedding (see Chapter 4). The dendrogram is not a binary tree in that case. Merge heights are determined by the depth in the recursion (i.e. the first clustering pass is at the root of the dendrogram).

### 3.3.4 Experiments

We run the agglomerative and divisive approaches on the Karate Club graph as illustrated in Figures 3.3 and 3.4. We make a cut of each dendrogram with 4 clusters.

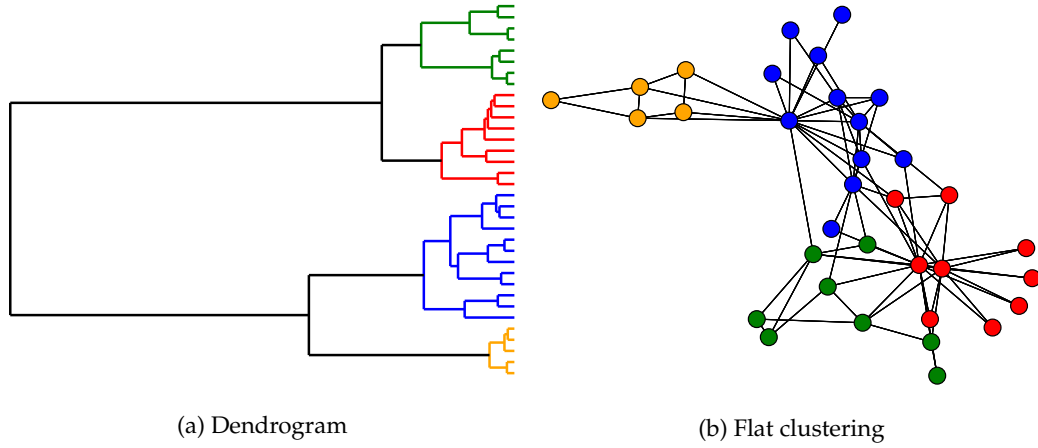


Figure 3.3: Agglomerative approach

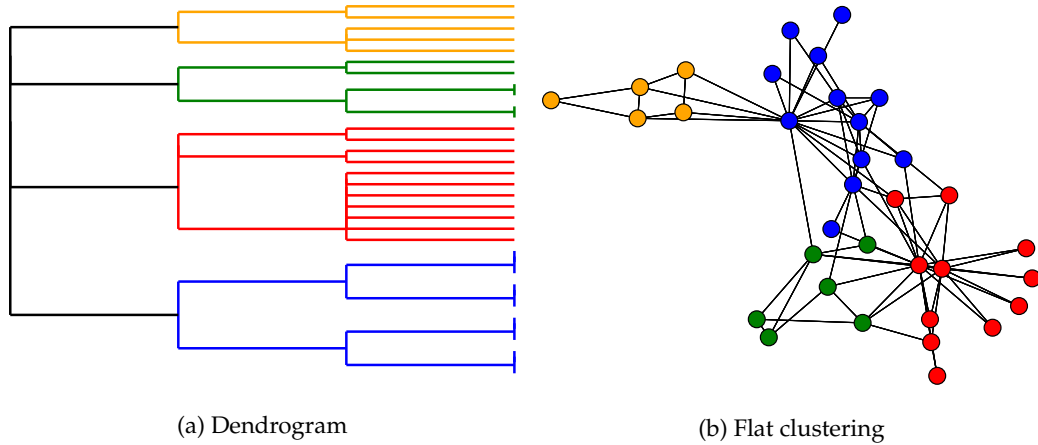


Figure 3.4: Divisive approach

The dendrogram of the agglomerative approach is densely packed at the leaves as the last few merges have a low similarity (i.e. a high distance). On the other hand, the divisive approach has a more spread out shape as the height information is built (somewhat arbitrarily) based on the recursion depth of the flat clustering method.

Notice how, despite having very different dendrograms (as seen in Figures 3.3a and 3.4a), the resulting flat clustering when cutting for 4 clusters is vastly similar (Figures 3.3b and 3.4b).

There are some cases where the data at hand is inherently hierarchical in that its labeling has a tree structure. Hierarchical clustering makes it possible to refine the existing hierarchy in such a situation.

### 3.4 Conclusion

In this chapter, we described some well-established methods for both flat and hierarchical node clustering. In particular, we considered variants of the Louvain algorithm (Blondel et al., 2008) and compared their performances. We also introduced a novel variant, the Chained Local Move (CLM) which aims at speeding up the initial optimization phases of the algorithm. We notice that the variants under study cover a fairly wide spectrum in terms of time performance and modularity. This could prove useful as one can choose a variant depending on the desired time/modularity trade-off.

Further work could focus on further speeding up the costly initial optimization phase of the Louvain algorithm. Indeed, this initial phase is the heaviest in terms of computations as many communities have to be considered for most of the nodes in the standard Louvain. We feel there is still room for improvement in terms of alleviating the cost of those computations.

## Chapter 4

# Node embedding

### 4.1 Introduction

In the context of network mining, a number of tasks can be performed by exploiting the graph structure of the data under study. Such tasks include the clustering, as illustrated by the Louvain method (Blondel et al., 2008), or ranking, as illustrated by PageRank (Page et al., 1998) and other centrality measures (Kleinberg, 1999; Brandes, 2001), of the nodes of the graph.

However, it is also possible to apply other methods that perform the same tasks but operate on vector data by *embedding* each node of the graph into a vector of  $\mathbb{R}^d$  in a way that preserves some of the properties of the graph. For instance, a "good" embedding of the nodes should make neighbors in the graph closer in terms of distance in  $\mathbb{R}^d$ .

#### 4.1.1 Motivation

There exists many methods for the embedding task (Hamilton et al., 2017), using a variety of techniques ranging from random projections (Zhang et al., 2018) to skip-gram models (Grover and Leskovec, 2016). Some techniques are unsupervised (Bhowmick et al., 2020) while others are supervised or semi-supervised (Kipf and Welling, 2017). While most methods put an emphasis on the relevance of the embedding, some also focus on the scalability of the process on large graphs (Zhang et al., 2019).

In this chapter, we introduce a method that builds upon the qualities of the Louvain method for clustering the nodes of a graph (Blondel et al., 2008). This method is:

- **scalable**, as it adds little to no overhead to the execution of the Louvain clustering method
- mostly **parameter-free**, as the Louvain method only has one tolerance parameter and we fix the resolution parameter to  $\gamma = 1$ . In particular, there is no need to set the embedding dimension  $d$ .
- **unsupervised**, as the Louvain method is also unsupervised

- **interpretable**, the underlying clustering makes it possible to describe each axis of the embedding space (see section 4.4.1).

To demonstrate the scalability and the effectiveness of this method, we run experiments against five competing algorithms on large and very large graphs.

### 4.1.2 Related work

Network embedding methods make use of a large variety of techniques (Chen et al., 2020). In the recent years, many embedding methods (Grover and Leskovec, 2016; Dong et al., 2017; Keikha et al., 2018) have been derived from techniques taken from the word representation literature like word2vec (Mikolov et al., 2013). These methods usually require a large number of random walks to train the model. As a result, these methods are often too slow to run on large graphs.

Earlier approaches rely on the factorization of the adjacency matrix using matrix factorization results (Ou et al., 2016; Abdi, 2007; Huang et al., 2012). While these methods are grouped together, their performances vary widely. For instance, Laplacian eigenmaps (Belkin and Niyogi, 2003) are unsuitable for large graphs while factorizations such as the non-negative matrix factorization can tackle large datasets (Huang et al., 2012). The interpretability of the resulting embedding also varies. For example, another family of factorization approaches are spectral techniques which rely on the eigendecomposition of matrices to learn embeddings (Luo et al., 2003). Some of those embeddings can be interpreted using equivalent physical systems (Bonald et al., 2018b).

Unlike spectral approaches that use the global information available in a graph, some learn embeddings by using local information (e.g. by looking at the neighborhood of the nodes) (Tang et al., 2015). This is the case of ClusterNE. Using local information often makes it possible to lower the overall complexity for training embeddings, for instance by using negative sampling.

One approach relies on a hierarchical clustering of the nodes (Bhowmick et al., 2020), they use the same hard node clustering algorithm as ClusterNE but use it iteratively to generate a hierarchy of the nodes. At each level of this hierarchy, a random vector is drawn for every cluster. Those level representations are then summed to obtain the embedding of each node. In terms of using the Louvain clustering method, their approach and ClusterNE both begin by applying Louvain on the whole graph. However, building the hierarchy calls for numerous other executions of the algorithm while ClusterNE directly builds the embedding from the first execution.

## 4.2 Embedding method

### 4.2.1 Algorithm

Let  $G = (V_1, V_2, E)$  be a bipartite graph. Consider the Louvain clustering method with no resolution parameter (i.e.  $\gamma = 1$ ) and using the Barber modularity. Assume this method yields  $k$  distinct clusters in  $V_1$  and  $V_2$ . Let  $B \in \mathbb{R}^{n \times m}$  be the biadjacency of  $G$  and  $Y \in \mathbb{R}^{m \times k}$

be the column membership matrix for the obtained clustering, i.e.  $Y_{iu} = 1$  if node  $i \in V_2$  is in cluster  $u$  and 0 otherwise. The embedding matrix for the nodes of  $V_1$  is  $\text{diag}(d)^{-1}BY$  where  $\text{diag}(x)$  is the diagonal matrix whose coefficients are the  $(x_i)_i$  and  $d$  is the degree vector of the nodes of  $V_1$ . Conversely, if  $X \in \mathbb{R}^{n \times k}$  is the row membership matrix, then the embedding matrix for the nodes of  $V_2$  is  $\text{diag}(f)^{-1}B^T X$  with  $f$ , the degree vector of the nodes of  $V_2$ .

For graphs that are not bipartite, one can use the previous method by cloning all the nodes of the graph and connecting each node to the clones of its original neighbors. It is equivalent to looking at the adjacency matrix of the graph as a biadjacency matrix of some bipartite graph.

This method can thus be applied on directed or undirected and weighted or unweighted graphs. It can also be applied on unconnected graphs although it should be noted that one inherent downside of this method is that Louvain yields at least as many clusters as connected components. This means that a graph with many connected components will have a high-dimensional embedding. It is thus more efficient to use this method on connected graphs. In particular, singleton clusters (i.e. clusters made up of one node only) are irrelevant as they contribute little information to the embedding. Those clusters are thus removed to keep the embeddings short. However, this truncation implies that isolated nodes have a zero embedding.

---

**Algorithm 4:** CLUSTERNE

---

**Input:** The adjacency matrix  $A$  of a graph  $G = (V, E)$ , a tolerance  $\epsilon$

**Output:** An embedding of the nodes of  $V$

$\mathcal{C} \leftarrow \text{GENERALIZEDLOUVAIN}(G, \epsilon)$

$s_1, \dots, s_l \leftarrow \text{SINGLETONCLUSTERS}(\mathcal{C})$

$Y \leftarrow \text{MEMBERSHIP}(\mathcal{C})$

$d \leftarrow A\vec{1}$

$Y \leftarrow Y(s_1, \dots, s_l)$

$E \leftarrow \text{diag}(d)^{-1}AY$

**return**  $E$

---

We describe the pseudo code in Algorithm 4 where GENERALIZEDLOUVAIN is the algorithm described in section 3.2.3 with the Barber modularity function from Equation (3.7), SINGLETONCLUSTERS returns the singleton clusters of some partition  $\mathcal{C}$  (this function is  $\Theta(n)$  with  $n$  the number of nodes of the graph  $G$ ), MEMBERSHIP returns the membership matrix of a partition and, if  $M \in \mathbb{R}^{n \times m}$  and  $\forall i, a_i \in \{1, \dots, m\}$ ,  $M(a_1, \dots, a_r)$  denotes the extracted matrix of  $\mathbb{R}^{n \times r}$  whose columns are the  $a_1$ -th up to  $a_r$ -th columns of  $M$ . This embedding method inherits some properties from the Louvain clustering method:

- Because the number of clusters is not known in advance, the embedding dimension is not known either and needs not be set in advance.
- The algorithm scales as well as the Louvain clustering method on large data as most of the complexity comes from the clustering itself.

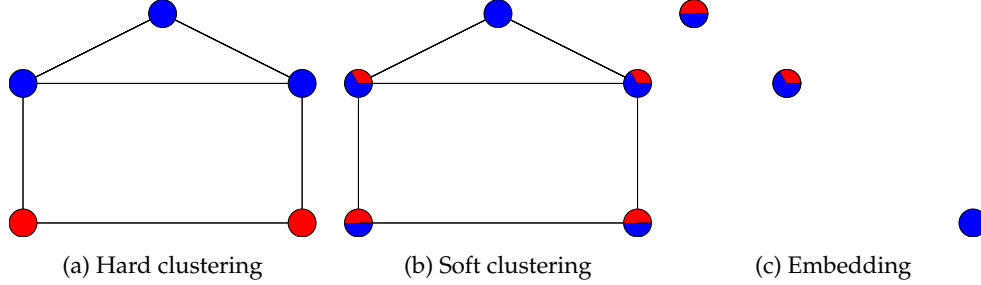


Figure 4.1: Clusterings and embedding of a graph.

Just as the clustering of the generalized Louvain is a co-clustering of both node sets, all the embeddings of the nodes from either node set share the same vector space (if one also computes the column embedding of the graph).

#### 4.2.2 Link with soft clustering

The described embedding method can be described not only in terms of the Louvain *hard* node clustering method described in Chapter 3, but also in terms of a *soft* clustering derived from the same method. Hard clustering denotes the case where each node is assigned exactly one cluster while soft clustering assigns a vector of probabilities denoting how likely a node is to belong to each cluster (Yu et al., 2005). Those vectors can be seen as the embeddings of the nodes.

We illustrate this on a simple graph in Figure 4.1. Starting from the hard node clustering of the graph obtained with the Louvain method in Figure 4.1a, we build a probability vector for each node by counting the number of neighbors in each community and normalizing those counts. This gives a soft clustering as seen in Figure 4.1b where each pie chart stands for the probability vector. This soft clustering is directly interpreted as an embedding in  $\mathbb{R}^2$  as depicted in Figure 4.1c.

### 4.3 Results

In order to evaluate the performance of the method described in section 4.2, which we will refer to as ClusterNE, we compare it against other embedding methods on two tasks and also compare the time performance of those algorithms. Let  $d$  be the embedding dimension, these methods are:

- **non-negative matrix factorization** (NMF) (Huang et al., 2012): an approximate matrix factorization method which yields matrices with positive elements from a matrix with positive elements. It is commonly used for dimensionality reduction. To compute this factorization, we set the number of iteration to 200 and the tolerance to  $10^{-4}$ .
- **singular value decomposition** (SVD) (Abdi, 2007): a matrix factorization technique based on a generalization of the eigendecomposition of matrices. We return the  $d$  first

	$n$	$m$	# Labels
PolBlogs	$1.10^3$	$2.10^4$	2
WikiVitals	$1.10^4$	$8.10^5$	11
WikiVitals+	$4.10^4$	$3.10^6$	11

Table 4.1: Dataset statistics.

left singular vectors of the adjacency matrix weighted by the corresponding singular values (in descending order).

- **RandNE** (Zhang et al., 2018): a network embedding approach using successive random projections. We set  $q = 3$  and the weights to  $(0.5, 0.5^2, 0.5^3)$ .
- **ProNE** (Zhang et al., 2019): a network embedding method using spectral propagation to improve the learned embeddings. We set the parameters to  $k = 10, \mu = 0.1, \theta = 0.5$ .
- **LouvainNE** (Bhowmick et al., 2020): a network embedding algorithm which also relies on the clustering obtained with the Louvain method. This method recursively applies Louvain on the obtained clusters and derives an embedding by picking random vectors. We use their stochastic variant with  $\alpha = 0.1$ .

As all these methods require to set  $d$  beforehand, we set this dimension to the one obtained after running ClusterNE. We rely on three open-source packages: the non-negative matrix factorization is found in *scikit-learn*<sup>1</sup>; ProNE is found in *nodevectors*<sup>2</sup>; RandNE, LouvainNE and the singular value decomposition are found in *scikit-network* of Chapter 2. Although all those implementations offer an API in Python, they rely on compiled backends but to different degrees. One should thus be careful when comparing their performance as results may differ (especially in terms of time performance) when using other implementations of the same algorithms.

In the experiments, we use publicly-available real-world web graphs. For link prediction and node classification, we use web graphs and graphs extracted from the Wikipedia hyperlink graph detailed in Table 4.1. For time performance, we use extractions of the Web Data Commons hyperlink graph. In each table, we report the best absolute value in bold and otherwise display the relative errors to that value.

### 4.3.1 Link prediction

The embeddings are expected to make it possible to reconstruct the graph. In order to assess this property, we run a network reconstruction task: we sample some pairs of nodes in the graph (we sample as much edges as non-edges) and compute the cosine similarities of the corresponding embedding vectors to estimate the probability that the nodes are connected in the original graph. We compare these similarities to the actual linked nodes of the graph using the area under the curve (AUC) score (Huang and Ling, 2005) as reported in

<sup>1</sup><https://scikit-learn.org>

<sup>2</sup><https://github.com/VHRanger/nodevectors>



Table 4.2. We observe that ClusterNE is the second-best performing algorithm with results close to the better-performing ProNE.

	PolBlogs ( $d = 8$ )	WikiVitals ( $d = 11$ )	WikiVitals+ ( $d = 16$ )
ClusterNE	0.70	0.98	0.98
NMF	0.71	0.92	0.97
SVD	0.74	0.94	0.95
LouvainNE	<b>0.89</b>	0.80	0.85
ProNE	0.83	<b>0.93</b>	<b>0.87</b>
RandNE	0.65	0.89	0.97

Table 4.2: AUC on the pairwise cosine similarities of the embedding of sampled nodes.

### 4.3.2 Node classification

Similarly, the embeddings are expected to capture the structure of the graph: we run a node classification task: using a dataset with a known ground truth, we use a supervised learning framework (Platt, 1999) to classify the embedding vectors. We supply 100 nodes per class for the training set and use the remaining nodes for validation. We report the accuracy of each algorithm with respect to the ground truth in terms of precision in table 4.3. We observe that ProNE performs best once again with a significant edge over RandNE, ClusterNE and the SVD.

	PolBlogs ( $d = 8$ )	WikiVitals ( $d = 11$ )	WikiVitals+ ( $d = 16$ )
ClusterNE	0.96	0.86	0.67
NMF	0.92	0.78	0.56
SVD	0.92	0.83	0.70
LouvainNE	<b>0.89</b>	0.68	0.65
ProNE	0.97	<b>0.63</b>	<b>0.54</b>
RandNE	0.91	0.81	0.80

Table 4.3: Precision of an SVC for the classification of embedding vectors.

### 4.3.3 Time performance

We also report the execution times of each algorithm on graphs with a number of edges  $m$  ranging from  $3 \cdot 10^7$  to  $8 \cdot 10^8$ . For reference, we use an AMD EPYC 7542 32-Core CPU with 250 gigabytes of RAM. All algorithms are run in a single thread. We set a timeout of 2 hours. The results are reported in table 4.4.

The time performance of LouvainNE and the difference to the results displayed in the corresponding publication (Bhowmick et al., 2020) can be explained by the choice of the data structure used to represent the graphs: we use a sparse matrix format with a compression along the rows (Buluç et al., 2009) as described in Chapter 2 (see Section 2.3.1) which makes extracting subgraphs very costly. However, as our method requires only one execution of the Louvain clustering method whereas LouvainNE iteratively applies the same

algorithm, it can be expected that our method will always be faster when using the same tolerance parameter as the first clustering run is common to both methods.

We note that the SVD and ProNE do not scale as well as the NMF and ClusterNE.









	$m = 3 \cdot 10^7$ ( $d = 347$ )	$m = 2 \cdot 10^8$ ( $d = 518$ )	$m = 8 \cdot 10^8$ ( $d = 1215$ )
ClusterNE	1.5	1.6	<b>277</b>
NMF	<b>8</b>	<b>39</b>	1.8
SVD	33	51	
LouvainNE			
ProNE	47	69	
RandNE	5.1	7.6	

Table 4.4: Execution times (in seconds).  $m$  is the number of edges. : Timeout ( $> 7200$ s). : Memory overflow.

## 4.4 Properties

We now go over some relevant properties of the ClusterNE method.

### 4.4.1 Interpretability

Thanks to the underlying clustering, the embedding of the nodes of a graph obtained with the method described in section 4.2 can be interpreted as the weighted membership matrix of the nodes of the graph for the clustering obtained by Louvain. Intuitively, the embedding of a node denotes how close it is to each cluster.

For example, we embed the nodes of the WikiVitals dataset. We obtain 11 clusters, which we label manually from their closest articles in terms of PageRank (Page et al., 1998) as given in Table 4.5.

Clusters	Closest articles
Biology	Taxonomy (biology), Animal, Protein
World	Bibliothèque nationale de France, United States, Geographic coordinate system
Europe	Latin, Roman Empire, Greek language
Society	Marriage, Incest, Adoption
Media	The New York Times, Encyclopædia Britannica, Time (magazine)
Asia	India, Buddhism, Chinese language
Mathematics	Mathematics, Real number, Function (mathematics)
Physics	Hydrogen, Oxygen, Kelvin
Geography	Earth, Atlantic Ocean, Pacific Ocean
Philosophy	Aristotle, Plato, Age of Enlightenment
Craftsmanship	Jewellery, Weaving, Shoe

Table 4.5: Clusters and their closest articles in terms of PageRank

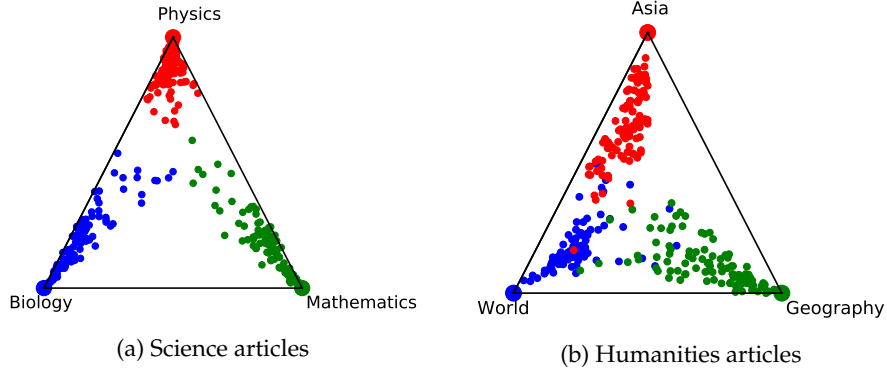


Figure 4.2: Visualization of 3-dimensional embeddings of topical Wikipedia articles and of the underlying clustering.

To illustrate the interpretability of the results, we choose two triplets of classes: Biology, Mathematics and Physics on one hand and World, Geography and Asia on the other hand. For each triplet we extract the articles (i.e. nodes) closest to each class in terms of PageRank (personalized by setting the nodes of the corresponding clusters as seeds). We plot the embedding of those articles projected on the 2-dimensional simplex. The color of each point denotes its cluster. Each vertex of the simplex is associated with a class of the chosen triplet as seen in Figure 4.2. We ignore the components of the embedding not associated with a class of the triplet under study.

In the case of Figure 4.2a, we observe that science articles are heavily polarized depending on the field they are related to. This is unlike what can be observed in Figure 4.2b where the clusters appear to be mixed together. In particular, some nodes belonging to the Asia class are very close to the World class. This denotes nodes whose embeddings have a higher component in clusters other than their own. In the clustering, this translates as nodes highly connected to other communities i.e. nodes at the border between communities.

It should also be noted that, as a byproduct of the Louvain clustering, two nodes belonging to two distinct connected components have orthogonal embeddings under ClusterNE. This is due to Louvain yielding at least as many clusters as there are connected components in the graph.

#### 4.4.2 Sparsity of the embedding

Another interesting feature of the ClusterNE method is that it yields sparse embeddings. This is illustrated by Table 4.6 where we report the fraction of entries of each embedding that are equal to zero for all the datasets and embedding methods under study. Methods such as LouvainNE and RandNE which are based on random projections and randomly picked vectors are expectedly dense. Likewise, the SVD and ProNE yield dense embeddings.

	PolBlogs ( $d = 8$ )	WikiVitals ( $d = 11$ )	WikiVitals+ ( $d = 16$ )
ClusterNE	<b>83</b>	45	<b>63</b>
NMF	69	<b>51</b>	57
SVD	28	0	0
LouvainNE	0	0	0
ProNE	28	0	0
RandNE	0	0	0

Table 4.6: Fraction of zero entries (in %).

ClusterNE gives sparse embeddings as the membership matrix  $Y$  and the adjacency matrix  $A$  are sparse. We thus also expect their dot product to be sparse. The NMF also gives a sparse embedding by construction. Having a sparse embedding makes it possible to use less memory for storing the embedding. It also makes it possible to combine the resulting embedding with other techniques that preserve the sparsity of their inputs.

## 4.5 Conclusion

In this chapter, we introduced ClusterNE, a node embedding method based on the Louvain clustering method discussed in Chapter 3. We illustrate how it compares to other embedding methods and show that it makes it possible to embed even very large graphs while being interpretable in terms of the underlying clustering.

Future work may use variants of the Louvain algorithm, especially parallelized ones (Bhowmick and Srinivasan, 2013) to make it possible to tackle even larger graphs. Creating embedding methods based on other clustering algorithms is also of particular interest. For instance, it may prove useful to design one around a clustering method that allows a number of clusters to be set as one may need to be able to set the dimension of the embedding.

## Chapter 5

# Dataset labeling

The use-case presented in this chapter was initially proposed to us by Maria-Laura Maag at Nokia. Initial work, made with Élie de Panafieu, consisted in defining the setting described in Section 5.1.2 and establishing the chordal conjecture. Further work, made with Élie de Panafieu, Alex Scott and Maya Stein within the context of the RandNet project <sup>1</sup>, focused on proving the results of Section 5.2. Those results have been published at NeurIPS 2021.

## 5.1 Introduction

### 5.1.1 Motivation and related work

Machine learning algorithms are split into two categories: unsupervised methods and supervised ones. The latter require labelled sets of elements for training from which they learn distinctive features, making it possible to process new items and predict some results for one or several tasks.

However the quality of those results depends heavily on that of the training dataset as well as its quantity (Brodley and Friedl, 1999). Acquiring a suitable corpus of items is not a trivial task (Nghiem and Ananiadou, 2018). In a general setting, as the purpose of a training set is to obtain a classifier, no classifier is available beforehand to build a training dataset. Thus, the creation of this dataset cannot be carried out automatically. This often means that humans must be involved which may prove time consuming. The need for an efficient human-based scheme for the creation of training data thus naturally arises.

A naive approach to this problem could consist in choosing an unlabelled set of data and having humans look at each item of this set and classify it. This requires:

- that an unlabelled set of data can be selected. As a thorough exploration of such a set is expensive for the reasons described above, it is expected that some elements may be ambiguous.

---

<sup>1</sup>See <https://cordis.europa.eu/project/id/101007705>

- that a set of labels has been determined. This is difficult in some cases as this set can be large.
- that the humans know the whole label set. This can also be difficult when this set is large.

To avoid some of the problems that would be induced by this solution, another possibility consists in presenting the experts with pairs of items of the set and asking if the two items should be put in the same category. This addresses the last two issues mentioned above but does not help with poor choices of the initial set of data to be labelled. This setting of pairwise comparisons is explored in this chapter.

This setting closely resembles what can be found in the active clustering literature (Xiong et al., 2017; Kim and Ghosh, 2017). One slight difference is that we seek to find the *exact* partition of the whole graph while minimizing the number of queries made to the oracle. Most approaches tackling related problems do not aim at finding the exact partition as labeling data is costly.

Recent papers (Chien et al., 2019; Mazumdar and Saha, 2017) acknowledge that humans prefer pairwise queries over pointwise queries as they are better suited for comparisons. Pairwise queries have been considered in semi-supervised clustering (Basu et al., 2004b; Wagstaff et al., 2001; Gribel et al., 2021) where they are called *must-link* and *cannot-link* constraints. The pairs of vertices linked by those constraints are random in general, but chosen adaptively in active semi-supervised clustering (Basu et al., 2004a). In both cases, the existence of a similarity measure between items is assumed. This is not the case in Eriksson et al. (2011); Krishnamurthy et al. (2012), where a hierarchical clustering is built by successively choosing pairs of items and measuring their similarity. There, the trade-off between the number of measurements and the accuracy of the hierarchical clustering is investigated. The difference with the current chapter is that the similarity measure takes real values, while we consider boolean queries, and that their output is a hierarchical clustering, while our clustering is flat.

The approach described in Section 5.2 does not involve supervised techniques. Rather, we investigate a data-agnostic generalization of this pairwise clustering setting. We consider a graph where each node is an item to be labelled in the original dataset. Rather than classifying each node, we seek to unveil the corresponding *clustering* of the graph (that is, we ignore the label names for the moment). This clustering (or *partition*) is what we try to find efficiently in this chapter.

In order to unveil the clustering, we suppose we have at our disposal an *oracle* that, when given a pair of nodes, is able to determine whether those two nodes belong to the same cluster or not. In reality, such queries correspond to questions asked to human experts about pairs of items. We make the strong assumption that the relation unveiled by the oracle is transitive, that is, if the oracle determines that items  $x$  and  $y$  on one hand, and  $y$  and  $z$  on the other hand, belong to the same cluster, then, we expect  $x$  and  $z$  to belong to the same cluster.

We consider the case where the oracle is *perfect*: it makes no mistakes. We want to characterize the algorithms whose average number of queries to the oracle is minimal.

### 5.1.2 Setting

The answers of the oracle can be organized in a graph where each element from the item set  $S$  is represented by a vertex, and each edge has one of two possible types. Two vertices are linked by a *positive edge* (resp. *negative edge*) if the oracle determined the corresponding elements belong (resp. do not belong) to the same part of the set partition. Thus, each past query to the oracle corresponds to an edge.

The *positive* (resp. *negative*) graph is the subgraph obtained by keeping only the positive (resp. negative) edges. A *positive component* is a connected component of the positive graph. The *contracted graph* is obtained by contracting each positive component to a vertex. The negative edges are then simply referred to as edges.

A set partition discovery algorithm (which we will now refer to as just "an algorithm") inputs a number of elements  $n$  and queries the oracle until the secret set partition on  $n$  elements is perfectly reconstructed. This final state is easily interpreted in terms of the graph:

**Theorem 1.** *A set partition discovery algorithm can stop if and only if the contracted graph is complete.*

Alternatively, this setting can be interpreted in terms of the equivalence relation  $\mathcal{R}$  on the item set  $S$  elements that we seek to unravel. For  $x, y \in S$ ,  $x\mathcal{R}y$  if, and only if,  $x$  and  $y$  are in the same class.  $\mathcal{R}$  is the (uniquely defined) equivalence relation such that the set partition we seek is exactly the quotient set  $S/\mathcal{R}$ . A set partition discovery algorithm then consists in growing two symmetric binary relations over  $S^2$ ,  $\mathcal{R}^+$  (for positive answers) and its "complement"  $\mathcal{R}^-$  (for negative answers). The transitive closure of  $\mathcal{R}^+$  has a quotient set  $Q$ .  $\mathcal{R}^-$  induces another binary relation  $\mathcal{R}_Q^-$  on  $Q^2$  by:  $a\mathcal{R}_Q^-b \Leftrightarrow \exists(x, y) \in a \times b, x\mathcal{R}^-y$ . The algorithm stops when  $\mathcal{R}_Q^-$  spans all pairs of distinct elements of  $Q$  and the sought relation  $\mathcal{R}$  is then the transitive closure of  $\mathcal{R}^+$ .

Let us recall that the *average complexity* is the average number of queries asked before reconstructing the correct set partition, when this set partition is chosen uniformly at random among all set partitions on  $n$  elements.

The complexity will depend on the distribution of the underlying partition. In this chapter, we will deal with the case where the partition is taken uniformly at random among all partitions on a given set. In particular, in this setting, the number of clusters is unknown a priori. In the situations we encountered at Nokia, we found that it may be very hard to get even an estimate of the number of clusters in practice. While it is likely that some partitions can be discarded by having some knowledge about the data at hand, we seek to solve this problem in a general case.

In Figure 5.1, we give an example of how two distinct algorithms unfold starting from the same graph represented as a binary tree. Each node of the tree at depth  $k$  is a possible realization of the contracted graph  $G_k$  after  $k$  queries. Starting from the graph  $G_0$  on the left (which is the root of the tree), the dotted line indicates what the next query is. The upper successor is obtained by adding the queried edge, which corresponds to a negative answer. The lower successor is obtained by merging both ends of the queried edge, which

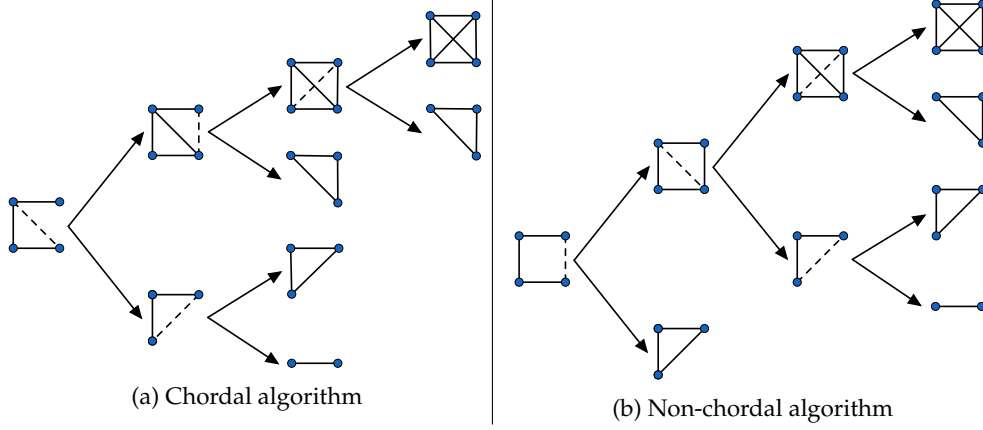


Figure 5.1: Unfolding of two different algorithms

corresponds to a positive answer. Notice that all leaves are complete graphs and that both trees have the same leaves.

Each leaf and the path that leads to it correspond to a different partition of the nodes of the graph. Conversely, each partition that is compatible with the starting graph has a corresponding leaf. The complexity associated with one particular partition for the algorithm under study is the depth of the associated leaf. Then, the average complexity of the algorithm under study is the average depth of the leaves (as we assume that the partitions are picked uniformly at random). Notice how, for Figure 5.1a, that complexity is  $\frac{3.2+2.3}{5} = \frac{12}{5}$  queries on average, while for Figure 5.1b, it amounts to  $\frac{1.1+3.4}{5} = \frac{13}{5}$ . This demonstrates that all algorithms do not have the same average complexity.

In fact, by running numerical simulations, we conjectured that the algorithms that have the best (i.e. lowest) average complexity are those that yield a chordal graph after each query. We define a chordal algorithm as an algorithm where, at each step, the contracted graph is chordal. We prove this conjecture in Section 5.2.

## 5.2 Chordal algorithms

In this section, we characterize optimal algorithms under the assumption that the partition of the set is chosen uniformly at random among all possible partitions.

In the rest of this section, we show that all partition discovery algorithms where the contracted graph is chordal after each step (denoted as chordal partition discovery algorithms or chordal algorithms) incur the same cost as stated in Corollary 1 and that they are the only optimal algorithms in our setting. We drew these properties from observations on small datasets and simulations on random data.

We introduce the following notations: for a graph  $G = (V, E)$  let  $G(uv) = (V, E \cup \{uv\})$ , let  $G_{uv}$  be the graph obtained from  $G$  by contracting nodes  $u$  and  $v$  into just one node and let  $G[A] = (A, \{uv | u, v \in A\})$  for  $A \subset V$  be the subgraph of  $G$  induced by  $A$ .



### 5.2.1 Chordal graphs

A graph is chordal if any of its cycles of four or more vertices has a chord. This means that all its induced cycles (i.e. chordless cycles) have length 3, hence chordal graphs are sometimes called triangulated graphs. An example of a chordal graph is given in Figure 5.2. We briefly go over some properties and characterizations of chordal graphs (see Golumbic (2004)).

Recall the definition of a  $uv$ -separator for two vertices  $u, v \in V$  as a subset  $S$  of the node set such that removing  $S$  leaves  $u$  and  $v$  in two distinct connected components. A graph is chordal if, and only if, all vertex separators which are minimal (for the inclusion) are cliques (i.e. complete subgraphs of the graph). This means that chordal graphs can be decomposed: the node set  $V$  is the union of a clique (a minimal vertex separator) and two subsets  $A$  and  $B$  such that there are no edges between  $A$  and  $B$ .

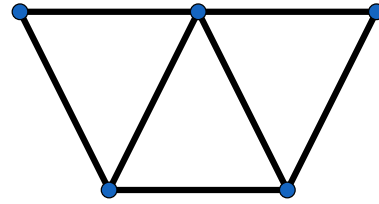


Figure 5.2: A chordal graph

Another characterization of chordal graphs relies on the notion of *perfect elimination schemes*. We say that a vertex is *simplicial* if its neighborhood is a clique. Any chordal graph has a simplicial vertex. Note that all the subgraphs of a chordal graph are also chordal. Thus, if one removes a simplicial vertex from a chordal graph, the resulting subgraph will also have a simplicial vertex. By iteratively removing those simplicial vertices, one creates a perfect elimination scheme of the graph which ends when there are no more vertices. Conversely, a graph with a perfect elimination scheme is chordal. This characterization can be used to check if a graph is chordal.

One should note that identifying two adjacent vertices in a chordal graph leaves the graph chordal. This notably simplifies how queries can be chosen in an efficient way (this is further discussed in Section 5.3.1).

**Lemma 1.** *Let  $G$  be a chordal graph, for any adjacent vertices  $u$  and  $v$ ,  $G_{uv}$  is chordal.*

*Proof.* Assume  $G_{uv}$  is not chordal, there is a minimal induced cycle  $C = (u, p_1, \dots, p_k, u)$  in  $G_{uv}$ , with  $k \geq 3$ . Then consider  $C' = (u, v, p_1, \dots, p_k, u)$  in  $G$ . If  $C'$  has a chord, it is either  $(u, p_1)$  or  $(v, p_k)$  and  $(u, p_1, \dots, p_k, u)$  or  $(v, p_1, \dots, p_k, v)$  is an induced cycle of length four or more. Otherwise,  $C'$  is. In either case, a contradiction.  $\square$

The following lemma is key for our subsequent work as it states that it is possible to "grow" a chordal graph into a larger chordal graph, i.e., any incomplete chordal graph has at least one non-edge that leaves the graph chordal once added to the edge set.

**Lemma 2.** *Let  $G$  be a chordal graph that is not complete. Then  $G$  has a non-edge  $e$  such that  $G(e)$  is chordal.*

*Proof.* Let  $u$  be a non-universal vertex of  $G$ . Among all non-neighbors of  $u$ , choose  $p_1$  such that  $|N(u) \cap N(p_1)|$  is maximized.

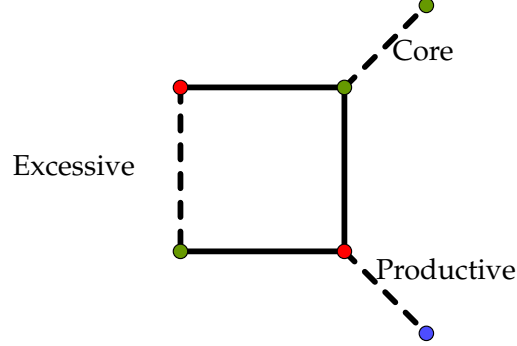


Figure 5.3: Different categories of queries (the colors of the vertices denote their label in the partition).

If  $G(up_1)$  is not chordal, there is an induced cycle  $C = (u, p_1, \dots, p_k, u)$ , with  $k \geq 3$ . As  $p_k \in N(u) \cap N(p_{k-1}) \setminus N(p_1)$ , our choice of  $p_1$  guarantees that there is a vertex  $w \in N(u) \cap N(p_1) \setminus N(p_{k-1})$ . Let  $j$  be the largest index in  $\{1, \dots, k-2\}$  such that  $wp_j \in E$ . Then, depending on whether the edge  $wp_k$  is present, either  $(w, p_j, \dots, p_k, u, w)$  or  $(w, p_j, \dots, p_k, w)$  is an induced cycle of length at least 4 in  $G$ , a contradiction.  $\square$

### 5.2.2 Optimality of chordal algorithms

In this section, we prove that the optimal algorithms are exactly the chordal algorithms.

We define three categories of queries as depicted in Figure 5.3:

- Queries that receive a positive answer are called *core* queries
- Queries at time  $t$  are *excessive* if they compare vertices  $x$  and  $y$  that are joined by an induced path in  $G_t$  on an even number of vertices that alternates between two partition classes
- Queries that are neither core nor excessive are *productive*

The first two categories are mutually exclusive as excessive queries always receive a negative answer. Thus, each query belongs to exactly one category. We now characterize the overall average complexity by looking at each category.

**Lemma 3.** *For any algorithm and any partition of a set of size  $n$  containing  $k$  classes, the number of core queries is exactly  $n - k$ .*  $\square$

*Proof.* If a query is core, the number of nodes of the contracted graph decreases by exactly one and it is the only case where this number changes. Since the algorithm starts from a graph with  $n$  nodes and ends with a graph of  $k$  nodes, there has to be  $n - k$  core queries.  $\square$

We then look at excessive queries.

**Lemma 4.** *For each non-chordal algorithm, there is an input partition and a time  $t$  such that  $G_t$  has an induced  $C_4$  one of whose edges comes from a negative query in step  $t - 1$ .*

*Proof.* Since the algorithm is not chordal, for some input partition one of the aggregated graphs  $G_t$  of the algorithm has an induced cycle  $C$  of length at least 4. Consider a realization of  $G_t$  where the vertices of  $C$  are all in distinct partition classes. Then there is a time  $t'$  when four of the vertices of  $C$  form an induced  $C_4$  in  $G_{t'}$ .  $\square$

Then, we can prove:

**Lemma 5.** *An algorithm makes no excessive queries if and only if it is chordal.*

*Proof.* By definition, a chordal algorithm has no aggregated graphs with induced cycles of length at least 4. So, since an excessive query, if answered negatively, creates an induced cycle of even length at least 4, chordal algorithms make no excessive queries.

Conversely, Lemma 4 ensures that for any non-chordal algorithm, there exists a partition and a contracted graph  $G_t$  such that  $G_t$  has an induced  $C_4 = (t, u, v, w)$  such that the last asked query was  $(w, t)$ . There exists a partition for which  $t$  and  $v$  are in one cluster and  $u$  and  $w$  are in another one, meaning that  $(w, t)$  was an excessive query. Thus for all non-chordal algorithms, there exists a partition for which an excessive query is made.  $\square$

We now consider productive queries. To that end, we first consider the set  $\mathcal{P}_2(n)$  of partitions with at most two blocks and  $\mathcal{P}_2^*(n) = \mathcal{P}_2(n) \setminus \mathcal{P}_1(n)$ , the set of partitions with exactly two blocks. In this particular subcase, an additional property can be used: if one considers three nodes  $u, v$  and  $w$  where  $(u, v)$  and  $(v, w)$  are both negative edges, then, assuming the underlying partition is in  $\mathcal{P}_2(n)$ , it is known that  $u$  and  $w$  are in the same block and that  $v$  is in the only other block. Intuitively, this corresponds to the fact that "the enemy of my enemy is my friend" applies only when there are two blocks.

**Lemma 6.** *The expected number of productive queries made by any algorithm on a random partition from  $\mathcal{P}_2(n)$  is exactly  $\frac{n-1}{2}$ .*

*Proof.* We run some algorithm on a partition from  $\mathcal{P}_2(n)$  and color each node depending on its cluster. At each  $t$ , the connected components of  $G_t$  are all bipartite graphs (where each of the two node sets corresponds to a cluster in the partition). Every such component can be colored in two ways; thus the number of possible colorings of  $G_t$  is  $2^{n_{\text{com}}(G_t)}$  where  $n_{\text{com}}$  denotes the number of connected components of  $G_t$ .

Let us prove by induction that:

- The  $2^{n_{\text{com}}(G_t)}$  colorings of  $G_t$  are equally likely.
- If the  $t$ -th query joins two components of  $G_{t-1}$  then it is productive with probability half.

The initialization is clear as in  $G_0$ , each node is in its own connected component. Let  $(u, v)$  be the query made at the  $t$ -th step. If  $u$  and  $v$  are in the same connected component, the query is excessive and the first induction hypothesis holds.

Let us then consider the case where  $u$  and  $v$  lie in distinct connected components  $H_u$  and  $H_v$ . The four colorings of  $H_u \cup H_v$  are equally likely and independent from the coloring of the other components of the graph. Then,  $u$  and  $v$  have the same color with probability  $\frac{1}{2}$  and the probability that  $(u, v)$  is productive is also  $\frac{1}{2}$ . Depending on the colors of  $u$  and  $v$ , there are only two possible colorings for  $H_u \cup H_v$  and they are equally likely. This concludes the induction.

As each query joining two components decreases the number of connected components by exactly one, there are exactly  $n - 1$  such queries and by linearity of the expected number of productive queries at each step, the total expected number of productive queries is  $(n - 1)/2$ .  $\square$

**Lemma 7.** *The expected number of productive queries made by any algorithm on a random partition from  $\mathcal{P}_2^*(n)$  is exactly*

$$\frac{2^n}{2^n - 2} \frac{n - 1}{2}.$$

*Proof.* Let us consider an algorithm and denote its expected number of productive queries by  $\alpha(n)$ . We pick a partition  $P$  uniformly at random from  $\mathcal{P}_2(n)$ . If  $P$  is constant, then the algorithm only makes core queries, in particular, it makes no productive queries. This happens with probability  $\frac{2}{2^n}$ . Conditioning on  $P$  being nonconstant is equivalent to picking it uniformly from  $\mathcal{P}_2^*(n)$ . By Lemma 6, we conclude that the expected number of productive queries satisfies:

$$\frac{n - 1}{2} = \alpha(n) \cdot \mathbb{P}(P \text{ nonconstant}) + 0 \cdot \mathbb{P}(P \text{ constant}) = \frac{2^n - 2}{2^n} \alpha(n).$$

Hence:

$$\alpha(n) = \frac{2^n}{2^n - 2} \frac{n - 1}{2}.$$

$\square$

We now use the previous result to prove that all algorithms have the same expected number of productive queries on any random partition picked uniformly from  $\mathcal{P}_k(n)$  (the set of partitions of  $n$ -sets into exactly  $k$  sets)

**Lemma 8.** *All algorithms have the same expected number of productive queries on a random partition from  $\mathcal{P}_k(n)$ .*

*Proof.* Fix  $k$  and consider a partition of  $[n]$  into exactly  $k$  sets  $C_1, C_2, \dots, C_k$ . Let  $\alpha_{ij}$  denote the expected number of productive queries that compare a vertex from  $C_i$  with a vertex from  $C_j$ . For  $i = j$ ,  $\alpha_{ij} = 0$ . We thus assume  $i \neq j$ .

Let  $q_{ij}$  be the number of productive queries comparing a vertex from  $C_i$  with a vertex from  $C_j$ . (so  $\alpha_{ij} = \mathbb{E}(q_{ij})$ ) Then, considering the set  $S = C_i \cup C_j$ , and applying Lemma 7, we

obtain

$$\begin{aligned}\alpha_{ij} &= \sum_{S \subseteq [n], |S| \geq 2} \mathbb{E}(q_{ij} | C_i \cup C_j = S) \mathbb{P}(C_i \cup C_j = S) \\ &= \sum_{S \subseteq [n], |S| \geq 2} \frac{|S| - 1}{2} \frac{2^{|S|}}{2^{|S|} - 2} \mathbb{P}(C_i \cup C_j = S)\end{aligned}$$

Because clusters cannot be distinguished a priori, all  $\alpha_{ij}$  are equal, and by linearity of expectation the expected number of productive queries is

$$\binom{k}{2} \sum_{S \subseteq [n], |S| \geq 2} \frac{|S| - 1}{2} \frac{2^{|S|}}{2^{|S|} - 2} \mathbb{P}(C_i \cup C_j = S)$$

which is independent from the choice of the algorithm. □

**Theorem 2.** *On partitions of size  $n$  chosen uniformly at random, an algorithm has minimal average complexity if and only if it is chordal.*

*Proof.* By Lemmas 3 and 8, all algorithms have the same expected number of core queries and productive queries. So the optimal algorithms are the ones with the minimum expected number of excessive queries. By Lemma 5, these are the chordal algorithms. □

### 5.2.3 Cost equivalence of chordal algorithms

Theorem 2 states that optimal algorithms are all the chordal algorithms. A simple corollary of this statement is that all chordal algorithms have the same (minimal) average complexity. This section is devoted to the proof of Corollary 1 which states that the complexity of all chordal algorithms have the same distribution.

The general idea for this theorem is to proceed by induction on the number of non-edges. The base case is trivial as a complete graph allows just one possible partition discovery algorithm which is chordal and necessarily optimal. The inductive step consists in showing that two consecutive queries can be interchanged without impacting the complexity. This is tricky as one has to deal with disjoint cases depending on whether the asked queries leave the graph chordal or not. When the resulting graph is not chordal, we derive some properties on the structure of the graph (depending on whether the queries asked are incident or not). Those structures are described in Lemmas 12 and 13.

**Lemma 9.** *Let  $G$  be a chordal graph, let  $u, v \in V$  be distinct and non-adjacent, and assume  $G(uv)$  is chordal. Then  $N(u) \cap N(v)$  is complete and separates  $u$  and  $v$  (in  $G$ ).*

*Proof.* Note that  $N(u) \cap N(v)$  is complete, as otherwise there are two non-adjacent vertices  $x, y \in N(u) \cap N(v)$ , and  $(u, x, v, y)$  is an induced cycle in  $G$ , resulting in a contradiction. It remains to show that  $N(u) \cap N(v)$  separates  $u$  from  $v$ . Assume this is not the case, then there is an induced path  $P = (u, p_1, \dots, p_k, v)$  (with  $k \geq 2$ ) that does not intersect

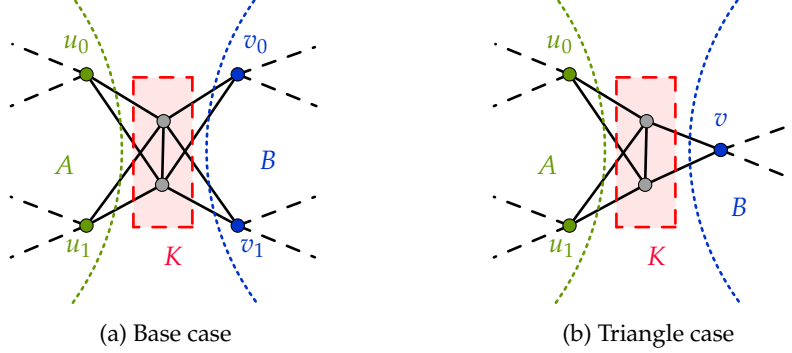


Figure 5.4: Structure of the graph when adding two edges prevents chordality.

$N(u) \cap N(v)$ . Adding the edge  $uv$  to  $P$ , we obtain an induced cycle of length at least 4, a contradiction to  $G(uv)$  being chordal.  $\square$

We now give a characterization of graphs that remain chordal when we add either one of two edges, but not if we add both. We give an illustration in Figure 5.4.

**Lemma 10.** *Let  $G$  be a chordal graph and let  $u_0, u_1, v_0, v_1 \in V$  such that for  $i = 0, 1$ , vertices  $u_0, u_1, v_i$  are all distinct,  $u_i v_i \notin E$ , and  $G(u_i v_i)$  is chordal. If  $G(u_0 v_0)(u_1 v_1)$  is not chordal, then  $K := N(u_0) \cap N(u_1) \cap N(v_0) \cap N(v_1)$  is complete, and  $G - K$  has two distinct components  $A$  and  $B$ , such that either  $u_0, u_1 \in A$  and  $v_0, v_1 \in B$ , or  $u_0, v_1 \in A$  and  $u_1, v_0 \in B$ .*

*Proof.* As  $G(u_0 v_0)(u_1 v_1)$  is not chordal, we know that  $G(u_0 v_0)(u_1 v_1)$  has an induced cycle  $C = (u_1, v_1, \dots, v_{\ell-1}, v_\ell, \dots, v_k, u_1)$ , with  $k \geq \ell \geq 2$ , where either  $v_{\ell-1} = v_0$  and  $v_\ell = u_0$ , or  $v_{\ell-1} = u_0$  and  $v_\ell = v_0$ . According to Lemma 9, since  $G(u_i v_i)$  is chordal, the set  $K_i := N(u_i) \cap N(v_i)$  is complete and separates  $u_i$  and  $v_i$  in  $G$ , for each  $i \in \{0, 1\}$ . Since  $C$  has at least four vertices, and  $K_i$  has neighbors  $u_i, v_i$ , we know that  $V(C) \cap K_i = \emptyset$ , for  $i = 0, 1$ .

Assume there is a vertex  $x \in K_0 \setminus K_1$ . Then  $(v_1, \dots, v_{\ell-1}, x, v_\ell, \dots, v_k, u_1)$  is a path in  $G - K_1$ , in contradiction to the fact that  $K_1$  separates  $u_1$  from  $v_1$ . So  $K_0 \subseteq K_1$ , and with the help of a symmetric argument we see that  $K_0 = K_1$ . In order to finish the proof it suffices to note that the paths  $(v_1, \dots, v_{\ell-1})$  and  $(v_\ell, \dots, v_k)$  ensure that there are components  $A$  and  $B$  as desired.  $\square$

We now see that a graph that is obtained by assembling two graphs along a complete subgraph is chordal if and only if the two smaller graphs are. This is a partial converse of the fact that all subgraphs of a chordal graph are chordal.

**Lemma 11.** *Let  $G$  be a graph, let  $A, B, K$  be a partition of  $V$  such that  $G[K]$  is complete and there are no edges between  $A$  and  $B$ . Then  $G$  is chordal if, and only if  $G[A \cup K]$  and  $G[K \cup B]$  are both chordal.*

*Proof.* As induced subgraphs of chordal graphs are chordal, we only need to show that if both  $G[A \cup K]$  and  $G[K \cup B]$  are both chordal, then so is  $G$ . For this, it suffices to observe

that any cycle of  $G$  that contains vertices from both  $A$  and  $B$  has to pass twice through  $K$ .  $\square$

The next two lemmas give a structural characterization of aggregated graphs where consecutive queries cannot easily be interchanged. In order to make their statement easier, let us say that a graph  $G$  has a *complete separation*  $(A, K, B)$  if  $V$  is the disjoint union of  $A, B, K$  so that  $A \neq \emptyset \neq B$ , each of  $A \cup K$  and  $B \cup K$  is complete, and there are no edges from  $A$  to  $B$  (observe that we allow  $K$  to be empty).

The next lemma asserts that, in case the two non-edges to be added prevent chordality when they are both added, there exists a third non-edge which makes can be added to either of the two previous non-edges and leave the graph chordal.

**Lemma 12.** *Let  $G$  be a chordal graph that does not have a complete separation. For  $i = 0, 1$  let  $u_i v_i$  be a non-edge of  $G$  such that  $G(u_i v_i)$  is chordal, and  $G(u_0 v_0)(u_1 v_1)$  is non-chordal. Then there is a non-edge  $uv$  of  $G$  such that  $G(uv)$  is chordal, and  $G(u_i v_i)(uv)$  is chordal for  $i = 0, 1$ .*

*Proof.* Use Lemma 10 to see that the intersection  $K$  of the neighborhoods of  $u_0, v_0, u_1, v_1$  is either a clique or empty, and  $G - K$  has two connected components  $A$  and  $B$  such that  $u_0, u_1 \in A$  and  $v_0, v_1 \in B$  (after possibly changing the roles of  $u_0$  and  $v_0$ ).

As  $G$  has no complete separation, and as there are no edges between  $A$  and  $B$ , one of  $A \cup K, B \cup K$  has to have a non-edge; because of symmetry we can assume this is  $A \cup K$ . According to Lemma 11, the subgraph of  $G$  induced by  $A \cup K$  is chordal. Then, according to Lemma 2, there is also a non-edge  $uv$  with  $u, v \in A \cup K$  having the additional property that  $G(uv)$  is chordal. As  $K$  is complete, we can assume that  $u \in A$ .

If there is no non-edge  $uv$  as desired, we have that  $G(u_i v_i)(uv)$  is non-chordal for some  $i \in \{0, 1\}$ ; by symmetry, let us assume  $G(u_1 v_1)(uv)$  is non-chordal. So, we may apply Lemma 10 to see that the intersection  $K'$  of the neighborhoods of  $u, v, u_1, v_1$  is either a clique or empty, and  $G - K'$  has two connected components  $A', B'$  such that  $u, u_1 \in A'$  and  $v, v_1 \in B'$  (after possibly changing the roles of  $u$  and  $v$ ). Note that  $K' \subseteq N(u_1) \cap N(v_1) \subseteq K$ . Furthermore,  $K \subseteq K'$ , since  $K'$  separates  $u_1$  from  $v_1$  and  $K \subseteq N(u_1) \cap N(v_1)$ . So  $K = K'$ .

In particular,  $v \notin K$ , that is,  $v \in A$ . So, as  $v_1 \in B$ , we know that  $v, v_1$  lie in distinct components of  $G - K$ . However, we also have that  $v, v_1$  belong to the same component (namely,  $A$ ) of  $G - K' = G - K$ , a contradiction. So the desired non-edge  $uv$  exists.  $\square$

**Lemma 13.** *Let  $G$  be a chordal graph that does not have a complete separation. Let  $u_0, u_1, v \in V$  such that for  $i = 0, 1$ , we have  $u_i v \notin E$  and  $G(u_i v)$  is chordal,  $G(u_0 v)(u_1 v)$  is non-chordal, and moreover,  $u_0 u_1 \in E$ . Then there is a non-edge  $uw$  of  $G$  such that  $G(uw)$  is chordal, and  $G(u_i v)(uw)$  is chordal for  $i = 0, 1$ .*

*Proof.* We start by proving that  $N(u_0) \cap N(v) = N(u_1) \cap N(v)$ . For this assume there is an  $i \in \{0, 1\}$  and a vertex  $x \in (N(u_{1-i}) \cap N(v)) \setminus N(u_i)$ . Then  $(u_{1-i}, x, v, u_i, u_{1-i})$  is an induced cycle of length 4 in  $G(u_i v)$ , a contradiction since this graph is chordal. This proves the equality, and we set  $K := N(u_0) \cap N(v) = N(u_1) \cap N(v)$ .

Because of Lemma 9,  $G[K]$  is complete and separates  $u_0, u_1$  from  $v$ . Since  $G$  does not have a complete separation, at least one of  $G[A \cup K], G[B \cup K]$  is not complete, but by Lemma 11 both are chordal. So by Lemma 2 and, again, Lemma 11, there is a non-edge

$uw$  such that  $G(uw)$  is chordal. Now, if  $uw$  is not as desired, say because  $G(u_0v)(uw)$  is non-chordal, then there is an induced cycle  $C$  of length at least 4 going through both  $uw$  and  $u_0v$ . However,  $C$  has to meet  $K$ , which implies  $C$  is a triangle, a contradiction.  $\square$

We will prove a more general result than needed which allows for the algorithm to start with any aggregated graph instead of starting with the empty graph. More precisely, if  $G$  is an aggregated graph at time  $t$  for some algorithm for an  $n$ -set, then we call the restriction of the algorithm to all queries after time  $t$  that eventually lead to a realization of  $G$  an *algorithm starting at  $G$* . We define the *complexity distribution* of this algorithm analogously to our earlier definition. In particular, if  $G$  is complete, then the complexity distribution is trivial as no questions are asked with probability 1.

**Theorem 3.** *For any chordal  $G$ , all chordal algorithms starting at  $G$  have the same complexity distribution.*

*Proof.* We proceed by induction on the number of non-edges of  $G$ . Proving the base case is trivial as, starting from a complete graph, the only algorithm has a trivial distribution for its complexity where the only possibility is to ask no questions.

For the induction step assume that for any chordal graph with  $k$  or less missing edges, all chordal algorithms have the same complexity distribution, and consider a graph  $G$  with  $k + 1$  missing edges. Let  $A_0, A_1$  be two distinct chordal algorithms for  $G$ . If their first queries are the same, say they query the edge  $e$ , then by induction we know that for both  $G_e$  and  $G(e)$ , the two algorithms have the same distribution if we let them start there. As the distribution for an algorithm starting at  $G$  is uniquely obtained from the complexity distributions of the same algorithm starting at  $G_e$  and at  $G(e)$ , we see that  $A_0$  and  $A_1$  have the same complexity distribution.

So we can assume that  $A_0$  and  $A_1$  differ in their first queries. Say the first query of  $A_i$  is  $u_i, v_i$ , for  $i = 0, 1$ . Then  $G(u_iv_i)$  is chordal for  $i = 0, 1$ . Note that we can assume that  $u_0 \neq u_1$ . We will distinguish two cases.

First, let us assume that  $G(u_0v_0)(u_1v_1)$  is chordal and moreover, if  $v_0 = v_1$  then  $u_0u_1 \notin E$ . Then, for  $i = 0, 1$ , the edge  $u_iv_i$  can be chosen as the first edge of a chordal algorithm for  $G(u_{1-i}v_{1-i})$  or for  $G_{u_{1-i}v_{1-i}}$ . As the induction hypothesis applies to  $G(u_{1-i}v_{1-i})$  and to  $G_{u_{1-i}v_{1-i}}$ , we can assume that  $u_iv_i$  is the second edge in  $A_{1-i}$ . Observe that for each  $i = 0, 1$  after the second query of  $A_i$ , we arrive at one of the four graphs  $(G_{u_0v_0})_{u_1v_1}$ ,  $G(u_0v_0)_{u_1v_1}$ ,  $G(u_1v_1)_{u_0v_0}$ ,  $G(u_0v_0)(u_1v_1)$ . Thus the complexity distribution of  $A_0$  and  $A_1$  is identical (as it can be computed from the complexity distribution for the algorithms starting at these four graphs).

Now, let us assume that either  $G(u_0v_0)(u_1v_1)$  is chordal,  $v_0 = v_1$  and  $u_0u_1 \in E$ , or  $G(u_0v_0)(u_1v_1)$  is not chordal. Then, by Lemmas 12 and 13, we know that either there is an edge  $uv \in E$  such that  $G(uv)$ ,  $G(uv)(u_0v_0)$  and  $G(uv)(u_1v_1)$  are chordal, or  $V$  can be partitioned into three sets,  $A, B$  and  $K$ , such that  $A \cup K$  and  $B \cup K$  are complete and  $K$  separates  $A$  from  $B$ . If the former is the case, we can proceed as in the previous paragraph to see that every chordal algorithm starting with  $u_0v_0$  has the same complexity distribution as any of the chordal algorithms starting with  $uv$  (note that such algorithms exist by Lemma 2), which, in turn, has the same complexity distribution as any of the chordal algorithms starting with  $u_1v_1$ , leading to the desired conclusion.



So assume there are sets  $A, B$  and  $K$  as above. By symmetry, we can assume that  $u_0, u_1 \in A$  and  $v_0, v_1 \in B$ . Consider the automorphism  $\sigma$  of  $G$  that maps  $u_0$  to  $u_1$  and  $v_0$  to  $v_1$  while keeping all other vertices fixed. We can now view  $A_0$  as an algorithm in  $\sigma(G)$  that starts with the edge  $u_1 v_1$ . By the induction hypothesis, we conclude that  $A_1$  (for  $G$ ) has the same distribution as  $A_0$  (for  $\sigma(G)$ ), and thus also for  $G$ .  $\square$

**Corollary 1.** *On partitions of size  $n$  chosen uniformly at random, all chordal algorithms have the same complexity distribution.*

## 5.2.4 Complexity estimates

In this section, we look for the asymptotic mean and standard deviation of chordal algorithms when the item set is large. Let  $X_n$  denote the number of pairwise queries needed to partition the  $n$ -set using any chordal algorithm. We seek to characterize the limit law of  $X_n$  when  $n \rightarrow \infty$ . In doing so, we obtain exact formulae for the complexity distribution of chordal algorithms for any  $n$ . They are found in Corollary 2 and Theorem 6.

**$q$ -integers** We will encounter  $q$ -integers (Ernst, 2000). A  $q$ -analog is a generalization of a known result where one introduces an additional parameter  $q$  such that the original result can be found when  $q \rightarrow 1$ .

$q$ -integers stem from the sum of a geometric sequence, for any non-negative integer  $n$ :

$$n = \lim_{q \rightarrow 1} (1 + q + q^2 + \cdots + q^{n-1}) = \lim_{q \rightarrow 1} \frac{1 - q^n}{1 - q}$$

This enables us to define a natural  $q$ -analog of any integer  $n$  as:

$$[n]_q = 1 + q + \cdots + q^{n-1} = \frac{1 - q^n}{1 - q}.$$

The  $q$ -factorial of the integer  $n$  is:

$$[n]_q! = \prod_{j=1}^n [j]_q.$$

The  $q$ -exponential is defined as:

$$e_q(z) = \sum_{n \geq 0} \frac{z^n}{[n]_q!}.$$

The  $q$ -Pochhammer symbol is defined as:

$$(a; q)_n = \prod_{k=0}^{n-1} (1 - aq^k)$$

Observe that all  $q$ -analogs reduce to their classic counterparts for  $q = 1$ .

We will need some alternative formulas which we gather in the following proposition for future reference:

**Proposition 5.**

$$\begin{aligned}
1. (x; q)_n &= \frac{(x; q)_\infty}{(xq^n; q)_\infty} & 3. \frac{1}{(x; q)_\infty} &= \sum_{n \geq 0} \frac{x^n}{(q; q)_n} \\
2. [n]_q! &= \frac{(q; q)_n}{(1 - q)^n} & 4. e_q(x) &= ((1 - q)x; q)_\infty^{-1}
\end{aligned}$$

*Proof.*

1. The result stems from the simplification of the products in the right hand member.

$$2. [n]_q! = \frac{\prod_{j=1}^n (1 - q^j)}{\prod_{j=1}^n (1 - q)} = \frac{\prod_{j=0}^{n-1} (1 - q \cdot q^j)}{\prod_{j=1}^n (1 - q)} = \frac{(q; q)_n}{(1 - q)^n}$$

3. The  $q$ -binomial theorem states (see, e.g., Andrews (1986)):

$$(x; q)_\infty^{-1} = \sum_{n=0}^{\infty} \frac{x^n}{[n]_q! (1 - q)^n} = \sum_{n=0}^{\infty} \frac{x^n}{(q; q)_n}$$

$$4. \text{ Using the two previous properties: } ((1 - q)x; q)_\infty^{-1} = \sum_{n \geq 0} \frac{(1 - q)^n x^n}{(q; q)_n} = \sum_{n \geq 0} \frac{x^n}{[n]_q!}$$

□

**Universal algorithm** We consider the universal algorithm, described (in a recursive fashion) in Algorithm 5 which consists in asking all possible questions related to one element before selecting the next element.  $\text{QUERY}(u, v)$  returns *True* if  $u$  and  $v$  are in the same class and *False* otherwise. Note that exactly one block of the partition is unveiled at each call of the  $\text{UNIALG}$  function.

**Proposition 6.** *The universal algorithm is a chordal algorithm.*

*Proof.* Using the notation of Algorithm 5, let us denote node  $u$  as the *central node* of its step. Let  $\mathcal{C} = (u, a_0, a_1, \dots, a_p, v)$  be a cycle of length  $n \geq 4$ ,  $a_0$  and  $a_p$  are distinct. Then:

- either  $a_0$  or  $a_p$  have already been central and  $a_0 a_p$  is a chord
- otherwise, if say  $a_0$  has not been central, then  $u$  and  $a_1$  have been and  $u a_1$  is a chord

□

Thanks to Corollary 1, we can assert that all chordal algorithms will share the asymptotics of the universal one.

Denote the universal algorithm by  $\mathcal{A}$ , the set of all partitions by  $\mathcal{P}$ . Let:

$$P(z, q) = \sum_{p \in \mathcal{P}} q^{\#\text{queries}_{\mathcal{A}}(p)} \frac{z^{|p|}}{|p|!}$$

---

**Algorithm 5:** The universal algorithm UNIALG
 

---

**Result:** The underlying partition

**Input:** A set  $S$

```

if  $S \neq \emptyset$  then
   $u \leftarrow \text{POP}(S)$ 
   $B \leftarrow \{u\}$ 
  for  $v \in S$  do
     $q \leftarrow \text{QUERY}(u, v)$ 
    if  $q = \text{True}$  then
       $B \leftarrow B \cup \{v\}$ 
    end
   $\mathcal{P} \leftarrow \text{UNIALG}(S \setminus B)$ 
  return  $\mathcal{P} \cup \{B\}$ 
end
else
  return  $\emptyset$ 
end

```

---

with  $\#_{\text{queries}}_{\mathcal{A}}(p)$ , the number of queries required by the universal algorithm to unveil partition  $p$  starting from the graph with  $|p|$  nodes and no edges.

**Theorem 4.**  $P(z, q)$  satisfies  $P(0, q) = 1$  and the following differential equation:

$$\partial_z P(z, q) = P(qz, q)e^{qz} \quad (5.1)$$

*Proof.* Let  $a_{n,k}$  be the number of partitions over  $n$ -sets that require  $\mathcal{A}$  to ask  $k$  queries to be found. Then  $P$  can be written:

$$P(z, q) = \sum_{n,k \geq 0} a_{n,k} q^k \frac{z^n}{n!} \quad (5.2)$$

We seek an inductive relation for  $a_{n,k}$ . To get all partitions on  $(n+1)$ -sets requiring  $\mathcal{A}$  to make  $k$  queries, consider the number  $m$  of elements in the same cluster as the first element.

In the first round of queries, where all elements are compared to the first one,  $n$  queries are asked and  $m$  of them yield a positive result. Then,  $\mathcal{A}$  behaves exactly as if on a  $n-m$  set over the remaining elements and all  $(n-m)$  sets can be reached that way.

From this, we derive:

$$a_{n+1,k} = \sum_{m=0}^n \binom{n}{m} a_{n-m,k-n} \quad (5.3)$$

as we are only interested in cases that result in a total number of queries equal to  $k$ . Summing Equation (5.3) over  $n$  and  $k$  so as to find  $P(z, q)$  as defined in 5.2, we get:

$$\sum_{n,k \geq 0} a_{n+1,k} q^k \frac{z^n}{n!} = \sum_{n,k \geq 0} \sum_{m=0}^n \binom{n}{m} a_{n-m,k-n} q^k \frac{z^n}{n!}$$

Setting,  $n' = n - m$  and  $k' = k - n$  so that  $n = n' + m$  and  $k = k' + n' + m$ :

$$\begin{aligned} \sum_{n,k \geq 0} a_{n+1,k} q^k \frac{z^n}{n!} &= \sum_{n',k',m \geq 0} a_{n',k'} q^{k'+n'+m} \frac{z^{n'+m}}{n'!m!} \\ &= e^{qz} \sum_{n',k' \geq 0} a_{n',k'} q^{k'} \frac{(qz)^{n'}}{n'!} \\ \partial_z P(z, q) &= P(qz, q) e^{qz} \end{aligned}$$

If the set under study is empty, only the empty partition can be found and thus  $P(0, q) = 1$   $\square$

We now seek to solve Equation (5.1). In particular, we would like to have an explicit expression of  $a_{n,k}$ . Let  $f(z, q) = e^{\frac{q}{1-q}z}$  be a solution of a similar equation as  $\partial_z f(z, q) = \frac{q}{1-q} f(z, q)$  and  $f(qz, q) e^{qz} = e^{\frac{1-q+q}{1-q}qz} = f(z, q)$  so  $\partial_z f(z, q) = \frac{q}{1-q} f(qz, q) e^{qz}$ .

**Theorem 5.** Let  $\text{Poch}(z, a, q)$  denote the exponential generating function associated to the  $q$ -Pochhammer symbol

$$\text{Poch}(z, a, q) = \sum_{k \geq 0} (a; q)_k \frac{z^k}{k!},$$

then the generating function of the universal algorithm complexity is

$$P(z, q) = \text{Poch}\left(-\frac{q}{1-q}z, \frac{1-q}{q}, q\right) e^{\frac{q}{1-q}z}$$

The following corollary derives the complexity distribution of chordal algorithms for any finite number of elements  $n$ .

**Corollary 2.** The generating function for the number of queries of the universal algorithm on  $n$ -sets is:

$$P_n(q) = \left(\frac{q}{1-q}\right)^n \sum_{m=0}^n \binom{n}{m} (-1)^m \left(\frac{1-q}{q}; q\right)_m$$

*Proof of Theorem 5 and Corollary 2.* We investigate solutions of the differential equation 5.1 of the form  $P(z, q) = A(z, q) e^{\frac{q}{1-q}z}$ . The differential equation on  $P(z, q)$  implies the following differential equation for  $A(z, q)$

$$\partial_z A(z, q) + \frac{q}{1-q} A(z, q) = A(qz, q).$$

with initial condition  $A(0, q) = 1$ . Decomposing  $A(z, q)$  as a series in  $z$

$$A(z, q) = \sum_{k \geq 0} a_k(q) \frac{z^k}{k!},$$

we obtain a recurrence on  $a_k(q)$ :

$$a_{k+1}(q) = -\frac{q}{1-q} a_k(q) + q^k a_k(q) = -\frac{q}{1-q} \left(1 - (1-q)q^{k-1}\right) a_k(q),$$

with  $a_0(q) = 1$ . We deduce

$$a_k(q) = \left(-\frac{q}{1-q}\right)^k \prod_{j=0}^{k-1} \left(1 - \frac{1-q}{q} q^j\right) = \left(-\frac{q}{1-q}\right)^k \left(\frac{1-q}{q}; q\right)_k$$

To conclude the proof of the theorem, we observe that  $\text{Poch}\left(-\frac{q}{1-q}z, \frac{1-q}{q}, q\right) e^{\frac{q}{1-q}z}$  is indeed solution of the differential equation characterizing  $P(z, q)$ .

To prove the corollary, notice how  $P_n(q)$  is obtained by computing the Cauchy product of the series  $A(z, q)$  and  $f(z, q)$  and extracting the coefficient of  $z^n$ :

$$P_n(q) = n! \sum_{m=0}^n \frac{a_m(q)}{m!} \frac{1}{(n-m)!} \left(\frac{q}{1-q}\right)^{n-m} = \left(\frac{q}{1-q}\right)^n \sum_{m=0}^n \binom{n}{m} (-1)^m \left(\frac{1-q}{q}; q\right)_m$$

□

We provide a second expression for the complexity generating function better suited for an asymptotic analysis as it deals with positive terms only. It is a  $q$ -analog of the following classic Dobinski formula for the Bell numbers (Flajolet and Sedgewick, 2009), which counts the number of partitions of size  $n$ :

$$B_n = \frac{1}{e} \sum_{m \geq 0} \frac{m^n}{m!}$$

**Theorem 6.** *The generating function of the number of queries used by the universal algorithm on partitions of size  $n$  is*

$$P_n(q) = \frac{1}{e_q(1/q)} \sum_{k \geq 0} \frac{[k]_q^n}{[k]_q!} q^{n-k}$$

*The sum converges for  $q > 1/2$ .*

*Proof.* Starting from the expression of Corollary 2 and using Proposition 5:

$$\begin{aligned}
P_n(q) &= \left(\frac{q}{1-q}\right)^n \sum_{m=0}^n \binom{n}{m} (-1)^m \left(\frac{1-q}{q}; q\right)_m \\
&= \left(\frac{q}{1-q}\right)^n \left(\frac{1-q}{q}; q\right)_\infty \sum_{m=0}^n \binom{n}{m} (-1)^m \frac{1}{(q^{m-1}(1-q); q)_\infty} \\
&= \left(\frac{q}{1-q}\right)^n \left(\frac{1-q}{q}; q\right)_\infty \sum_{m=0}^n \binom{n}{m} (-1)^m \sum_{k \geq 0} \frac{(q^{m-1}(1-q))^k}{(q; q)_k} \\
&= \left(\frac{q}{1-q}\right)^n \left(\frac{1-q}{q}; q\right)_\infty \sum_{k \geq 0} \frac{(1-q)^k}{(q; q)_k} q^{-k} (-1)^n \sum_{m=0}^n \binom{n}{m} (-1)^{n-m} q^{mk} \\
&= \left(\frac{1-q}{q}; q\right)_\infty \sum_{k \geq 0} \frac{(1-q)^k}{(q; q)_k} \cdot \frac{(1-q^k)^n}{(1-q)^n} \cdot q^{n-k} \\
&= \frac{1}{e_q(1/q)} \sum_{k \geq 0} \frac{[k]_q^n}{[k]_q!} q^{n-k}.
\end{aligned}$$

We apply d'Alembert's criteria to find the values of  $q$  for which this formal sum converges:

$$\lim_{k \rightarrow \infty} \frac{\frac{[k+1]_q^n}{[k+1]_q!} q^{n-k-1}}{\frac{[k]_q^n}{[k]_q!} q^{n-k}} = \frac{1-q}{q}$$

is smaller than 1 when  $q > 1/2$ . □

**Asymptotics** Using the expression obtained in Theorem 6, we now seek to find the asymptotic mean and standard deviation of  $X_n$  for large  $n$ . Lemma 14 gathers some expressions of common  $q$ -integer derivatives for large  $n$  and  $m$ . Theorem 7 gives the asymptotic mean and standard deviation for the universal algorithm. To do so, we prove that the Laplace transform of the random variable  $X_n$  counting the number of queries converges to that of the desired result. This leads us to using the Laplace method in order to estimate the asymptotics of a sum derived from the previous lemmas.

**Lemma 14.** *As  $n$  and  $m$  tend to infinity,  $q = e^s$ ,  $ms$  and  $ns$  tend to 0, we have*

$$\begin{aligned}
[m]_q^n &= m^n \exp\left(\frac{1}{2}nms + \frac{1}{12}nm^2\frac{s^2}{2}\right) \left(1 + O(nm^3s^3 + ns)\right) \\
[m]_q! &= m^m e^{-m} \sqrt{2\pi[m]_q} \exp\left(\frac{1}{4}m^2s + \frac{1}{36}m^3\frac{s^2}{2}\right) \left(1 + O(m^4s^3 + ms) + o(1)\right).
\end{aligned}$$

*Proof.* Let us first introduce the function  $S(x) = \frac{e^x - 1}{x} - 1$ . Note that  $1 + x(1 + S(x)) = e^x$ .

Also,  $\log(1 + S(x)) = \log\left(\sum_{k \geq 0} \frac{x^k}{(k+1)!}\right)$  so that:

$$\log(1 + S(x)) = \log\left(1 + \frac{x}{2} + \frac{x^2}{6} + O(x^3)\right) = \frac{x}{2} + \frac{x^2}{24} + O(x^3)$$

Then,

$$[m]_q^n = \left(\frac{1 - e^{sm}}{1 - e^s}\right)^n = m^n \left(\frac{1 + S(sm)}{1 + S(s)}\right)^n = m^n e^{n \log(1 + S(sm)) - n \log(1 + S(s))}.$$

and

$$[m]_q^n = m^n \exp\left(\frac{1}{2}nms + \frac{1}{12}nm^2\frac{s^2}{2} + O(nm^3s^3 + ns)\right)$$

According to Moak (1984), we have the following  $q$ -analog of Stirling's formula when  $x \rightarrow \infty$  while  $x \log(q) \rightarrow 0$

$$\log(\Gamma_q(x)) = (x - 1/2) \log([x]_q) + \frac{\text{Li}_2(1 - q^x)}{\log(q)} + \frac{1}{2} \log(2\pi) + o(1)$$

where  $\text{Li}_2(z)$  denotes the Dilogarithm function

$$\text{Li}_2(z) = \sum_{k \geq 1} \frac{z^k}{k^2}$$

We deduce

$$\begin{aligned} [m]_q! &= [m]_q \Gamma_q(m) = [m]_q \exp\left((m - 1/2) \log([m]_q) + \frac{\text{Li}_2(1 - q^m)}{\log(q)} + \frac{1}{2} \log(2\pi) + o(1)\right) \\ &= [m]_q^m \exp\left(\frac{\text{Li}_2(1 - q^m)}{\log(q)}\right) \sqrt{2\pi [m]_q} (1 + o(1)) \end{aligned}$$

The first part of the lemma provides

$$[m]_q^m = m^m \exp\left(\frac{1}{2}m^2s + \frac{1}{12}m^3\frac{s^2}{2}\right) \left(1 + O(m^4s^3 + ms)\right).$$

The Dilogarithm is expanded as

$$\frac{\text{Li}_2(1 - q^m)}{\log(q)} = \frac{1}{s} \sum_{k \geq 1} \frac{1}{k^2} (-ms(1 + S(ms)))^k = -m \left(1 + \frac{1}{4}ms + \frac{1}{18}m^2\frac{s^2}{2} + O(ms)^3\right).$$

Injecting those past two expansions in the previous one concludes the proof.  $\square$

**Theorem 7.** *The asymptotic mean  $E_n$  and standard deviation  $\sigma_n$  of the number  $X_n$  of queries used*

by the universal algorithm on a partition of size  $n$  chosen uniformly at random are

$$E_n = \frac{1}{4}(2\zeta - 1)e^{2\zeta} \quad \text{and} \quad \sigma_n = \frac{1}{3}\sqrt{\frac{3\zeta^2 - 4\zeta + 2}{\zeta + 1}}e^{3\zeta},$$

where  $\zeta$  is the unique positive solution of

$$\zeta e^\zeta = n.$$

The normalized random variable

$$X_n^* = \frac{X_n - E_n}{\sigma_n}$$

follows in the limit a normalized Gaussian law.

*Proof.* To prove the limit law, we show that the Laplace transform  $\mathbb{E}(e^{sX_n^*})$  converges point-wise to the Laplace transform of the normalized Gaussian  $e^{s^2/2}$  (see e.g., Billingsley (2008)). We have

$$\mathbb{E}(e^{sX_n^*}) = e^{-sE_n/\sigma_n} \mathbb{E}(e^{sX_n/\sigma_n}) = \frac{e^{-sE_n/\sigma_n}}{B_n} P_n(e^{s/\sigma_n})$$

For any fixed real value  $s$ , we compute the asymptotics of  $P_n(e^{s/\sigma_n})$ . Let  $q := e^{s/\sigma_n}$ , so  $q$  tends to 1, then

$$P_n(e^{s/\sigma_n}) = \frac{1}{e_q(e^{-s/\sigma_n})} \sum_{m \geq 0} \frac{[m]_q^n}{[m]_q!} e^{(n-m)s/\sigma_n}.$$

Motivated by the asymptotics from Lemma 14, we rewrite this expression as

$$P_n(e^{s/\sigma_n}) = \frac{1}{e_q(e^{-s/\sigma_n})} \sum_{m \geq 0} A_{n,s}(m) e^{-\phi_{n,s}(m)} \quad (5.4)$$

where

$$A_{n,s}(m) = \frac{[m]_q^n}{m^n \exp\left(\frac{1}{2}nm\frac{s}{\sigma_n} + \frac{1}{12}nm^2\frac{(s/\sigma_n)^2}{2}\right)} \frac{m^m e^{-m} \exp\left(\frac{1}{4}m^2\frac{s}{\sigma_n} + \frac{1}{36}m^3\frac{(s/\sigma_n)^2}{2}\right)}{[m]_q!} e^{(n-m)s/\sigma_n},$$

$$\phi_{n,s}(m) = -n \log(m) + m \log(m) - m - \frac{1}{4}(2n - m)m\frac{s}{\sigma_n} - \frac{1}{36}(3n - m)m^2\frac{(s/\sigma_n)^2}{2}.$$

The dominant contribution to the sum comes from integers  $m$  close to the minimum of



$\phi_{n,s}(m)$ , so we study this function. The successive derivatives of  $\phi_{n,s}(m)$  are

$$\begin{aligned}\phi'_{n,s}(m) &= -\frac{n}{m} + \log(m) - \frac{1}{2}(n-m)\frac{s}{\sigma_n} - \frac{1}{12}(2n-m)m\frac{(s/\sigma_n)^2}{2}, \\ \phi''_{n,s}(m) &= \frac{n}{m^2} + \frac{1}{m} + \frac{s}{2\sigma_n} - \frac{1}{6}(n-m)\frac{(s/\sigma_n)^2}{2} \\ \phi'''_{n,s}(m) &= -\frac{2n}{m^3} - \frac{1}{m^2} + \frac{(s/\sigma_n)^2}{12}\end{aligned}$$

When  $n$  is large enough, the second derivative of  $\phi_{n,s}(m)$  is strictly positive for all  $m > 0$ , so the function is convex. It reaches its unique minimum at a value denoted by  $m(s)$  and characterized by  $\phi'_{n,s}(m(s)) = 0$ . Injecting the Taylor expansion

$$m(s) = m_0 + m_1 \frac{s}{\sigma_n} + m_2 \frac{(s/\sigma_n)^2}{2} + O\left((s/\sigma_n)^3\right)$$

in this equation, extracting the coefficients of the powers of  $s$ , we obtain:

$$\begin{cases} m_0 \log(m_0) - n = 0 \\ 2m_1 (\log(m_0) + 1) - nm_0 + m_0^2 = 0 \\ m_0^4 + 24m_0^2 m_1 + 12m_0 m_2 (\log(m_0) + 1) - 2(m_0^3 + 6m_0 m_1) n + 12m_1^2 = 0 \end{cases}$$

Hence, by replacing  $n$  with  $\zeta e^\zeta$ :

$$\begin{cases} m_0 = e^\zeta \\ m_1 = \frac{1}{2} \frac{\zeta-1}{\zeta+1} e^{2\zeta} \\ m_2 = \frac{1}{3} \frac{2\zeta^3-3\zeta^2+2}{(\zeta+1)^3} e^{3\zeta} \end{cases}$$

The dominant contribution to the sum defining  $P_n(e^{s/\sigma_n})$  comes from values of  $m$  close to  $m(s)$ . The *central part*  $C_n$  is defined as the integers  $m$  such that  $|m - m(s)| < c_n$ . A heuristic proposed by Flajolet and Sedgewick (2009) is to find  $c_n$  such that

$$|\phi''_{n,s}(m(s))|c_n^2 \rightarrow +\infty \quad \text{and} \quad |\phi'''_{n,s}(m(s))|c_n^3 \rightarrow 0.$$

As  $n$ , and thus  $\zeta$ , tend to infinity, we have:

$$m(s) \sim e^\zeta \sim \frac{n}{\log(n)}, \quad |\phi''_{n,s}(m(s))| \sim \zeta e^{-\zeta} \sim \frac{\log(n)^2}{n}, \quad |\phi'''_{n,s}(m(s))| \sim \zeta e^{-2\zeta} \sim \frac{\log(n)^3}{n^2},$$

so we choose  $c_n = e^{3\zeta/5}$ .

We define  $p(s) = m_0 + m_1 \frac{s}{\sigma_n} + m_2 \frac{(s/\sigma_n)^2}{2}$  and  $r(s) = C e^{-\zeta/2}$  for some constant  $C$ . Note that  $\phi'_{n,s}(p(s) + r(s)) > 0$ . Since  $\phi'_{n,s}$  is an increasing function,  $0 \leq m(s) - p(s) \leq r(s)$  and

$m(s) = p(s) + O(r(s))$ . Then for  $m$  in  $C_n$ :

$$\begin{aligned} m(s) &= e^\zeta + \frac{1}{2} \frac{\zeta - 1}{\zeta + 1} e^{2\zeta} \frac{s}{\sigma_n} + \frac{1}{3} \frac{2\zeta^3 - 3\zeta^2 + 2}{(\zeta + 1)^3} e^{3\zeta} \frac{(s/\sigma_n)^2}{2} + O(e^{-\zeta/2}), \\ \phi_{n,s}(m) &= -(\zeta^2 - \zeta + 1)e^\zeta - E_n \frac{s}{\sigma_n} - \frac{s^2}{2} + (\zeta + 1)e^{-\zeta} \frac{(m - m(s))^2}{2} + O(e^{-\zeta/4}), \\ A_{n,s}(m) &= \frac{1}{\sqrt{2\pi e^\zeta}} (1 + o(1)). \end{aligned}$$

We chose the values of  $E_n$  and  $\sigma_n$  so that the coefficients in  $s$  and  $s^2$  are the ones presented in the above equation. The error term is then obtained using the Lagrange form of the remainder in Taylor's Theorem. We deduce the following asymptotics for the central part of the sum:

$$\sum_{m \in C_n} A_{n,s}(m) e^{-\phi_{n,s}(m)} \sim \frac{1}{\sqrt{2\pi e^\zeta}} e^{(\zeta^2 - \zeta + 1)e^\zeta + E_n \frac{s}{\sigma_n} + \frac{s^2}{2}} \sum_{m \in C_n} e^{-(\zeta + 1)e^{-\zeta} \frac{(m - m(s))^2}{2}}.$$

Applying the Euler-Maclaurin formula to turn the sum into an integral, we obtain:

$$\sum_{m \in C_n} A_{n,s}(m) e^{-\phi_{n,s}(m)} \sim \frac{1}{\sqrt{2\pi e^\zeta}} e^{(\zeta^2 - \zeta + 1)e^\zeta + E_n s / \sigma_n + s^2 / 2} \int_{-c_n}^{c_n} e^{-(\zeta + 1)e^{-\zeta} \frac{x^2}{2}} dx.$$

After the variable change  $y = \sqrt{(\zeta + 1)e^{-\zeta}} x$ , observing that  $\sqrt{(\zeta + 1)e^{-\zeta}} c_n$  tends to infinity, the integral is approximated as a Gaussian integral and we conclude:

$$\sum_{m \in C_n} A_{n,s}(m) e^{-\phi_{n,s}(m)} \sim \frac{1}{\sqrt{\zeta + 1}} e^{(\zeta^2 - \zeta + 1)e^\zeta + E_n s / \sigma_n + s^2 / 2}.$$

When we compare the asymptotics of the central part to the asymptotics of the Bell numbers (see, e.g. Flajolet and Sedgewick (2009))

$$B_n \sim \frac{e^{(\zeta^2 - \zeta + 1)e^\zeta - 1}}{\sqrt{\zeta + 1}},$$

we see from Equation (5.4) that, as expected,

$$\frac{e^{-sE_n/\sigma_n}}{B_n} \frac{1}{e_q(e^{-s/\sigma_n})} \sum_{m \in C_n} A_{n,s}(m) e^{-\phi_{n,s}(m)} \sim e^{s^2/2}.$$

Let us now prove that the part of the sum corresponding to  $m \leq m(s) - c_n$  is negligible compared to the central part. According to Lemma 14, we have:

$$A_{n,s}(m) = \left(1 + O(nm^3/\sigma_n^3)\right) \left(1 + O(m^4/\sigma_n^3)\right) \frac{1}{\sqrt{2\pi[m]_q}} = O(m^8).$$

Since  $\phi_{n,s}(m)$  is convex (for large enough  $n$ ), we have  $\phi_{n,s}(m) \geq \phi_{n,s}(m(s) - c_n)$  for all  $m < m(s) - c_n$ . Since

$$\phi_{n,s}(m(s) - c_n) = -(\zeta^2 - \zeta + 1)e^\zeta - E_n \frac{s}{\sigma_n} - \frac{s^2}{2} + (\zeta + 1)e^{-\zeta} \frac{c_n^2}{2} + O(e^{-\zeta/4})$$

and  $e^{-\zeta} c_n^2$  tends to infinity as  $e^{\zeta/5}$ , we obtain for all  $m < m(s) - c_n$ :

$$\phi_{n,s}(m) \geq -(\zeta^2 - \zeta + 1)e^\zeta - E_n \frac{s}{\sigma_n} - \frac{s^2}{2} + \Theta(e^{\zeta/5}).$$

We conclude:

$$\begin{aligned} \sum_{m < m(s) - c_n} A_{n,s}(m) e^{-\phi_{n,s}(m)} &\leq \sum_{m < m(s) - c_n} O(m^8) e^{(\zeta^2 - \zeta + 1)e^\zeta + E_n s / \sigma_n + s^2 / 2 - \Theta(\exp(\zeta/5))} \\ &\leq e^{(\zeta^2 - \zeta + 1)e^\zeta + E_n s / \sigma_n + s^2 / 2} m(s)^9 e^{-\Theta(\exp(\zeta/5))}. \end{aligned}$$

Since  $m(s) \sim e^\zeta$ , this result is exponentially small, with respect to  $n$ , compared to the central part. Let us now prove that the part of the sum beyond the central part is negligible as well. There is a constant  $C$  such that for  $n$  large enough and any  $m \geq Ce^{3\zeta/2}$ , we have

$$-\frac{1}{4}(2n - m)m \frac{s}{\sigma_n} - \frac{1}{36}(3n - m)m^2 \frac{(s/\sigma_n)^2}{2} \geq 0.$$

In that case, we obtain the simple bound

$$\phi_{n,s}(m) \geq -n \log(m) + m \log(m) - m \geq m - n \log(m).$$

Injecting this bound and  $A_{n,s}(m) = O(m^8)$  in the sum, we obtain:

$$\sum_{m \geq Ce^{3\zeta/2}} A_{n,s}(m) e^{-\phi_{n,s}(m)} \leq O(1) \sum_{m \geq Ce^{3\zeta/2}} m^{n+8} e^{-m}.$$

The sum is bounded by an integral and  $n + 8$  integrations by part are applied

$$\begin{aligned} \sum_{m \geq Ce^{3\zeta/2}} A_{n,s}(m) e^{-\phi_{n,s}(m)} &\leq O(1)(n + 8)!(Ce^{3\zeta/2})^{n+8} e^{-C \exp(3\zeta/2)} \\ &\leq O(1)n^n (Ce^{3\zeta/2})^{n+8} e^{-C \exp(3\zeta/2)} \end{aligned}$$

Since  $n = \zeta e^\zeta$ , we have  $n^n = e^{O(\zeta^2) \exp(\zeta)}$ , so

$$\sum_{m \geq Ce^{3\zeta/2}} A_{n,s}(m) e^{-\phi_{n,s}(m)} \leq O(1) e^{-\Theta(\exp(3\zeta/2))}$$

which is negligible compared to the central part of the sum. The last part we consider is

$m(s) + c_n \leq m \leq Ce^{3\zeta/2}$ . Since  $\phi_{n,s}(m)$  is decreasing there and  $A_{n,s}(m) = O(m^8)$ , we have

$$\sum_{m=m(s)+c_n}^{Ce^{3\zeta/2}} A_{n,s}(m) e^{-\phi_{n,s}(m)} = O(e^{27\zeta/2}) e^{-\phi_{n,s}(m(s)+c_n)}$$

As for the case  $m = m(s) - c_n$ , we find

$$\phi_{n,s}(m(s) + c_n) = -(\zeta^2 - \zeta + 1)e^\zeta - E_n \frac{s}{\sigma_n} - \frac{s^2}{2} + \Theta(e^{\zeta/5})$$

and conclude

$$\sum_{m=m(s)+c_n}^{Ce^{3\zeta/2}} A_{n,s}(m) e^{-\phi_{n,s}(m)} = e^{(\zeta^2 - \zeta + 1)e^\zeta + E_n \frac{s}{\sigma_n} + \frac{s^2}{2}} O(e^{27\zeta/2}) e^{-\Theta(e^{\zeta/5})},$$

which is negligible compared to the central part. In conclusion, we have

$$\mathbb{E}(e^{sX_n^*}) = \frac{e^{-sE_n/\sigma_n}}{B_n} P_n(e^{s/\sigma_n}) \sim \sum_{m \in C_n} A_{n,s}(m) e^{-\phi_{n,s}(m)} \sim e^{s^2/2}.$$

Since the Laplace transform of  $X_n^*$  converges pointwise to the Laplace transform of the normalized Gaussian law,  $X_n^*$  converges in distribution to this Gaussian.  $\square$

We can derive a simpler equivalent of  $E_n$  for large  $n$ :

**Corollary 3.** *The asymptotic mean  $E_n$  of the number  $X_n$  of queries used by the universal algorithm on a partition of size  $n$  chosen uniformly at random satisfies  $E_n \sim \frac{n^2}{2 \log n}$*

*Proof.*  $\zeta e^\zeta = n$  means that  $\zeta = W(n)$  where  $W$  is the principal branch of the Lambert  $W$  function. This in turn means that  $\zeta = \log n - \log \log n + o(\log \log n)$  (see e.g. Corless et al. (1996)). Thus  $2\zeta - 1 \sim 2 \log n$  and  $e^{2\zeta} \sim \frac{n^2}{\log^2 n}$  hence the result.  $\square$

By comparison with the expression of Corollary 3, an exhaustive algorithm (i.e. one that queries all distinct pairs of elements) has the worst possible performance at  $\binom{n}{2} \sim \frac{n^2}{2}$  queries. This slim  $\frac{1}{\log n}$ -margin can be explained by the underlying partition distribution. A partition picked uniformly at random has  $\frac{n}{\log n}$  blocks on average (Flajolet and Sedgewick, 2009) and the number of negative queries is at least that number squared meaning that  $\frac{n^2}{\log^2 n}$  is a lower bound for  $E_n$  (as additional negative queries and positive queries remain).

### 5.3 Practical use

We set the theoretical foundations for chordal algorithms optimality in the setting of random partitions. While the results of Theorems 1 and 2 give some strong theoretical guarantees for chordal algorithms, we seek to know how those hold against a more practical

setting. Indeed, a number of assumptions made in the setting described in this chapter can be challenged. Additionally, parts of the work done in the previous section can be used to implement a chordal algorithm efficiently.

### 5.3.1 Considerations for implementations

When implementing any chordal algorithm, one needs to find which queries make it possible to keep the graph chordal at each step. Note that any query that keeps the graph chordal in the event of it being answered negatively also keeps the graph chordal when it is answered positively (see Lemma 1). Indeed, identifying two neighbors in a chordal graph results in a chordal graph. Suitable queries thus consist of non-edges that preserve chordality when added. A naive approach could consist in trial and error, i.e. adding any non-edge and checking that the resulting graph is chordal. Each such test can be performed in linear time in the size of the graph (Tarjan and Yannakakis, 1984). Alternatively, Lemma 9 makes it possible to rely on the chordality of graph to find a suitable query by checking that the intersection of the neighborhood of two nodes  $u$  and  $v$  separates them. In the worst case that is also done in linear time.

In a practical situation, it is preferable that several humans can work in parallel on the same item set. In particular, this requires that one can find a suitable query for an available human while some served queries are still pending. This is achievable thanks to the remark made earlier: it suffices to assume that any pending query will receive a negative answer. Thus, by choosing the next suitable query by considering the graph where all pending queries have received a negative answer, one can keep the graph chordal while allowing multiple humans to answer queries simultaneously.

### 5.3.2 Assistance by data-specific techniques

All the work done in the previous sections of this chapter make no assumption on the nature of the data to be labeled. This enables us to use the very general setting introduced earlier but, in practice, we expect to be able to use ad hoc techniques depending on the data. For instance, if all data items are texts, one could use text classifiers, after a few queries are asked, in order to reduce the number of queries (e.g. by finding positive queries and having the oracle answer them). Indeed, such classifiers and other ad hoc methods make use of additional information that was left aside: that of the data itself. If the data is labeled so that it can be subsequently used to train a supervised classifier, said classifier can be trained and used to predict the class of items as the data is labeled. This falls under the context of active learning which we do not explore here. We evaluate the importance of two distinct features for any algorithm:

- the chordality of the algorithm
- the use of supervised techniques

As classifiers proceed in a pointwise fashion, we proceed as follows: we pick the first edges to query at random until a negative answer is obtained and then train the classifier on the

items of the largest connected component of the contracted graph and their corresponding clustering (which is not yet final as some clusters may still be merged); the classifier is then used to predict the cluster of one node outside the largest component and the non-edge between that node and the corresponding node in the contracted graph is queried. When the largest component comprises all the graph, the classifier is not used anymore and queries are picked either at random or according to some chordal strategy. To illustrate this, we run experiments on the well-known 20 News Group dataset. We use two well-known text classifiers: a linear classifier with stochastic gradient descent (SGD) for training (Zadrozny and Elkan, 2002) and a multinomial naive Bayes (MNB) classifier (Christopher et al., 2008). We extract an increasing number of elements in the dataset at random. For each such set size, we perform 100 random extractions and average the results of each algorithm. We report the results in Figure 5.5.

In Figures 5.5a and 5.5b, we notice that supervised techniques give a consistent edge over unsupervised approaches be it in the random or chordal context. This is due to the fact that classifiers use the data to find positive edges with a higher probability than unsupervised algorithms. Those positive edges reduce the number of nodes in the contracted graph early on and thus make it possible to reduce the overall number of queries. In Figures 5.5c and 5.5d, one can notice that chordal algorithms have a slight edge over random ones both when combined with classifiers or not. While this was expected for the unsupervised case given the results of Section 5.2, it should be noted that keeping the contracted graph chordal also reduces the number of queries in the supervised case. This is partly explained by the fact that when the whole contracted graph consists of its largest connected component, classifiers no longer play any role so that random and chordal algorithms behave as if they were unsupervised.

The use of supervised techniques yields noticeable but not necessarily significant decreases in the number of queries. While they make it harder to implement a labeling algorithm, it should be noted that they do not incur significant costs in terms of added computation times. Indeed, we expect the number of items to cluster to be small (in the thousands at most) as it would otherwise be impracticable for humans. The classifiers we mention have minimal execution times at that scale and those execution times should thus remain negligible compared to the time spent answering queries by humans. It is also possible to use classifiers in a parallelized fashion, as long as there are items left out of the largest connected component (the reasoning described in Section 5.3.1 still applies).

### 5.3.3 Imperfect oracle

In Section 5.2, we made the assumption that the oracle did not make any mistake. In a more practical context where the oracle is human, we expect that some mistakes will be made. We discuss how these errors could be handled.

False negatives (when the oracle wrongly answered negatively) are corrected more easily than false positives. For instance, the former can be corrected a posteriori by asking additional questions between positive components which seem close (making it possible to correct a false negative in a few queries). On the other hand, false positives require more redundancy as finding the mislabeled edge in a positive component is cumbersome. To

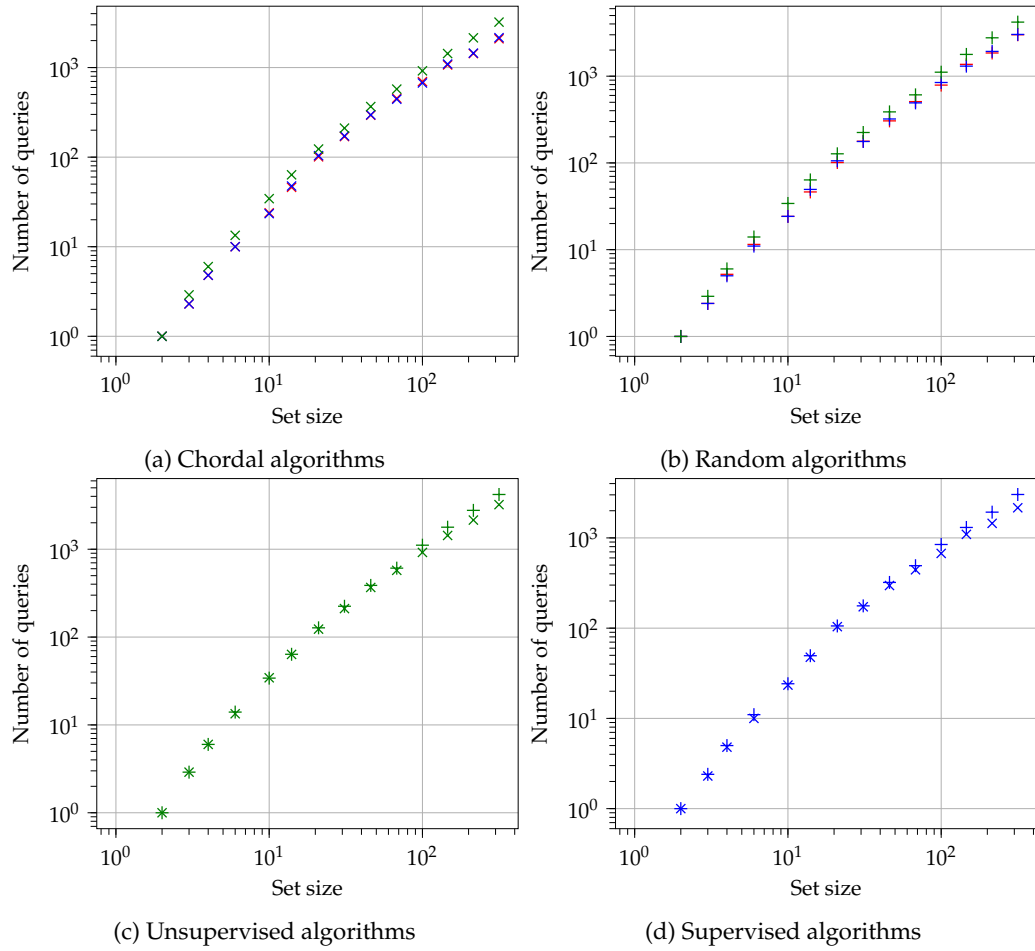


Figure 5.5: Average number of queries needed over 100 runs as a function of the number of elements.  $\times$ : Chordal algorithm,  $+$ : Random algorithm;  $\bullet$ : Unsupervised;  $\bullet$ : SGD;  $\bullet$ : MNB

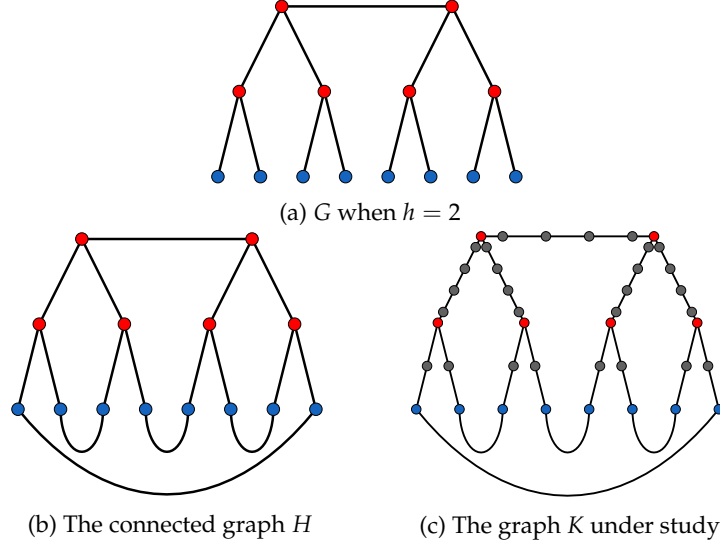


Figure 5.6: Different steps in the construction of a robust graph

address this imbalance, it is preferable to work on the expanded graph of queries (as opposed to the contracted graph under study in Section 5.2) where positive components are not merged.

Furthermore, there are many ways to model errors. For instance, one can assume that the overall number of errors will be bounded. Here we investigate, another error model which assumes mistakes are made with a (small) probability  $p$  at each query. To help detect inconsistencies in the oracle's answers, we create cycles in the expanded graph. To assess how robust to errors the graphs we create this way are, we rely on the notion of reliability polynomials (Page and Perry, 1994).

The coefficient  $c_i$  of  $p^i$  in the reliability polynomial  $R_G(p)$  of a graph  $G$  is the number of ways of disconnecting  $G$  by removing  $i$  edges. If, for all the edges of  $G$ , each is removed with probability  $p$ , then  $R_G(p)$  is the probability the graph is disconnected after the edges are deleted. We want to minimize  $R_G(p)$ . Using the added cycles in the graph, one can detect errors by checking that no cycle contains exactly one negative edge. Finding the smallest such contradictory cycle makes it possible to identify an ambiguous element by recursively splitting the cycle into two. To create such cycles, we build a graph whose reliability polynomial can be estimated.

Let us consider the tree graph  $G$  of depth  $h$  depicted in Figure 5.6a. We highlight the internal nodes in red and the leaves in blue. We seek to turn the previously described tree into a 2-connected graph. To achieve this, we allow ourselves to add  $2^{h-1}$  edges binding the  $2^h$  leaves of the tree. This yields the graph  $H$  depicted in Figure 5.6b.

Numbering the leaves in the most simple fashion (from left to right using the representation depicted in Figure 5.6a), we link each leaf with an even index to the next leaf (with the last leaf being connected to the first one). We now consider a graph  $K$  built from the



previously constructed graph by adding  $l \in 2\mathbb{N}$  nodes of degree 2 on each edge between the internal nodes and  $\frac{l-2}{2}$  nodes on the edges between the internal nodes and the leaves (as depicted in Figure 5.6c). We would like to show that  $K$  is optimally robust whenever  $H$  is. To prove that, we would like to express  $R_K(p)$ , the reliability polynomial of  $K$ , using  $R_H(p)$ .

We need to express the number of ways  $\tilde{c}_i$  of disconnecting  $K$  by removing  $i$  edges in terms of  $c_i$ . Since removing more than one edge between a pair of given nodes of degree 3 always disconnects the graph, we distinguish the cases where more than one edge between any pair of nodes of degree 3 has been removed from the other cases.

**Removing at most one edge per segment** There are  $l^i c_i$  ways to choose the edges to remove in order to yield no such pair of nodes (from any configuration that disconnects  $G$ , many configurations can be derived to disconnect  $K$ ).

**Removing at least two edges on at least one segment** On the other hand, the number of ways to remove at least two edges between at least one pair of nodes of degree 3 is  $\binom{ml}{i} - \binom{m}{i} l^i$  because  $\binom{ml}{i}$  is the number of ways of removing  $i$  edges in  $K$  (with or without disconnecting it) and  $\binom{m}{i} l^i$  is the number of ways to remove  $i$  edges without removing more than one edge between a pair of given nodes of degree 3 (still with no regard as to whether or not the yielded graph is connected).

In the end, both cases make up all the configurations which disconnect  $K$ :

$$\tilde{c}_i = c_i l^i + \binom{ml}{i} - \binom{m}{i} l^i$$

Thus:

$$\begin{aligned} R_K(p) &= \sum_{i=0}^{ml} \tilde{c}_i p^i \\ &= \sum_{i=0}^{ml} \left[ c_i l^i p^i + \binom{ml}{i} p^i - \binom{m}{i} l^i p^i \right] \\ R_K(p) &= R_H(lp) + (1+p)^{ml} - (1+lp)^m \end{aligned}$$

Notice how  $R_K(p) = R_H(lp) + o(p)$  for small  $p$ .

We are thus able to correlate the reliability of the graphs  $H$  and  $K$ . In particular, the parameter  $l$  allows one to choose the maximum size of the cycles in the graph thus enabling a trade-off between additional queries and error detection. Also, note that the graph  $K$  minus the edges between its leaves (i.e. the blue nodes in Figure 5.6c) can be obtained via a chordal algorithm. Aiming at graphs of the same form thus makes it possible to correct errors while maintaining control over the number of queries.

## 5.4 Conclusion

In this chapter, we introduced a particular case of dataset labeling and characterized its optimal solutions. We first introduced a formal setting based on a graph representation of the labeling. We then proved optimal solutions (on average) consisted of a particular family of algorithms. We proved that the complexity distribution of any two algorithms of this family is the same on any given graph and then gave an approximation of the mean and variance of those distributions when the number of elements to be labeled is large. We also discussed the potential improvements or changes that can be made to both the setting and the algorithms for use in real use-cases.

We previously assumed that the distribution of the underlying partition of the graph is the uniform one. As discussed at the end of Section 5.2.4, optimal solutions offer just a slim  $\frac{1}{\log n}$ -gain over the worst case. This is due in part to the fact that, under the uniform model, there are many blocks on average which makes for a high lower bound on the number of queries whatever the algorithm that is used. Indeed, in the experiments of Section 5.3.1, we find that chordal and random algorithms produce close results.

However, it is possible to have at least some information about that partition which biases the distribution. One information that may be found in some particular cases is the number of clusters. Though it is difficult to evaluate in general, when it is possible, the partition is picked from a limited set. Limiting the number of clusters alleviates the lower bound discussed earlier and enables greater improvements when comparing chordal algorithms to random ones. When running experiments with that fixed-size distribution, we find that all chordal algorithms are no longer equivalent and that the clique algorithm seems to reach a minimal number of queries (with the precision that the clique algorithm compares the selected vertex with the largest positive components first). Future work could focus on proving or disproving this conjecture.

## Chapter 6

# Conclusion and perspectives

In this work, we have tackled a number of problems in graph mining and machine learning.

First, we formally introduced graphs and depicted basic examples for the related yet distinct fields of graph mining and graph theory. We gave an outline of the thesis.

Then, we introduced Scikit-network, an open-source graph analysis library aimed at providing both an API that is consistent with the current Python ecosystem in machine learning packages and a state-of-the-art performance typically only found in C/C++ packages. We gave a sketch of practical considerations for representing graphs. We would like to emphasize that, in addition to the experiments of Chapter 2, results given in Chapters 3 and 4 illustrate the influence of the representation of graphs depending on the operations that have to be performed. In the future, the library could benefit from broader adoption of multi-threaded implementations of some algorithms along with the permanent maintenance efforts required to keep the package up to date. There are also a number of potential additions that could be made to the package such as anomaly detection algorithms.

In Chapter 3, we considered two clustering algorithms, one for flat clustering and one for hierarchical clustering. In the former case, we looked at the Louvain algorithm and its numerous variants. We first considered the various modularity functions present in the literature under a unified setting. We also described some improvements of the Louvain method and illustrated how the implementation of those algorithms (in particular the graph representation they use) has a considerable influence on their time performance. As regards hierarchical clustering, we considered the Paris algorithm and gave some examples of how it can be used with inherently hierarchical data. Future work could be focused on further improvements of the Louvain method, especially of the costly initial optimization phase.

We introduced a simple novel method for embedding the nodes of a graph, be it undirected, directed or bipartite. It is strongly linked to soft clustering as it is based on the formerly introduced Louvain clustering algorithm which enables easy interpretability and scalability. It has the particularity of not allowing the user to set an embedding dimen-

sion. Future improvements of this method could make use of the improvements of the Louvain method described in Chapter 3. In particular, those improvements can be aimed at speeding up the algorithm to tackle even larger graphs.

In Chapters 2, 3 and 4, we tackled situations where graphs could be used to model data. In this case, the accuracy of any method is usually measured against some ground truth of the data at hand and its time performance is expectedly dependent on the implementation of the method. Those empiric criteria are thus subjected to a careful understanding of those external factors (i.e. the data at hand and the implementation in use). To that end, making data and information readily available in the form of repositories enables increased reproducibility (Baker, 2016; Pineau et al., 2021). Even then, implementations may differ due to choices made when designing them such as the programming language that was used or the data structures. When proposing a new method, depicting those parameters may seem cumbersome. It is however often necessary as it affects what can be inferred from the experiments.

In Chapter 5 we studied a setting motivated by the construction of labeled datasets for supervised learning. While close to the problems depicted in the active learning literature, we make some strong assumptions that radically alter the approach to that setting. We describe and prove the optimality of a class of algorithms for this setting. In addition to these theoretical results, we provide some numerical experiments closer to real-world considerations. In the future, more of the strong assumptions made to solve the problem at hand could be challenged.

## Chapter 7

# Bibliography

- Abdi, H. (2007). Singular value decomposition (svd) and generalized singular value decomposition. *Encyclopedia of measurement and statistics*, pages 907–912.
- Ahuja, R., Mehlhorn, K., Orlin, J., and Tarjan, R. (1990). Faster algorithms for the shortest path problem. *J. ACM*, 37:213–223.
- Airoldi, E. M., Blei, D. M., Fienberg, S. E., and Xing, E. P. (2008). Mixed membership stochastic blockmodels. *Journal of Machine Learning Research*, 9(Sep):1981–2014.
- Andrews, G. E. (1986). *q-Series: Their Development and Application in Analysis, Number Theory, Combinatorics, Physics and Computer Algebra: Their Development and Application in Analysis, Number Theory, Combinatorics, Physics, and Computer Algebra*. Number 66. American Mathematical Soc.
- Bacchelli, A. and Bird, C. (2013). Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 712–721. IEEE.
- Baker, M. (2016). Reproducibility crisis. *Nature*, 533(26):353–66.
- Barber, M. J. (2007). Modularity and community detection in bipartite networks. *Physical Review E*, 76(6):066102.
- Basu, S., Banerjee, A., and Mooney, R. J. (2004a). Active semi-supervision for pairwise constrained clustering. In *Proceedings of the 2004 SIAM international conference on data mining*, pages 333–344. SIAM.
- Basu, S., Bilenko, M., and Mooney, R. J. (2004b). A probabilistic framework for semi-supervised clustering. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 59–68.
- Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D. S., and Smith, K. (2011). Cython: The best of both worlds. *Computing in Science & Engineering*, 13(2):31–39.

- Belkin, M. and Niyogi, P. (2003). Laplacian eigenmaps for dimensionality reduction and data representation. *Neural Computation*, 15(6):1373–1396.
- Belkin, M. and Niyogi, P. (2003). Laplacian eigenmaps for dimensionality reduction and data representation. *Neural Computation*, 15:1373–1396.
- Bhowmick, A. K., Meneni, K., Danisch, M., Guillaume, J.-L., and Mitra, B. (2020). LouvainNE: Hierarchical Louvain Method for High Quality and Scalable Network Embedding \*. In *the 13th ACM International WSDM Conference*, Houston, United States.
- Bhowmick, S. and Srinivasan, S. (2013). A template for parallelizing the louvain method for modularity maximization. In *Dynamics On and Of Complex Networks, Volume 2*, pages 111–124. Springer.
- Billingsley, P. (2008). *Probability and measure*. John Wiley & Sons.
- Bloch, F., Jackson, M. O., and Tebaldi, P. (2019). Centrality measures in networks. *Available at SSRN 2749124*.
- Blondel, V. D., Guillaume, J.-L., Lambiotte, R., and Lefebvre, E. (2008). Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008.
- Bonald, T., Charpentier, B., Galland, A., and Hollocou, A. (2018a). Hierarchical graph clustering based on node pair sampling. In *Proceedings of the 14th International Workshop on Mining and Learning with Graphs (MLG)*.
- Bonald, T., Hollocou, A., and Lelarge, M. (2018b). Weighted spectral embedding of graphs. In *2018 56th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 494–501. IEEE.
- Brandes, U. (2001). A Faster Algorithm for Betweenness Centrality. In *Journal of Mathematical Sociology*, volume 25, pages 163–177.
- Brodley, C. E. and Friedl, M. A. (1999). Identifying mislabeled training data. *Journal of artificial intelligence research*, 11:131–167.
- Buluç, A., Fineman, J. T., Frigo, M., Gilbert, J. R., and Leiserson, C. E. (2009). Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In auf der Heide, F. M. and Bender, M. A., editors, *SPAA*, pages 233–244. ACM.
- Campigotto, R., Céspedes, P. C., and Guillaume, J.-L. (2014). A generalized and adaptive method for community detection.
- Chen, F., Wang, Y.-C., Wang, B., and Kuo, C.-C. J. (2020). Graph representation learning: a survey. *APSIPA Transactions on Signal and Information Processing*, 9:e15.

- Chien, I. E., Zhou, H., and Li, P. (2019). Hs2: Active learning over hypergraphs with point-wise and pairwise queries. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pages 2466–2475. PMLR.
- Christopher, D. M., Prabhakar, R., Hinrich, S., et al. (2008). Introduction to information retrieval. *An Introduction To Information Retrieval*, 151(177):5.
- Corless, R. M., Gonnet, G. H., Hare, D. E., Jeffrey, D. J., and Knuth, D. E. (1996). On the lambertw function. *Advances in Computational mathematics*, 5(1):329–359.
- Csardi, G. and Nepusz, T. (2006). The igraph software package for complex network research. *InterJournal, Complex Systems*:1695.
- De Meo, P., Ferrara, E., Fiumara, G., and Proveti, A. (2011). Generalized louvain method for community detection in large networks. In *2011 11th international conference on intelligent systems design and applications*, pages 88–93. IEEE.
- Dong, Y., Chawla, N. V., and Swami, A. (2017). Metapath2vec: Scalable representation learning for heterogeneous networks. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '17*, page 135–144, New York, NY, USA. Association for Computing Machinery.
- Dugué, N. and Perez, A. (2015). *Directed Louvain: maximizing modularity in directed networks*. PhD thesis, Université d’Orléans.
- Eriksson, B., Dasarathy, G., Singh, A., and Nowak, R. (2011). Active clustering: Robust and efficient hierarchical clustering using adaptively selected similarities. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 260–268. JMLR Workshop and Conference Proceedings.
- Ernst, T. (2000). *The history of q-calculus and a new method*. Citeseer.
- Euler, L. (1956). The seven bridges of königsberg. *The world of mathematics*, 1:573–580.
- Farber, M. (1983). Characterizations of strongly chordal graphs. *Discrete Mathematics*, 43(2-3):173–189.
- Flajolet, P. and Sedgewick, R. (2009). *Analytic combinatorics*. cambridge University press.
- Ford Jr, L. R. (1956). Network flow theory. Technical report, Rand Corp Santa Monica Ca.
- Fortunato, S. (2010). Community detection in graphs. *Physics reports*, 486(3-5):75–174.
- Golumbic, M. C. (2004). *Algorithmic graph theory and perfect graphs*. Elsevier.
- Gonthier, G. et al. (2008). Formal proof—the four-color theorem. *Notices of the AMS*, 55(11):1382–1393.
- Gribel, D., Gendreau, M., and Vidal, T. (2021). Semi-supervised clustering with inaccurate pairwise annotations. *arXiv preprint arXiv:2104.02146*.

- Grover, A. and Leskovec, J. (2016). node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864.
- Gurevich, Y. and Shelah, S. (1987). Expected computation time for hamiltonian path problem. *SIAM Journal on Computing*, 16(3):486–502.
- Hagberg, A., Swart, P., and S Chult, D. (2008). Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States).
- Hamilton, W. L., Ying, R., and Leskovec, J. (2017). Representation learning on graphs: Methods and applications. cite arxiv:1709.05584Comment: Published in the IEEE Data Engineering Bulletin, September 2017; version with minor corrections.
- Hopcroft, J. and Tarjan, R. (1974). Efficient planarity testing. *Journal of the ACM (JACM)*, 21(4):549–568.
- Huang, J. and Ling, C. X. (2005). Using auc and accuracy in evaluating learning algorithms. *IEEE Transactions on knowledge and Data Engineering*, 17(3):299–310.
- Huang, Z., Zhou, A., and Zhang, G. (2012). Non-negative matrix factorization: A short survey on methods and applications. In *International Symposium on Intelligence Computation and Applications*, pages 331–340. Springer.
- Hubert, L. and Arabie, P. (1985). Comparing partitions. *Journal of classification*, 2(1):193–218.
- Jensen, T. R. and Toft, B. (2011). *Graph coloring problems*, volume 39. John Wiley & Sons.
- Keikha, M. M., Rahgozar, M., and Asadpour, M. (2018). Community aware random walk for network embedding. *Knowledge-Based Systems*, 148:47–54.
- Kim, T. and Ghosh, J. (2017). Semi-Supervised Active Clustering with Weak Oracles. *arXiv:1709.03202 [cs, stat]*. arXiv: 1709.03202.
- Kipf, T. N. and Welling, M. (2017). Semi-Supervised Classification with Graph Convolutional Networks. In *Proceedings of the 5th International Conference on Learning Representations*, ICLR ’17.
- Kleinberg, J. M. (1999). Authoritative sources in a hyperlinked environment. *Journal of the ACM (JACM)*, 46(5):604–632.
- Krishnamurthy, A., Balakrishnan, S., Xu, M., and Singh, A. (2012). Efficient active algorithms for hierarchical clustering. *arXiv preprint arXiv:1206.4672*.
- Kunegis, J. (2013). KONECT: The Koblenz Network Collection. In *Proceedings of the 22Nd International Conference on World Wide Web, WWW ’13 Companion*, pages 1343–1350, New York, NY, USA. ACM. event-place: Rio de Janeiro, Brazil.



- Landherr, A., Friedl, B., and Heidemann, J. (2010). A critical review of centrality measures in social networks. *Business & Information Systems Engineering*, 2(6):371–385.
- Li, B., Springer, J., Bebis, G., and Gunes, M. H. (2013). A survey of network flow applications. *Journal of Network and Computer Applications*, 36(2):567–581.
- Lovász, L. (1972). A characterization of perfect graphs. *Journal of Combinatorial Theory, Series B*, 13(2):95–98.
- Luo, B., Wilson, R. C., and Hancock, E. R. (2003). Spectral embedding of graphs. *Pattern recognition*, 36(10):2213–2230.
- Mazumdar, A. and Saha, B. (2017). Clustering with noisy queries.
- McGibbon, R. T. and Smith, N. J. (2016). A platform tag for portable linux built distributions. PEP 513, -.
- Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. (2013). Distributed representations of words and phrases and their compositionality. In *NIPS*, pages 3111–3119. Curran Associates, Inc.
- Mitra, R., Nandy, D., et al. (2012). A survey on clustering techniques for wireless sensor network. *International Journal of Research in Computer Science*, 2(4):51–57.
- Moak, D. S. (1984). The q-analogue of stirling’s formula. *The Rocky Mountain Journal of Mathematics*, pages 403–413.
- Murtagh, F. and Contreras, P. (2012). Algorithms for hierarchical clustering: an overview. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 2(1):86–97.
- Newman, M. E. (2006). Modularity and community structure in networks. *Proceedings of the national academy of sciences*, 103(23):8577–8582.
- Nghiem, M.-Q. and Ananiadou, S. (2018). Aplenty: annotation tool for creating high-quality datasets using active and proactive learning. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 108–113.
- Okamoto, K., Chen, W., and Li, X.-Y. (2008). Ranking of closeness centrality for large-scale social networks. In *International workshop on frontiers in algorithmics*, pages 186–195. Springer.
- Ou, M., Cui, P., Pei, J., Zhang, Z., and Zhu, W. (2016). Asymmetric Transitivity Preserving Graph Embedding. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD ’16*, pages 1105–1114, San Francisco, California, USA. ACM Press.
- Ozaki, N., Tezuka, H., and Inaba, M. (2016). A simple acceleration method for the louvain algorithm. *International Journal of Computer and Electrical Engineering*, 8(3):207.

- Page, L., Brin, S., Motwani, R., and Winograd, T. (1998). The pagerank citation ranking: Bringing order to the web. In *Proceedings of the 7th International World Wide Web Conference*, pages 161–172, Brisbane, Australia.
- Page, L. B. and Perry, J. E. (1994). Reliability polynomials and link importance in networks. *IEEE Transactions on Reliability*, 43(1):51–58.
- Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al. (2011). Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(Oct):2825–2830.
- Peixoto, T. P. (2014). The graph-tool python library. *figshare*.
- Pineau, J., Vincent-Lamarre, P., Sinha, K., Larivière, V., Beygelzimer, A., d’Alché Buc, F., Fox, E., and Larochelle, H. (2021). Improving reproducibility in machine learning research: a report from the neurips 2019 reproducibility program. *Journal of Machine Learning Research*, 22.
- Platt, J. C. (1999). Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. In *Advances in Large Margin Classifiers*, pages 61–74. MIT Press.
- Que, X., Checconi, F., Petrini, F., and Gunnels, J. A. (2015). Scalable community detection with the louvain algorithm. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 28–37. IEEE.
- Rand, W. M. (1971). Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical association*, 66(336):846–850.
- Rehman, S. U., Khan, A. U., and Fong, S. (2012). Graph mining: A survey of graph mining techniques. In *Seventh International Conference on Digital Information Management (ICDIM 2012)*, pages 88–92. IEEE.
- Rosenberg, A. and Hirschberg, J. (2007). V-measure: A conditional entropy-based external cluster evaluation measure. In *Proceedings of the 2007 joint conference on empirical methods in natural language processing and computational natural language learning (EMNLP-CoNLL)*, pages 410–420.
- Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., and Monfardini, G. (2008). The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80.
- Staudt, C. L., Sazonovs, A., and Meyerhenke, H. (2016). Networkit: A tool suite for large-scale complex network analysis. *Network Science*, 4(4):508–530.
- Tang, J., Qu, M., Wang, M., Zhang, M., Yan, J., and Mei, Q. (2015). LINE: Large-scale Information Network Embedding. *Proceedings of the 24th International Conference on World Wide Web - WWW ’15*, pages 1067–1077. arXiv: 1503.03578.

- Tarjan, R. E. and Yannakakis, M. (1984). Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on computing*, 13(3):566–579.
- Traag, V. A. (2015). Faster unfolding of communities: Speeding up the louvain algorithm. *Physical Review E*, 92(3):032801.
- Traag, V. A., Van Dooren, P., and Nesterov, Y. (2011). Narrow scope for resolution-limit-free community detection. *Physical Review E*, 84(1):016114.
- Traag, V. A., Waltman, L., and Van Eck, N. J. (2019). From louvain to leiden: guaranteeing well-connected communities. *Scientific reports*, 9(1):1–12.
- Vinh, N. X., Epps, J., and Bailey, J. (2010). Information theoretic measures for clusterings comparison: Variants, properties, normalization and correction for chance. *The Journal of Machine Learning Research*, 11:2837–2854.
- Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Jarrod Millman, K., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., Carey, C., Polat, İ., Feng, Y., Moore, E. W., VanderPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E. A., Harris, C. R., Archibald, A. M., Ribeiro, A. H., Pedregosa, F., van Mulbregt, P., and Contributors, S. . . (2020). SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272.
- Vishwanathan, S. V. N., Schraudolph, N. N., Kondor, R., and Borgwardt, K. M. (2010). Graph kernels. *Journal of Machine Learning Research*, 11:1201–1242.
- Wagstaff, K., Cardie, C., Rogers, S., Schroedl, S., et al. (2001). Constrained k-means clustering with background knowledge. In *Icml*, volume 1, pages 577–584.
- Walt, S. J. v. d., Colbert, S. C., and Varoquaux, G. (2011). The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30.
- Waltman, L. and Van Eck, N. J. (2013). A smart local moving algorithm for large-scale modularity-based community detection. *The European physical journal B*, 86(11):1–14.
- Xiong, C., Johnson, D. M., and Corso, J. J. (2017). Active Clustering with Model-Based Uncertainty Reduction. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(1):5–17.
- Yang, Z., Algesheimer, R., and Tessone, C. J. (2016). A comparative analysis of community detection algorithms on artificial networks. *Scientific reports*, 6(1):1–18.
- Yu, K., Yu, S., and Tresp, V. (2005). Soft clustering on graphs. *Advances in neural information processing systems*, 18:1553–1560.
- Zadrozny, B. and Elkan, C. (2002). Transforming classifier scores into accurate multiclass probability estimates. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 694–699.

- Zhan, F. B. and Noon, C. E. (1998). Shortest path algorithms: an evaluation using real road networks. *Transportation science*, 32(1):65–73.
- Zhang, J., Dong, Y., Wang, Y., Tang, J., and Ding, M. (2019). Prone: Fast and scalable network representation learning. In *IJCAI*, volume 19, pages 4278–4284.
- Zhang, L. and Tu, W. (2009). Six degrees of separation in online society.
- Zhang, Z., Cui, P., Li, H., Wang, X., and Zhu, W. (2018). Billion-scale network embedding with iterative random projection.

**Titre :** Contributions à base de graphes à l'apprentissage automatique

**Mots clés :** graphes, apprentissage automatique, annotation de données

**Résumé :** Un graphe est un objet mathématique permettant de représenter des relations entre des entités (appelées nœuds) sous forme d'arêtes. Les graphes sont depuis longtemps un objet d'étude pour différents problèmes allant d'Euler au PageRank en passant par les problèmes de plus courts chemins. Les graphes ont plus récemment trouvé des usages pour l'apprentissage automatique.

Avec l'avènement des réseaux sociaux et du web, de plus en plus de données sont représentées sous forme de graphes. Ces graphes sont toujours plus gros, pouvant contenir des milliards de nœuds et arêtes. La conception d'algorithmes efficaces s'avère nécessaire pour permettre l'analyse de ces données. Cette thèse étudie l'état de l'art et propose de nouveaux algorithmes pour la recherche de communautés et le plongement de nœuds dans des données massives. Par ailleurs, pour faciliter la manipulation de grands graphes et leur appliquer les techniques

étudiées, nous proposons Scikit-network, une librairie libre développée en Python dans le cadre de la thèse. De nombreuses tâches, telles que le calcul de centralités et la classification de nœuds, peuvent être accomplies à l'aide de Scikit-network.

Nous nous intéressons également au problème d'annotation de données. Les techniques supervisées d'apprentissage automatique nécessitent des données annotées pour leur entraînement. La qualité de ces données influence directement la qualité des prédictions de ces techniques une fois entraînées. Cependant, obtenir ces données ne peut pas se faire uniquement à l'aide de machines et requiert une intervention humaine. Nous étudions le problème d'annotation, sous un formalisme utilisant des graphes, avec pour but de décrire les solutions qui limitent cette intervention de façon optimale. Nous caractérisons ces solutions et illustrons comment elles peuvent être appliquées.

**Title :** Graph-based contributions to machine learning

**Keywords :** graphs, machine learning, data labeling

**Abstract :** A graph is a mathematical object that makes it possible to represent relationships (called edges) between entities (called nodes). Graphs have long been a focal point in a number of problems ranging from work by Euler to PageRank and shortest-path problems. In more recent times, graphs have been used for machine learning.

With the advent of social networks and the world-wide web, more and more datasets can be represented using graphs. Those graphs are ever bigger, sometimes with billions of edges and billions of nodes. Designing efficient algorithms for analyzing those datasets has thus proven necessary. This thesis reviews the state of the art and introduces new algorithms for the clustering and the embedding of the nodes of massive graphs. Furthermore, in order to facilitate the handling of large graphs and to apply the techniques

under study, we introduce Scikit-network, a free and open-source Python library which was developed during the thesis. Many tasks, such as the classification or the ranking of the nodes using centrality measures, can be carried out thanks to Scikit-network.

We also tackle the problem of labeling data. Supervised machine learning techniques require labeled data to be trained. The quality of this labeled data has a heavy influence on the quality of the predictions of those techniques once trained. However, building this data cannot be achieved through the sole use of machines and requires human intervention. We study the data labeling problem in a graph-based setting, and we aim at describing the solutions that require as little human intervention as possible. We characterize those solutions and illustrate how they can be applied in real use-cases.