HAL open science

Novel Approaches to the Clustering of Large Graphs

Alexandre Hollocou

To cite this version:

Alexandre Hollocou. Novel Approaches to the Clustering of Large Graphs. Computer Science [cs]. ENS, 2018. English. NNT : . tel-01987048v1

HAL Id: tel-01987048

https://hal.science/tel-01987048v1

Submitted on 25 Jan 2019 (v1), last revised 4 Mar 2020 (v2)

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.

THÈSE DE DOCTORAT

de l'Université de recherche Paris Sciences et Lettres
PSL Research University

Préparée à l'École Normale Supérieure

Nouvelles approches pour le partitionnement de grands graphes
Novel Approaches to the Clustering of Large Graphs

Ecole doctorale n°386

SCIENCES MATHÉMATIQUES DE PARIS CENTRE

Spécialité MATHÉMATIQUES

COMPOSITION DU JURY :

M. CHANG Cheng-Shang
National Tsing Hua University, Rapporteur

M. LAMBIOTTE Renaud
University of Oxford, Rapporteur

M. VAYATIS Nicolas
ENS Paris-Saclay, Président du jury

M. BONALD Thomas
Télécom Paristech, Membre du jury

M. D'ALCHÉ-BUC Florence
Télécom Paristech, Membre du jury

M. LELARGE Marc
INRIA Paris, Membre du jury

M. VIENNOT Laurent
INRIA Paris, Membre du jury

Soutenue par ALEXANDRE **HOLLOCOU**
le 19 décembre 2018

Dirigée par **Marc LELARGE**
et **Thomas BONALD**

ENS | PSL

Abstract

Graphs are ubiquitous in many fields of research ranging from sociology to biology. A graph is a very simple mathematical structure that consists of a set of elements, called *nodes*, connected to each other by *edges*. It is yet able to represent complex systems such as protein-protein interaction or scientific collaborations. Graph clustering is a central problem in the analysis of graphs whose objective is to identify dense groups of nodes that are sparsely connected to the rest of the graph. These groups of nodes, called *clusters*, are fundamental to an in-depth understanding of graph structures. There is no universal definition of what a *good* cluster is, and different approaches might be best suited for different applications. Whereas most of classic methods focus on finding node partitions, i.e. on coloring graph nodes so that each node has one and only one color, more elaborate approaches are often necessary to model the complex structure of real-life graphs and to address sophisticated applications. In particular, in many cases, we must consider that a given node can belong to more than one cluster. Besides, many real-world systems exhibit multi-scale structures and one much seek for hierarchies of clusters rather than flat clusterings. Furthermore, graphs often evolve over time and are too massive to be handled in one batch so that one must be able to process stream of edges. Finally, in many applications, processing entire graphs is irrelevant or expensive, and it can be more appropriate to recover local clusters in the neighborhood of nodes of interest rather than color all graph nodes.

In this work, we study alternative approaches and design novel algorithms to tackle these different problems. First, we study *soft clustering* algorithms that aim at finding a degree of membership for each node to each cluster, allowing a given node to be a member of multiple clusters. Then, we explore the subject of *edge clustering* whose objective is to find partitions of edges instead of node partitions. Third, we study *hierarchical clustering* techniques to find multi-scale clusters in graphs. We then examine *streaming methods* for graph clustering, able to handle graphs with billions of edges. Finally, we focus on *local* graph clustering approaches, whose objective is to find clusters in the neighborhood of so-called *seed nodes*. The novel methods that we propose to address these different problems are mostly inspired by variants of *modularity*, a classic measure that accesses the quality of a node partition, and by *random walks*, stochastic processes whose properties are closely related to the graph structure. We provide analyses that give theoretical guarantees for the different proposed techniques, and endeavour to evaluate these algorithms on real-world datasets and use cases.

List of publications

Multiple local community detection.
Hollocou, A., Bonald, T., and Lelarge, M. (2017).
ACM SIGMETRICS Performance Evaluation Review, 45(2):76–83.
Reference [Hollocou et al., 2017a].

A streaming algorithm for graph clustering.
Hollocou, A., Maudet, J., Bonald, T., and Lelarge, M. (2017).
NIPS Workshop.
Reference [Hollocou et al., 2017b].

Hierarchical graph clustering by node pair sampling.
Bonald, T., Charpentier, B., Galland, A., and Hollocou, A. (2018).
KDD Workshop.
Reference [Bonald et al., 2018a].

Weighted spectral embedding of graphs.
Bonald, T., Hollocou, A., and Lelarge, M. (2018).
In *Communication, Control, and Computing (Allerton), 2018 56th Annual Allerton Conference on*.
Reference [Bonald et al., 2018c].

Modularity-based sparse soft graph clustering.
Hollocou, A., Bonald, T., and Lelarge, M. (2019).
AISTATS.
Reference [Hollocou et al., 2019].

A fast agglomerative algorithm for edge clustering.
Hollocou, A., Lutz, Q., and Bonald, T. (2018).
Preprint.
Reference [Hollocou et al., 2018].

Forward and backward embedding of directed graphs.
Bonald, T., De Lara, N., and Hollocou, A. (2018).
Preprint.
Reference [Bonald et al., 2018b].

Acknowledgements

First, I would like to express my sincere gratitude to my advisors Marc Lelarge and Thomas Bonald for their support during the last three years. I would like to thank you for encouraging my research and for your invaluable contribution to this work. It has been a truly great pleasure to work with you. I hope that we will have the opportunity to continue discussions in the future.

I am also extremely grateful to my colleagues and friends at the Ministère des Armées. It was an amazing experience working with you, and I will never forget these moments. Special thanks to Fabrice, without whom my Ph.D. would not have been possible, and Pierre, for his extensive knowledge and his always helpful insights.

I would also like to thank my committee members, professor Cheng-Shang Chang, professor Renaud Lambiotte, professor Florence D'Alche-Buc, professor Nicolas Vayatis and professor Laurent Viennot for serving as my committee members even at hardship and busy schedules.

I am also grateful to all my friends for their encouragements. Special thanks to the Joko team, Xavier, Nicolas and Arthur, for your patience and your support.

Lastly, I would like to thank my family and Nathalie for their unwavering love and for supporting me for everything. I cannot thank you enough for encouraging me throughout this experience.

Contents

1 Introduction **13**
1.1 Motivations . . . 13
1.1.1 Graphs . . . 13
1.1.2 Graph clustering . . . 16
1.2 Contributions . . . 18
1.3 Notations . . . 19
1.4 Classic graph clustering methods . . . 20
1.4.1 Modularity . . . 20
1.4.2 Spectral methods . . . 27
1.4.3 Some other graph clustering approaches . . . 31
1.5 Evaluation of graph clustering algorithms . . . 32
1.5.1 Synthetic graph models . . . 32
1.5.2 Real-life graphs and ground-truth clusters . . . 34

2 Soft clustering **37**
2.1 Introduction . . . 37
2.2 Fuzzy c-means: an illustration of soft clustering . . . 38
2.2.1 The k-means problem . . . 38
2.2.2 Fuzzy c-means . . . 39
2.3 Soft modularity . . . 41
2.4 Related work . . . 42
2.4.1 The softmax approach . . . 42
2.4.2 Other approaches . . . 44
2.5 Alternating projected gradient descent . . . 45
2.6 Soft clustering algorithm . . . 48
2.6.1 Local gradient descent step . . . 48
2.6.2 Cluster membership representation . . . 48
2.6.3 Projection onto the probability simplex . . . 49
2.6.4 MODSOFT . . . 49
2.7 Link with the Louvain algorithm . . . 49
2.8 Algorithm pseudo-code . . . 51
2.9 Experimental results . . . 53
2.9.1 Synthetic data . . . 53
2.9.2 Real data . . . 53

3 Edge clustering **57**
3.1 Introduction . . . 57
3.2 Related work . . . 58
3.3 A modularity function for edge partitions . . . 58
3.4 The approach of Evans and Lambiotte . . . 60
3.4.1 Incidence graph and bipartite graph projections . . . 60
3.4.2 Line graph . . . 61
3.4.3 Weighted line graph . . . 61
3.4.4 Random-walk-based projection . . . 61

3.4.5 Limitation . . . 62
3.5 An agglomerative algorithm for maximizing edge modularity . . . 62
3.5.1 Agglomerative step . . . 63
3.5.2 Greedy maximization step . . . 64
3.5.3 Algorithm . . . 64
3.6 Experimental results . . . 64
3.6.1 Synthetic data . . . 64
3.6.2 Real-life datasets . . . 65

4 Hierarchical clustering **69**
4.1 Introduction . . . 69
4.2 Related work . . . 69
4.3 Agglomerative clustering and nearest-neighbor chain . . . 70
4.3.1 Agglomerative algorithms . . . 70
4.3.2 Classic distances . . . 71
4.3.3 Ward's method . . . 72
4.3.4 The Lance-Williams family . . . 72
4.3.5 Nearest-neighbor chain algorithm . . . 72
4.4 Node pair sampling . . . 73
4.5 Clustering algorithm . . . 75
4.6 Link with modularity . . . 76
4.7 Experiments . . . 77

5 Streaming approaches **85**
5.1 Introduction . . . 85
5.2 Related work . . . 86
5.2.1 Edge streaming models . . . 86
5.2.2 Connected components and pairwise distances . . . 86
5.2.3 Minimum cut . . . 87
5.3 A streaming algorithm for graph clustering . . . 88
5.3.1 Intuition . . . 88
5.3.2 Algorithm . . . 89
5.3.3 Complexity . . . 89
5.3.4 Parameter setting . . . 89
5.4 Theoretical analysis . . . 90
5.4.1 Modularity optimization in the streaming setting . . . 90
5.4.2 Streaming decision . . . 90
5.5 Experimental results . . . 93
5.5.1 Datasets . . . 93
5.5.2 Benchmark algorithms . . . 93
5.5.3 Performance metrics and benchmark setup . . . 93
5.5.4 Benchmark results . . . 93

6 Local approaches **95**
6.1 Introduction . . . 95
6.2 Related work . . . 95
6.2.1 Scoring and sweeping . . . 96
6.2.2 Personalized PageRank . . . 96
6.2.3 Heat kernel diffusion . . . 99
6.2.4 Local spectral analysis . . . 100
6.2.5 Other approaches . . . 100
6.2.6 Finding seed sets . . . 100
6.3 Multiple Community Detection . . . 100
6.3.1 Scoring gap . . . 101
6.3.2 Local embedding . . . 101
6.3.3 Finding new seeds . . . 101

6.4 Algorithm . . . 101
6.4.1 Inputs . . . 104
6.4.2 Description . . . 104
6.4.3 Post-processing . . . 105
6.5 Experiments . . . 105
6.5.1 Case study: Wikipedia . . . 105
6.5.2 Real-world data . . . 106
6.5.3 Synthetic data . . . 108

7 Conclusion **109**

Chapter 1

Introduction

1.1 Motivations

1.1.1 Graphs

A graph is a mathematical structure that consists of a set of entities, called *nodes*, and a set of connections between pairs of these entities, called *edges*. More formally, a graph G is a pair (V, E), where V is the set of nodes and $E \subset V \times V$ is the set of edges. Graph nodes can also be called *vertices*, and the term *network* is often used to refer to a graph. In Figure 1.1, we give a simple example of a graph with 5 nodes and 6 edges.

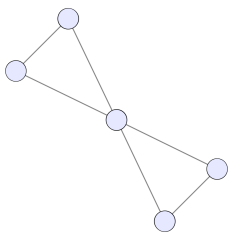


Figure 1.1: A simple graph with 5 nodes

This structure is very simple and yet extremely powerful to model an incredible variety of data. For instance, it is natural to model the interactions in a *social network* as a graph where the nodes correspond to the users of the social network, and the edges correspond to the friendship links between these users [Leskovec and Mcauley, 2012]. Another common example is the *web graph* [Barabási et al., 2016], whose nodes are defined as the web pages on the Internet and whose edges correspond to the hyperlinks between these pages. Graphs are also very useful for many problems in biology, to model the *protein-protein* interactions for example [Breitkreutz et al., 2007], or in medical sciences, where the researchers are interested among other things in *disease-symptom graphs* [Hassan et al., 2015], i.e. graphs where nodes can represent either diseases or symptoms, and where edges are used to connect each disease to its symptoms.

In the later example, we see that the graphs are a bit particular: nodes are of two types (namely diseases or symptoms) and edges can only connect nodes of different types. Such graphs are called *bipartite graphs* and are widespread to model a wide range of information. For instance, they also can be used to build the *actor-movie graph* from a database as IMDb[1] (the Internet Movie Database). In this graph, nodes correspond to actors or movies, and we put an edge between an actor and a movie in the graph if the actor appears in the movie's cast. Bipartite graphs can be used to model musical tastes using datasets from streaming music services like Spotify, Apple Music or Deezer too. For example, we can consider the graph where nodes can either represent the users or the artists on the platform, and where there is an edge between a user and an artist if the user has listened to a song interpreted by the artist.

[1] `https://datasets.imdbws.com/`

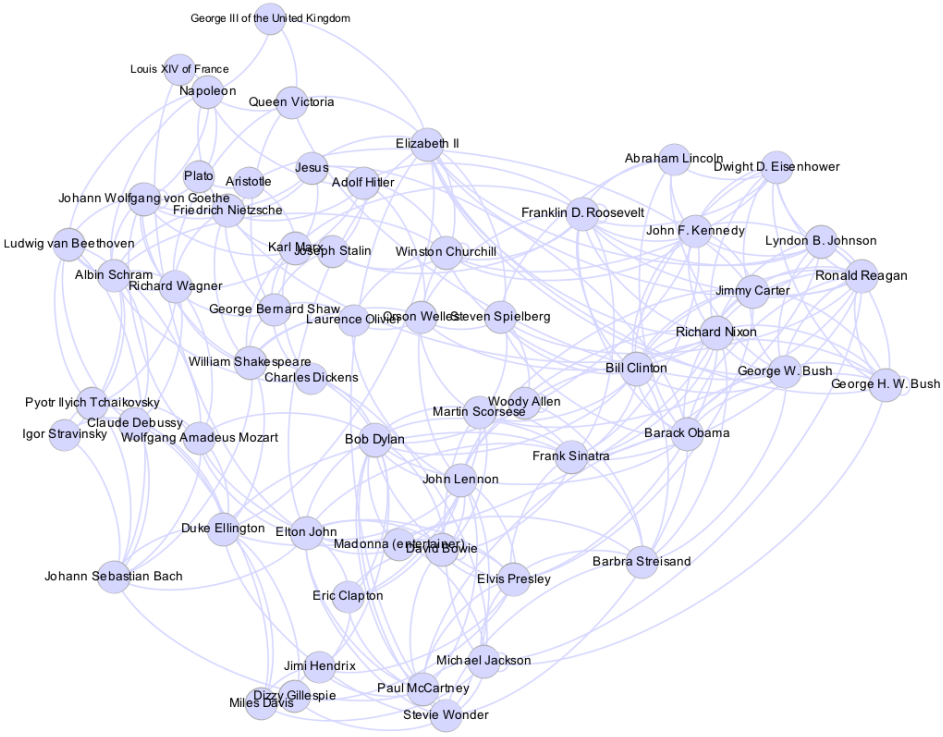


Figure 1.2: Wikipedia – Hyperlinks between the top 58 human articles

Note that multiple graphs can be built from the same dataset. For instance, if we consider Twitter, we can build various types of graph from the data collected by this social network. We can for example consider the graph whose nodes are Twitter users and where there is an edge between two users u and v if u follows v; or the graph where there is an edge between u and v if u has retweeted a Tweet published by v. We can also study a graph whose nodes are *hashtags*, like *#christmas* or *#datascience*, and where an edge is put between two hashtags if they appear in the same Tweet. Such a graph, where entities are linked by an edge if they are *cited* together, is called a *co-citation* graph. We can use this type of graph to describe scientific collaborations for instance, by looking at the graph where two researchers are linked by an edge if they are both cited in the same paper. A co-citation graph can also be built from Wikipedia data: one can consider the graph where nodes are Wikipedia articles and where an edge is put between two articles u and v if they are cited by the same article w (e.g. *Lady Diana* and *Winston Churchill* are linked because they are both cited in the *Elizabeth II* article). Note that we can also consider the graph where there is an edge between two articles if they cite the same article (e.g. *Gustave Flaubert* and *Francis Scott Fitzgerald* would be connected because they both cite the *Paris* article), or the even more natural graph where we put an edge between two articles u and v, if u directly cites v (e.g. *Albert Einstein* is connected to *Sigmund Freud* because the *Einstein* article cites the *Freud* article). In Figure 1.2, we show a subgraph of the later graph, where we only keep articles corresponding to human beings. In this figure, we display the Top 58 nodes in terms of degrees (the *degree* of a node corresponds to its number of connections to other nodes).

Finally, as it is impossible to give an exhaustive list of the data types that can be represented as graphs, we give two specific examples that can be easily visualized and that we will use throughout this work. First, we use the *OpenFlights database*[2] to build a graph whose nodes correspond to airports, and where we put an edge between two airports if there is an airline operating services between them. We draw this graph on Figure 1.3. Note that, in this figure, airports are positioned using their geographical coordinates. We will also use the *OpenStreet Map* database[3] to define a similar graph where edges correspond to roads and nodes to road intersections. We display a section of this graph that corresponds to the center of

[2]https://openflights.org/data.html
[3]https://wiki.openstreetmap.org/wiki/

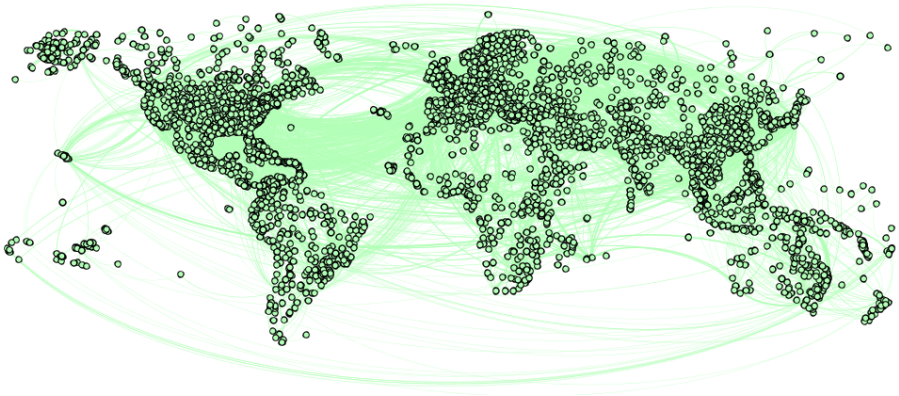


Figure 1.3: OpenFlights graph

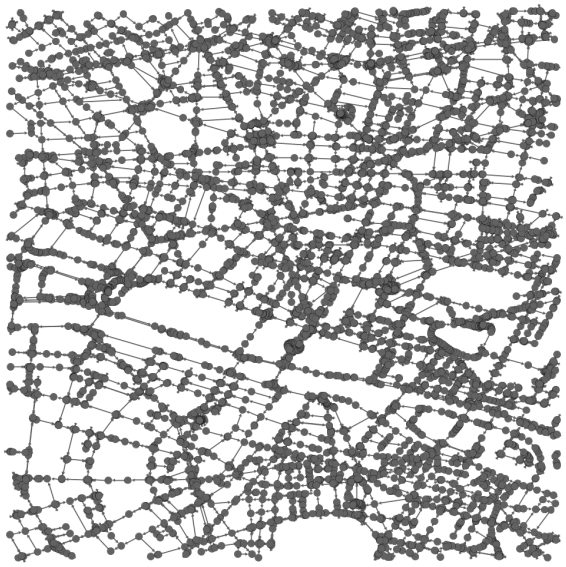


Figure 1.4: Open Street Map graph (Paris)

Paris in Figure 1.4.

All the graphs cited above can be of different types. They can be *undirected*, i.e. their edges have no direction and we have $(u, v) \in E \Leftrightarrow (v, u) \in E$, or *directed*, i.e. their edges have orientations and one can have $(u, v) \in E$ while $(v, u) \notin E$. For instance, to model friendships in a social network as Facebook we will typically use an undirected graph, as the friendships on Facebook are two-way relationships. However, to model the following interactions on Twitter, it is more relevant to use a directed graph. Indeed, when a user u follows a user v, v does not necessarily follows u. It is also more relevant to use a directed graph for the web graph, as the hyperlinks are themselves directed. On a more exotic note, note that directed graphs can be used to represent the interactions between players or teams in a sport championship (e.g. the interaction between the tennis players in the ATP World Tour), by taking different players or teams as nodes and by adding a directed edge (u, v) between these nodes if u as won against v.

Another important characteristic of graphs revolves around edge weights. Indeed, graphs can be *weighted*, i.e. a non-negative real number w_{uv}, called the *weight*, is assigned to each edge (u, v). They can also be *unweighted*, and, in this case, we will often consider that each edge weight is equal to 1. For example, it is natural to weight the edges of the Spotify or Deezer user-artist graph described above by defining the weight w_{uv} as the number of times the user u has listened to the artist v. We will also typically use a weighted graph to represent the shopping behavior of users on an e-commerce website or a

marketplace as Amazon, by taking the users and the products as nodes of the graph, and by putting an edge of weight w_{uv} between u and v if the user u has bought a quantity w_{uv} of the product v. The list of examples we could give is long, but we will now focus on our specific subject of interest, graph clustering.

1.1.2 Graph clustering

As Kleinberg puts it, "*whenever one has a heterogeneous set of objects, it is natural to seek methods for grouping them together*" [Kleinberg, 2003]. Therefore, one of the most natural task that arises when analyzing a graph $G = (V, E)$ consists in subdividing the set of nodes V into cohesive subsets of nodes, the so-called *clusters*. Taking the social network example described above, one would like to be able to identify the different social circles for instance (groups of friends, families, groups of colleagues, associations...). In the movie-actor graph, one would be interested in pinpointing groups of nodes corresponding to different film genres (action, adventure, comedy, drama etc.) or different origin countries (Hollywood movies, Bollywood movies, French movies, Italian movies etc.). And, in the OpenFlights graph considered above, one would like to identify different regions of the world with dense local flight connections for instance. The problem of finding relevant clusters in a graph is known as *graph clustering*. It is also commonly referred to as *community detection*, and in this case, the word "*community*" is used to designate a cluster. This denomination is best suited for graphs whose nodes represent people whom we wish to study the social interactions. We will use the more general term "cluster" in the rest of this work.

One of the greatest challenge with graph clustering lies in the fact that there is no universal definition of what a good clustering is. Graph clustering is actually frequently referred to as an *ill-posed problem* [Kleinberg, 2003, Peel et al., 2017]. Researchers often agree that a good node cluster is a group of nodes with dense internal connections and sparser connections with the nodes outside the cluster [Newman and Girvan, 2004], but such a definition is vague and can lead to multiple interpretations. As a result, the number of distinct approaches to tackle graph clustering is considerable [Fortunato and Castellano, 2012].

However, some constants can be identified: a majority of graph clustering methods are interested in finding *node partitions*, i.e. in grouping the nodes in non-empty subsets (the clusters), in such a way that every node is included in *one and only one* subset. In other word, a node partition corresponds to a *coloring* of the nodes where every node has one and only one color. In Figure 1.5, we show such a partition of the nodes of the Wikipedia subgraph presented above. The different colors code for the different clusters of this node partition. These clusters correspond to the result of a classic graph clustering algorithm called Louvain [Blondel et al., 2008]. We see that the clusters correspond to different historical periods and different spheres of interest. The blue cluster seems to correspond to twentieth century artists, like Bod Dylan, Miles Davis and Martin Scorsese, whereas the salmon-colored cluster mostly corresponds to twentieth century political leaders, as Winston Churchill and John F. Kennedy. Beyond the few inaccuracies of this node coloring, we see that such a partition of the nodes is too restrictive to model the complexity of the real-world interactions between these historical figures. In particular, we might want a node to belong to multiple clusters (i.e. to be colored with more than one color), or to obtain multi-scale clusters. In the following, we list some challenges associated with clustering real-world graph data.

Overlapping clusters

In a partition of the graph nodes, a given node belongs to a unique cluster. In many examples presented above, we see that such a constraint is too limited to model the interactions between nodes. In a social network for instance, we know that a given individual often belongs to multiple social circles corresponding to her family, here friends, her colleagues etc. Hence, a simple node coloring is not able to model the complexity of this reality. We can find another example in the Wikipedia graph. In Figure 1.6, we show a zoom on the Wikipedia subgraph consisting of human articles. The nodes of the graph are partitioned (and colored) by the Louvain algorithm. We see that George Gershwin is in the same cluster as other classical music composers as Igor Stravinsky, whereas one could argue that he might also be put in the same cluster as jazz musicians, as Duke Ellington.

To address this issue, a common solution consists in finding subsets of nodes that are not necessarily disjoint. In other words, this solution considers that clusters are subsets of nodes that might *overlap* [Xie et al., 2013]. Another solution consists in considering partitions of the graph edges instead of the graph

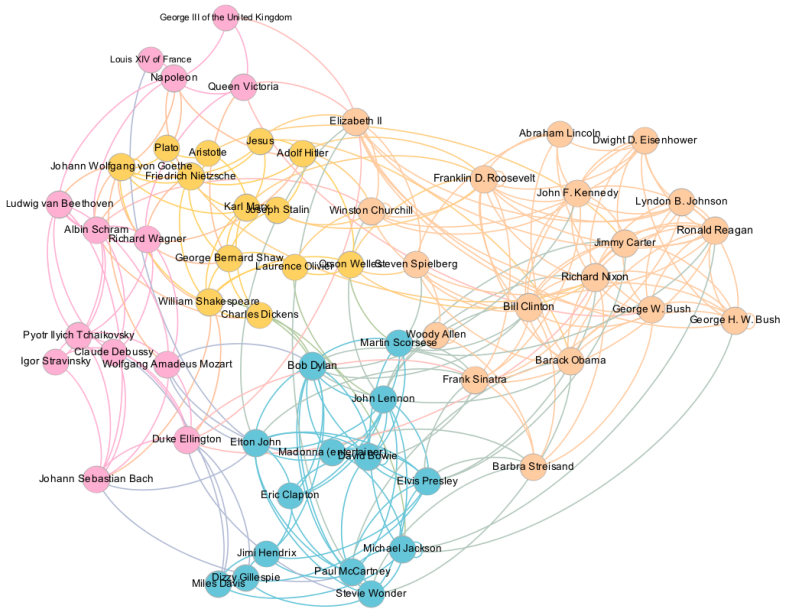


Figure 1.5: Wikipedia Top 58 Human Articles: node partition returned by the Louvain algorithm

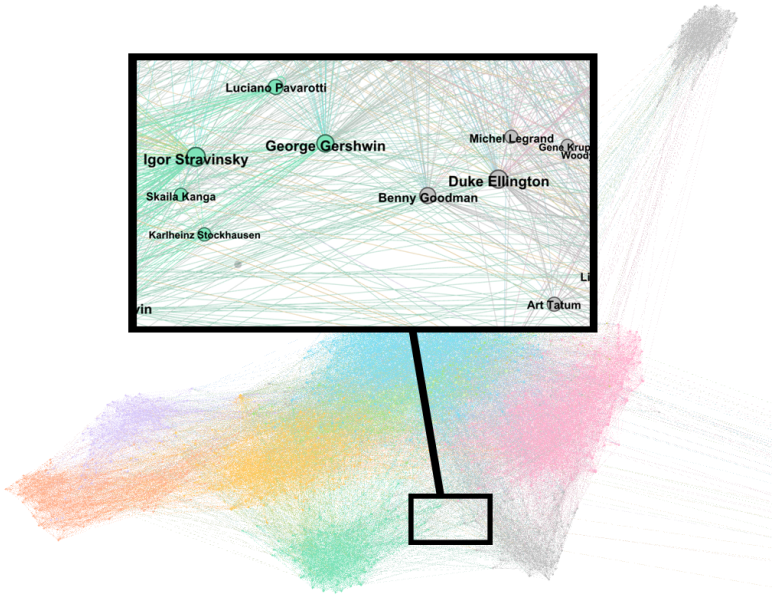


Figure 1.6: Wikipedia Human Articles: node partition returned by the Louvain algorithm

nodes [Evans and Lambiotte, 2009]. One could also be interested in calculating a *degree of membership* of each node to each cluster instead of simply assigning nodes to clusters [Chang and Chang, 2017]. This problem constitutes the subject of Chapter 2 and Chapter 3.

Cluster hierarchy

In most of applications, the notion of "*good*" cluster is intrinsically related to the scale we are interested in. For instance, in the actor-movie graph, we could be interested in grouping the nodes by movie genres (comedy, action etc.), or by sub-genres (biopics, disaster movies etc.). We see that a partition of this graph's nodes can only represent one of this breakdown. The same reasoning applies to our user-artist music graph, where one might be interested in main genres, as electronic music, rock or classical music, or be interested in sub-genres, as brit-pop, grunge or ambient music, which cannot be solved by a simple node coloring.

A common solution to this issue consists in studying cluster hierarchies [Girvan and Newman, 2002]. These hierarchies can typically be seen as trees whose nodes are clusters and where the children of a given cluster in this tree form a partition of this cluster. In a *good* hierarchical clustering, the different levels of the tree corresponds to the different *scales* in the cluster structure of the graph. This problem is the subject of Chapter 4.

Local approaches

Classic clustering approaches focus on clustering the *whole* graph given as an input, i.e. they consider that every node of the graph must belong to a cluster. Moreover, they rely on the knowledge of the entire graph structure. Yet, in many practical applications, we are only interested in small portions of the graph. Typically, we are often interested in finding the cluster (or the clusters) to which a given node belongs, without regard to the other clusters. For instance, if we want to recommend new friends to a given user on a social network with a graph clustering approach, we will typically focus on the clusters in the neighborhood of this user. Besides, in many real-life situations, exploring the whole graph can be expensive or even impossible. Consider the Twitter Retweet graph defined above for instance. One can explore this graph locally using the Twitter API with queries such as "*who did retweet the Tweet X*". However there is a rate limit that bounds the number of queries to this API to 15 calls every 15 minutes[4], which prevent us to build the whole graph.

For these different reasons, one might be interested in local approaches that focus on finding local clusters around given nodes of interests, commonly called the *seed nodes*, and that require only a local knowledge of the graph structure around these nodes [Kloumann and Kleinberg, 2014]. This problem is the subject of Chapter 6.

Streaming approaches

Classic graph clustering methods process a graph in *one batch*. In other words, they need to handle all the nodes and all the edges of the graph at the same time to identify the clusters. In real-life applications, the amount of data is often too massive to be handled in a single batch. As an example, it takes about 30 GB of RAM to load the Friendster[5] graph [Yang and Leskovec, 2015], which represents more memory than what classic computers can handle. Besides, in many situations, edges naturally arrive in a streaming fashion. For instance, on our Twitter graphs, each new Tweet can make a new edge appears in the graphs.

For both of these reasons, streaming algorithms [McGregor, 2014] are very interesting to cluster real-world graphs. A streaming algorithm will typically process the graph edges *one by one* and update the node clustering at each new edge arrival without storing the whole graph structure in memory. This will be the subject of Chapter 5.

1.2 Contributions

In all the examples cited above, we see that it is difficult to identify a single *right* approach to graph clustering. There exist diverse strategies addressing different angles of this complex problem. In the

[4] https://developer.twitter.com/en/docs/basics/rate-limiting.html

[5] Friendster was a social network similar to Facebook. In June 2011, the website repositioned itself as a social gaming site.

present work, we propose to study some of the *various facets of the graph clustering problem.*

- First, we study graph clustering approaches that go beyond simple *crisp* clusterings of graph nodes. More precisely, we examine two kinds of methods to tackle this problem:
  - *soft-clustering* approaches that compute for each node a degree of membership to each cluster instead of a node partition,
  - and *edge clustering* approaches whose objective is to partition the graph edges instead of the graph nodes.
- Second, we focus on methods for recovering *multi-scale clusters* and *hierarchies* instead of partitions corresponding to a single scale.
- Third, we analyze *streaming* techniques for graph clustering that handle graph edges one by one and update the clusters at each new edge arrival for both memory efficiency and the ability to handle constantly updating graphs.
- Finally, we study *local graph clustering* approaches that only explore a local neighborhood of a set of nodes of interest in order to find clusters to which these nodes belong.

This manuscript is organized as follows. In the rest of the present chapter, we introduce notations that are used throughout this work and key notions on which most of our work is based. In Chapter 2, we study soft graph clustering methods. In Chapter 3, we focus on the edge clustering problem. In Chapter 4, we study hierarchical graph clustering techniques. We study streaming algorithms for graph clustering in Chapter 5. Finally, in Chapter 6, we explore the problem of local graph clustering, and Chapter 7 concludes this work.

1.3 Notations

Unless mentioned otherwise, we consider that we are given a graph $G = (V, E)$, where V is the set of nodes and E is the set of edges. We use $n = |V|$ to denote the number of nodes, and $m = |E|$ to denote the number of edges. We use $A = (A_{ij})_{i,j \in V}$ to denote the adjacency matrix of the graph: A_{ij} corresponds to the weight of the edge (i, j) if $(i, j) \in E$, and we have $A_{ij} = 0$ if $(i, j) \notin E$. In the case of an unweigted graph, we take $A_{ij} = 1$ if $(i, j) \in E$. $\mathrm{Nei}(i)$ is used to denote the set of neighbors of node i,

$$\mathrm{Nei}(i) = \{j \in V : (i, j) \in E\}.$$

We sometimes use the notation $i \sim j$ for $j \in \mathrm{Nei}(i)$. The weighted degree w_i of a node $i \in V$, is defined as

$$w_i = \sum_{j \in V} A_{ij}.$$

If the graph is unweighted, we also use the notation d_i for the degree, $d_i = w_i \in \mathbb{N}$. We use w to denote the sum of the node weights,

$$w = \sum_{i \in V} w_i.$$

Note that we also have

$$w = \sum_{i,j \in V} A_{ij}.$$

The weighted volume of a set of node $C \subset V$ is defined as the sum of the weighted degrees of the nodes of C: $\mathrm{Vol}(C) = \sum_{i \in C} w_i$.

In the following, we consider the general case of weighted and directed graphs, unless mentioned otherwise.

1.4 Classic graph clustering methods

In this section, we present the major graph clustering concepts on which our work is based. It does on no account constitute a comprehensive list of graph clustering techniques. We especially focus on *modularity*, *random walks* and *spectral methods*. Modularity [Newman and Girvan, 2004] measures how well a given clustering explains the observed graph structure compared to a situation where edges would be distributed at random. Random walks [Lovász et al., 1993] describe the process of a walker who jumps from a node to one of its neighbor picked at random and are tightly linked with important graph properties such as cluster structures. Spectral methods are based on the spectral decomposition of the so-called Laplacian and are related to random-walks and other key notions in graphs. These notions constitute key pillars throughout our work. For a more complete field review on the subject of graph clustering we refer the reader to the work of Fortunato [Fortunato and Castellano, 2012]. Please also note that each chapter of this document includes a specific bibliography on the subject that is tackled.

1.4.1 Modularity

Modularity is arguably the most widely used method for clustering graphs . It was introduced by Newman and Girvan in 2004 in [Newman and Girvan, 2004]. The modularity function is a quality function defined on node partitions. Given a partition P of the nodes of an undirected graph $G = (V, E)$, $P = \{C_1, \ldots, C_K\}$ with $C_1 \sqcup \ldots \sqcup C_K$, the modularity of P, $Q(P)$ is defined as

$$Q(P) = \frac{1}{w} \sum_{C \in P} \sum_{i,j \in C} \left(A_{ij} - \frac{w_i w_j}{w} \right). \tag{1.1}$$

The modularity is the difference between two terms. The first term $\frac{1}{w} \sum_C \sum_{i,j \in C} A_{ij}$ corresponds to the ratio between the weight of the edges within clusters (the *intra-cluster edges*) and the total edge weight. This first term can be trivially maximized by putting all the nodes in one cluster, and is therefore not a good quality measure alone. The second term $\frac{1}{w} \sum_C \sum_{i,j \in C} w_i w_j / w$ counterbalances the first term. It corresponds to the ratio between the expected weight of intra cluster edges and the total edge weight in a random graph model where the expected weight of an edge (i, j) is $\frac{w_i w_j}{w}$. Note that, in such a model, the expected weighted degree of a node i corresponds to the weighted degree of i in G, w_i. This kind of random graph model is well-known in graph theory and corresponds to a weighted variant of the *configuration model* [Frieze and Karoński, 2016].

In other words, the modularity does not only measures the ratio of in-cluster edges but the difference between that ratio and the expected ratio in a graph where edges would be placed at random. Modularity can be interpreted in different manners. In the following, we present some of these interpretations.

Node pair sampling

Consider the following sampling process where an edge (i, j) is chosen with a probability proportional to its weight. In this process, each node pair (i, j) is chosen with probability

$$p(i, j) = \frac{A_{ij}}{w}.$$

The distribution p is a symmetric joint distribution with marginal distribution

$$p(i) = \sum_{j \in V} p(i, j) = \frac{w_i}{w},$$

which corresponds to the probability distribution on nodes where a node is picked with a probability proportional to its weighted degree. Then, the modularity function can be rewritten in function of this joint distribution and its marginal

$$Q(P) = \sum_{C} \sum_{i,j \in C} \left[p(i, j) - p(i)p(j) \right].$$

In other words, the modularity of P corresponds to the difference between the probability of sampling an edge within a cluster and the probability of sampling a node pair independently, under the marginal distribution, within a cluster.

The sampling process considered here naturally induces a joint distribution on the clusters of a partition $P = \{C_1, \ldots, C_K\}$:

$$p(C, C') = \sum_{i \in C} \sum_{j \in C'} p(i, j),$$

with marginal distribution

$$p(C) = \sum_{C' \in P} p(C, C') = \sum_{i \in C} p(i).$$

Then we can write the modularity as

$$Q(P) = \sum_{C \in P} (p(C, C) - p(C)^2).$$

In other words, the modularity of P also corresponds to the difference between the probability of sampling a pair of identical clusters with the joint distribution p, and the probability of sampling the same cluster twice independently under the marginal distribution of p.

If we use $w^{(E)}(C)$ to denote the sum of the weights of the edges within cluster C, $w^{(E)} = \sum_{i,j \in C} A_{ij}$, and $w^{(V)}(C)$ to denote the sum of the weighted degrees of the nodes within cluster C, $w^{(V)} = \sum_{i \in C} w_i$, then we get

$$Q(P) = \sum_{C \in P} \frac{w^{(E)}(C)}{w} - \sum_{C \in P} \left(\frac{w^{(V)}(C)}{w} \right)^2.$$

In this form, the first term can still be interpreted as the weighted proportion of in-group edges. Now, we see that the second term corresponds to the Simpson index associated with the probability distribution $\frac{w^{(V)}(C_1)}{w}, \ldots, \frac{w^{(V)}(C_K)}{w}$. The Simpson index is a popular measure of diversity in biology: if we interpret $p(C) = w^{(V)}(C)/w$ as the proportion of individuals of *species* C, it corresponds to the probability of getting two individuals in the same species when sampled uniformly at random in the total population [Simpson, 1949]. The minimal value of the Simpson index for a partition with K clusters is $\frac{1}{K}$. It corresponds to the most *diverse* distribution. As a consequence, we see that the modularity of a partition with K clusters cannot exceed $1 - \frac{1}{K}$.

Random walk interpretation

The modularity function can also be interpreted in terms of random walks on the nodes of the graph $G = (V, E)$. Consider a random walk on the nodes of G, where the probability of moving from node i to node j is $p_{ij} = A_{ij}/w_i$. The successive nodes $X_0, X_1, X_2, \ldots$ visited by the random walk form a Markov chain $(X_t)_{t \geq 0}$ on V. We have:

$$\forall t \geq 0, \forall i \in V, \quad \mathbb{P}(X_{t+1} = i) = \sum_{j \in V} \mathbb{P}(X_t = j) p_{ji}.$$

From this equation, we see that π is a stationary distribution of $(X_t)_{t \geq 0}$ if and only if

$$\forall i \in V, \quad \pi_i = \sum_{j \in V} \pi_j p_{ji}.$$

If the graph G is undirected, it is easy to see that the distribution where the probability of being at a node i is proportional to its weighted degree w_i, defined as $\pi_i = w_i/w$, is a stationary distribution of the Markov chain $(X_t)_{t \geq 0}$. If the graph is also connected and not bipartite, the Perron-Frobenius theorem [Perron, 1907] implies that $\pi_i = w_i/w$ is the unique stationary distribution of the Markov chain. Moreover, in this case, π is the limiting distribution of X_t when $t \to \infty$, independently of the distribution of the initial state X_0.

The modularity of a partition P can be written

$$Q(P) = \sum_{C \in P} (\mathbb{P}_\pi[X_0 \in C, X_1 \in C] - \mathbb{P}_\pi[X_0 \in C, X_\infty \in C])$$

where $\mathbb{P}_\pi[X_0 \in C, X_1 \in C]$ is the probability for the walk to be in C at initial time (with X_0 initialized with the distribution π) and after one step, and $\mathbb{P}_\pi[X_0 \in C, X_\infty \in C]$ is the probability for the walk to be in C at initial time and in the stationary regime.

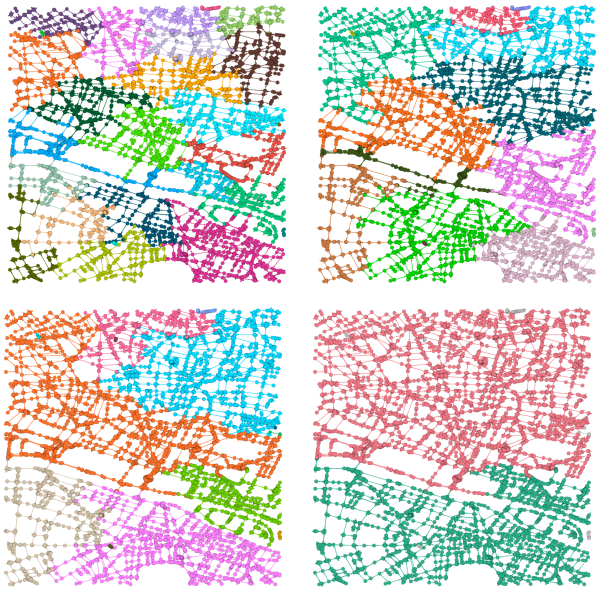


Figure 1.7: Open Street Map graph (Paris) – results of the Louvain algorithm for different resolutions

Resolution

The modularity function defined in (1.1) has an important limitation called the *resolution limit*. As studied in [Fortunato and Barthelemy, 2007], the modularity function $Q(P)$ is not fit for finding many small clusters in large graphs. Indeed, maximizing the modularity of a partition P is equivalent to maximizing $\sum_{C \in P} \sum_{i,j \in C} (A_{ij} - w_i w_j / w)$, and we see that, while the first term $\sum_C \sum_{i,j \in C} A_{ij}$ does not depend on the size of the graph, the second term $\sum_C \sum_{i,j \in C} w_i w_j / w$ depends on the total edge weight w and vanishes as w tends to infinity. In particular, this leads to the following surprising situation for disconnected graphs (i.e. graphs with multiple connected components): adding or removing edges to a connected component A has an impact on the clusters found by modularity maximization in another connected component B (because w is impacted), whereas it should have no effect on the community structure in B, as A and B are disconnected.

To get around this resolution limit, Reichardt and Bornholdt [Reichardt and Bornholdt, 2006] proposed a generalized modularity function

$$Q_\gamma(P) = \frac{1}{w} \sum_{C \in P} \sum_{i,j \in C} \left(A_{ij} - \gamma \frac{w_i w_j}{w} \right), \tag{1.2}$$

where $\gamma \geq 0$ is called the *resolution parameter*. We see that, if $\gamma = 1$, then Q_γ corresponds to the standard modularity function. Otherwise, γ counterbalances the normalization factor $1/w$ in the second term. In other works [Lambiotte et al., 2008], authors consider another definition for the generalized modularity function:

$$Q_t(P) = \frac{1}{w} \sum_{C \in P} \sum_{i,j \in C} \left(t A_{ij} - \frac{w_i w_j}{w} \right), \tag{1.3}$$

with $t \geq 0$. Note that both of these definitions are equivalent in the sense that the maximization of one or the other of these functions leads to the same results if we take $t = 1/\gamma$.

In order to illustrate the impact of this resolution parameter, we show in Figure 1.7 the partitions obtained for the Open Street Map graph centered on Paris with a modularity optimization method (the Louvain algorithm) for different values of resolutions. The images are ordered by increasing values of t (or, equivalently, by decreasing values of γ). We see on the top left image, which corresponds to the smallest value of t, that the clusters correspond to local areas of Paris, like *Saint Michel* or *Saint André des Arts*, that are smaller than Paris *arrondissements*. On the bottom right image, we see that we have only two clusters that correspond to the left bank and right bank of Paris.

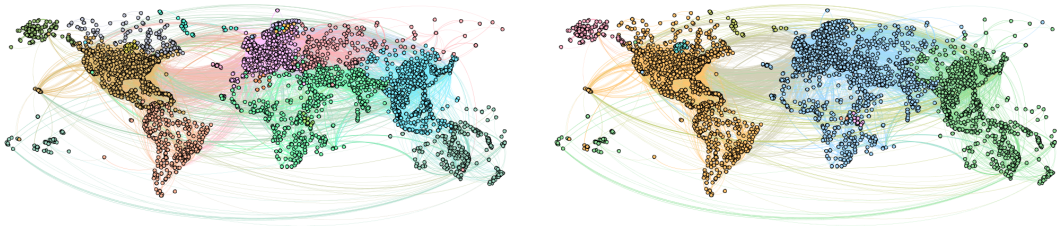


Figure 1.8: Open Flight graph – results of the Louvain algorithm for different resolutions

We show similar results for the Open Flight graph in Figure 1.8. We see on the left image, that corresponds to the smallest value of t, that the clusters nearly correspond to the continents, whereas, on the right image, the clusters correspond to larger regions (for instance, Europe and Africa, or South and North America have been grouped together).

Equivalence with the maximum likelihood method on a degree-corrected SBM

Newman has established a link between the modularity maximization problem and the maximum likelihood method applied to a random graph model called the degree-corrected stochastic block model [Newman, 2016]. Stochastic block models, or SBM, are a popular class of models for random graphs with community structures [Holland et al., 1983]. The standard stochastic block model is defined as follows. We are given as parameters of this model the set of nodes V and a partition $P = \{C_1, \dots, C_K\}$ of this set, as well as a matrix $p = (p_{kl})_{1 \leq k,l \leq K}$ of parameters such that $p_{kl} \in [0, 1]$. These parameters determine the probability of edges within and between each pair of cluster (C_k, C_l). More precisely, in this model, for each pair of node $(i, j) \in V \times V$, an edge is put between i and j with probability $p_{C(i)C(j)}$, where $C(i)$ (respectively $C(j)$) denotes the cluster $C \in P$ to which i belongs (resp. to which j belongs).

There exists many variants of this stochastic block model. A common variant consists in putting an edge between each pair of node $(i, j) \in V \times V$ with a weight drawn with a Poisson distribution $\mathrm{Poi}(p_{C(i)C(j)})$ of parameter $p_{C(i)C(j)}$. In this case, p_{kl} is the expected weight of an edge between nodes in groups k and l rather than the probability for such an edge to exist. Another variant of this model considers that the weight of an edge $(i, j) \in V \times V$ is drawn with the distribution $\mathrm{Poi}(\frac{w_i w_j}{w} p_{C(i)C(j)})$. Such a model is referred to as a *degree-corrected* block model [Zhao et al., 2012], and is particularly interesting because graphs generated with standard SBM (i.e. with non-degree-corrected models) have degree distributions very different from the real-world graphs, whereas, with degree-corrected block models, we can generate graphs that fit any degree distribution. We use the notation $p'_{C(i)C(j)} = \frac{w_i w_j}{w} p_{C(i)C(j)}$ in the following to make the calculations more concise.

Here, we consider a slight variation of the degree-corrected model described above where the weight of an edge $(i, j) \in V \times V$ is drawn with the distribution $\mathrm{Poi}(\frac{w_i w_j}{w} p_{C(i)C(j)})$ if $i \neq j$, and with the distribution $\mathrm{Poi}(\frac{w_i w_j}{2w} p_{C(i)C(i)})$ if $i = j$ (i.e. if (i, j) is a self-loop). This model can be used to generate a graph but also to infer the parameters p and $P = (C_1, \dots, C_K)$ from an observed graph. Indeed, if we observe a graph G and make the assumption that this graph has been generated with the block model described above, we can be interested in finding the parameters of this model that best explains the graph G. To do so, we study the probability that an observed graph G, with the adjacency matrix A, is generated with given values of parameters $p = (p_{kl})_{k,l \in [\![1,K]\!]}$ and P. This probability, or *likelihood*, is given by

$$\mathbb{P}(A|P, p) = \prod_{i<j} \mathbb{P}\left[\mathrm{Poi}\left(p'_{C(i)C(j)}\right) = A_{ij}\right] \prod_i \mathbb{P}\left[\mathrm{Poi}\left(\frac{p'_{C(i)C(i)}}{2}\right) = \frac{A_{ii}}{2}\right],$$

where $\mathbb{P}\left[\mathrm{Poi}\left(\lambda\right) = x\right]$ denotes the probability for a Poisson random variable with parameter λ to be equal to x. Note that we have adopted in this expression the convention where the weight of a self-edge (i, i) is $A_{ii}/2$, and not A_{ii}. Knowing that $\mathbb{P}\left[\mathrm{Poi}\left(\lambda\right) = x\right] = \frac{\lambda}{x!} e^{-\lambda}$, we get

$$\mathbb{P}(A|P, p) = \prod_{i<j} \frac{(p'_{C(i)C(j)})^{A_{ij}}}{A_{ij}!} e^{-p'_{C(i)C(j)}} \prod_i \frac{\left(\frac{p'_{C(i)C(i)}}{2}\right)^{A_{ii}/2}}{(A_{ii}/2)!} e^{-p'_{C(i)C(i)}/2}.$$

We are interested in the parameters P and p that maximize this quantity, or, equivalently, the logarithm of this quantity, i.e. the *log-likelihood*:

$$\log \mathbb{P}(A|P,p) = \sum_{i<j} \left[A_{ij} \log({p'}_{C(i)C(j)}) - \log A_{ij}! - p'_{C(i)C(j)} \right] - \sum_i \left[\frac{A_{ii}}{2} \log \left(\frac{{p'}_{C(i)C(i)}}{2} \right) - \log(A_{ii}/2)! - \frac{p'_{C(i)C(i)}}{2} \right]$$

Replacing p'_{kl} with its value, we get

$$\log \mathbb{P}(A|P,p) = \sum_{i<j} \left[A_{ij} \log \left(\frac{w_i w_j}{w} \right) + A_{ij} \log(p_{C(i)C(j)}) - \log A_{ij}! - \frac{w_i w_j}{w} p_{C(i)C(j)} \right] - \sum_i \left[\frac{A_{ii}}{2} \log \left(\frac{w_i w_i}{2w} \right) + \frac{A_{ii}}{2} \log(p_{C(i)C(i)}) - \log(A_{ii}/2)! - \frac{w_i w_i}{2w} p_{C(i)C(i)} \right].$$

We can group the terms that do not depend on the parameters P and p in a constant c. We obtain

$$\log \mathbb{P}(A|P,p) = \frac{1}{2} \sum_{i,j} \left[A_{ij} \log(p_{C(i)C(j)}) - \frac{w_i w_j}{w} p_{C(i)C(j)} \right] + c$$

In order to establish a connection with the modularity, Newman consider the special case where $p_{kl} = p_{\text{in}}$ if $k = l$ and $p_{kl} = p_{\text{out}}$ otherwise. Therefore, the average weight of intra-cluster edges is p_{in} and the average weight of inter-cluster edges is p_{out} for all clusters. Using the Kronecker delta (δ such that $\delta_{kl} = 1$ if $k = l$, and $\delta_{kl} = 0$ otherwise), we have

$$\log p_{kl} = \log \frac{p_{\text{in}}}{p_{\text{out}}} \delta_{kl} + \log p_{\text{out}} \quad \text{and} \quad p_{kl} = (p_{\text{in}} - p_{\text{out}})\delta_{kl} + p_{\text{out}}.$$

This gives us

$$\begin{aligned} \log \mathbb{P}(A|P,p) &= \frac{1}{2} \sum_{i,j} \left[A_{ij} \left(\delta_{C(i)C(j)} \log \frac{p_{\text{in}}}{p_{\text{out}}} + \log p_{\text{out}} \right) - \frac{w_i w_j}{w} \left(\delta_{C(i)C(j)} (p_{\text{in}} - p_{\text{out}}) + p_{\text{out}} \right) \right] + c \\ &= \frac{1}{2} \log \frac{p_{\text{in}}}{p_{\text{out}}} \sum_{ij} \left(A_{ij} - \frac{p_{\text{in}} - p_{\text{out}}}{\log(p_{\text{in}}/p_{\text{out}})} \frac{w_i w_j}{w} \right) \delta_{C(i)C(j)} + c + \frac{w}{2} (\log p_{\text{out}} - p_{\text{out}}). \end{aligned}$$

If we try to maximize this quantity with respect to the planted partition P for fixed values of the parameters p_{in} and p_{out}, we obtain

$$\log \mathbb{P}(A|P,p) = \alpha \sum_{C \in P} \sum_{i,j \in C} \left[A_{ij} - \gamma \frac{w_i w_j}{w} \right] + \beta,$$

where α and β are constants that depends on A, p_{in} and p_{out} but not on P, and where $\gamma = \frac{p_{\text{in}} - p_{\text{out}}}{\log(p_{\text{in}}/p_{\text{out}})}$. Finally, we see that maximizing the maximum likelihood for this degree-corrected stochastic block model for fixed values of p_{in} and p_{out} is *strictly equivalent* to maximizing the generalized modularity (1.2) with the resolution γ.

Stability and continuous-time random walks

We have seen above that the modularity can be interpreted in terms of random walk on the graph nodes with the expression

$$Q(P) = \sum_{C \in P} \left(\mathbb{P}_\pi[X_0 \in C, X_1 \in C] - \mathbb{P}_\pi[X_0 \in C, X_\infty \in C] \right).$$

A natural generalization of the modularity based on this expression is the stability [Lambiotte et al., 2008], defined as

$$R(t) = \sum_{C \in P} \left(\mathbb{P}_\pi[X_0 \in C, X_t \in C] - \mathbb{P}_\pi[X_0 \in C, X_\infty \in C] \right). \tag{1.4}$$

In this definition, we consider the discrete-time random walk, whose dynamics is governed by the equation

$$\mathbb{P}(X_{t+1} = i) = \sum_j \frac{A_{ji}}{w_j} \mathbb{P}(X_t = j).$$

Writing the distribution of X_t as a vector $\pi(t)$, and using $D = \mathrm{diag}(w_1, \ldots, w_n)$, the diagonal matrix with the weighted node degrees, this becomes

$$\pi(t+1)^T = \pi(t)^T D^{-1} A.$$

Instead of this discrete-time random walk, we can consider the continuous-time process associated with this random walk. The dynamics of this process is governed by the Kolmogorov equation

$$\frac{\partial \pi(t)^T}{\partial t} = \pi(t)^T (D^{-1}A - I),$$

which leads to $\pi(t)^T = \pi(0)^T e^{t(D^{-1}A - I)}$. If we use this continuous-time process in the definition (1.4), the stability becomes

$$R'(t) = \sum_{C \in P} \sum_{i,j \in C} \left(\pi_i \left(e^{t(D^{-1}A - I)} \right)_{ij} - \pi_i \pi_j \right).$$

The stationary distribution π of this process for an undirected graph is equal to the stationary distribution of the discrete-time random walk, $\pi_i = w_i/w$. Therefore, we get

$$R'(t) = \sum_{C \in P} \sum_{i,j \in C} \left(\frac{w_i}{w} \left(e^{t(D^{-1}A - I)} \right)_{ij} - \frac{w_i w_j}{w^2} \right).$$

In the limit of small diffusion times t, we have

$$\begin{aligned} R'(t) &= \sum_{C \in P} \sum_{i,j \in C} \left(\frac{w_i}{w} \left[(1-t)I + tD^{-1}A + o(t) \right]_{ij} - \frac{w_i w_j}{w^2} \right) \\ &= \frac{1}{w} \sum_{C \in P} \sum_{i,j \in C} \left(tA_{ij} - \frac{w_i w_j}{w} \right) + (1-t)I + o(t). \end{aligned}$$

We see that maximizing the linearization of $R(t)$ in the limit $t \to 0$ is equivalent to maximizing

$$\frac{1}{w} \sum_{C \in P} \sum_{i,j \in C} \left(tA_{ij} - \frac{w_i w_j}{w} \right)$$

which corresponds to the generalized modularity presented above (1.3). Moreover, we see in this interpretation that the resolution t corresponds to the diffusion time t considered in the stability $R'(t)$. Therefore, the smaller t, the more local the information we take into account.

Note that the notion of stability proposed by Lambiotte et al. and the concept of modularity can be unified in a probabilistic framework defined in [Chang et al., 2015, Chang et al., 2018b]. In this work, the authors consider any symmetric bivariate distribution $p(\cdot, \cdot)$ for sampling pairs of nodes. The marginal distribution $p(i) = \sum_{j \in V} p(i,j)$ of such a distribution can naturally be used to sample individual nodes. From such a distribution, the authors define the notion of *centrality* for any set of node $S \subset V$ as the probability of sampling a node in S, i.e.

$$C(S) = \sum_{i \in S} \sum_{j \in V} p(i,j) = \sum_{i \in S} p(i),$$

and the notion of *relative centrality* of a set of nodes S_1 with respect to another set of nodes S_2 as the probability of sampling a node pair (i,j) with $i \in S_1$ knowing that j is in S_2, i.e.

$$C(S_1 | S_2) = \frac{\sum_{i \in S_1} \sum_{j \in S_2} p(i,j)}{C(S_2)}.$$

Using these notions, Chang et al. define the notion of cluster *strength* as the difference of the relative centrality of a set $S \in V$ with respect to itself on the one hand, and its centrality on the other hand:

$$Str(S) = C(S|S) - C(S).$$

Based on this new concept, they propose a unified definition for the modularity of a partition P, which is defined as the weighted average of cluster strengths using cluster centralities as weights, i.e.,

$$Q(P) = \sum_{S \in P} C(S) Str(C).$$

This notion unifies the different concepts presented above. Indeed, taking the sampling distribution $p(i,j) = A_{i,j}/w$, we find Newman's modularity, and taking $p(i,j) = \pi_i \left(e^{t(D^{-1}A-I)}\right)_{ij}$, we find Lambiotte's stability.

Newman's greedy algorithms

The modularity maximization problem that consists in finding a partition P that maximizes $Q(P)$ cannot be solved exactly in acceptable time as it has been proven to be NP-hard [Brandes et al., 2007]. However, many methods have been proposed to find good approximations of the modularity optimum in reasonable time. These techniques includes greedy algorithms, spectral approaches [Newman, 2006], and simulated annealing methods [Guimera et al., 2004]. A thorough review of existing algorithms can be found in [Fortunato and Castellano, 2012].

The greedy algorithm [Newman, 2004] proposed by Newman is certainly the most simple heuristic for maximizing the modularity. It starts from the situation where every node is isolated in its own cluster, i.e. $P = \{\{i\}, i \in V\}$. Then, at each step, it merges the pair of clusters that leads to the largest increase in modularity. In other words, if $P = \{C_1, \ldots, C_K\}$ is the partition at the beginning of a step, the algorithm considers the partitions $P_{kl} = \{C_k \cup C_l\} \cup \{C_{k'}, k' \neq k, k' \neq l\}$, and picks the one that corresponds to the highest value of $\Delta Q = Q(P_{kl}) - Q(P)$. The algorithm stops when no merge yields to a positive increase of modularity.

Since merging two clusters with no edge between them can only lead to a non-positive modularity increase ΔQ, we only need to consider the pairs of clusters between which there are edges. Therefore, the number of pairs to consider at each step is at most m. The modularity increase is given by

$$\Delta Q = Q(P_{kl}) - Q(P) = \frac{w^{(E)}(C_k, C_l) + w^{(E)}(C_l, C_k)}{w} - 2\frac{w^{(V)}(C_k) w^{(V)}(C_l)}{w^2},$$

where $w^{(E)}(C_k, C_l) = \sum_{i \in C_k} \sum_{j \in C_l} A_{ij}$. If we store the coefficients $w^{(E)}(C_l, C_k)$ and $w^{(V)}(C_k)$ for all clusters throughout the execution of the algorithms, the update ΔQ can be computed in constant time. The update of these coefficients after each merge takes $O(n)$ in the worst-case scenario. Hence, each step is in $O(n+m)$. Since, there is at most $n-1$ steps, the time complexity of the algorithm is in $O(n(n+m))$.

The Louvain algorithm

Note that, in the Newman's algorithm presented above, if two clusters have been merged during a step of the algorithm, they cannot be split at a later step. For this reason, this algorithm is often too elementary for recovering clusters in real-world graphs. The widespread Louvain algorithm uses a similar but more sophisticated approach. The algorithm was named after the university of its inventors [Blondel et al., 2008].

Like in the Newman's algorithm, each node is placed in its own cluster at the initialization of the Louvain algorithm. Then the algorithm repeats the following two steps.

1. (Greedy maximization) While modularity Q increases, we cycle over nodes and move each node to the neighboring cluster that results in the largest modularity increase.

2. (Aggregation) We build a new graph whose nodes are the clusters found during step 1 and where edge weights correspond to the sum of edge weights between clusters (self loops with a weight corresponding to the sum of the intra-cluster edge weights are also added to each node).

The algorithm stops when the modularity increase is lower than a threshold $\epsilon > 0$ given as a parameter. The result of the first step depends on the order in which the nodes are considered. For this reason, the nodes are typically shuffled at the beginning of the algorithm. The second step forces the algorithm to explore more solutions by merging clusters. This aggregation step is build on the key result that states that modularity Q is invariant under the aggregation step. In other words, the modularity of the partition obtained after step 1 is the same as the modularity of the partition where each node is isolated in the aggregated graph.

The aggregated graph is typically much smaller than the original graph, which allows the Louvain algorithm to handle massive real-world graphs [Prat-Pérez et al., 2014]. However, the complexity of the algorithm cannot be easily bounded as it depends on the number of iterations during the first step which is graph specific. Note that the increase considered during step 1, where a node i is moved from its current cluster C_k to a neighboring cluster C_l, can be written

$$\Delta Q = \frac{w^{(E)}(\{i\}, C_l) - w^{(E)}(\{i\}, C_k)}{w} - 2\frac{w_i(w^{(V)}(C_k) - w^{(V)}(C_l) + w_i)}{w^2}.$$

In order to speed up the computation of ΔQ, we typically store the node-cluster weights, $w^{(E)}(\{i\}, C_k)$, throughout the algorithm. This requires $O(m)$ memory and allows one iteration during step 1 to be in $O(m)$ (but the number of iterations cannot be bounded).

1.4.2 Spectral methods

Another popular class of methods for graph clustering that plays an important role in our work is known as *spectral methods*. Like the modularity, we will see that spectral methods can be linked to random walks. These methods are based on the spectral decomposition of the *Laplacian* matrix, a key matrix in graph theory. This matrix is defined as

$$L = D - A$$

where $D = \mathrm{diag}(w_1, \ldots, w_n)$ is the diagonal matrix containing the weighted degrees of the nodes. Note that there exists alternative definitions of the Laplacian matrix that we will review below.

Connection with the Laplacian operator

The Laplacian matrix corresponds to the discrete version of the negative Laplacian operator $f \mapsto -\Delta f = -\sum_i \partial^2 f/\partial x_i^2$. Indeed, if $f : V \to \mathbb{R}$ is a function on the nodes of the graph, then we can represent this function as a vector $f = (f_x)_{x \in V} \in \mathbb{R}^n$ and get

$$(Lf)_x = \sum_y (D_{xy} - A_{xy}) f_y = w_x f_x - \sum_y A_{xy} f_y = \sum_y A_{xy}(f_x - f_y).$$

We consider the case where the graph $G = (V, E)$ is an infinite square lattice grid in $\mathbb{R}^K$ defined as

$$V = \{\epsilon(a_1, \ldots, a_K) | a_i \in \mathbb{Z}\} \subset \mathbb{R}^K$$
$$A_{xy} = \begin{cases} 1/\epsilon^2 & \text{if } \exists i, y = x \pm \epsilon e_i \\ 0 & \text{otherwise,} \end{cases}$$

where $(e_1, \ldots, e_K)$ is the canonical orthonormal basis of $\mathbb{R}^K$. Then we have a direct connection between the Laplacian matrix and the negative continuous Laplacian operator:

$$(Lf)_x = \sum_y A_{xy}(f_x - f_y) = \sum_i \frac{(f(x) - f(x - \epsilon e_i)) + (f(x) - f(x + \epsilon e_i))}{\epsilon^2} \underset{\epsilon \to 0}{\to} -\sum_i \frac{\partial^2 f}{\partial x_i^2} = -\Delta f.$$

Spectral decomposition

As stated above, spectral methods are based on the spectral decomposition of the Laplacian matrix. We have, for all $v \in \mathbb{R}^n$,

$$v^T L v = v^T (D - A) v = \sum_{ij} v_i (D_{ij} - A_{ij}) v_j = \sum_i w_i v_i^2 - \sum_{ij} A_{ij} v_i v_j = \sum_{ij} A_{ij}(v_i^2 - v_i v_j).$$

In the following, we assume that G is undirected. Therefore, A is symmetric, and we get

$$v^T L v = \sum_{ij} A_{ij}(\frac{v_i^2 + v_j^2}{2} - v_i v_j) = \frac{1}{2}\sum_{ij} A_{ij}(v_i - v_j)^2. \tag{1.5}$$

As a consequence, the Laplacian matrix L is *positive semi-definite*. Thus, the spectral theorem can be applied to L and yields to

$$L = V \Lambda V^T$$

where $\Lambda = \mathrm{diag}(\lambda_1, \ldots, \lambda_n)$ is the diagonal matrix of the eigenvalues of L, with $0 \leq \lambda_1 \leq \ldots \leq \lambda_n$, and $V = (v_1, \ldots, v_n)$ is the matrix whose columns are the corresponding eigenvectors, with $V^T V = I$. Using (1.5) we see that $v^T L v = 0$ if and only if $\forall i, j, A_{ij} > 0 \Rightarrow v_i = v_j$. If the graph is connected, this yields to $v \propto 1$. In this case we have $\lambda_1 = 0$, $\lambda_2 > 0$ and $v_1 = 1/\sqrt{n}$. The vector v_2 of the Laplacian associated with the second smallest eigenvalue is called the *Fiedler vector*. It is the first non trivial vector of the Laplacian. We show below that this vector has important properties.

Minimum cut interpretation

Given a partition of the graph nodes into two clusters, A and $\bar{A} = V \setminus A$, the cut is defined as

$$\mathrm{cut}(A, \bar{A}) = \sum_{i \in A} \sum_{j \in \bar{A}} A_{ij}.$$

A classical problem consists in finding the set A such that $A \neq \emptyset$ and $A \neq V$ that minimizes $\mathrm{cut}(A, \bar{A})$. A solution to this problem is a simple clustering of G into two clusters. We define the indicator vector associated with the partition $\{A, \bar{A}\}$ as $v_i = 1$ if $i \in A$ and $v_i = -1$ otherwise. We have

$$v^T L v = \frac{1}{2}\sum_{ij} A_{ij}(v_i - v_j)^2 = \frac{1}{2}\sum_{i \in A, j \in \bar{A}} 4A_{ij} + \frac{1}{2}\sum_{i \in \bar{A}, j \in A} 4A_{ij} = 4\mathrm{cut}(A, \bar{A})$$

Thus, minimizing $v^T L v$ with $v \in \{-1, 1\}^n$ is equivalent to minimizing $\mathrm{cut}(A, \bar{A})$. As explained above, a trivial solution of this problem is $v \propto 1$, i.e. $v = 1$ or $v = -1$, which corresponds to $A = \emptyset$ or $A = V$. If we add the constraint $|A| = |\bar{A}|$, the equivalent constraint on v is $\sum_i v_i = v^T 1 = 0$. Then, we get the minimization problem

$$\begin{array}{ll} \underset{v \in \{-1,1\}^n}{\text{minimize}} & v^T L v \\ \text{subject to} & v^T 1 = 0. \end{array}$$

A natural relaxation of this problem is

$$\begin{array}{ll} \underset{v \in \mathbb{R}^n}{\text{minimize}} & v^T L v \\ \text{subject to} & v^T 1 = 0 \\ & \|v\| = 1. \end{array}$$

As the solution is orthogonal to the first eigenvector $v_1 = 1$ of L, it is easy to see that a solution to this problem is proportional to the Fiedler vector v_2, i.e. the eigenvector associated with the second smallest eigenvalue of L.

Normalized Laplacian

So far, we have considered the Laplacian matrix defined as $L = D - A$. In many papers, researchers do not consider the Laplacian L but is *normalized* version [Chung, 1997]

$$\mathcal{L} = D^{-1/2} L D^{-1/2} = I - D^{-1/2} A D^{-1/2}$$

where $D^{-1/2} = \mathrm{diag}(w_1^{-1/2}, \ldots, w_n^{-1/2})$. The normalized Laplacian is closely related to the canonical random walk on the nodes of the graph. Consider the random walk in the graph where the probability

of moving from node i to node j is $p_{ij} = A_{ij}/w_i$. We call X_t the tth node visited by the walk. As seen above, $(X_t)_{t\geq 0}$ is a Markov chain on V, and we have

$$\forall i, \mathbb{P}(X_{t+1} = i) = \sum_j \mathbb{P}(X_t = j) p_{ji}.$$

Writing the distribution of X_t as a vector $\pi(t)$, this equation becomes

$$\pi(t+1)^T = \pi(t)^T P$$

where $P = D^{-1}A$ is the transition matrix of the Markov chains. A stationary distribution π of the Markov chain satisfies $\pi_i = \sum_j \pi_j p_{ji}$ i.e.

$$\pi^T = \pi^T P.$$

Thus π^T is the left eigenvector of P associated with the eigenvalue 1 such that $\pi^T 1 = 1$. As seen previously, if the graph is connected and not bipartite, by Perron-Frobenius this stationary distribution is unique and it is the limit of $\pi(t)$ when $t \to \infty$.

The normalized Laplacian is closely related to the transition matrix P since we have

$$P = D^{-1/2}(I - \mathcal{L})D^{1/2}. \tag{1.6}$$

Unlike P, $\mathcal{L}$ is symmetric. Therefore, by the spectral theorem, there exists some matrix X such that

$$\mathcal{L} = X\Lambda X^T$$

where $X^T X = I$ and $\Lambda = \mathrm{diag}(\lambda_1, \ldots, \lambda_n)$ is the diagonal matrix with the eigenvalues of $\mathcal{L}$, with $|\lambda_1| \leq \ldots \leq |\lambda_n|$. The columns of X are the eigenvectors of $\mathcal{L}$ for the respective eigenvalues $\lambda_1, \ldots, \lambda_n$. From (1.6), we get

$$P = V(I - \Lambda)U^T$$

with $U = \frac{D^{1/2}X}{\sqrt{w}}$ and $V = D^{-1/2}X\sqrt{w}$. Observe that $U^T V = I$. As a consequence, the lines of U^T and the columns of V are the left and right eigenvectors of P for the respective eigenvalues $1 - \lambda_1, \ldots, 1 - \lambda_n$. We can apply the Perron-Frobenius theorem to P if the graph is connected and not bipartite. Then we obtain $1 - \lambda_1 = 1 > 1 - \lambda_2 \geq 1 - \lambda_3 \geq \ldots \geq 1 - \lambda_n$, which gives us

$$\lambda_1 = 0 < \lambda_2 \leq \ldots \leq \lambda_n.$$

We can easily verify that the eigenvector of $\mathcal{L}$ associated with the eigenvalue $\lambda_1 = 0$ is $x_1 = \pm\sqrt{\pi}$. The right and left eigenvectors of P associated with the eigenvalue $1 - \lambda_1 = 1$ are $u_1 = \pi$ and $v_1 = 1$.

Link with the modularity matrix

The modularity of a partition P can be rewritten as

$$Q(P) = \sum_{C \in P} \sum_{i,j \in C} Q_{ij},$$

with $Q_{ij} = \frac{1}{w}\left(A_{ij} - \frac{w_i w_j}{w}\right)$. We call $\boldsymbol{Q} = (Q_{ij})_{1\leq i,j\leq n}$ the *modularity matrix*. We will see that $\boldsymbol{Q}$ can be rewritten using the spectral decomposition of the normalized Laplacian $\mathcal{L}$. If $\boldsymbol{w}$ is the vector whose components are the weighted degrees w_i, we have

$$\begin{aligned}
\boldsymbol{Q} = \frac{1}{w}\left(A - \frac{\boldsymbol{w}\boldsymbol{w}^T}{w}\right) &= \frac{1}{w} D^{1/2}\left[D^{-1/2}AD^{-1/2} - \frac{\sqrt{\boldsymbol{w}}\sqrt{\boldsymbol{w}}^T}{w}\right] D^{1/2} \\
&= \frac{1}{w} D^{1/2}\left[I - \underbrace{\mathcal{L}}_{=X\Lambda X^T} - x_1 x_1^T\right] D^{1/2} \\
&= Y\Lambda_Q Y^T,
\end{aligned}$$

with $Y = \frac{1}{\sqrt{w}} D^{1/2} X$ and $\Lambda_Q = \mathrm{diag}(0, 1 - \lambda_1, \ldots, 1 - \lambda_n)$.

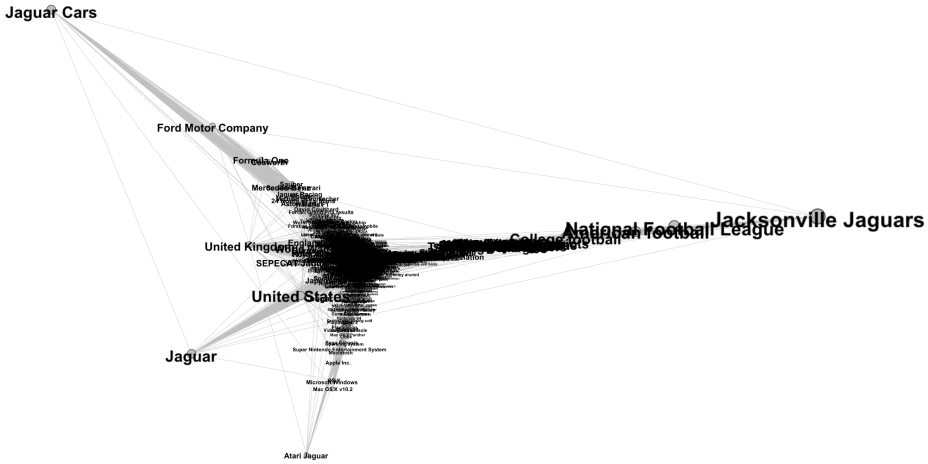


Figure 1.9: Spectral embedding in 2D of the *Jaguar* subgraph of Wikipedia with the normalized Laplacian

Therefore, the modularity matrix $\boldsymbol{Q}$ and the diagonal matrix Λ_Q are congruents. By Sylvester's law of inertia [Sylvester, 1852], $\boldsymbol{Q}$ and Λ_Q have the same number of positive, negative and zero eigenvalues. Hence, if $\mu_1, \ldots, \mu_n$ are the eigenvalues of $\boldsymbol{Q}$, we have

$$
\begin{aligned}
|i : \mu_i = 0| &= |i : \lambda_i = 1| + 1 \\
|i : \mu_i < 0| &= |i : \lambda_i > 1| \\
|i : \mu_i > 0| &= |i : \lambda_i < 1|.
\end{aligned}
$$

Spectral embedding

Graph embedding aims at representing a graph in a low-dimensional Euclidean space $\mathbb{R}^K$ with $K \ll n$. More precisely, each node $i \in V$ is mapped to a vector $x_i \in \mathbb{R}^K$. A *good* embedding must encode the graph structure in the sense that, if two nodes i and j are *close* in the graph G, they must be represented by two vectors x_i and x_j that are *close* in $\mathbb{R}^K$, i.e. for the Euclidean distance $\|x_i - x_j\|$.

The spectral decomposition of the Laplacian matrix can be used to build such an embedding [Von Luxburg, 2007]. To do so, we compute the first K eigenvectors of the Laplacian L, i.e. the eigenvectors $v_1, \ldots, v_K$ corresponding to the smallest K eigenvalues $\lambda_1 = 0 \leq \lambda_2 \leq \ldots \leq \lambda_K$. These eigenvectors can be computed efficiently with the Lanczos algorithm [Lanczos, 1950]. In the so-called *spectral embedding*, each node $i \in V$ is mapped to the vector $x_i = (v_{1i}, \ldots, v_{Ki}) \in \mathbb{R}^K$. Note that similar embeddings can be performed by using the first K eigenvectors of the normalized Laplacian $\mathcal{L}$, or the first K left-eigenvectors $u_1, \ldots, u_K$ or the K right-eigenvectors $v_1, \ldots, v_K$ of the random walk transition matrix P.

In Figure 1.9, we show the result of the embedding of a subgraph of Wikipedia in two dimensions using the first two non-trivial eigenvectors x_2 and x_3 of the normalized Laplacian matrix $\mathcal{L}$. This graph corresponds to the subgraph induced by the articles at 2 hops from the *Jaguar (disambiguation)* article in the Wikipedia graph. In other words, this subgraph consists of neighbors of the *Jaguar (disambiguation)* node and of neighbors of these neighbors. This subgraph is quite interesting because we find articles that refer to very different concepts named Jaguar (e.g. Jaguar cars, the animal, Fender Jaguar guitars etc.). In Figure 1.9, we see that articles corresponding to different notions are mapped to different areas of the vector space $\mathbb{R}^2$. For instance, a direction in this embedding space corresponds to articles about American football as there exists a NFL team called the Jacksonville Jaguars. Another direction corresponds to sport cars related to the Jaguar cars. We can also distinguish thematic regions corresponding to the jaguar animal or the Atari Jaguar video game console. Zooming in, we could also observe a direction corresponding to military aircraft and, in particular, a node corresponding to the British-French aircraft SEPECAT Jaguar.

Figure 1.10: Embedding in 2D of the *Jaguar* subgraph of Wikipedia with the node2vec algorithm

Spectral clustering

Embedding methods can easily be used to cluster graph nodes. Indeed, since these embeddings are supposed to preserve the graph structure, we can expect node clusters in G to correspond to clusters of points in the embedding space $\mathbb{R}^K$. Therefore, a natural approach consists in first embedding the graph nodes $i \in V$ and then using a clustering algorithm in the embedding space $\mathbb{R}^K$ to cluster the images $x_i \in \mathbb{R}^K$ of these nodes. Such methods based on a spectral embedding are referred to as *spectral clustering* techniques. They traditionally use the k-means algorithm [Lloyd, 1982] to cluster the points in the Euclidean space $\mathbb{R}^K$. There exists a legion of variants for spectral clustering [Von Luxburg, 2007], using different Laplacian matrices and/or different normalization techniques for the eigenvectors. The most classic methods [Shi and Malik, 2000, Ng et al., 2002] are simply based on the Laplacian matrix L or its normalized version $\mathcal{L}$ and can be summarized as follows

- computing the first K eigenvectors of the Laplacian L (or the normalized Laplacian $\mathcal{L}$): $x_1, \ldots, x_K$;
- mapping each node $i \in V$ to the vector $(x_{1i}, \ldots, x_{Ki}) \in \mathbb{R}^K$;
- clustering the points $(y_i)_{i \in V}$ with the k-means algorithm into clusters $C_1, \ldots, C_k$;
- outputing the node clusters $C'_1, \ldots, C'_k$ where $C'_l = \{i \in V | y_i \in C_k\}$.

We see that the number of clusters must be given as a parameter of the algorithm. Note that the first step of this algorithm can easily be replaced by any other embedding method. As a matter of fact, there exists many similar graph clustering approaches that use this scheme with non-spectral embedding algorithms [Perozzi et al., 2014, Tang et al., 2015]. In Figure 1.10, we show the embedding ot the *Jaguar* subgraph of Wikipedia with an embedding algorithm called *node2vec* [Grover and Leskovec, 2016] that is based on the classic *Word2vec* algorithm [Mikolov et al., 2013], traditionally used to embed words in a document in Natural Language Processing (NLP). We see in Figure 1.10 that, like with the spectral embedding, the different regions in the vector space correspond to different topics in this dataset.

1.4.3 Some other graph clustering approaches

As stressed above, the objective of this introductory chapter is not to provide a comprehensive list of graph clustering algorithms. However, let us cite some other state-of-the-art techniques that we will use in

our different benchmarks throughout this work. First, note that there exists many greedy algorithms that optimize objective functions other than modularity. This is the case for instance of the SCD algorithm, that partitions the graph by maximizing the WCC [Prat-Pérez et al., 2014], which is a community quality metric based on triangle counting. Another example is OSLOM [Lancichinetti et al., 2011], that finds node partitions by locally optimizing a fitness function which measures the statistical significance of a community. Many other algorithms are based on *random walks*. For instance, Infomap [Rosvall and Bergstrom, 2008] splits the network into modules by compressing the information flow generated by random walks. Another example is Walktrap, that uses random walks to estimate the similarity between nodes, which is then used to cluster the network [Pons and Latapy, 2005]. Finally, let us cite a family of methods using non-negative matrix factorization [Lee and Seung, 2001] on the adjacency matrix A or on variants of this matrix, such as BigClam [Yang and Leskovec, 2013], that can be used to detect overlapping clusters in graph. We refer the reader to comparative studies [Lancichinetti and Fortunato, 2009, Fortunato and Castellano, 2012, Xie et al., 2013] for a more exhaustive list of graph clustering methods.

1.5 Evaluation of graph clustering algorithms

We presented above some classic graph clustering algorithms. A big question arises when we try to compare these different methods between each other: how to evaluate the performance of a graph clustering algorithm? For some approaches, such as spectral methods, theoretical guarantees about the quality of the recovered clusters can be obtained. But, most of these analyses only apply to restricted graph models that are often not in line with real-life graphs. The quality of the results of graph clustering algorithms can also be evaluated through experiments. These experiments can be conducted on synthetic graphs with *planted* clusters or on real-life graphs using *ground-truth* clusters. In this section, we introduce these different techniques to evaluate graph clustering algorithms.

1.5.1 Synthetic graph models

Synthetic graphs are mainly used in the theoretical analysis of graph clustering algorithms but can also be used in experiments [Lancichinetti et al., 2008]. These models are generally random graphs models, i.e. models to generate graphs from a specific probability distribution or a certain random process. Some models take the set of clusters $\mathcal{C} = \{C_1, \dots, C_K\}$, $C_k \in V$, as a parameter and generate the set of edges E from these so-called *planted* clusters. This is the case of the Stochastic Block Model for instance. Other models, such as the LFR model presented in this section, generate both the clusters and the edges.

Stochastic Block Models

We have already briefly introduced the Stochastic Block Model (SBM) above to give an interpretation of the modularity measure. Recall that, in the standard SBM [Holland et al., 1983], we are given a partition $P = \{C_1, \dots, C_K\}$ of the nodes V and a matrix $p = (p_{kl})_{1 \leq k,l \leq K} \in [0,1]^{K \times K}$, and recall that graphs are generated by connecting any pairs of nodes with the probability $p_{C(i)C(j)}$ independently of other pairs of nodes. Note that we use $C(i)$ to denote the cluster of P to which the node i belongs. Therefore, the probability that an observed graph with the adjacency matrix A is generated with this model is given by

$$\mathbb{P}[A|P,p] = \prod_{i<j} p_{C(i)C(j)}^{A_{ij}} (1 - p_{C(i)C(j)})^{1-A_{ij}}.$$

The matrix p, which is referred to as the *connectivity* matrix, determines the way different clusters interact with each other with the values p_{kl}, $k \neq l$, outside the diagonal, and the way nodes within the same cluster connect to each other with its diagonal coefficients, p_{kk}, $k \in [\![1, K]\!]$. In order to simplify this model, one commonly consider the restrained model where p takes the same value a on the diagonal, and the same value b outside the diagonal. This model is referred to as the *symmetric SBM*.

There exist many variants of the standard stochastic block model. Additional parameters can be added to the model to control the expected degree of each node [Karrer and Newman, 2011], which leads to the so-called *degree-corrected SBM*. Several definitions of *overlapping SBMs* have bee introduced to overcome the fact that a node can only belong to one cluster in the standard SBM [Airoldi et al., 2008, Fortunato, 2010]. Finally, it is worth mentioning that models, called *Labelled SBMs*, have been introduced to allow edges to carry labels such as a weight [Heimlicher et al., 2012, Lelarge et al., 2015].

The objective of graph clustering algorithms in a SBM is to recover the partition $P = \{C_1, \dots, C_K\}$ from the observed graph G. The performance of a graph clustering algorithm can simply be measured with a function f called the *agreement*, which measures the similarity between two community membership vectors $x \in [\![1, K]\!]^n$ and $y \in [\![1, K]\!]^n$, and is defined as

$$f(x, y) = \max_{\pi \in S_K} \frac{1}{n} \sum_{i \in V} \mathbb{1}_{\{x_i = \pi(y_i)\}},$$

where S_K is the group of permutation of $[\![1, K]\!]$. Therefore, if $x_i = C(i)$ for all $i \in V$, and $\hat{x}_i$ is the membership vector returned by a graph clustering algorithm, $f(x, \hat{x})$ is the maximum number of common components between x and any relabelling of $\hat{x}$.

Based on this measure, one can define different tasks for a graph clustering algorithm.

- (Detection) This task consists in determining if an observed graph has been generated from a SBM or from an Erdos-Renyi model without any cluster structure. More formally, a detection is solved in an SBM with a planted membership vector $x \in [\![1, K]\!]^n$ if, for an observed graph G, there exists $\epsilon > 0$ and an algorithm $\mathcal{A}$ that takes G as an input and outputs a membership vector $\hat{x}$ such that

$$\mathbb{P}\left[f(x, \hat{x}) \geq \frac{1}{K} + \epsilon\right] = 1 - o(1),$$

  when $n \times +\infty$.

- (Partial recovery) This task consists in approximately recovering the planted membership vector x. The quality of the recovery is measured through a parameter $\alpha \in (0, 1)$. More formally, an algorithm solves this task for α if it outputs $\hat{x}$ such that

$$\mathbb{P}[f(x, \hat{x}) \geq \alpha] = 1 - o(1).$$

- (Exact recovery) This task consists in recovering the planted membership vector x exactly. In other words, an algorithm solves this task if it outputs $\hat{x}$ such that

$$\mathbb{P}[f(x, \hat{x}) = 1] = 1 - o(1).$$

The difficulty of these tasks highly depends on the parameters p_{kl} (a and b in the symmetric SBM), and we typically observe phase transitions, i.e. range of parameters where these tasks are feasible or impossible. Besides, for different values of the parameters, different algorithms have been proven to be the most efficient [Abbe, 2017].

Note that spectral approaches based on different versions of the Lapacian matrix have widely been used for clustering graphs generated with the stochastic block model. But recently it has been proven that the spectral analysis of a more complicated, higher dimensional, and non-symmetric matrix, known as the Bethe Hessian, leads to better performances, and even reached the optimal performance on the classic SBM [Saade et al., 2014]. Whereas, the normalized Laplacian is related to the standard random walk, the Bethe Hessian is related to the *non-backtracking* random walk on the graph.

The LFR model

The Lancichinetti-Fortunato-Radicchi (LFR) model [Lancichinetti et al., 2008] is another popular model used to generate graphs with a cluster structure. Whereas the SBM is inspired by the Erdos-Renyi model, the LFR model takes its inspiration from the configuration model [Molloy and Reed, 1995]. The graphs generated from this model are supposed to be closer to real-world graphs because their degree distribution and their cluster size distribution follow power laws that are typically observed in real-life datasets. The LFR model takes 4 parameters, n, γ, β, μ and $\bar{d}$. n corresponds to the number of nodes in the generated graph. γ and β are the exponents of the power law distributions of the node degrees and respectively the community sizes. The *mixing parameter* μ is the fraction of edges between any node i and neighbors j outside the cluster of i (*external neighbors*). Finally $\bar{d}$ is the average degree of a node in the generated graph. The LFR model generates a graph with the following steps.

- The node degrees d_i are generated from the power-law distribution with parameter γ. The extremes of the distribution $d_{\min}$ and $d_{\max}$ are chosen so that the average degree is $\bar{d}$.
- The cluster sizes s_k are generated from a power-law distribution with parameter β. The extremes of the distribution $s_{\min}$ and $s_{\max}$ are chosen so that the sum of cluster sizes is n, and so that the size of the smallest cluster $s_{\min}$ (resp. the size of the largest cluster $s_{\max}$) is larger than the size of the smallest degree $d_{\min}$ (resp. the largest degree $d_{\max}$).
- The edges of the graph are generated with the configuration model in order to follow the degree sequence $(d_1, \ldots, d_n)$.
- Then, the model assign nodes to clusters as follows. Initially, nodes do not belong to any cluster. They are said to be *homeless*. Then, we iteratively try to assign each homeless node i to a cluster k chosen uniformly at random. If the size s_k of the chosen cluster is larger than the number of neighbors of i in this clusters (i.e. its number of *internal neighbors*), then we effectively assign i to the cluster k. Otherwise, the node i remains homeless. If the cluster k is complete (i.e. its size is equal to s_k), then we remove a randomy selected node before inserting i. This procedure naturally stops when there are no more homeless nodes.
- Note that the previous step generates clusters that follow the size distribution $(s_1, \ldots, s_K)$ but the mixing condition μ is not enforced. To do so, we perform rewiring operations similar to the ones performed by the configuration model, so that the node degrees remained unchanged and so that the fraction of internal neighbors is approximately $1 - \mu$.

In the different chapters of this document, we perform some experiments on synthetic graphs but we mainly focus on evaluating algorithms on real-life graphs as explained below.

1.5.2 Real-life graphs and ground-truth clusters

Synthetic graph models often fail to model the wide variety of graphs that are observed in real-life applications. For instance, graphs generated with the standard SBM have a degree distribution that is way different from what we typically observe in real-world networks (one could use the degree-corrected SBM, but the number of triangles in the generated graphs is inconsistent with what we commonly have in real-life datasets). For this reason, benchmarks on real-world graphs are essential to evaluate the performance of graph clustering algorithms. However, using a real-life graph G is often complex since we need to know what are the *real clusters* in G to compare them with the result of graph clustering algorithms. These *real clusters* are referred to as *ground-truth clusters*. To define these clusters, one typically choose a specific type of metadata on graph nodes that can be used as a proxy for the *real* cluster membership. For instance, in a graph between football players, the team of each player can be used to defined ground-truth clusters. However, we see that such a choice of metadata is somewhat arbitrary and that other choices could lead to different clusters [Peel et al., 2017]. In our football example, one could for instance choose the nationality of each player to define the ground-truth clusters. Most of the time, this issue is ignored and datasets where each graph G comes with a specific set of ground-truth clusters $\mathcal{C} = \{C_1, \ldots, C_K\}$ are used. One of the most widely-used source of such datasets is the Stanford Network Analysis Project[6] (SNAP) [Yang and Leskovec, 2015]. In order to compare the result of a graph clustering algorithm with ground-truth clusters, different score functions can be used, such as the Normalized Mutual Information or the F_1-score.

Normalized Mutual Information

The Normalized Mutual Information (NMI), which is a standard measure of dependence between random variables in information theory, can be used to measure the similarity between a ground-truth partition $\mathcal{C} = \{C_1, \ldots, C_K\}$ and the partition $\hat{\mathcal{C}} = \{\hat{C}_1, \ldots, \hat{C}_{K'}\}$ returned by a graph clustering algorithm. It is defined as

$$NMI(\mathcal{C}, \hat{\mathcal{C}}) = \frac{2I(\mathcal{C}, \hat{\mathcal{C}})}{H(\mathcal{C}) + H(\hat{\mathcal{C}})},$$

[6] `https://snap.stanford.edu/data/`

where $H(\mathcal{C})$ is the entropy of the clustering $\mathcal{C}$, i.e.

$$H(\mathcal{C}) = -\sum_{C \in \mathcal{C}} \frac{|C|}{n} \log \frac{|C|}{n},$$

and where $I(\mathcal{C}, \hat{\mathcal{C}})$ is the mutual information between $\mathcal{C}$ and $\hat{\mathcal{C}}$, i.e.

$$I(\mathcal{C}, \hat{\mathcal{C}}) = \sum_{C \in \mathcal{C}} \sum_{\hat{C} \in \hat{\mathcal{C}}} \frac{|C \cap \hat{C}|}{n} \log \left[\frac{|C \cap \hat{C}|/n}{\frac{|C|}{n} \frac{|\hat{C}|}{n}} \right] = H(\mathcal{C}) - H(\hat{\mathcal{C}}|\mathcal{C}).$$

It is easy to see that $NMI(\mathcal{C}, \hat{\mathcal{C}}) = 1$ if $\mathcal{C} = \hat{\mathcal{C}}$. Moreover, we have by Jensen's inequality $I(\mathcal{C}, \hat{\mathcal{C}}) \geq 0$, and thus $NMI(\mathcal{C}, \hat{\mathcal{C}}) \geq 0$.

The normalized mutual information is well-suited to compare node partitions. However, it cannot directly be applied to the evaluation of overlapping graph clustering algorithms [McDaid et al., 2011]. For this reason, the F_1-score presented below is often more convenient to evaluate the performance of general clustering algorithms.

F_1-Score

The F_1-score is a classic measure in statistics to measure the accuracy of a test. It is defined as the harmonic mean of *precision* and *recall*. In the case of graph clustering, the precision for a recovered cluster $\hat{C}$ with respect to a ground-truth cluster C is defined as

$$\text{precision}(\hat{C}, C) = \frac{|C \cap \hat{C}|}{|\hat{C}|},$$

and the recall is defined as

$$\text{recall}(\hat{C}, C) = \frac{|C \cap \hat{C}|}{|C|}.$$

Then, the F_1-score of the cluster $\hat{C}$ with respect to C is defined as

$$F_1(\hat{C}, C) = 2 \frac{\text{precision}(\hat{C}, C)\text{recall}(\hat{C}, C)}{\text{precision}(\hat{C}, C) + \text{recall}(\hat{C}, C)}.$$

It is easy to verify that the F_1-score $F_1(\hat{C}, C)$ takes values between 0 and 1 and that we have $F_1(\hat{C}, C) = 1$ if and only if we have perfect precision and recall, i.e. $\text{precision}(\hat{C}, C) = \text{recall}(\hat{C}, C) = 1$. Note that we have defined the F_1-score for a pair of clusters, but we still need to define the F_1-score between two sets of clusters, $\mathcal{C} = \{C_1, \ldots, C_K\}$ and $\hat{\mathcal{C}} = \{\hat{C}_1, \ldots, \hat{C}_{K'}\}$. A natural way to define such a score [Prat-Pérez et al., 2014] consists in first defining the F_1-score of a cluster $\hat{C}$ with respect to a set of clusters as

$$F_1(\hat{C}, \mathcal{C}) = \max_{C \in \mathcal{C}} F_1(\hat{C}, C),$$

and then defining the average F_1-score between two sets of clusters $\mathcal{C}$ and $\hat{\mathcal{C}}$ as

$$F_1(\mathcal{C}, \hat{\mathcal{C}}) = \frac{1}{2|\mathcal{C}|} \sum_{C \in \mathcal{C}} F_1(C, \hat{\mathcal{C}}) + \frac{1}{2|\hat{\mathcal{C}}|} \sum_{\hat{C} \in \hat{\mathcal{C}}} F_1(\hat{C}, \mathcal{C}).$$

We have $F_1(\mathcal{C}, \hat{\mathcal{C}}) \in [0, 1]$ and $F_1(\mathcal{C}, \hat{\mathcal{C}}) = 1$ if and only if the clusters of $\mathcal{C}$ and $\hat{\mathcal{C}}$ are identical, i.e. $\forall C \in \mathcal{C}$, $\exists \hat{C} \in \hat{\mathcal{C}}$, such that $C = \hat{C}$, and $\forall \hat{C} \in \hat{\mathcal{C}}$, $\exists C \in \mathcal{C}$, such that $\hat{C} = C$. Therefore, this measure is *normalized* and well-suited to compare different graph clustering algorithms, even if they return overlapping clusters. We widely use it in the experiments performed in the different chapters of this document.

Chapter 2

Soft clustering

In this chapter, we study soft graph clustering methods derived from the modularity score. In particular, we introduce an efficient algorithm based on a relaxation of the modularity optimization problem. A solution of this relaxation gives to each element of the dataset a probability to belong to a given cluster, whereas a solution of the standard modularity problem is a partition. The algorithm that we introduce is based on sparse matrices, and can therefore handle large graphs unlike existing approaches. Furthermore, we prove that our method includes, as a special case, the Louvain optimization scheme. Experiments on both synthetic and real-world data illustrate that our approach provides meaningful information on various types of data. This chapter is based on the work presented in [Hollocou et al., 2019].

2.1 Introduction

As stressed in Chapter 1, many classic graph clustering algorithms are focused on finding partitions of nodes. In such approaches, a node cannot belong to more than one cluster. Yet, in many real-life applications, it makes more sense to allow some elements to belong to multiple clusters. For instance, if we consider the problem of clustering the users of a social network such as Facebook or LinkedIn, we see that a person can belong to several social circles, such as her family, her colleagues and her friends, and we might be interested in recovering all these circles with a clustering algorithm, even if they do not form a partition. In this chapter, we study graph clustering methods that overcome this problem by determining for each node a *degree of membership* to each cluster, instead of returning a simple node partition. In particular, we are interested in techniques that aim at finding a probability $p_{ik} \in [0, 1]$ for each node i to belong to a given cluster k, whereas the node partioning methods implicitly consider that this probability can be either 0 or 1. The problem that these techniques try to solve is referred to as *soft clustering*, or *fuzzy clustering* [Dunn, 1973, Bezdek, 1981].

In this chapter, we introduce a relaxation of the classic modularity optimization problem and we study soft-clustering algorithms based on this relaxation. In particular, we introduce an efficient algorithm to find an approximation of this *soft* version of the modularity maximization problem. This algorithm has the following characteristics:

1. the number of clusters does not need to be specified;
2. the algorithm is local;
3. the solution found by the algorithm is sparse;
4. the algorithm includes the Louvain algorithm as a special case.

More precisely, property 2 indicates that each update of the membership information of a given node depends only on its direct neighbors in the graph, which implies a fast computation of these updates. Property 3 states that most of the membership probabilities p_{ik} returned by our algorithm are equal to 0, which guarantees an efficient storage of the solution. Finally, property 4 translates into the fact that an update performed by our algorithm reduces to an update performed by the Louvain algorithm as soon as the unique parameter of our algorithm is large enough.

The remainder of the chapter is organized as follows. In section 2.2, we introduce the concept of soft-clustering with the *soft* version of the k-means algorithm, called fuzzy c-means. We introduce the relaxation of the modularity maximization problem in section 2.3. In section 2.4, we present the related work on the topic of soft graph clustering. In section 2.5, we present our optimization method and obtain theoretical guarantees about its convergence. In section 2.6, we take benefits of the specificities of our optimization technique to propose an algorithm that is both local and memory efficient. In section 2.7, we study the ties of our approach to the Louvain algorithm. Finally, in section 2.9, we present experimental results on synthetic and real-world data.

2.2 Fuzzy c-means: an illustration of soft clustering

In order to introduce the concept of soft clustering, we present the *fuzzy c-means* algorithm that arises from a relaxation of the *k-means* problem. The fuzzy c-means algorithm is a soft clustering algorithm for observations that live in an Euclidean space. It gives to each observation a degree of membership to each cluster, whereas the Lloyd's algorithm [Lloyd, 1982] used to approximate the solution of the k-means problem simply returns a partition of these observations.

2.2.1 The k-means problem

The k-means clustering problem can be stated as follows. Given a set of observations $X = \{x_1, \dots, x_n\}$ in an Euclidean space (let us say $\mathbb{R}^K$), we want to find a partition of these observations into k subsets, $S = \{S_1, \dots, S_k\}$, and a set of k vectors of $\mathbb{R}^K$, $\mu = \{\mu_1, \dots, \mu_k\}$, that minimize the objective

$$J(S, \mu) = \sum_{l=1}^{k} \sum_{x \in S_l} \|x - \mu_l\|^2.$$

$k \in \mathbb{N}^*$ is given as a parameter of this problem. The vectors μ are referred to as the *means*. It is easy to see that, for a fixed partition S, the optimal vector μ_l corresponds to the *centroid* of the observations in S_l. Indeed, the gradient of the objective function with respect to μ_l is

$$\nabla_{\mu_l} J(S, \mu) = -2 \sum_{x \in S_l} (x - \mu_l).$$

Setting $\nabla_{\mu_l} J(S, \mu) = 0$, we get

$$\mu_l = \frac{1}{|S_l|} \sum_{x \in S_l} x.$$

If the vectors μ_l correspond to the centroids of the sets S_l, then the objective can be rewritten as

$$\begin{aligned} J(S, \mu) = \sum_{l=1}^{k} \sum_{x \in S_l} \left\| x - \frac{1}{|S_l|} \sum_{y \in S_l} y \right\|^2 &= \sum_{l=1}^{k} \left(\sum_{x \in S_l} x^T x - \frac{1}{|S_l|} \sum_{x, y \in S_l} (x^T y + y^T x) + \frac{1}{|S_l|^2} \sum_{x, y, y' \in S_l} y^T y' \right) \\ &= \sum_{l=1}^{k} \frac{1}{|S_l|} \sum_{x, y \in S_l} (x^T x - x^T y) \\ &= \sum_{l=1}^{k} \frac{1}{2|S_l|} \sum_{x, y \in S_l} \|x - y\|^2. \end{aligned}$$

Thus, when the means μ correspond to the centroids of the clusters S, minimizing the objective $J(S, \mu)$ is equivalent to minimizing the pairwise squared deviations of observations in the same cluster.

The k-means problem cannot be solved exactly in reasonable time as it has been proven to be NP-hard, event for two clusters and in a plane ($K = 2$) [Garey et al., 1982, Mahajan et al., 2009]. The classic heuristic that is used to find an approximation of the solution to this problem is the Lloyd's algorithm [Lloyd, 1982]. It is an iterative method that alternates between two steps:

- (Assignment step) Each observation is assigned to the cluster whose means is the nearest,

$$S_l = \{x : x \in X, \forall l', \|x - \mu_l\| \leq \|x - \mu_{l'}\|\}.$$

- (Mean update step) The means μ are chosen to be the centroids of the observations in the clusters

$$\mu_l = \sum_{x \in S_l} \frac{x}{|S_l|}.$$

Note that the initial means μ are generally picked at random. There exists many variants of the Lloyd's algorithm, that either use different distance function, such as the k-medoids algorithm [Kaufman and Rousseeuw, 1987], or specific initialization methods for the means, such as the k-means++ algorithm [Arthur and Vassilvitskii, 2007].

2.2.2 Fuzzy c-means

We can associate a binary matrix $p = (p_{il}) \in \mathbb{R}^{n \times k}$ to a partition $S = \{S_1, \ldots, S_k\}$ as follows:

$$\forall i \in [\![1, n]\!], \forall l \in [\![1, k]\!], \quad p_{il} = \begin{cases} 1 & \text{if } x_i \in S_l \\ 0 & \text{otherwise.} \end{cases}$$

We call such a matrix the *membership matrix* associated with the partition S. The k-means objective can be rewritten using this membership matrix:

$$J(S, \mu) = \sum_{i=1}^{n} \sum_{l=1}^{k} p_{il} \|x_i - \mu_l\|^2.$$

Note that a general matrix with non-negative integer coefficients, $p \in \mathbb{N}^{n \times k}$, is a membership matrix for some partition S if and only if

$$\forall i \in [\![1, n]\!], \quad \sum_{l=1}^{k} p_{il} = 1.$$

Indeed, as the coefficients can only take non-negative integer values, it is easy to see that this equation is equivalent to say that, for each i, there is one and only one l such that $p_{il} = 1$ (and the other coefficients are $= 0$). Using this result, we can rewrite the k-means problem as follows

$$\begin{aligned} \underset{p \in \mathbb{N}^{n \times k}}{\text{minimize}} \quad & \sum_{i=1}^{n} \sum_{l=1}^{k} p_{il} \|x_i - \mu_l\|^2 \\ \text{subject to} \quad & \forall i \in [\![1, n]\!], \sum_{l=1}^{k} p_{il} = 1. \end{aligned} \tag{2.1}$$

The idead behind the fuzzy c-means algorithm [Dunn, 1973, Bezdek, 1981] consists in considering a relaxation of the problem (2.1) where the coefficients of p can take real values, i.e.

$$\begin{aligned} \underset{p \in \mathbb{R}^{n \times k}}{\text{minimize}} \quad & J(p, \mu) = \sum_{i=1}^{n} \sum_{l=1}^{k} p_{il} \|x_i - \mu_l\|^2 \\ \text{subject to} \quad & \forall i \in [\![1, n]\!], \sum_{l=1}^{k} p_{il} = 1. \end{aligned}$$

In this relaxation, p_{il} can now be interpreted as the *degree of membership* of the observation x_i to the cluster S_l. The objective function can further generalized by introducing a new parameter m

$$J(p, \mu) = \sum_{i=1}^{n} \sum_{l=1}^{k} p_{il}^m \|x_i - \mu_l\|^2.$$

We will see that $m \in [1, \infty)$ is a parameter that is used to control the *fuzziness* of the result. The Lagrange function associated with this problem is

$$\mathcal{L}(p, \mu, \lambda) = \sum_{i=1}^{n} \sum_{l=1}^{k} p_{il}^m \|x_i - \mu_l\|^2 - \sum_{i=1}^{n} \lambda_i \left[\sum_{l=1}^{k} p_{il} - 1 \right],$$

where the λ_i are the Lagrange multipliers associated with the constraints $\sum_l p_{il} = 1$. The stationary points of the objective function $J(p, \mu)$ can be found by setting the gradients of $\mathcal{L}$ with respect to p, μ and λ to zero. We get

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial p_{il}} &= m p_{il}^{m-1} \|x_i - \mu_l\|^2 - \lambda_i = 0 \\
\frac{\partial \mathcal{L}}{\partial \lambda_i} &= \sum_{l=1}^{k} p_{il} - 1 = 0 \\
\nabla_{\mu_l} \mathcal{L} &= -\sum_{i=1}^{n} 2 p_{il}^m (x_i - \mu_l) = 0.
\end{aligned}$$

From the last of these equations, we can get an expression of the mean μ_l in function of x and p:

$$\mu_l = \frac{\sum_i p_{il}^m x_i}{\sum_i p_{il}^m}. \tag{2.2}$$

Note that if the coefficients of p are in $\{0, 1\}$, then we have the same equation as in the k-means problem and μ_l corresponds to the centroid of the observations of the lth cluster. In the general case where the coefficients p_{il} take real values, μ_l corresponds to the *weighted centroid* of the lth cluster using the weights p_{il}^m.

Combining the other two stationary equations, we can get rid of the Lagrange multipliers λ_i and obtain an expression of p_{il} in function of x and μ. On the one hand, we have

$$p_{il} = \left(\frac{\lambda_i}{m \|x_i - \mu_l\|^2} \right)^{\frac{1}{m-1}}. \tag{2.3}$$

On the other hand, we have $\sum_{l=1}^{k} p_{il} = 1$. Using the expression (2.3) for p_{il}, this gives us

$$\sum_{l=1}^{k} \left(\frac{\lambda_i}{m \|x_i - \mu_l\|^2} \right)^{\frac{1}{m-1}} = 1 \Leftrightarrow \lambda_i^{1/(m-1)} = \left(\sum_{l=1}^{k} \frac{1}{m \|x_i - \mu_l\|^{\frac{2}{m-1}}} \right)^{-1}.$$

Finally, we can replace the value of $\lambda_i^{1/(m-1)}$ in (2.3) by the expression that we have just obtained. This gives us

$$p_{il} = \left(\sum_{l'=1}^{k} \left(\frac{\|x_i - \mu_l\|}{\|x_i - \mu_{l'}\|} \right)^{\frac{2}{m-1}} \right)^{-1}. \tag{2.4}$$

The fuzzy c-means algorithm alternatively uses the equations (2.4) and (2.2) to update the means and the membership matrix. More precisely, it can be decomposed into 2 steps that are repeated until convergence:

- (Assignment step) The membership matrix is updated with the formula (2.4).
- (Mean update step) The means are updated with the formula (2.2).

This algorithm converges to a local minimum or a saddle point of the objective function $J(p, \mu)$. Note that the coefficients p_{il} obtained with this method are in $[0, 1]$ and verify $\forall i$, $\sum_l p_{il} = 1$. Therefore, p_{il} can be interpreted as the probability for the observation x_i to belong to the lth cluster.

Now we study the impact of the parameter m on the fuzziness of the result by looking at the limit cases $m \to 1$ and $m \to \infty$. We start with the case $m \to 1$. We have

$$\lim_{m \to 1} \left(\frac{\|x_i - \mu_l\|}{\|x_i - \mu_{l'}\|} \right)^{\frac{2}{m-1}} = \begin{cases} +\infty & \text{if } \|x_i - \mu_l\| > \|x_i - \mu_{l'}\|; \\ 1 & \text{if } \|x_i - \mu_l\| = \|x_i - \mu_{l'}\|; \\ 0 & \text{if } \|x_i - \mu_l\| < \|x_i - \mu_{l'}\|. \end{cases}$$

Knowing that there exists a $l' \in [\![1, k]\!]$ such that $\|x_i - \mu_l\| > \|x_i - \mu_{l'}\|$ if and only if $l \neq \arg\min_{l'} \|x_i - \mu'_l\|$, we have

$$\lim_{m \to 1} \sum_{l'=1}^{k} \left(\frac{\|x_i - \mu_l\|}{\|x_i - \mu_{l'}\|} \right)^{\frac{2}{m-1}} = \begin{cases} 1 & \text{if } l = \arg\min_{l'} \|x_i - \mu'_l\| \\ +\infty & \text{otherwise.} \end{cases}$$

This leads to $\lim_{m \to 1} p_{il} = 1$ if $l = \arg\min_{l'} \|x_i - \mu'_l\|$, and $\lim_{m \to 1} p_{il} = 0$ otherwise. We see that in this limit case, we have a *crisp* partitioning that corresponds to the assignment made by Lloyd's algorithm, i.e. each observation is assigned to the nearest cluster. In this limit, the update made in the second step of the fuzzy c-means algorithm consists in taking the centroid of each cluster, which also corresponds to the update performed by Lloyd's approach.

In the limit $m \to +\infty$, we have

$$\lim_{m \to \infty} \left(\frac{\|x_i - \mu_l\|}{\|x_i - \mu_{l'}\|} \right)^{\frac{2}{m-1}} = 1,$$

which gives us

$$\lim_{m \to \infty} p_{il} = \lim_{m \to \infty} \left[\sum_{l'=1}^{k} \left(\frac{\|x_i - \mu_l\|}{\|x_i - \mu_{l'}\|} \right)^{\frac{2}{m-1}} \right]^{-1} = \frac{1}{k}.$$

Therefore, in the limit $m \to \infty$, the soft clustering obtained corresponds to the situation of *maximum fuzziness*, where each observation x_i has an uniform probability to belong to each cluster.

2.3 Soft modularity

Going back to the problem of graph clustering, we will now see how the problem of modularity maximization can be relaxed into a soft clustering problem. The problem of modularity maximization introduced in Chapter 1 can be written as

$$\underset{P \in \mathcal{P}}{\text{maximize}} \quad Q(P) = \frac{1}{w} \sum_{C \in P} \sum_{i,j \in C} \left(A_{ij} - \frac{w_i w_j}{w} \right), \tag{2.5}$$

where $\mathcal{P}$ is the set of all partitions of V. The modularity function $Q(P)$ can be reformulated using the notation $C(i)$ for the cluster of P to which the node i belongs, and with the Kronecker delta δ:

$$Q(P) = \frac{1}{w} \sum_{i,j \in V} \left(A_{ij} - \frac{w_i w_j}{w} \right) \delta_{C(i)C(j)}.$$

As described in the previous section, we can introduce the *membership matrix* $p \in \{0, 1\}^{n \times n}$ associated with the partition $P = \{C_1, \ldots, C_K\}$ defined as

$$\forall i \in V, \forall k \in [\![1, n]\!], \quad p_{ik} = \begin{cases} 1 & \text{if } i \in C_k \\ 0 & \text{otherwise.} \end{cases}$$

Note that the size of such a membership matrix is $n \times n$ and not $n \times k$ as in the analysis of the k-means algorithm. Indeed, we no longer bound the maximum number of clusters (but there can be at most n clusters). The coefficient $\delta_{C(i)C(j)}$ can be expressed in function of the coefficients of the membership matrix associated with P,

$$\delta_{C(i)C(j)} = \sum_{k=1}^{n} p_{ik} p_{jk}.$$

Therefore, it is easy to see that P is an optimal solution of the modularity maximization problem if and only if its associated membership matrix p is an optimal solution of

$$\begin{aligned} \underset{p \in \mathbb{Z}^{n \times n}}{\text{maximize}} \quad & \frac{1}{w} \sum_{i,j \in V} \sum_{k=0}^{n} \left(A_{ij} - \frac{w_i w_j}{w} \right) p_{ik} p_{jk} \\ \text{subject to} \quad & \forall i \in V, \sum_{k=1}^{n} p_{ik} = 1 \\ & \forall i \in V, \forall k \in [\![1, n]\!], p_{ik} \geq 0. \end{aligned}$$

Given a membership matrix p, we use $p_{i\cdot}$ to denote the vector that corresponds to the ith line of p and $p_{\cdot k}$ to denote the vector that corresponds to its kth column. Using this notation, we have the following simple expression for the coefficient $\delta_{C(i)C(j)}$

$$\delta_{C(i)C(j)} = p_{i\cdot}^T p_{j\cdot},$$

and the problem can be rewritten

$$\begin{aligned} \underset{p \in \mathbb{Z}^{n\times n}}{\text{maximize}} \quad & \frac{1}{w} \sum_{i,j \in V} \left(A_{ij} - \frac{w_i w_j}{w} \right) {p_{i\cdot}}^T p_{j\cdot} \\ \text{subject to} \quad & \forall i \in V, 1^T p_{i\cdot} = 1 \\ & \forall i \in V, p_{i\cdot} \geq 0. \end{aligned} \tag{2.6}$$

In a similar way to the fuzzy c-means approach described above, we can consider a relaxation of the modularity optimization problem (2.6) where the coefficient of the membership matrix p can take real values:

$$\begin{aligned} \underset{p \in \mathbb{R}^{n\times n}}{\text{maximize}} \quad & \frac{1}{w} \sum_{i,j \in V} \left(A_{ij} - \frac{w_i w_j}{w} \right) {p_{i\cdot}}^T p_{j\cdot} \\ \text{subject to} \quad & \forall i \in V, 1^T p_{i\cdot} = 1 \\ & \forall i \in V, p_{i\cdot} \geq 0. \end{aligned} \tag{2.7}$$

Note that if p is a feasible solution of this problem, then the vector $p_{i\cdot}$ can be interpreted as a probability distribution over clusters, as we have $p_{i\cdot} \geq 0$ and $1^T p_{i\cdot} = 1$. Therefore, p_{ik} corresponds to the probability for a node i to belong to cluster k. Nodes can have a positive degree of membership for multiple clusters, unlike in the classical modularity maximization problem. We use $Q(p)$ to refer to this new objective function, that we call *soft modularity*:

$$Q(p) = \frac{1}{w} \sum_{i,j \in V} \left(A_{ij} - \frac{w_i w_j}{w} \right) {p_{i\cdot}}^T p_{j\cdot}. \tag{2.8}$$

The soft modularity can also be written in function of the columns of the membership matrix p,

$$Q(p) = \frac{1}{w} \sum_{k=1}^{n} p_{\cdot k}^T \left(A - \frac{\boldsymbol{w}\boldsymbol{w}^T}{w} \right) p_{\cdot k}, \tag{2.9}$$

where $\boldsymbol{w}$ denotes the vector whose coefficients are the weighted degrees w_i.

In the rest of this chapter, we study how to solve the *soft modularity maximization* problem.

2.4 Related work

Several methods have been proposed to solve the soft modularity maximization problem described above or variants of this problem [Nicosia et al., 2009, Griechisch and Pluhár, 2011, Havens et al., 2013, Chang and Chang, 2017, Kawase et al., 2016]. In this section, we study in detail the softmax algorithm introduced in [Chang and Chang, 2017]. We also present the characteristics of the other existing approaches without giving a detailed definition of each of these algorithms.

2.4.1 The softmax approach

Here, we present the approach introduced in [Chang and Chang, 2017] from a slightly different angle than the original paper. The optimization problem considered here is a slight variant of the relaxation introduced above where the number of clusters is limited to K. In other words, the membership matrix considered here has size $n \times K$, and the corresponding problem can be stated as

$$\begin{aligned} \underset{p \in \mathbb{R}^{n\times K}}{\text{maximize}} \quad & \frac{1}{w} \sum_{i,j \in V} \left(A_{ij} - \frac{w_i w_j}{w} \right) {p_{i\cdot}}^T p_{j\cdot} \\ \text{subject to} \quad & \forall i \in V, 1^T p_{i\cdot} = 1 \\ & \forall i \in V, p_{i\cdot} \geq 0. \end{aligned} \tag{2.10}$$

The approach of Chang et al. is based on the *softmax* function defined as follows.

Definition 2.4.1 (Softmax)**.** *We define the* softmax *function* $\sigma : \mathbb{R}^{n\times K} \to (0,\infty)^{n\times K}$ *as follows,*

$$\forall q \in \mathbb{R}^{n\times K}, \forall i \in [\![1,n]\!], \forall k \in [\![1,K]\!] \quad [\sigma(q)]_{ik} = \frac{\exp(q_{ik})}{\sum_{l=1}^{K}\exp(q_{il})}.$$

Note that, for all matrix $q \in \mathbb{R}^{n\times K}$, the softmax function maps q to a matrix $\sigma(q)$ that verifies the constraints $\forall i$, $1^T\sigma(q)_{i\cdot} = 1$ and $\sigma(q) \geq 0$. In particular, the lines of $\sigma(q)$ can be interpreted as probability distributions over the clusters. However, note that the image of $\mathbb{R}^{n\times K}$ under σ is not the set of constraints of (2.10), i.e. the set of matrices $p \in \mathbb{R}^{n\times K}$ s.t. $\forall i$, $1^T p_{i\cdot} = 1$ and $p \geq 0$. Indeed, we have $\forall q$, $\sigma(q) > 0$. Nevertheless, we can obtain matrices $\sigma(q)$ with coefficients equal to zero in the limit where the coefficients of q tend to $\pm\infty$. In other word, the image of the closure of $\mathbb{R}^{n\times K}$ under σ is the set of constraints of (2.10).

Let us study some properties of the softmax function. We see that if $q' \in \mathbb{R}^{n\times K}$ verifies $\forall i$, $q'_{ik} = q_{ik} + c_i$ with $c \in \mathbb{R}^n$, then $\sigma(q') = \sigma(q)$. In particular, σ is not *injective*. However, we have the following lemma that characterizes the q such that $\sigma(q) = p$ for a given p.

Lemma 2.4.2. *For all* $p \in \mathbb{R}^{n\times K}$ *such that* $\forall i$, $1^T p_{i\cdot} = 1$ *and* $p_{i\cdot} > 0$*, we have*

- $\sigma(q) = p$ *for* $q \in \mathbb{R}^{n\times K}$ *defined as*

$$\forall i,k, \quad q_{ik} = \log(p_{ik});$$

- $\forall q'$ *s.t.* $\sigma(q') = p$*, there exists* $c \in \mathbb{R}^n$ *s.t.*

$$\forall i,k, q'_{ik} = q_{ik} + c_i.$$

Proof. It is easy to verify that $\sigma(q) = p$. If q' verifies $\sigma(q') = p = \sigma(q)$, then, taking the logarithm, we get

$$q'_{ik} = q_{ik} + \log\left(\frac{\sum_l \exp(q'_{il})}{\sum_l \exp(q_{il})}\right).$$

□

The softmax approach of the soft modularity maximization problem consists in using the softmax function to parameterized the solution p. In other words, we set $p = \sigma(q)$ and we study the objective function

$$Q(\sigma(q)) = \frac{1}{w}\sum_{i,j}\left(A_{ij} - \frac{w_i w_j}{w}\right)\sigma(q)^T_{i\cdot}\sigma(q)_{j\cdot}.$$

We want to find an updated rule for q for which the objective function $q \mapsto Q(\sigma(q))$ is non-decreasing. In order to simplify the following calculations, we use B to denote the matrix $B = \frac{1}{w}\left(A - \frac{\boldsymbol{w}\boldsymbol{w}^T}{w}\right)$, so that $Q(\sigma(q)) = \sum_{ij} B_{ij}\sigma(q)^T_{i\cdot}\sigma(q)_{j\cdot}$. In their approach, Chang et al. consider the case where $\forall i, B_{ii} = 0$. Note that it simply corresponds to a particular normalization of the diagonal coefficients of A. They also consider that B is symmetric, i.e. that the graph G is undirected. We use these hypotheses in the following.

We consider variations Δq of the variable q that only impact the i_0^{th} row of q, for some i_0, i.e. we have $\Delta q_i = 0$ for all $i \neq i_0$. The variation of modularity ΔQ for such a variation Δq of q can be written

$$\Delta Q = Q(\sigma(q+\Delta q)) - Q(\sigma(q)) = 2\left(\sigma(q+\Delta q)_{i_0\cdot} - \sigma(q)_{i_0\cdot}\right)^T \sum_j B_{i_0 j}\sigma(q)_{j\cdot} \tag{2.11}$$

$\sigma(q+\Delta q)$ can be written

$$\forall i,k, \quad \sigma(q+\Delta q)_{ik} = \frac{\sigma(q)_{ik}e^{\Delta q_{ik}}}{\sum_l \sigma(q)_{il}e^{\Delta q_{il}}}.$$

We use $x \in \mathbb{R}^K$ to denote the vector $\sigma(q)_{i_0\cdot}$ and $y \in \mathbb{R}^n$ the vector defined as $y = \sum_j B_{i_0 j}\sigma(q)_{jk}$. The equation (2.11) becomes

$$\Delta Q = 2\sum_k \left[\frac{x_k e^{\Delta q_{i_0 k}}}{\sum_l x_l e^{\Delta q_{i_0 l}}} - x_k\right] y_k.$$

We see that if we take the variation Δq defined as $\Delta q_{i_0\cdot} = \theta y$ and $\forall i \neq i_0$, $\Delta q_i = 0$ for some $\theta \in \mathbb{R}$, then we have

$$\Delta Q = 2\sum_k \left[\frac{x_k y_k e^{\theta y_k}}{\sum_l x_l e^{\theta y_l}} - \frac{x_k}{\sum_l x_l}\right],$$

where we used $\sum_l x_l = 1$. Now, if we introduce the function $f : \theta \mapsto \frac{\sum_k x_k y_k e^{\theta y_k}}{\sum_k x_k e^{\theta y_k}}$, then we have

$$\Delta Q = 2(f(\theta) - f(0)).$$

Note that f is the derivative of $F : \theta \mapsto \log\left(\sum_k x_k e^{\theta y_k}\right)$. The derivative of f is

$$f'(\theta) = \frac{\sum_k x_k y_k^2 e^{\theta y_k} \sum_k x_k e^{\theta y_k} - \left(\sum_k x_k y_k e^{\theta y_k}\right)^2}{\left(\sum_k x_k e^{\theta y_k}\right)^2}.$$

The Cauchy-Schwartz inequality $\left(\sum_k a_k b_k\right)^2 \leq \sum_k a_k^2 \sum_k b_k^2$ applied with $a_k = \sqrt{x_k} y_k e^{\theta y_k/2}$ and $b_k = \sqrt{x_k} e^{\theta y_k/2}$ gives us $f'(\theta) \geq 0$, with equality if and only if the vectors a and b are proportional, i.e. if and only if y_k is constant for all k. Finally, this yields to the following Lemma.

Lemma 2.4.3. *For all i_0, for a variation Δq of q such that $\Delta q_{i_0\cdot} = \theta \sum_j B_{i_0\cdot}\sigma(q)_{j\cdot}$ with $\theta > 0$, and $\Delta q_{i\cdot} = 0$ for $i \neq i_0$, we have*

$$\Delta Q = Q(\sigma(q + \Delta q)) - Q(\sigma(q)) \geq 0 \tag{2.12}$$

with equality if and only if $\sum_j B_{i_0 j}\sigma(q)_{j\cdot}$ is constant for all k.

This leads to the following algorithm.

1. (Initialization) Take an initial matrix q such that each row $q_{i\cdot}$ is not a constant vector.
2. (Main loop) At each step, for each node $i \in V$, we do

$$q_{i\cdot} \leftarrow q_{i\cdot} + \theta \sum_j B_{ij}\sigma(q)_{j\cdot}.$$

3. (Output) We return $p = \sigma(q)$.

$\theta > 0$ is given as a parameter of the algorithm. We can stop the algorithm when the modularity $Q(\sigma(q))$ does not significantly increase anymore. The analysis performed above shows that $Q(\sigma(q))$ is non-decreasing under each update. The choice of the initialization made in the algorithm is due to the fact that, if the ith row of q is a constant vector, then we will have an equality in (2.12) (i.e. $\Delta Q = 0$) when we update the ith row.

The main limitation of this softmax approach is that the membership matrix p computed by the algorithm is the densest possible matrix i.e. none of its coefficients are equal to zero. As a consequence, we need to store $n \times K$ coefficients which can be very expensive in term of memory when the number of nodes n or the number of clusters K increase. In most of real-life graphs, for a given node, we expect the degree of membership to most of the clusters to be equal to zero, as a node will generally be *connected* to only a small number of clusters. In other words, we expect the membership matrix p to be sparse, which is not the case in this softmax approach.

2.4.2 Other approaches

The relaxation of the modularity optimization problem and the optimization of the soft modularity (2.8) (or slight variants of this objective function) have been tackled by other articles [Nicosia et al., 2009, Griechisch and Pluhár, 2011, Havens et al., 2013, Kawase et al., 2016]. Like for the softmax approach,

the maximum number of clusters K must be specified for all these approaches. They all rely on membership matrices $p \in \mathbb{R}^{n \times K}$, with generally $K \ll n$.

These works differ in the optimization techniques they use to solve the problem. In [Nicosia et al., 2009], the authors use a genetic algorithm to optimize a more general relaxation of the modularity problem. In [Griechisch and Pluhár, 2011], the authors do not directly study the optimization of the relaxation problem, but they rely on an external quadratic solver. In [Havens et al., 2013], the authors use a spectral approach that does not directly solve the relaxation of the modularity problem, but where modularity is used as a selection criterion. The main limitation of these methods lies in the maximum number of clusters K that must be specified. We could get around this issue by taking large values for K, but all the approaches cited above do not scale well to large K. Indeed, like for the softmax approach [Chang and Chang, 2017] described above, the solutions $p \in \mathbb{R}^{n \times K}$ found by these methods are dense matrices. In other words, the number of parameters to store in memory and to optimize is in $O(nK)$, which quickly becomes prohibitive for large values of K. For instance, in the approach of [Nicosia et al., 2009], the genetic algorithm starts with dense random matrices of $\mathbb{R}^{n \times K}$, and its hybridation and mutation mechanisms do not lead to sparser matrices, so that all the coefficients of the solution p found by this algorithm are positive with probability 1.

In this chapter, we introduce an efficient algorithm to solve a relaxation of the modularity optimization problem with a membership matrix $p \in \mathbb{R}^{n \times n}$. Thus, the maximum number of clusters does not need to be specified. Besides, unlike the approaches presented above, the updates performed by our algorithm preserve the sparsity of the solution p and are *local*, in the sense that the membership vector $p_{i\cdot}$ of a node $i \in V$ is updated using only membership information from its neighbors in the graph. Thus, our algorithm can easily scale up to large datasets with large number of clusters, which is not the case of the algorithms listed above.

As proposed in [Whang et al., 2015], the problem of soft graph clustering can also be tackled by a combination of spectral embedding [Von Luxburg, 2007, Fowlkes et al., 2004] and soft-clustering in the vector space, with algorithms such as the fuzzy C-means algorithm [Dunn, 1973] or the NEO k-means algorithm [Whang et al., 2015]. However, the number of clusters K has to be specified for these techniques. Besides, spectral embedding methods do not scale well to large size graphs [Perozzi et al., 2014]. For instance, they are unable to handle the subgraph of Wikipedia that we use in our experiments in Section 2.9. Finally, it is worth mentioning that several algorithms have been proposed for the related problem of overlapping community detection in networks [Lancichinetti et al., 2011, Yang and Leskovec, 2013]. Whereas these approaches return clusters $C \subset V$, soft-clustering methods give for each node the degrees of membership to all clusters.

2.5 Alternating projected gradient descent

We can rewrite the relaxation of the modularity optimization problem as a classic minimization problem $\min_{p \in \mathcal{X}} J(p)$, where $J : p \mapsto -Q(p)$ is the cost function and $\mathcal{X} = \{p \in \mathbb{R}^{n \times n} : \forall i, p_{i\cdot} \geq 0, 1^T p_{i\cdot} = 1\}$ is the set of constraints.

As seen in Chapter 1, the normalized weighted Laplacian of the graph is defined as $\mathcal{L} = I - D^{-1/2} A D^{-1/2}$, where D is the diagonal matrix whose entries are the weighted degrees of the nodes: $D = \mathrm{diag}(w_1, \ldots, w_n)$. We have seen that the Laplacian matrix is symmetric and semi-definite positive [Chung, 1997]. We use $0 \leq \lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n$ to denote its eigenvalues.

Theorem 2.5.1. *The function J is convex if and only if the second lowest eigenvalue λ_2 of the normalized Laplacian $\mathcal{L}$ verifies $\lambda_2 \geq 1$.*

Proof. The cost function J can be written as $J(p) = \frac{1}{w} \sum_k \tilde{J}(p_{\cdot k})$ where $\tilde{J}(q) = \sum_i \sum_j (A_{ij} - w_i w_j / w) q_i q_j$. So J is convex if and only if $\tilde{J}$ is convex. Moreover, the function $\tilde{J}$ is convex iff its hessian matrix H is semi-definite positive. An expression of H is given by $H = -2\left(A - \frac{\boldsymbol{w}\boldsymbol{w}^T}{w}\right)$, where $\boldsymbol{w}$ denotes the vector $(w_i)_{i \in V}$. We see that H is closely related to the modularity matrix introduced in Chapter 1.
Let x be a vector of $\mathbb{R}^n$, and $y = D^{1/2} x$. Then we have:

$$\begin{aligned} x^T H x &= -2\left(y^T D^{-1/2} A D^{-1/2} y - \|u_1^T y\|^2\right) \\ &= 2(y^T(\mathcal{L} - I)y + \|u_1^T y\|^2) \end{aligned}$$

where $u_1 = (1/\sqrt{w})D^{-1/2}\boldsymbol{w}$. It is easy to verify that u_1 is a normalized eigenvector associated with the first eigenvalue $\lambda_1 = 0$ of the normalized Laplacian $\mathcal{L}$. We use $u_1, \ldots, u_n$ to denote the orthonormal basis of eigenvectors corresponding to the eigenvalues $(\lambda_1, \ldots, \lambda_n)$. Thus, we can write:

$$\begin{aligned} x^T H x &= 2(\sum_{k=1}^{n}(\lambda_k - 1)\|u_k^T y\|^2 + \|u_1^T y\|^2) \\ &= 2(\sum_{k=2}^{n}(\lambda_k - 1)\|u_k^T y\|^2) \end{aligned}$$

We see that $x^T H x \geq 0$ for all x if and only if $1 \geq \lambda_2 \geq \ldots \geq \lambda_n$, which concludes the proof. □

The condition $\lambda_2 \geq 1$, which corresponds to a large spectral gap, is in general not satisfied. For instance, Lemma 1.7. in [Chung, 1997] proves that $\lambda_2 \leq 1$ if G is unweighted and not complete. Therefore, the loss function associated with our problem is in general non-convex. However, it is convex in $p_{i\cdot}$ for a graph with no self-loop as shown in the following proposition. In the rest of the chapter, we assume that the graph G does not contain any self-loop.

Proposition 2.5.2. *If the graph does not contain any self loops, i.e. if $A_{ii} = 0$ for all node i, the function $p \mapsto J(p)$ is convex with respect to variable $p_{i\cdot}$ for all $i \in V$.*

Proof. Let i be a given node. We have for all $k, k' \in [\![1, n]\!]$,

$$\begin{aligned} \frac{\partial J}{\partial p_{ik}} &= -\frac{2}{w}\sum_{j\in V}(A_{ij} - w_i w_j/w)p_{jk}, \\ \frac{\partial J}{\partial p_{ik}\partial p_{ik'}} &= \frac{2}{w}(w_i^2/w - A_{ii})\delta(k, k'), \end{aligned}$$

where $\delta(k, k') = 1$ if $k = k'$ and 0 otherwise. If $A_{ii} = 0$, the hessian matrix H_i of J with respect to $p_{i\cdot}$ can be written $H_i = 2(w_i/w)^2 I$. It is thus definite positive, which proves the convexity of J in $p_{i\cdot}$. □

With this result, we can apply the methods of convex optimization with convex constraints to the problem $\min_{p_{i\cdot}\in\mathcal{Y}} J(p)$, with fixed $p_{j\cdot}$ for $j \neq i$, where the set $\mathcal{Y}$ corresponds to the probability simplex of $\mathbb{R}^n$:

$$\mathcal{Y} = \{q \in \mathbb{R}^n : q \geq 0, 1^T q = 1\}.$$

We use $\pi_{\mathcal{Y}}$ to denote the euclidean projection onto the probability simplex $\mathcal{Y}$,

$$\pi_{\mathcal{Y}}(x) = \arg\min_{y\in\mathcal{Y}}\|x - y\|^2.$$

Then, using the projected gradient descent method [Goldstein, 1964, Levitin and Polyak, 1966], we can define an update rule for variable $p_{i\cdot}$ such that modularity is non-decreasing at each step. The gradient of J with respect to $p_{i\cdot}$ is:

$$\nabla_i J = -\frac{2}{w}\sum_{j\in V}(A_{ij} - w_i w_j/w)p_{j\cdot}.$$

Theorem 2.5.3. *The soft modularity objective function $Q(p)$ is non-decreasing under the update rule*

$$p_{i\cdot} \leftarrow \pi_{\mathcal{Y}}\left(p_{i\cdot} + \frac{2t}{w}\sum_{j\in V}\left(A_{ij} - \frac{w_i w_j}{w}\right)p_{j\cdot}\right) \tag{2.13}$$

for all node $i \in V$ if the step size t verifies $t < (w/w_i)^2$.
$Q(p)$ is invariant under all these update rules iff $p_{i\cdot}$ maximizes $Q(p)$ with fixed $p_{j\cdot}$, $j \neq i$, for all node $i \in V$.

Proof. Let i be a given node of V and $p \in \mathcal{X}$ be a feasible solution of the relaxation of the soft modularity optimization problem. We define p^+ with $p^+_{i\cdot} = \pi_{\mathcal{Y}}\left[p_{i\cdot} + \frac{2t}{w}\sum_{j\in V}\left(A_{ij} - \frac{w_i w_j}{w}\right)p_{j\cdot}\right]$, and $p^+_{j\cdot} = p_{j\cdot}$ for all $j \neq i$.

The objective function $J(p) = -Q(p)$ is quadratic in p_{ik}, $k \in [\![1, n]\!]$, so we have:

$$\begin{aligned} J(p^+) &= J(p) + \nabla_i J(p)^T({p_{i\cdot}}^+ - p_{i\cdot}) + \frac{1}{2}({p_{i\cdot}}^+ - p_{i\cdot})H_i({p_{i\cdot}}^+ - p_{i\cdot}) \\ &= J(p) - t\nabla_i J(p)^T G_i(p) + \left(\frac{w_i t}{w}\right)^2 \|G_i(p)\|^2 \end{aligned} \tag{2.14}$$

where $\nabla_i J(p)$ is the gradient of J with respect to $p_{i\cdot}$, H_i is the hessian matrix of J with respect to $p_{i\cdot}$ whose expression was given in the proof of Proposition 2.5.2, and $G_i(p) = (p_{i\cdot} - {p_{i\cdot}}^+)/t$.

Besides, by definition of $\pi_{\mathcal{Y}}$, and using the expression of $\nabla_i J(p)$ given in the proof of Proposition 2.5.2, we have

$$\begin{aligned} p^+_{i\cdot} &= \arg\min_{q\in\mathcal{Y}} \left\{\|q - (p_{i\cdot} - t\nabla_i J(p))\|^2\right\} \\ &= \arg\min_{q\in\mathcal{Y}} \left\{2t\nabla_i J(p)^T(q - p_{i\cdot}) + \|q - p_{i\cdot}\|^2\right\} \\ &= \arg\min_{q\in\mathbb{R}^n} \left\{\nabla_i J(p)^T(q - p_{i\cdot}) + \frac{\|q - p_{i\cdot}\|^2}{2t} + h(q)\right\} \end{aligned} \tag{2.15}$$

where $h(q) = 0$ if $q \in \mathcal{Y}$, and $h(q) = +\infty$ otherwise.

$\mathcal{Y}$ is a convex set, so h is a convex function. Note that h is non-differentiable. We use $\partial h(q)$ to denote the subdifferential of h at q i.e. the set of all its subgradients. Remember that a vector v is defined as a subgradient of h at q if it verifies, for all q', $h(q') - h(q) \geq v^T(q' - q)$.

Equation (2.15) can be written $p^+_i = \arg\min_q L(q)$, where L is a non-differentiable convex function. The optimality of $p^+_{i\cdot}$ gives us $0 \in \partial L(p^+_{i\cdot})$. Therefore, there exists a $v \in \partial h(p^+_{i\cdot})$, such that

$$\nabla_i J(p) + \frac{1}{t}({p_{i\cdot}}^+ - p_{i\cdot}) + v = 0.$$

Using this result in equation (2.14), we obtain

$$J(p^+) = J(p) + tv^T G_i(p) - t\left(1 - t\left(\frac{w_i}{w}\right)^2\right)\|G_i(p)\|^2.$$

We have $tv^T G_i(p) = v^T(p_{i\cdot} - {p_{i\cdot}}^+) \leq h(p_{i\cdot}) - h({p_{i\cdot}}^+)$ because $v \in \partial h(p^+_{i\cdot})$. Both $p_{i\cdot}$ and $p^+_{i\cdot}$ belongs to $\mathcal{Y}$, thus $v^T G_i(p) \leq 0$.

Finally, we obtain

$$J(p^+) \leq J(p) - t\left(1 - t\left(\frac{w_i}{w}\right)^2\right)\|G_i(p)\|^2.$$

We have $Q(p^+) \geq Q(p)$ if $t < (w/w_i)^2$. The inequality becomes an equality if and only if $G_i(p) = 0$, which gives us $\nabla_i J(p) + v = 0$. This is equivalent to say that p is an optimum of $J + h$ with respect to $p_{i\cdot}$. □

We now consider the natural algorithm that cycles through the nodes of the graph to apply the update (2.13).

Theorem 2.5.4. *The algorithm converges to a local maximum of the soft modularity function $p \mapsto Q(p)$ which is a fixed point of the updates of (2.13).*

Proof. The sequence of matrices p built by the algorithm converges to a limit $p^* \in \mathcal{Y}$, since the soft modularity Q is non-decreasing under each update, $\mathcal{Y}$ is a compact, and $p \mapsto Q(p)$ is continuous and upper bounded. From the proof of Theorem 2.5.3, we have for all $i \in V$, $G_i(p) = 0$, which gives us $(p^*_{i\cdot})^+ = p^*_{i\cdot}$, thus p^* is a fixed point for the update relative to node i. Moreover, we have for all node $i \in V$, $\nabla_i J(p^*) + v = 0$ for some $v \in \partial h(p^*_{i\cdot})$, which proves that p^* is a local optimum of the function $p \mapsto J(p) = -Q(p)$. □

2.6 Soft clustering algorithm

In the previous section, we have presented updates rules that can be applied in an alternating fashion to find a local maximum of the soft modularity $Q(p)$. However, there are three major issues with a naive implementation of this method.

1. The gradient descent update for each node i is computationally expensive because the update rule (2.13) involves a sum over all nodes of V.

2. The size of a solution p is n^2, which is prohibitive for large datasets.

3. The computational cost of the projection $\pi_{\mathcal{Y}}$ onto the probability simplex is classically in $O(n \log n)$ [Duchi et al., 2008], which can be a limiting factor in practice.

In the present section, we present an efficient implementation of the algorithm that solves these three problems. In particular, our implementation guarantees that the update step for node i only requires local computation, is memory efficient, and uses a fast mechanism for projection.

2.6.1 Local gradient descent step

The update of Theorem 2.5.3 relative to node $i \in V$ can be decomposed into two steps:

1. $\hat{p}_{i\cdot} \leftarrow p_{i\cdot} + \frac{2t}{w} \sum_{j \in V} \left(A_{ij} - \frac{w_i w_j}{w} \right) p_{j\cdot}$

2. $p_{i\cdot} \leftarrow \pi_{\mathcal{Y}}(\hat{p}_{i\cdot})$.

At first sight, step 1 seems to depend on information from all nodes of V, and to require the computation of a sum over n terms. However, the gradient descent step can be written as

$$\hat{p}_{i\cdot} \leftarrow p_{i\cdot} + \frac{2t}{w} \sum_{j \sim i} A_{ij}(p_{j\cdot} - \bar{p}),$$

where $\bar{p}$ is weighted average of vector $p_{j\cdot}$:

$$\bar{p} = \sum_{j \in V} \frac{w_j}{w} p_{j\cdot}.$$

The vector $\bar{p} \in \mathbb{R}^n$ can be stored and updated throughout the algorithm. Then, the computation of $\hat{p}_{i\cdot}$ is purely local, in the sense that it only requires information from neighbors of node i. Moreover, the sum to compute contains only d_i terms, where $d_i = |\mathrm{Nei}(i)|$ is the unweighted degree of node i. In most real-life graphs, d_i is much smaller than the number of nodes n.

2.6.2 Cluster membership representation

The cluster membership variable p is a $n \times n$ matrix. In a naive implementation, the memory cost of storing p is therefore in $O(n^2)$. However, we will see below that the projection step can be written $\pi_{\mathcal{Y}}(\hat{p}_{i\cdot}) = (\hat{p}_{i\cdot} - \theta 1)_+$ with $\theta \geq 0$, where $[(x)_+]_k = \max(x_k, 0)$. Thus, the projection acts as a thresholding step. We expect it to have the same effect as a Lasso regularization[1] and to lead to a sparse vector $p_{i\cdot}$.

To take benefit of this sparsity, we only store the non-zero coefficients of p. If, for all node $i \in V$, the number of non-zero values in vector $p_{i\cdot}$ is lower than a given L, then the memory cost of storing p is in $O(nL)$. We will see, in our experiments in Section 2.9, that the average number of positive components that we observe for vectors $p_{i\cdot}$ throughout the execution of our algorithm is lower than 2, and that the maximum number of positive components that we record is $L = 65$ for a graph with $731,293$ nodes.

[1] Note that the l_1 proximal operator for one component is $x \mapsto \mathrm{sign}(x)(|x| - \lambda)_+$.

2.6.3 Projection onto the probability simplex

The classic algorithm to perform the projection $\pi_{\mathcal{Y}}(\hat{p}_{i\cdot})$ onto $\mathcal{Y}$ is described in [Duchi et al., 2008]. It is defined as follows.

- Sort the vector $\hat{p}_{i\cdot}$ into μ: $\mu_1 \geq \mu_2 \geq \cdots \geq \mu_n$.
- Find $\rho = \max\left\{j \in [n],\ \mu_j - \frac{1}{j}\left(\sum_{r=1}^{j} \mu_r - 1\right) > 0\right\}$.
- Define $\theta = \frac{1}{\rho}\left(\sum_{i=1}^{\rho} \mu_i - 1\right)$ so that $\pi_{\mathcal{Y}}\left(\hat{p}_{i\cdot}\right) = (\hat{p}_{i\cdot} - \theta 1)_+$.

This algorithm runs in $O(n \log n)$ because of the sorting step.

In our case, the complexity can be reduced to $O(L_i \log L_i)$, with $L_i = |\mathrm{supp}_i(p)|$, where $\mathrm{supp}_i(p) = \{k \in [\![1, n]\!], \exists j \in \mathrm{Nei}(i) \cup \{i\}, p_{ik} > 0\}$ is the union of the supports of vectors $p_{j\cdot}$ for $j \in \mathrm{Nei}(i) \cup \{i\}$. L_i is upper-bounded by $L(d_i + 1)$ which is typically much smaller than the total number of nodes n.

Proposition 2.6.1. *In order to compute $\pi_{\mathcal{Y}}(\hat{p}_{i\cdot})$, we only need to sort the components $k \in \mathrm{supp}_i(p)$ of $\hat{p}_{i\cdot}$ to determine ρ and θ. All components $k \notin \mathrm{supp}_i(p)$ of $\pi_{\mathcal{Y}}\left(\hat{p}_{i\cdot}\right)$ are set to zero.*

Proof. First note that

$$\sum_{k=1}^{n} \hat{p}_{ik} = \sum_k p_{ik} + \frac{2t}{w} \sum_j A_{ij} \left(\sum_k p_{ik} - \sum_k \bar{p}_k \right) = 1$$

since $\sum_k \sum_j \frac{w_j}{w} p_{jk} = \sum_j \frac{w_j}{w} \sum_k p_{jk} = 1$.

Let j_0 be defined by $j_0 = \max\{j : \mu_j \geq 0\}$. The function $j \mapsto \left(\sum_{r=1}^{j} \mu_r - 1\right)$ increases for $j \leq j_0$ and then decreases to 0 when $j = n$. In particular, this function is non-negative for $j \geq j_0$. This implies that $\rho \leq j_0$ and $\theta \geq 0$.

Now, for $k \notin \mathrm{supp}_i(p)$, we have $\hat{p}_{ik} = -\frac{2tw_i}{w}\bar{p}_k \leq 0$ so that the k-th component of $\pi_{\mathcal{Y}}\left(\hat{p}_{i\cdot}\right)$ will be zero since $\theta \geq 0$. Moreover, since $\rho \leq j_0$, the value of $\hat{p}_{ik}$ is not used for the determination of ρ and θ. □

2.6.4 MODSOFT

Finally, the algorithm can be described as follows.

- **Initialization:** $p \leftarrow I$ and $\bar{p} \leftarrow \boldsymbol{w}/w$.
- **One epoch:** For each node $i \in V$,
  - $\forall k \in \mathrm{supp}_i(p)$, $\hat{p}_{ik} \leftarrow p_{ik} + t' \sum_{j \sim i} A_{ij}(p_{jk} - \bar{p}_k)$
  - $p_{i\cdot}^{+} \leftarrow \mathrm{project}(\hat{p}_{i\cdot})$
  - $\bar{p} \leftarrow \bar{p} + (w_i/w)(p_{i\cdot}^{+} - p_{i\cdot})$ and $p_{i\cdot} \leftarrow p_{i\cdot}^{+}$.

where $\boldsymbol{w}$ is the vector $(w_i)_{1 \leq i \leq n}$, project is the adaptation of the algorithm of [Duchi et al., 2008] presented above, and t' is the effective learning rate $t' = 2t/w$. One epoch of the algorithm is typically repeated until the increase of modularity falls below a given threshold $\epsilon > 0$, as in the Louvain algorithm. Note that we chose, in this implementation, to put each node in its own cluster at initialization, but our algorithm could also be initialized with the results of another algorithm (e.g. Louvain). We refer to this algorithm as MODSOFT (MODularity SOFT clustering). We give the algorithm pseudo-code as working python code in Section 2.8.

2.7 Link with the Louvain algorithm

In this section, we show that if the learning rate t is larger than a graph-specific threshold, one update performed by our algorithm reduces to a local update performed by the Louvain algorithm.

First, we observe that, if the membership matrix p verifies $p \in \{0,1\}^{n^2}$, then the update rule (2.13) for a node $i \in V$ becomes

$$p_{i\cdot} \leftarrow \pi_{\mathcal{Y}}\left(\left[\mathbf{1}_{\{i \in C_k\}} + \frac{2t}{w}\left(w_i(C_k) - w_i\frac{\mathrm{Vol}(C_k)}{w}\right)\right]_k\right)$$

where $C_k = \{i \in V : p_{ik} = 1\}$ refers to the kth cluster defined by p, $w_i(C)$ denotes the degree of node i in cluster $C \subset V$, $w_i(C) = \sum_{j \in C} A_{ij}$, and $\mathrm{Vol}(C)$ denotes the volume of cluster C, $\mathrm{Vol}(C) = \sum_{j \in C} w_j$.

In the following, we assume that: $\forall C, C' \subset V, \forall i \in V$,

$$C \neq C' \Rightarrow w_i(C) - \tfrac{w_i \mathrm{Vol}(C)}{w} \neq w_i(C') - \tfrac{w_i \mathrm{Vol}(C')}{w}. \tag{H}$$

The hypothesis (H) is non-binding in real cases, because, if the weights $(A_{ij})_{(i,j)\in E}$ are continuously distributed, then we will have (H) with probability 1; and if they follow a discrete distribution, we can always add a small noise to the weights, so that (H) is verified with probability 1.

Proposition 2.7.1. *Let $p \in \mathcal{X}$ be a membership matrix. If $p \in \{0,1\}^{n^2}$, if the hypothesis (H) is verified, and if $t > w/\delta$ where*

$$\delta = \min_{\substack{C,C' \subset V \\ i \in V \\ C \neq C'}} \left| \left(w_i(C) - \frac{w_i \mathrm{Vol}(C)}{w}\right) - \left(w_i(C') - \frac{w_i \mathrm{Vol}(C')}{w}\right) \right|,$$

then the update rule (2.13) for node $i \in V$ reduces to $p_{ik} \leftarrow 1$ if $k = \underset{l: j \in C_l, j \sim i}{\arg\max} \left[w_i(C_l) - w_i \frac{\mathrm{Vol}(C_l)}{w}\right]$, and $p_{ik} \leftarrow 0$ otherwise, where C_k denotes the kth cluster defined by p.

Proof. Given $p \in \mathcal{X} \cap \{0,1\}^{n^2}$ and $i \in V$, we use p^+ to denote the vector obtained by applying the update rule of Theorem 2.5.3 for node i. We have, for all $k \in [\![1,n]\!]$, $p^+_{ik} = \max(\hat{p}_{ik} - \lambda, 0)$ where $\hat{p}_{ik} = p_{ik} + \frac{2t}{w}[w_i(C_k) - w_i \mathrm{Vol}(C_k)/w]$ and λ is chosen so that $\sum_k p^+_{ik} = 1$.

Let $k^* = \arg\max_k [w_i(C_k) - w_i \mathrm{Vol}(C_k)/w]$. We assume that (H) is verified and that $t > w/\delta$. Thus, for all $k \in [\![1,n]\!]$ s.t. $k \neq k^*$,

$$\hat{p}_{ik^*} - \hat{p}_{ik} \geq p_{ik^*} - p_{ik} + \frac{2t}{w}\delta \geq -1 + \frac{2t}{w}\delta > 1. \tag{2.16}$$

In particular, this implies $p^+_{ik^*} > p^+_{ik}$. Now, note that if we had $p^+_{ik} > 0$ for a certain $k \neq k^*$, we would have $p^+_{ik^*} \in (0,1]$, $p^+_{ik} \in (0,1]$, and therefore $p^+_{ik^*} - p^+_{ik} < 1$. Yet, we would also have $p^+_{ik^*} - p^+_{ik} = (\hat{p}_{ik^*} - \lambda) - (\hat{p}_{ik} - \lambda) = \hat{p}_{ik^*} - \hat{p}_{ik}$, which leads to a contradiction with (2.16).

Therefore, we have $\forall k \neq k^*$, $p^+_{ik} = 0$, which immediately gives us $p^+_{ik^*} = 1$.

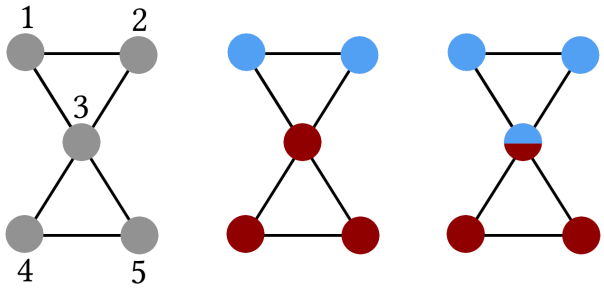
Finally, we see have seen in our simplification of the projection onto the probability simplex that the $\arg\max$ in the definition of k^* can be taken over the communities in the neighborhood of i. □

In particular, this result shows that if $p \in \{0,1\}^{n^2}$ and t is large enough, then for each update of our algorithm, the updated membership matrix p^+ verifies $p^+ \in \{0,1\}^{n^2}$, which corresponds to the maximum sparsity that can be achieved by a feasible solution $p \in \mathcal{X}$.

As described in Chapter 1, one epoch of the main routine of the Louvain algorithm considers successively each node $i \in V$, and (1) removes i from its current cluster, (2) applies an update rule, that we refer to as LouvainUpdate(i), which consists in transferring i to the cluster that leads to the largest increase in hard modularity. Proposition 2.7.2 gives an explicit formula to pick the cluster C_{k^*} to which node i is transferred by LouvainUpdate(i).

Proposition 2.7.2. *The update performed by* LouvainUpdate(i) *is equivalent to transferring node i to the cluster C^*_k such that:*

$$k^* = \underset{k: j \in C_k, j \sim i}{\arg\max} \left[w_i(C_k) - w_i \frac{\mathrm{Vol}(C_k)}{w}\right]$$



(a) Graph (b) Louvain (c) Modsoft

Figure 2.1: Bow tie graph

Proof. The variation of *hard* modularity when node $i \in V$ joins cluster C_k can be written as

$$\Delta Q(i \to C_k) = \frac{1}{w}\left[2\left(w_i(C_k) - w_i\frac{\mathrm{Vol}(C_k)}{w}\right) - \frac{w_i^2}{w}\right]. \tag{2.17}$$

Therefore, $\arg\max_k \Delta Q(i \to C_k) = \arg\max_k \left(w_i(C_k) - \frac{w_i \mathrm{Vol}(C_k)}{w}\right)$. □

Using Proposition 2.7.1, we see that the update performed by LouvainUpdate(i) and the update rule (2.13) are equivalent if (H) is verified and $t > w/\delta$ (with δ defined in Proposition 2.7.1). Therefore, under these assumptions, one epoch of our algorithm is strictly equivalent to one epoch of a slightly modified version of the Louvain algorithm in which we do not apply step (1) before applying LouvainUpdate(i). Note that step (1) should have no impact on LouvainUpdate(i) if clusters are large enough. Indeed, it only replaces in Proposition 2.7.2 the volume of the current cluster C_k of i, $\mathrm{Vol}(C_k)$, with $\mathrm{Vol}(C_k) - w_i$. It is also worth noting that the Louvain algorithm uses an aggregation routine that is not considered in the present analysis.

2.8 Algorithm pseudo-code

We give the pseudo-code of the algorithm as working python code in in Algorithm 1. The algorithm only requires one parameter, the learning rate `lr`, which corresponds to the $\frac{2t}{w}$ factor in the previous section.

The algorithm stores two variables `p` and `avg_p`. The variable `p` corresponds to the membership matrix p, i.e. the output of our algorithm, and is stored as a dictionary of dictionary, so that each `p`$[i][k]$ corresponds to a positive coefficient of p, p_{ik}. The variable `avg_p` is a dictionary used to store the average cluster membership vector $\bar{p}$, so that `avg_p`$[k]$ corresponds to $\bar{p}_k$.

The graph is given to the algorithm as four variables: `nodes`, `edges`, `degree` and `w`. The variable `nodes` contains the list of the nodes. The variable `edges` is a dictionary of dictionary, where `edges`$[i][j]$ contains the weight A_{ij}. The variable `degree` is a dictionary containing the node degrees, and `w` is the total weight of the graph.

One epoch in our algorithm corresponds to the application of all the update rules of Theorem 2.5.3, each update rule corresponding to the update of the membership probabilities of one node. Several strategies can be adopted regarding the choice of the number of epochs. A fixed number of epochs can be given in advance as an additional parameter to the algorithm, or the soft-modularity can be computed at the end of each epoch, and be used as a stopping criterion. For instance, we can stop the algorithm when the modularity increase becomes smaller than a given precision factor $\epsilon > 0$. This later strategy is the equivalent of the strategy used by the Louvain algorithm for the *hard* modularity problem.

Algorithm 1 Soft-modularity optimization

Require: `nodes`, `edges`, `degree`, `w`, `lr` (learning rate)

Initialization

```
p = dict()
avg_p = dict()
for node in nodes:
    p[node] = {node: 1.}
    avg_p[node] = (1./w) * degree[node]
```

One epoch (update the membership matrix `p`)

```
for node in nodes:
    new_p = dict()
    # Gradient descent step
    for com in p[node]:
        new_p[com] = p[node][com]
    for neighbor in edges[node]:
        weight = edges[node][neighbor]
        for com in p[neighbor]:
            if com not in new_p:
                new_p[com] = 0.
            new_p[com] += lr * weight * p[neighbor][com]
    for com in new_p:
        new_p[com] -= lr * degree[node] * avg_p[com]
    # Projection step
    new_p = project(new_p)
    # Updating average membership vector
    for com in p[node]:
        avg_p[com] -= (1./w) * degree[node] * p[node][com]
    p[node] = dict()
    for com in new_p:
        avg_p[com] += (1./w) * degree[node] * new_p[com]
        p[node][com] = new_p[com]
```

Sub-routine `project`

```
def project(in_dict):
    # Sort the values of in_dict in decreasing order
    values = sorted(in_dict.values(), reverse=True)
    # Find the value of lambda
    cum_sum = 0.; lamb = 0.
    i = 1
    for val in values:
        cum_sum += val
        new_lamb = (1. / i) * (cum_sum - 1.)
        if val - new_lamb <= 0.:
            break
        else:
            lamb = new_lamb
            i += 1
    # Create the output dictionary
    out_dict = dict()
    for key in in_dict:
        out_dict[key] = max(in_dict[key] - lamb, 0)
    return out_dict
```

2.9 Experimental results

2.9.1 Synthetic data

Bow tie

We first consider a toy example that we call the *bow tie* graph, that is defined in Figure 2.1 (a). The Louvain algorithm applied to this graph returns partition $\{\{1,2,3\},\{4,5\}\}$ or partition $\{\{1,2\},\{3,4,5\}\}$ depending on the order in which the updates LouvainUpdate are performed. It can be easily verified that these partitions correspond to the maximum of hard modularity over all possible partitions. In Figure 2.1 (b), we represent the results of the Louvain algorithm where colors code for clusters. In Figure 2.1 (c), we represent the results of our algorithm where the cluster membership is coded with a pie chart for each node. We see that our algorithm assigns nodes 1 and 2 to one cluster, and nodes 4 and 5 to another cluster, whereas node 3 has a mixed membership, with a probability 0.5 to belong to each cluster. It is easy to check that the membership matrix p returned by our algorithm corresponds to the optimal solution of the problem (2.7). The modularity Q_{soft} found by our algorithm is 50% higher than that found by the Louvain algorithm Q_{hard}.

Overlapping SBM

Now, we consider a variant with overlaps of the Stochastic Block Model (SBM) presented in Chapter 1, defined as follows. We have $V = C_1 \cup \cdots \cup C_K$ with $|C_1| = \ldots = |C_K| = c$ and $|C_k \cap C_{k+1}| = o$ for all k. For all nodes i and j such that $i \neq j$, the edge (i,j) exists with probability p_{in} if $i,j \in C_k$ for some k, and with probability p_{out} otherwise.

In order to numerically evaluate the performance of our algorithm on this model, we first compute the relative difference between the modularity Q_{soft} obtained with our algorithm and the modularity Q_{hard} obtained with the Louvain algorithm, $\Delta Q = (Q_{\text{soft}} - Q_{\text{hard}})/Q_{\text{hard}}$. We also compare the planted clusters of our model C_k with the clusters $\hat{C}_k$ returned by our algorithm. We consider that these clusters are the non-empty sets $\hat{C}_k$ where $\hat{C}_k = \{i \in V : p_{ik} > 0\}$. Note that the clusters $\hat{C}_k$ returned by our algorithm can overlap just as the C_k. In order to compare the clusters C_k and $\hat{C}_k$, we use the average of the F_1-score introduced in the introductory chapter. Remember that it takes values between 0 and 1, and is equal to 1 if the estimated clusters $\hat{C}_k$ correspond exactly to the initial clusters C_k.

We compute the same two metrics for Chang's algorithm [Chang and Chang, 2017]. Since the membership matrix returned by this algorithm is not sparse, we define $\hat{C}_k$ as $\hat{C}_k = \{i \in V : p_{ik} > \alpha\}$ for the F_1-score evaluation. We present the results for $\alpha = 0.05$, but other values of α leads to similar results.

For different values of the cluster size c, we run our algorithm, Chang's algorithm and the Louvain algorithm on 100 random instances of this model, with $o = 2$, $K = 2$, $p_{\text{in}} = 0.9$, and $p_{\text{out}} = 0.1$, We display the results in Figure 2.2. The average F_1-scores for our algorithm and Chang's algorithm are close to 1 (> 0.99 on average), and are higher than those obtained by Louvain (0.94 on average). Besides, the relative difference to the modularity found by Louvain ΔQ for Chang's algorithm and our algorithm is always positive. However, we note that ΔQ and the differences between the F_1-scores become negligible as the cluster size c increases. This can be explained by the fact that, as the cluster size c increases, the proportion of nodes with mixed membership, i.e. nodes that belong to multiple clusters C_k, decreases. If this proportion is low, the misclassification of nodes with mixed membership has a low impact on the modularity score Q and on the F_1-score.

We also represent in Figure 2.2 the sparsity of the solution in term of the number of non-zero coefficients of the matrix p, and the average execution time in milliseconds for the different algorithms. We observe that our algorithm provides performance comparable to Louvain (but with richer membership information). Besides, we see that MODSOFT runs more than 10 times faster and uses 5 times less memory than Chang's algorithm.

2.9.2 Real data

Wikipedia

We consider a graph built from Wikipedia data, where nodes correspond to Wikipedia articles and edges correspond to hyperlinks between these articles (we consider that the edges are undirected and unweighted). Wikipedia articles can correspond to a wide variety of entities: persons, locations, organizations, concepts

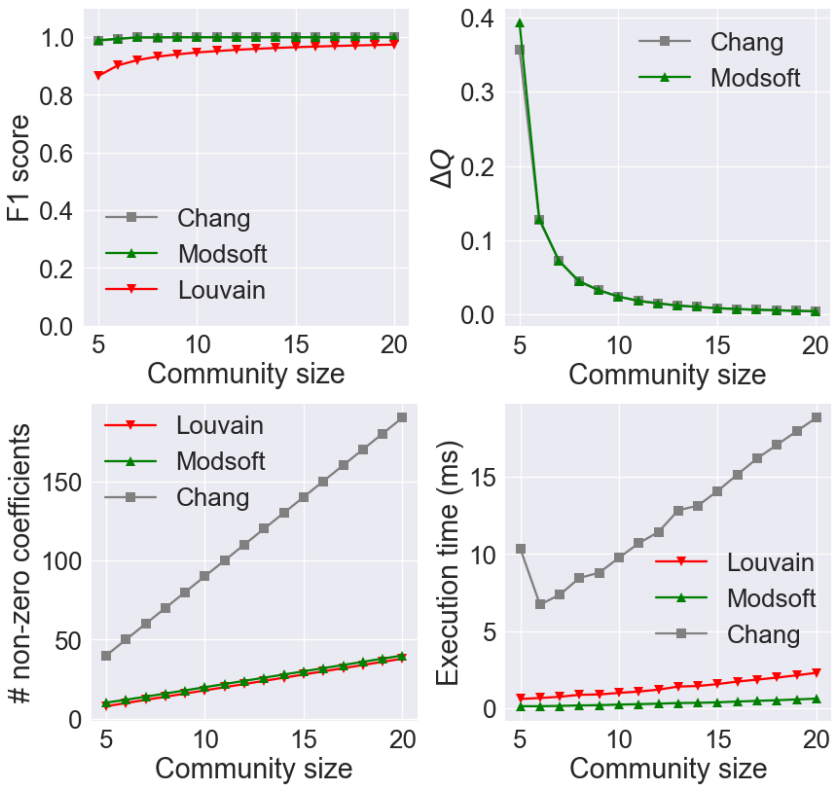


Figure 2.2: Overlapping SBM

etc. In order to restrict ourselves to homogeneous entities, we choose to limit ourselves to Wikipedia pages that correspond to humans. For this purpose, we use Wikidata, a knowledge base that provides structured data about Wikipedia articles. In particular, Wikidata objects have an `instance_of` field that we use to determine if a Wikipedia article represents a human or not. The subgraph induced by the human entities has 731,293 nodes and 3,266,258 edges.

We run the Louvain algorithm, Chang's algorithm and MODSOFT on this subgraph. As described in Section 2.4, Chang's algorithm requires the maximum number of clusters K to be specified. We run it with $K = n$ (which corresponds to the implicit choice made for our algorithm), K equal to the number of clusters returned by Louvain (i.e. $K = 12,820$) and $K = 100$. Chang's algorithm is unable to handle the graph (the execution raises a memory error) for all these choices of K, whereas our algorithm runs in 38 seconds on an Intel Xeon Broadwell CPU with 32 GB of RAM. As a baseline for comparison, the Louvain algorithm runs in 20 seconds on this dataset. The difference in memory consumption between our algorithm and Chang's algorithm can easily be explained by the fact that the membership matrix p in Chang's approach is a dense $n \times K$ matrix (see Section 2.4), i.e. the algorithm needs to store and perform operations on $73,129,300$ coefficients for $K = 100$. By contrast, the matrix returned by our algorithm is very sparse, since it contains only $760,546$ non-zero coefficients in our experiments, which represents a proportion of approximately 10^{-6} of the coefficients of the $n \times n$ matrix. The average number of positive components of vectors $p_{i\cdot}$ throughout the execution of the algorithm is 1.21, and the densest vector $p_{i\cdot}$ has 65 positive components.

Our algorithm leads to a higher modularity than the Louvain algorithm on this dataset. However, this increase represents a modularity increase ΔQ of slightly less than 1%. This can be explained by the fact that the nodes with mixed membership found by our algorithm represent less than 5% of the total number of nodes. Even if the performance increase brought by our algorithm in terms of modularity is marginal, we qualitatively observe that the results of our algorithm, and in particular the nodes with mixed membership identified by our approach, are particularly relevant.

In order to be able to graphically represent the results of our algorithm, we consider the subgraph induced by the top 100 nodes in term of degree. We display the results of our algorithm on this smaller graph in Figure 2.3. In particular, we observe that Napoleon (1769-1821), who became Emperor of the

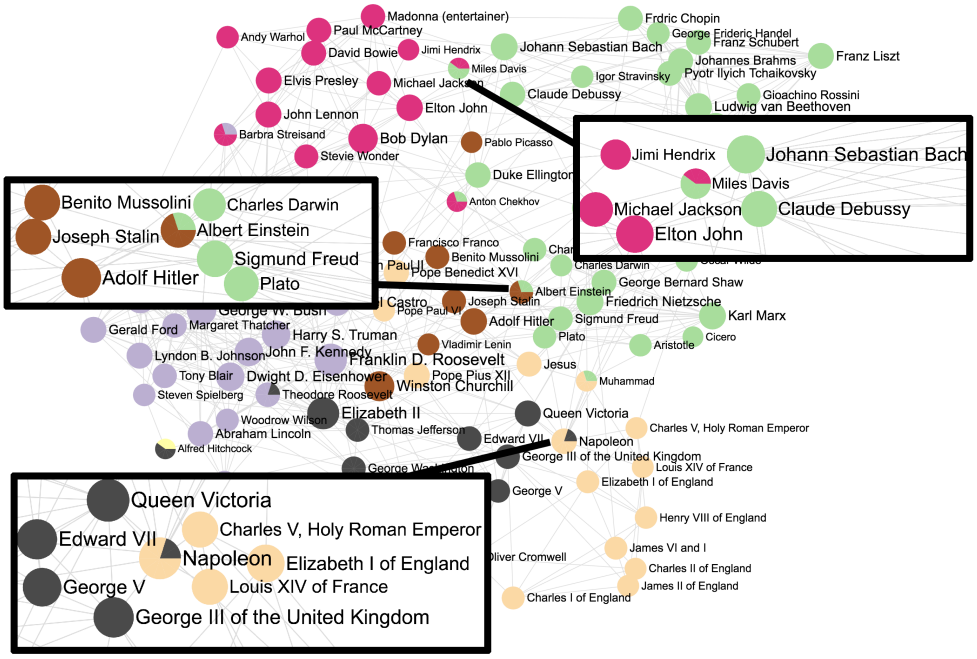


Figure 2.3: MODSOFT on a small subgraph of Wikipedia

French in 1804, appears to belong to the same cluster as European leaders from the old regime, such as Charles V (1338-1380) and Louis XIV (1638-1715), and, at the same time, to the same cluster as post-French-revolution leaders such as Queen Victoria (1819-1901) and Edward VII (1841-1910). We also see that Miles Davis (1926-1951), a famous American jazz trumpeter, is found to belong to the same cluster as classical music composers, such as Claude Debussy (1862-1918) and Johan Sebastian Bach (1685-1750), but also to the same cluster as pop/rock musicians such as Jimi Hendrix (1942-1970) and Michael Jackson (1958-2009). Finally, we observe that Albert Einstein who became politically involved during World War II, belongs to the same cluster as thinkers, intellectuals and scientists, such as Sigmond Freud and Plato, and to the same cluster as political leaders during World War II such as Adolf Hitler and Joseph Stalin.

Open Flights

We consider a graph built from the Open Flights database that regroups open-source information on airports and airlines. The graph we consider is built as follows: the nodes correspond to the airports and the edges correspond to the airline routes, i.e. there is an edge between two airports if an airline operates services between these airports.

We apply our algorithm and the Louvain algorithm to this graph. During the execution of the algorithm, the number of positive components of vectors $p_{i\cdot}$ remains lower than 11 and is on average 1.17. We measure a relative increase in modularity ΔQ of less that 1%, and yet we find that our algorithm brings insightful information about the dataset by identifying bridges between important zones of the world. In Figure 2.4, we display the results of our algorithm. We use plain colors to code for the cluster membership of *pure* nodes, i.e. nodes that belong to a unique cluster, and bigger red dots to identify nodes with a mixed membership. Note that most of these nodes with mixed membership are located on the borders between large regions of the world. For instance, we remark that numbers of nodes are located on the frontier between Europe, Africa and Middle East, such as the Dubai airport in the United Arab Emirates. We also observe that six airports with mixed membership are located all along the border between the United States on the one hand, and Canada and Alaska on the other hand.

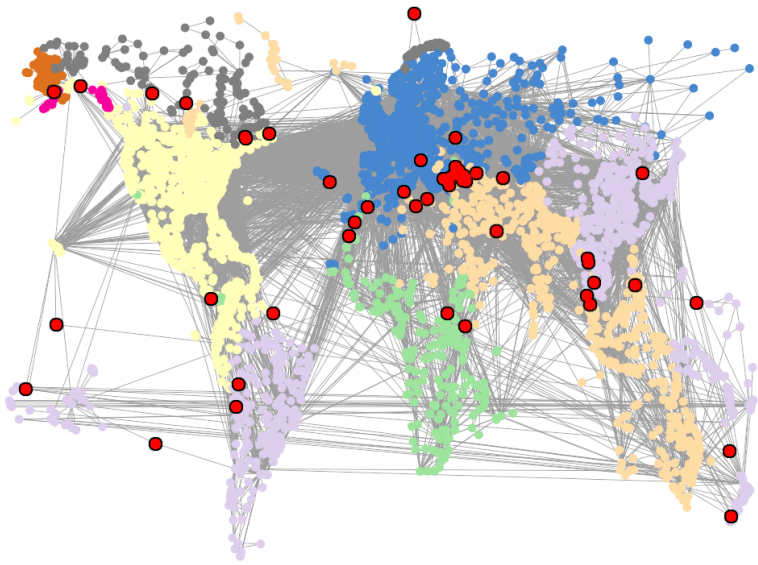


Figure 2.4: MODSOFT on the OpenFlights graph

	Graph size		F_1-score			Execution time (s)		
	$\|V\|$	$\|E\|$	L	O	M	L	O	M
Amazon	334,863	925,872	0.28	0.47	0.53	15	1038	32
DBLP	317,080	1,049,866	0.14	0.35	0.38	17	1717	33
YouTube	1,134,890	2,987,624	0.01	-	0.15	45	-	98

Table 2.1: F_1-scores and execution times in seconds on SNAP datasets

SNAP datasets

We present further experiments on the datasets from SNAP[2] that come with ground-truth clusters [Yang and Leskovec, 2015]. The Amazon dataset corresponds to a product co-purchasing network. The nodes of the graph represent Amazon products and the edges correspond to frequently co-purchased products. The ground-truth clusters are defined as the product categories. The DBLP dataset corresponds to a scientific collaboration network. The nodes of the graph represent the authors and the edges the co-authorship relations. The scientific conferences are used as ground-truth clusters. In the YouTube dataset, nodes represent users and edges connect users who have a friendship relation. User groups are used as ground-truth clusters. We use Louvain and the state-of-the-art algorithm OSLOM [Lancichinetti et al., 2011] as baselines for our experiments. OSLOM is known as one of the most effective overlapping community detection algorithm [Xie et al., 2013], and its C++ code is open-source. In Table 2.1, we give the F_1-scores and the execution times for Louvain (L), OSLOM (O) and Modsoft (M). The experiments were carried out on an `m4.xlarge` AWS instance with 64GB RAM and 16 vCPUs with Intel Xeon. A learning of $t = 0.1$ was used for Modsoft. Note that we did not include the results of OSLOM on the YouTube dataset because its the execution time exceeded 6 hours. We see that Modsoft outperforms OSLOM both in terms of F_1-score (11% higher on average) and execution time (OSLOM is about 40 times slower). Moreover, although Modsoft is slower than Louvain (about twice slower on average), it offers F_1-scores that are at least twice as large as the ones provided by Louvain. This can be explained by the fact that the clusters returned by Louvain are necessary disjoint whereas the ground-truth clusters in all three datasets can overlap. In contrast, Modsoft is able to capture these mixed memberships.

Note that such results using ground-truth clusters must be handled with care as they highly depend on the definition of the ground-truth as underlined in Chapter 1. Qualitative results provided above give additional guarantees about the performance of the Modsoft algorithm.

[2]Stanford Network Analysis Project, https://snap.stanford.edu/data/

Chapter 3

Edge clustering

In this chapter, we study another approach to overcome the limits of node partitioning algorithms. We study the problem of edge clustering and introduce an algorithm based on a variant of the modularity function for edge partitions. This extension naturally emerges from the interpretation of the modularity in terms of random walks. In this paper, we propose a novel agglomerative algorithm to efficiently maximize this alternative *edge* modularity function. This algorithm is based on an important conservation result that guarantees that modularity is invariant under the aggregation operation that we consider. We show in our experiments that our method outperforms existing algorithms, mainly because, unlike other methods, our approach does not require the construction of a memory-expensive line graph. This chapter is based in the work presented in [Hollocou et al., 2018].

3.1 Introduction

As pointed in the previous chapter, most of existing approaches focus on finding partitions of the graph nodes, even if recovering node partitions is often too restrictive in practice, as a node often belongs to several clusters in real-life graphs. We gave the example of social networks, where users can potentially belong to multiple social circles (friends, colleagues, family etc.). For this reason, it can be more interesting to *cluster the graph edges instead of the nodes*. In the social network example, this allows us to make a distinction between the different types of relationships (friendships, business relationships, family ties etc.) by putting the corresponding edges in different clusters.

In this chapter, we study graph clustering algorithm that output clusters of edges, i.e. subsets $C \subset E$. In particular, we introduce a generalization of the classic modularity measure to score partition of edges. We refer to this alternative objective function as the *edge* modularity. Whereas the standard modularity function can be interpreted using a random walk on graph nodes [Delvenne et al., 2010, Lambiotte et al., 2008] as seen in Chapter 1, the edge modularity arises by considering the sequence of edges crossed by this very same random walk. We propose a novel algorithm to maximize the edge modularity using an agglomerative strategy similar to the one used by the Louvain algorithm that enables it to scale to large graphs. Our method is based on a key conservation result that states that the edge modularity is invariant under the particular aggregation scheme performed by our algorithm. This aggregation strategy relies on a distinction between *border* nodes that are linked to edges from different clusters and *internal* nodes that are only connected to edges from one cluster.

We show on both synthetic and real-life datasets that our algorithm significantly outperforms, both in terms of execution time and memory consumption, the existing approaches that rely on the construction of a weighted line graph [Evans and Lambiotte, 2009] whose size is much larger than the original graph. Another important benefit of our method is the aggregated graph produced by the algorithm, as it is easy to visualize and interpret, and can be used to obtain better insights on large datasets.

The rest of the chapter is organized as follows. We briefly present the related work in section 3.2. In section 3.3, we define the edge modularity that we consider and study some of its properties. We study in detail the algorithm introduced by Evans and Lambiotte in Section 3.4. In section 3.5, we present our aggregation scheme, state the edge modularity conservation result, and define our edge modularity optimization algorithm. We present experimental results on synthetic and real-world data in section 3.6.

3.2 Related work

The problem of edge clustering has been tackled in different papers [Evans and Lambiotte, 2009, Ahn et al., 2010, Shi et al., 2013, Zhou et al., 2008]. We study in details the methods proposed in [Evans and Lambiotte, 2009] in Section 3.4. This approach relies on the definition of three scoring functions for edge partitions based on the interpretation of the modularity function in terms of random walks [Delvenne et al., 2010, Lambiotte et al., 2008] presented in Chapter 1. Evans and Lambiotte introduce these new functions but do not propose any specific algorithm to maximize them. As explained in Section 3.4, the optimization of these functions requires the construction of three variants of the line graph (graphs whose nodes are the edges of the original graph) and the use of classic modularity optimization techniques on these graphs. In practice, the size of these line graphs is prohibitive, as the number of edges in these new graphs is in $O(\sum_u d_u^2)$ where d_u is the degree of node u, and their method cannot be applied to large real-life graphs (see Section 3.6). In this chapter, we do not review in details the other existing approaches presented in [Ahn et al., 2010, Shi et al., 2013, Zhou et al., 2008]. But, as pointed in the respective papers, these algorithms only apply to small graphs an cannot be use to cluster real-life graphs as the ones considered in Section 3.6.

3.3 A modularity function for edge partitions

As explained in our introductory chapter, the modularity function can also be interpreted in terms of random walk on the nodes of G [Delvenne et al., 2010, Lambiotte et al., 2008]. Let us consider the classic random walk $(X_t)_{t\geq 0}$ on the nodes of G where the walk moves from a node u to one of its neighbors v with probability A_{uv}/w_u. $(X_t)_{t\geq 0}$ is a Markov chain. If G is connected and not bipartite, $(X_t)_{t\geq 0}$ has a unique stationary distribution π, defined as $\pi(u) = \frac{w_u}{w}$, and we have for all $u \in V$, $\lim_{t\to\infty} \mathbb{P}[X_t = u] = \pi(u)$ [Chung, 1997]. Then, the modularity of a partition $\mathcal{C}$ can be written as

$$Q(\mathcal{C}) = \sum_{C\in\mathcal{C}} \left(\mathbb{P}_\pi[X_0 \in C, X_1 \in C] - \mathbb{P}_\pi[X_0 \in C, X_\infty \in C]\right), \tag{3.1}$$

where $\mathbb{P}_\pi[X_0 \in C, X_1 \in C]$ is the probability for the walk to be in C at initial time (with X_0 initialized with the distribution π) and after one step, and $\mathbb{P}_\pi[X_0 \in C, X_\infty \in C]$ is the probability for the walk to be in C at initial time and in the stationary regime.

From the random walk $(X_t)_{t\geq 0}$ on the nodes of G, we can naturally define a Markov chain $(Y_t)_{t\geq 0}$ on the edges E of G, with, for all $t \geq 0$, $Y_t = \{X_t, X_{t+1}\} \in E$. For $e = \{u, v\} \in E$, we have:

$$\mathbb{P}[Y_t = e] = w_e \sum_{u'\in e} \frac{\mathbb{P}[X_t = u']}{w_{u'}},$$

where w_e denotes the weight of the edge e, i.e. $w_e = A_{uv}$.

Building on the random walk interpretation of the modularity, we can define the modularity for a partition of the edges E using $(Y_t)_{t\geq 0}$. Given a partition $\mathcal{C}$ of the edges of G, we define the *edge* modularity of $\mathcal{C}$ as

$$\tilde{Q}(\mathcal{C}) = \sum_{C\in\mathcal{C}} \left(\mathbb{P}_\pi[Y_0 \in C, Y_1 \in C] - \mathbb{P}_\pi[Y_0 \in C, Y_\infty \in C]\right), \tag{3.2}$$

where $\mathbb{P}_\pi[Y_0 \in C, Y_1 \in C]$ is the probability for the first two edges crossed by the walk to be in C (with X_0 initialized with the distribution π), and $\mathbb{P}_\pi[Y_0 \in C, Y_\infty \in C]$ is the probability for an edge crossed by the walk to be in C for the first jump and in the stationary regime.

The following proposition gives explicit expressions of the *edge* modularity.

Proposition 3.3.1. *The* edge *modularity of a partition $\mathcal{C}$ of E can be written as*

$$\tilde{Q}(\mathcal{C}) = \sum_{C\in\mathcal{C}} \sum_{e,f\in C} \left(\frac{w_e w_f}{w} \sum_{u\in e\cap f} \frac{1}{w_u} - \frac{w_e w_f}{(w/2)^2} \right) \tag{3.3}$$

and

$$\tilde{Q}(\mathcal{C}) = \sum_{C\in\mathcal{C}} \left[\sum_{u\in V} \frac{(w_u(C))^2}{ww_u} - \left(\frac{w(C)}{w(E)}\right)^2 \right] \tag{3.4}$$

where $w_u(C) = \sum_{e\in C} \mathbb{1}_{\{u\in e\}} w_e$ *and* $w(C) = \sum_{e\in C} w_e$.

Proof. The first equality follows from

$$\begin{aligned}
\mathbb{P}_\pi[Y_0 \in C, Y_1 \in C] &= \sum_{e\in C} \mathbb{P}_\pi[Y_0 \in C, Y_1 = e] = \sum_{e\in C} w_e \sum_{u\in e} \frac{\mathbb{P}_\pi[X_1 = u, Y_0 \in C]}{w_u} \\
&= \sum_{e\in C} w_e \sum_{u\in e} \frac{1}{w_u} \sum_{v\in V} \underbrace{\mathbb{P}_\pi[X_0 = v]}_{=w_v/w} \frac{A_{vu}}{w_v} \mathbb{1}_{\{\{v,u\}\in C\}} \\
&= \frac{1}{w} \sum_{e\in C} w_e \sum_{u\in e} \frac{1}{w_u} \sum_{f\in C} \mathbb{1}_{\{u\in f\}} w_f = \sum_{e,f\in C} \frac{w_e w_f}{w} \sum_{u\in e\cap f} \frac{1}{w_u} \\
\mathbb{P}_\pi[Y_0 \in C, Y_\infty \in C] &= \sum_{e,f\in C} \frac{4w_e w_f}{(w)^2}.
\end{aligned}$$

where we used the fact that, for all $e \in E$,

$$\lim_{t\to\infty} \mathbb{P}[Y_t = e] = w_e \sum_{u\in e} \frac{1}{w_u} \underbrace{\lim_{t\to\infty} \mathbb{P}[X_t = u]}_{w_u/w} = \frac{2w_e}{w}.$$

For the second equality, we remark that

$$\sum_{e,f\in C} \frac{w_e w_f}{w} \sum_{u\in e\cap f} \frac{1}{w_u} = \sum_{u\in V} \frac{1}{ww_u} \left(\sum_{e\in C} \mathbb{1}_{\{u\in e\}} w_e \right)^2$$

and

$$\sum_{e,f\in C} \frac{w_e w_f}{(w/2)^2} = \frac{\left(\sum_{e\in C} w_e\right)^2}{\left(\sum_{e\in E} w_e\right)^2}.$$

□

The *edge* modularity $\tilde{Q}(\mathcal{C})$ of an edge partition $\mathcal{C}$ is actually equal to the standard modularity of $\mathcal{C}$ in a graph H whose nodes correspond to the edges of G. This result is stated in the following proposition.

Proposition 3.3.2. *Let us define the graph* $H = (V^H, G^H)$ *with* $V^H = E$, $E^H = \{\{e, f\} \in E \times E : \exists u \in V, u \in e \cap f\}$, *and the adjacency matrix* $A^H_{ef} = \frac{w_e w_f}{2} \sum_{u\in e\cap f} \frac{1}{w_u}$.

For all partitions $\mathcal{C}$ *of* E, *we have* $\tilde{Q}(\mathcal{C}) = Q^H(\mathcal{C})$. *In other words, the* edge modularity *of* $\mathcal{C}$ *in* G *is equal to the standard modularity of* $\mathcal{C}$ *in* H.

Proof. First, we have for all $e \in V^H$.

$$\begin{aligned}
w^H_e &= \sum_{f\in V^H} A^H_{ef} = \sum_{f\in E} \frac{w_e w_f}{2} \sum_{u\in e\cap f} \frac{1}{w_u} \\
&= \frac{w_e}{2} \sum_{u\in e} \frac{1}{w_u} \sum_{f\in E} \mathbb{1}_{\{u\in f\}} w_f \\
&= \frac{w_e}{2} \sum_{u\in e} \frac{w_u}{w_u} = w_e.
\end{aligned}$$

Then, observe that

$$w^H = \sum_{e\in V^H} w^H_e = \sum_{e\in E} w_e = w/2.$$

This gives us

$$\sum_{C\in\mathcal{C}}\sum_{e,f\in C}\frac{A^H_{ef}}{w^H}=\sum_{C\in\mathcal{C}}\sum_{e,f\in C}\frac{w_e w_f}{w}\sum_{u\in e\cap f}\frac{1}{w_u}$$

and

$$\sum_{C\in\mathcal{C}}\sum_{e,f\in C}\frac{w^H_e w^H_f}{(w^H)^2}=\sum_{C\in\mathcal{C}}\sum_{e,f\in C}\frac{w_e w_f}{(w/2)^2}$$

Using (3.3), we can conclude the proof. □

Proposition 3.3.2 has important consequences. First, it follows from the work of Brandes et al. [Brandes et al., 2007] that the problem of maximizing $\tilde{Q}(\mathcal{C})$ is NP-hard and cannot be solved exactly in reasonable time. Then, we see that the standard modularity maximization heuristics as the Louvain algorithm can be used on the graph H to find an approximation of the solution of $\max_{\mathcal{C}}\tilde{Q}(\mathcal{C})$. However, we will see that this approach does not scale to large graphs.

3.4 The approach of Evans and Lambiotte

From Proposition 3.3.2, we see that the edge modularity can be written as the *standard* modularity using a different adjacency matrix A^H of size $m\times m$ that corresponds to a graph H whose nodes are the edges of the original graph:

$$Q(\mathcal{C})=\frac{1}{w^H}\sum_{C\in\mathcal{C}}\sum_{e,f}\left(A^H_{ef}-\frac{w^H_e w^H_f}{w^H}\right), \tag{3.5}$$

where $w^H_e=\sum_f A^H_{ef}$ and $w^H=\sum_e w^H_e$. We will see that the matrix A^H can be seen as the result of an operation called a *projection* of a particular bipartite graph, the *incidence graph*. In [Evans and Lambiotte, 2009], the authors study similar approaches with other $m\times m$ matrices that correspond to different projections of the incidence graph on the set of edges.

3.4.1 Incidence graph and bipartite graph projections

The *incidence matrix* of the graph G is a $n\times m$ matrix, that we note B, such that, for all node $i\in V$ and for all edge $e\in E$, $B_{ie}=1$ if the edge e is incident to the node i, and $B_{ie}=0$ otherwise. This matrix can be seen as the adjacency matrix of a bipartite graph, the so-called *incidence graph* $I(G)$, whose nodes are, on the one hand, the nodes V of the original graph, and, on the other hand, its edges E, and such that there is an edge between $i\in V$ and $e\in E$ if and only if i is an extremity of e.

If the graph G is unweighted and has no self-loop, which is an hypothesis of Evans et Lambiotte that we will use in the rest of this section, the adjacency matrix of the original graph can be written in function of the incidence matrix B:

$$\forall i,j,\quad A_{ij}=\sum_{e\in E}B_{ie}B_{je}(1-\delta_{ij}). \tag{3.6}$$

Such an operation, where we go from the matrix $B\in\{0,1\}^{n\times m}$ to $A\in\{0,1\}^{n\times n}$ is called a *projection*. More precisely, a projection of a bipartite graph between two sets X and Y is an operation that compresses the information contained in the bipartite graph by building a new graph that contains nodes of only either of the two sets X and Y.

In (3.6), we see that the graph G is a projection of the incidence graph $I(G)$ on the set of nodes. In this section, we are interested in projections of $I(G)$ to build graphs whose nodes are the edges E rather that the nodes V. The graph H defined in Proposition 3.3.2 is such a projection. However, there exists other ways to project the incidence graph $I(G)$ that leads to alternative definitions of the edge modularity in (3.5). Evans and Lambiotte study three of such projections that we present below.

3.4.2 Line graph

The simplest projection of $I(G)$ on the set of edges E consists in using the formula (3.6), that defines G as a projection of $I(G)$ on the set of nodes, and replace V with E. As in [Evans and Lambiotte, 2009], we refer to this projection as C. The adjacency matrix of C is defined as

$$\forall e, f, \quad A^C_{ef} = \sum_{i \in V} B_{ie} B_{if} (1 - \delta_{ef}).$$

In other words, $A^C_{ef} = 1$ if $e \neq f$ and $e \cap f \neq \emptyset$, i.e. if $e \neq f$ and e and f have a common extremity, and $A^C_{ef} = 0$ otherwise. C corresponds to the standard definition of the *line graph* that we find in the graph theory literature [Balakrishnan, 1997]. Note that we can derive C from the graph H defined in Proposition 3.3.2 by removing the edge weights and the self loops from H. The degree of $e \in E$ in C can be written in function of the (unweighted) degrees d_i of nodes in G:

$$d^C_e = \sum_{f \in E} A^C_{ef} = \sum_{i \in V} \sum_{f \in E} B_{ie} B_{if} (1 - \delta_{ef}) = \sum_{i \in e} (d_i - 1).$$

In other words, $d^C_e = d_i + d_j - 2$ if $e = (i, j)$. Besides, we see that a node of degree d_i in G corresponds to a clique of d_i nodes in the line graph C. This gives too much importance to the nodes with large degrees in G and motivates the definition of weighted line graphs.

3.4.3 Weighted line graph

We have seen above that the degree of a node $e = (i, j) \in E$ in the line graph C is in $O(d_i + d_j)$. The idea behind the following projection of the incidence graph $I(G)$ studied by Evans and Lambiotte is to obtain a graph whose degrees are in $O(1)$. This can be done by considering the graph D with the adjacency matrix

$$A^D_{ef} = \sum_{i : d_i > 1} \frac{B_{ie} B_{if}}{d_i - 1} (1 - \delta_{ef}).$$

Note that D is a weighted version of the line graph C. The weighted degree of a node $e = (i, j)$ in D is

$$w^D_e = \sum_f A^D_{ef} = \mathbb{1}_{d_i > 1} + \mathbb{1}_{d_j > 1}.$$

In other words, the degree of an edge $e \in E$ is 0 if it is an isolated edge, it is equal to 1 if it is a leaf, and it is equal to 2 otherwise. Note that we have $w^D_e = O(1)$ as wanted. As noted in [Evans and Lambiotte, 2009], this type of weighted projection is well suited for bipartite graphs corresponding to scientific collaborations. Indeed, if B corresponds to the adjacency matrix of a graph whose nodes are either scientists X or scientific papers Y, such that x is connected to y if x has written y, the projection defined as

$$\forall x, x', \forall y, \quad A_{xx'} = \sum_{y : d^B_y > 1} \frac{B_{xy} B_{x'y}}{d^B_y - 1} (1 - \delta_{xx'})$$

gives a larger weight to edges between scientists who have written papers together with few co-authors than to edges between scientists who have joint publications with many co-authors.

3.4.4 Random-walk-based projection

In Proposition 3.3.2, we have introduced a projection H of the incidence graph based on the canonical random walk $(X_t)_{t \geq 0}$ on G. This projection is not directly studied in [Evans and Lambiotte, 2009], but the authors consider a slight variation of H. Note that we have

$$A^H_{ef} = w^H \mathbb{P}_\pi [Y_0 = e, Y_1 = f],$$

where $(Y_t)_{t \geq 0}$ is the Markov chain on edges E introduced in the previous section. Evans and Lambiotte consider the projection E_1 defined as

$$A^{E_1}_{ef} = w^H \mathbb{P} [Y_0 = e, Y_2 = f],$$

which corresponds to 2 steps of $(Y_t)_{t\geq 0}$ instead of 1. A^{E_1} can also be written as

$$\forall e, f, \qquad \sum_{i,j:d_i>0,d_j>0} \frac{B_{ie}A_{ij}B_{jf}}{d_i d_j}.$$

This projection is particularly interesting in light of Chapter 2. Indeed, recall that the *soft modularity* can be written as

$$Q(p) = \frac{1}{w}\sum_{ij}\left(A_{ij} - \frac{w_i w_j}{w}\right) p_{ik}p_{jk},$$

where p_{ik} is the degree of membership of node i to the kth cluster. If we define a membership degree for each edge e to each cluster k, q_{ek}, and we write p in function of q as

$$\forall i, k, \quad p_{ik} = \sum_{e\in E} \frac{B_{ie}q_{ek}}{d_i},$$

then the soft modularity becomes

$$\begin{aligned} Q(p) &= \frac{1}{w}\sum_k \sum_{e,f} \sum_{i,j} \left(\frac{B_{ie}A_{ij}B_{jf}}{d_i d_j} - \frac{B_{ie}B_{jf}}{w}\right) q_{ek}q_{fk} \\ &= \frac{1}{w}\sum_k \sum_{e,f} \left(A^{E_1}_{ef} - \frac{w_e w_f}{w}\right) q_{ek}q_{fk} \equiv Q^{E_1}(q). \end{aligned}$$

In other words, the soft modularity for nodes $Q(p)$ is equal to the soft modularity $Q^{E_1}(q)$ for the membership matrix q in the graph E_1.

3.4.5 Limitation

In [Evans and Lambiotte, 2009], the authors do not present any specific technique to optimize the modularity (3.5) with the three projections C, D and E_1 that they introduce. However, this optimization can be done by applying standard modularity maximization algorithms in these three variants of the line graph. Nevertheless, as described in the following section, this technique is prohibitive in practice because the construction of the line graph is too expensive for large graphs (the number of edges in this graph is in $O(\sum_i d_i^2)$). The projection H that we introduced in the previous section have specific properties that enable us to overcome this problem and design an efficient algorithm for the edge modularity maximization problem that does not require the construction of the graph H. The rest of the chapter is dedicated to the presentation and analysis of this technique.

Besides, note that the method presented in [Evans and Lambiotte, 2009] do not directly applies to weighted graphs or graphs with self-loops. In the following, we consider general weighted graphs that can contain self-loops.

3.5 An agglomerative algorithm for maximizing edge modularity

We have seen above that the edge modularity can be expressed as the standard modularity in a weighted line graph H. Therefore, a natural approach for maximizing the edge modularity consists in building the graph H corresponding to G using Proposition 3.3.2 and applying the Louvain algorithm to this new graph. However there is an important limit to this naive approach. Indeed, building the graph H is prohibitive in practice for large graphs G as the number of edges in H is

$$\left(\sum_{u\in V}(d_u)^2 - m\right)/2,$$

where d_u denotes the unweighted degree of a node u in G and m denotes the number of edges in G. This number typically explodes as the sizes of the considered graphs increase, because node degrees follow scale-free power-law distributions in most of real-life graphs [Barabási and Albert, 1999].

In this section, we define an algorithm that does not require the construction of the weighted line graph H. This algorithm is based on an aggregation scheme that preserves the edge modularity (which is the central result of this section) and that can be easily interpreted.

3.5.1 Agglomerative step

In order to define our aggregation scheme, we define two sets of nodes for each edge cluster: the internal nodes and the border nodes.

Definition 3.5.1. *Given a partition of the edges of G, $\mathcal{C} = \{C_1, \dots, C_K\}$, we define the internal nodes for cluster C_k as*

$$\mathcal{I}(k) = \{u \in V : \forall e \in E, u \in e \Rightarrow e \in C_k\}$$

and the border nodes for cluster C_k as

$$\mathcal{B}(k) = \{u \in V : \exists e \in C_k, \exists f \notin C_k, u \in e \cap f\}.$$

In other words, the internal nodes correspond to nodes that are bound to edges that all belong to the same cluster, whereas border nodes are bound to edges that belong to distinct clusters. These sets of nodes follow the following properties, for all k:

$$\text{(a) } \mathcal{B}(k) \cap \mathcal{I}(k) = \emptyset \quad \text{(b) } \forall u \in \mathcal{B}(k), \exists l \neq k, u \in \mathcal{B}(l) \quad \text{(c) } \forall l \neq k, I(k) \cap I(l) = \emptyset.$$

In the following definition, we consider an aggregation operation that consists in aggregating the internal nodes for each cluster C_k into a single node v_k with a self-loop.

Definition 3.5.2. *Given a graph G and an edge partition $\mathcal{C} = \{C_1, \dots, C_K\}$, we define the* aggregated graph relative to G and $\mathcal{C}$ *as $G' = (V', E')$ with the set of node*

$$V' = \left(\bigcup_k \mathcal{B}(k)\right) \cup \{v_k : \mathcal{I}(k) \neq \emptyset\},$$

where $v_k = \mathcal{I}(k)$. The weighted edges E' of G' are defined by the adjacency matrix A'

$$\forall u, v \in V', \quad A'_{uv} = \begin{cases} A_{uv} & \text{if } u, v \in \mathcal{B}(k) \\ \sum_{u', v' \in \mathcal{I}(k)} A_{u'v'} & \text{if } u = v = v_k \\ \sum_{v' \in \mathcal{I}(k)} A_{uv'} & \text{if } u \in \mathcal{B}(k),\ v = v_k \\ \sum_{u' \in \mathcal{I}(k)} A_{u'v} & \text{if } u = v_k,\ v \in \mathcal{B}(k) \\ 0 & \text{otherwise.} \end{cases}$$

We also define the corresponding aggregated partition $\mathcal{C}' = \{C'_1, \dots, C'_K\}$ as

$$\begin{aligned} C'_k =& \{\{u, v\} \in C_k : u, v \in \mathcal{B}(k)\} \cup \{\{u, v_k\} : u \in \mathcal{B}(k), \exists v \in \mathcal{I}(k), \{u, v\} \in C_k\} \\ & \cup \{\{v_k, v_k\} : A'_{v_k v_k} > 0\}. \end{aligned}$$

In other words, this aggregation operation can be decomposed into two steps. (i) For each cluster C_k, all the internal nodes are aggregated into a single node v_k bearing a self-loop of weight $\frac{1}{2}\sum_{u,v \in \mathcal{I}(k)} A_{uv}$. (ii) For each border node $u \in \mathcal{B}(k)$, an edge between u and v_k is created if $\exists v \in \mathcal{I}(k)$ s.t. $\{u, v\} \in E$. The weight of this edge is then $\sum_{v \in \mathcal{I}(k)} A_{uv}$. Note that the result of this aggregation operation is easy to interpret. Indeed, the nodes in the new graph G' correspond to single nodes or groups of nodes in the original graph, and the edge weights in G' correspond to the interactions between these groups of nodes (or to internal interactions for self-loops). We will illustrate this point on real-life data in Section 3.6.

The following theorem states that the edge modularity is invariant under this aggregation operation.

Theorem 3.5.3. *The edge modularity $\tilde{Q}(\mathcal{C})$ of the edge partition $\mathcal{C}$ in G is equal to the edge modularity $\tilde{Q}(\mathcal{C}')$ of the edge partition $\mathcal{C}'$ in the aggregated graph G'.*

Proof. We use the formulation (3.3) to prove this result. On the one hand, it is easy to verify that we have $w'(C'_k) = w(C_k)$ and $w'(E') = w(E)$ using Definition 3.5.2, where w' denotes the weights in the new graph G'. On the other hand, similar calculations lead to $w'_u(C'_k) = w_u(C_k)$ and $w'_u = w_u$ for all nodes $u \in \mathcal{B}(k)$ for all k. Therefore, we only need to prove that

$$\sum_k \sum_{u \in \mathcal{I}(k)} \frac{w_u(C_k)^2}{w_u} = \sum_k \frac{w'_{v_k}(C'_k)^2}{w'_{v_k}}. \tag{3.7}$$

We remark that, for all k, for all $u \in \mathcal{I}(k)$, $w_u(C_k) = w_u$ and $w'_{v_k}(C'_k) = w'_{v_k}$. Thus (3.7) is equivalent to $\sum_{u \in \mathcal{I}(k)} w_u = w'_{v_k}$, which follows from Definition 3.5.2. □

Thanks to this key conservation result, we can use the same strategy as the Louvain algorithm to optimize the edge modularity, i.e. alternate greedy maximization of $\tilde{Q}$ and the agglomerative operation described above.

3.5.2 Greedy maximization step

The simplest greedy method to maximize the edge modularity consists in cycling over the graph edges and, for each edge, move it to the neighboring cluster that leads to the highest increase in edge modularity.

However, this greedy maximization strategy is only applicable to the first step of the algorithm when the graph has not been aggregated. Indeed, the aggregation operation generally yields to aggregated clusters C'_k that contain more than one edge. The edges in these clusters must then be moved all together in order for the modularity conservation result from Theorem 3.5.3 to apply. To handle this issue, we consider a method that consists in cycling through the cluster C'_k returned by the aggregation operation, and moves the edges of each of these clusters in groups. We move all the edges of such a group to the neighboring cluster that results to the largest increase in edge modularity.

The variation of edge modularity when a group of edges $M \subset E$ is moved from one cluster C_k to another cluster C_l is given by

$$\Delta\tilde{Q}(M : C_k \to C_l) = \sum_{u : w_u(M) > 0} \frac{w_u(M)}{w(E) w_u} (w_u(M) + w_u(C_l) - w_u(C_k)) - \frac{2w(M)(w(M) + w(C_l) - w(C_k)}{w(E)^2}.$$

3.5.3 Algorithm

In Algorithm 2, we give the pseudo-code of the algorithm that iteratively repeats the greedy maximization step and the aggregation operation described above. At initial time, the algorithm puts each edge in its own cluster. Like the Louvain algorithm, the algorithm takes a unique parameter $\epsilon > 0$. The greedy maximization step and the aggregation step are repeated until the modularity increase is lower than this sensibility threshold ϵ.

In order to compute the increase $\Delta\tilde{Q}(M : C_k \to C_l)$ and to perform the aggregation operation in an efficient manner, the algorithm can store the weights $w(C_k)$, $w_u(C_k)$, and the sets $\mathcal{B}(C_k)$ and $\mathcal{I}(C_k)$ for each cluster C_k.

3.6 Experimental results

3.6.1 Synthetic data

As in Chapter 2, we use a stochastic block model (SBM) with overlapping blocks to evaluate the performance of our algorithm on synthetic data. Given a set $\{B_1, \ldots, B_I\}$ of node blocks $B_i \subset V$, we consider the random graph model where the edge $\{u, v\}$ appears in the graph with probability p_{in} if u and v belongs to the same block B_i, and with probability p_{out} otherwise. The probabilities p_{in} and p_{out} are parameters of our model. The blocks that we consider have the same size $c > 0$ ($\forall i, |B_i| = c$) and overlap as follows: for all i, $|B_i \cap B_{i+1}| = o$ and $B_i \cap B_j = \emptyset$ if $j > i + 1$.

We compare our algorithm to the algorithms that consist in building the line graphs defined by Evans and Lambiotte C, D and E_1, and applying the Louvain algorithm to these graphs. We refer to the algorithm that corresponds to the graph C as EL(C) (and respectively we call EL(D) and EL(E_1) the algorithms that correspond to the graphs D and E_1). All these algorithms have been implemented in Python and will be made available online.

First, we compare the execution time and the memory consumption of these algorithms on multiple instances of our random graph model for different numbers of blocks I. The other parameters of the model are set to $p_{\text{in}} = 0.95$, $p_{\text{out}} = 0.05$, $c = 10$ and $o = 2$. We display the results in Figure 3.1. We see that our algorithm runs on average more than 5 times faster than EL(C), EL(D) and EL(E_1). Besides,

Algorithm 2 Agglomerative edge modularity optimization algorithm

Require: Graph G and sensibility threshold $\epsilon \geq 0$.
  Initialization: $\mathcal{C} \leftarrow \{\{e\}, e \in E\}$ (edge partition) and $\mathcal{M} \leftarrow \mathcal{C}$ (groups of edges to consider)
  $\tilde{Q}_{\text{old}} \leftarrow -\infty$, $\tilde{Q}_{\text{new}} \leftarrow \tilde{Q}(\mathcal{C})$
  while $\tilde{Q}_{\text{new}} - \tilde{Q}_{\text{old}} > \epsilon$ **do**
    Greedy maximization step
    increase $\leftarrow$ true
    while increase **do**
      increase $\leftarrow$ false
      for $M \in \mathcal{M}$ **do**
        $C_k \leftarrow$ current cluster of M in $\mathcal{C}$
        Compute $\Delta\tilde{Q}(M : C_k \to C_l)$ for neighboring clusters C_l in $\mathcal{C}$
        $l^* \leftarrow \arg\max_l \Delta\tilde{Q}(M : C_k \to C_l)$
        if $\Delta\tilde{Q}(M : C_k \to C_{l^*}) > \epsilon$ **then**
          increase $\leftarrow$ true
          Modifiy $\mathcal{C}$ by moving M from cluster C_k to C_{l^*}
        end if
      end for
    end while
    Aggregation step
    $G, \mathcal{C} \leftarrow$ aggregated graph and corresponding partition given by Definition 3.5.2
    $\mathcal{M} \leftarrow \mathcal{C}$, $\tilde{Q}_{\text{old}} \leftarrow \tilde{Q}_{\text{new}}$, $\tilde{Q}_{\text{new}} \leftarrow \tilde{Q}(\mathcal{C})$
  end while

our algorithm consumes on average 10 times less memory than EL(C) and EL(D) and almost 100 times less memory than EL(E_1). This can be easily explained by the fact that the size of the line graphs considered by the approach of Evans and Lambiotte is much larger than the size of the original graph G.

In order to compare the quality of the results of the different algorithms, we define the notion of estimated blocks $\hat{B}_k$ for the different algorithms. If $C_1, \ldots, C_K$ are the edge clusters returned by one algorithm, we define the corresponding estimated blocks for this algorithm as

$$\forall k, \quad \hat{B}_k = \left\{ u \in V : \frac{w_u(C_k)}{w_u} \geq \alpha \right\}$$

where $\alpha \in [0, 1]$ is a given parameter. Then, we use the F_1-score to compare the original blocks (B_i) with the estimated blocks ($\hat{B}_k$) for the different algorithms. The F_1-score is equal to 1 if and only if the blocks ($\hat{B}_k$) exactly correspond to the planted blocks (B_i).

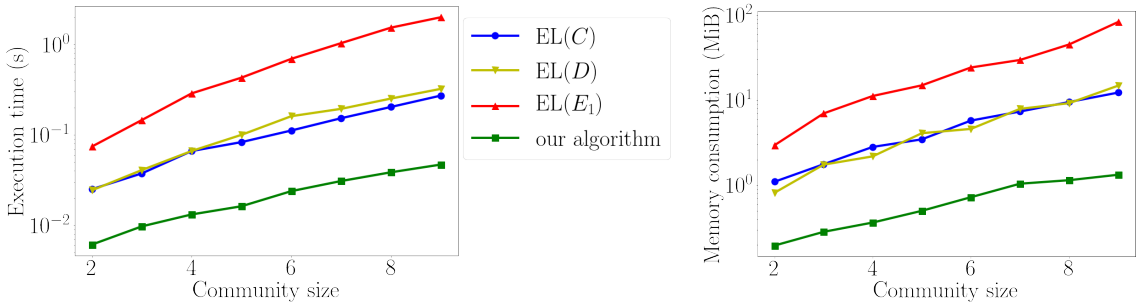
In Figure 3.1, we plot the average F_1-scores for values of α ranging from 0 to 1. We see that the F_1-score for our algorithm is on average 50% higher than the score observed for EL(C) and is very comparable to the score obtained by EL(D) (within 1% on average). This score is slightly lower (3% on average) than the score achieved by EL(E_1). However, we see that the benefit of EL(E_1) is marginal knowing that the memory consumption of this algorithm is almost 100 times higher than the one measured for our algorithm.

3.6.2 Real-life datasets

OpenFlights

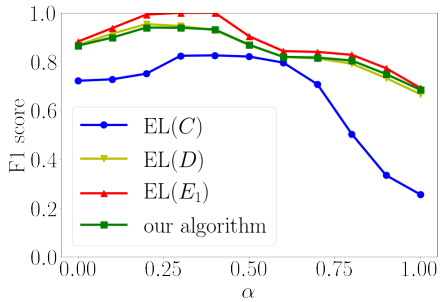
We first use the OpenFlights Airport Database[1] for our benchmarks on real-life data. We consider the graph whose nodes correspond to airports and whose edges correspond to airline routes between these airports. We run our algorithm and EL(C), EL(D) and EL(E_1) on this graph that contains 3,097 nodes and 18,193 edges. Our algorithm runs in 16.9 seconds and uses 15 MiB of memory, whereas EL(C) and EL(D) run respectively in 95 and $7.2 \cdot 10^3$ seconds and use 817 and 815 MiB of memory. The execution EL(E_1) raises a out-of-memory error on a computer with 16GB of RAM.

[1] `https://openflights.org`



(a) Execution time in seconds

(b) Memory consumption in MiB



(c) F_1-score

Figure 3.1: **Overlapping SBM** - Execution times, memory consumption and F_1-scores

Cluster 1	Frankfurt am Main, Charles de Gaulle, Amsterdam Schiphol, Munich, Barcelona
Cluster 2	Hartsfield Jackson Atlanta, Chicago O'Hare, Dallas Fort Worth, Houston, Denver
Cluster 3	Atatürk, Dubai, Suvarnabhumi, Kuala Lumpur, King Abdulaziz
Cluster 4	Domodedovo, Sheremetyevo, Pulkovo, Ben Gurion, Vnukovo
Cluster 5	Mohammed V, OR Tambo, Jomo Kenyatta, Murtala Muhammed, Quatro de Fevereiro

Figure 3.2: **OpenFlights** - Edge clusters around London Heathrow

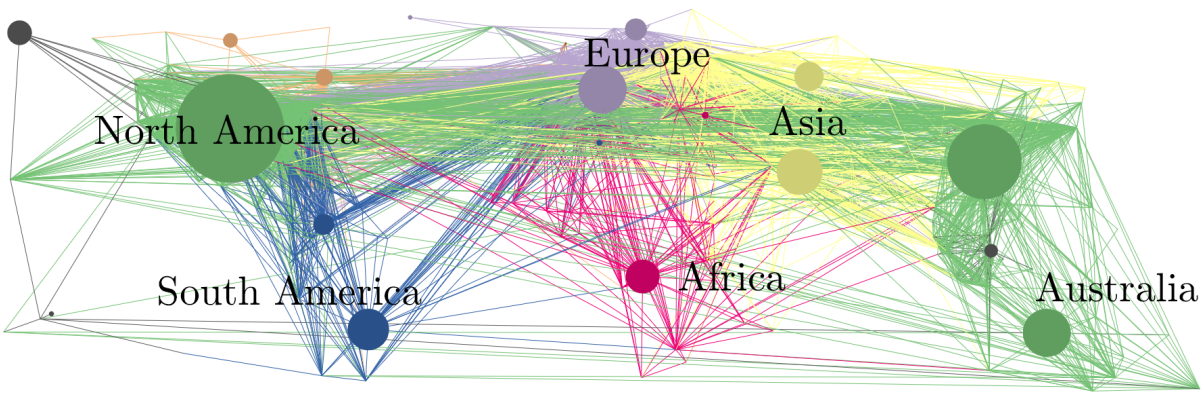


Figure 3.3: **OpenFlights** - Aggregated graph

			Run time (in seconds)				F_1-scores			
	# nodes	# edges	Louvain	Infomap	Walktrap	Ours	Louvain	Infomap	Walktrap	Ours
Amazon	$335 \cdot 10^3$	$926 \cdot 10^3$	12.7	31.8	261	16.2	0.29	0.30	0.39	0.42
DBLP	$317 \cdot 10^3$	$1050 \cdot 10^3$	14.1	27.6	1725	26.2	0.14	0.10	0.22	0.25

Table 3.1: Benchmarks on two SNAP datasets

To illustrate the results of our algorithm on this dataset, we first focus on the node corresponding to the London Heathrow Airport and consider its adjacent edges. We are interested in the clusters in which these adjacent edges have been categorized by our algorithm. In Figure 3.2, we list the end-nodes of these edges (i.e. the reachable airports from London Heathrow) and categorize them by edge clusters. We only display the first 5 clusters in terms of number of edges and we only list the top 5 airports in term of degree in the graph for each edge cluster. We see that the different clusters correspond to flights to different regions of the world (Europe for cluster 1, North America for cluster 2, Middle East and South Asia for cluster 3 etc.).

In addition to the clusters returned by our algorithm, the aggregated graph itself is very useful to have a synthetic view of the world airline traffic. This graph only contains 426 nodes, i.e. almost 10 times less nodes than the original graph, and 6,940 edges. We display this graph in Figure 3.3. We use edges of different colors to indicate the different edge clusters. We use colored circles to represent aggregated nodes, with a size proportional to the number of nodes that have been aggregated and a color corresponding to the edge cluster to which the aggregated nodes are connected. We position the aggregated nodes at the centroid of the airports that have been merged. We see that these aggregated nodes correspond to important regions of the world (North America, Central America, South America, Central Europe, North Europe etc.). Besides, we observe that the airports that have not been aggregated correspond to interfaces between these different regions of the world.

Wikipedia subset

In order to compare the performance of the algorithms on a larger graph, we run them on a subgraph of the Wikipedia graph that contain 4,589 nodes and 106,534 edges. In this graph, nodes correspond to Wikipedia articles and edges to hyperlinks between those articles. Whereas our algorithm runs in 281 seconds and use 229 MiB of memory on this graph, the three algorithms EL(C), EL(D) and EL(E_1) raise out-of-memory errors on a computer with 16 GB of RAM.

SNAP datasets

In order to provide more quantitative results, we perform experiments on SNAP datasets[2] with ground-truth communities. We use the Amazon co-purchasing network and the DBLP co-citation network datasets and compare our performance to two algorithms that are not based on modularity optimization: Walktrap [Pons and Latapy, 2005] and Infomap [Rosvall and Bergstrom, 2008] (that can be used for overlapping community detection). Note that we tried to run the EL(C), EL(D) and EL(E1) algorithms on these datasets, but they were not able to handle the graphs on a computer with 16GB of RAM. We report the results in Table 3.1. We see that our algorithm outperforms both the modularity-based Louvain algorithm and the other two algorithms, Infomap and Walktrap, in terms of F_1-score. Furthermore, our algorithm offers a competitive execution time compared to the other three algorithms.

[2] `https://snap.stanford.edu/data/`

Chapter 4

Hierarchical clustering

In this chapter, we introduce a novel hierarchical clustering method for graphs. The algorithm is agglomerative and based on a simple distance between clusters induced by the probability of sampling node pairs. We prove that this distance is reducible, which allows us to speed up the node aggregation through a technique known as the nearest-neighbor chain scheme. Our algorithm is parameter-free and provides a list of the most relevant resolutions of the graph. Furthermore, we show that this approach is tightly related to modularity, and that it can be used to choose the correct resolution parameter in the Louvain algorithm. This chapter is based on the work presented in [Bonald et al., 2018a].

4.1 Introduction

Real-life graphs, such as the ones presented in the introductory chapter, often exhibit complex, multi-scale cluster structures where each node is involved in many groups of nodes of different sizes. Classic graph clustering algorithms that output node partitions can only capture only one of the multiple scales in the cluster structure of such graphs. To overcome this problem, we need to use hierarchical clustering techniques that aim at finding trees of nested clusters, the so-called *hierarchies*. As seen in Chapter 1, modularity maximization does not directly enable an analysis of the graph at different scales. However, a resolution parameter can be introduced in the definition of modularity [Reichardt and Bornholdt, 2006, Lambiotte et al., 2008], but it is not directly related to the number of clusters or the cluster sizes and is therefore hard to adjust in practice.

In this chapter, we present a novel algorithm for hierarchical clustering that captures the multi-scale nature of real graphs. The algorithm is fast, memory-efficient and parameter-free. This approach is related to modularity optimization but does not require fine-tuning of the resolution parameter. It relies on a novel notion of distance between clusters induced by the probability of sampling node pairs. We prove that this distance is reducible, which guarantees that the resulting hierarchical clustering can be represented by regular dendrograms and enables a fast implementation of our algorithm through the nearest-neighbor chain scheme, a classical technique for agglomerative algorithms [Murtagh and Contreras, 2012].

The rest of the chapter is organized as follows. We present the related work in Section 4.2. We introduce classic agglomerative clustering techniques such as the nearest-neighbor chain algorithm in Section 4.3. The distance between clusters used to aggregate nodes and the clustering algorithm are presented in Sections 4.4 and 4.5. The link with modularity and the Louvain algorithm is explained in Section 4.6. Section 4.7 shows the experimental results.

4.2 Related work

As stressed in our introductory chapter, most graph clustering algorithms are not hierarchical and rely on some resolution parameter that allows one to adapt the clustering to the dataset and to the intended purpose [Reichardt and Bornholdt, 2006, Arenas et al., 2008, Lambiotte et al., 2014, Newman, 2016]. This parameter is hard to adjust in practice, which motivates the work presented in this chapter.

Usual hierarchical clustering techniques apply to vector data [Ward Jr, 1963, Murtagh and Contreras, 2012]. They do not directly apply to graphs, unless the graph is embedded in some metric space, through spectral

techniques such as the ones presented in Chapter 1 for instance [Von Luxburg, 2007, Donetti and Munoz, 2004].

A number of hierarchical clustering algorithms have been developed specifically for graphs. The most popular algorithms are agglomerative and characterized by some distance between clusters, see [Newman, 2004, Pons and Latapy, 2005, Huang et al., 2010, Chang et al., 2011]. None of these distances has been proved to be reducible, a key property of our algorithm. Among non-agglomerative algorithms, the divisive approach of [Newman and Girvan, 2004] is based on the notion of edge betweenness while the iterative approach of [Sales-Pardo et al., 2007] and [Lancichinetti et al., 2009] look for local maxima of modularity or some fitness function; other approaches rely on statistical inference [Clauset et al., 2008], replica correlations [Ronhovde and Nussinov, 2009] and graph wavelets [Tremblay and Borgnat, 2014]. To our knowledge, non of these algorithms has been proved to lead to regular dendrograms (that is, without inversion).

Finally, the Louvain algorithm also provides a hierarchy, induced by the successive aggregation steps of the algorithm [Blondel et al., 2008]. This is not a full hierarchy, however, as there are typically a few aggregation steps. Moreover, the same resolution is used in the optimization of modularity across all levels of the hierarchy, while the numbers of clusters decrease rapidly after a few aggregation steps. We shall see that our algorithm may be seen as a modified version of Louvain using a *sliding* resolution, that is adapted to the current agglomeration step of the algorithm.

4.3 Agglomerative clustering and nearest-neighbor chain

In this section, we present classic hierarchical clustering methods that do not traditionally apply to graph nodes but to observations in a vector space $\mathbb{R}^k$. In particular, we present a classic family of algorithms, the agglomerative algorithms, and we present the nearest-neighbor chain technique [Benzécri, 1982, Juan, 1982] that we use to cluster graph nodes in this chapter.

4.3.1 Agglomerative algorithms

We are interested in clustering a set of n observations $X = \{x_1, \dots, x_n\}$ in $\mathbb{R}^k$. Note that, even if these observations cannot directly be graph nodes, they can be the result of an embedding algorithms that maps nodes of G to vectors in $\mathbb{R}^k$ (e.g. spectral embedding algorithms, see Chapter 1). We assume that we are given a norm in $\|\cdot\|$ in $\mathbb{R}^k$ (typically the Euclidean norm, but it can also be any p-norm for instance). There exist two large families of hierarchical clustering algorithms that correspond to two approaches: the divisive approach and the agglomerative approach.

Divisive approach

In the *divisive* (or *top-down*) approach, we are typically given a *cut* function $(A, B) \mapsto \mathrm{cut}(A, B)$ that measures the quality of a split between two clusters $A, B \subset X$. A small value of $\mathrm{cut}(A, B)$ corresponds to two *distant* or *dissimilar* clusters A and B, whereas a large value of $\mathrm{cut}(A, B)$ corresponds to clusters that are *close* to each other. Divisive algorithms start from a situation where all observations are grouped in one cluster, and then iteratively split clusters until there are only clusters of size 1. The typical divisive algorithm splits the largest cluster C at each step into two subsets A and B, choosing the best cut, i.e. the smallest $\mathrm{cut}(A, B)$. More precisely, this greedy divisive algorithm can be described as follows.

- (Initialization) $\mathcal{C} \leftarrow \{\{x_1, \dots, x_n\}\}$
- While $\mathcal{C}$ contains a cluster with at least two observations.
  - $C \leftarrow \arg\max_{C' \in \mathcal{C}} |C'|$
  - $A, B \leftarrow \arg\min_{a,b \in \mathcal{C}: a \cup b = C} \mathrm{cut}(a, b)$
  - $\mathcal{C} \leftarrow \mathcal{C} \cup \{A, B\} \setminus \{C\}$
  - Output A, B and $\mathrm{cut}(A, B)$.

Note that, the maximum cluster size decreases at each step until we only obtain clusters with isolated nodes. We see that the algorithm does not only outputs one partition but a sequence of splits. The results that we observe at different steps of the algorithm correspond to different scales in the dataset.

Agglomerative approach

In the *agglomerative* (or *bottom-up*) approach, that we will adopt in this chapter, we start from individual clusters and merge iteratively the two *closest* clusters according to a distance function $(A, B) \mapsto d(A, B)$ that measures the dissimilarity between clusters. More precisely, the standard greedy agglomerative algorithm can be described as

- (Initialization) $\mathcal{C} \leftarrow \{\{x_1\}, \ldots, \{x_n\}\}$
- For $t = 1, \ldots, n-1$.
  - $A, B \leftarrow \arg\min_{a,b \in \mathcal{C}: a \neq b} d(a, b)$
  - $\mathcal{C} \leftarrow \mathcal{C} \cup \{C\} \setminus \{A, B\}$
  - Output A, B and $d(A, B)$.

Like the divisive method presented above, we see that the algorithm outputs at each step two clusters and the distance between these clusters. The result of this approach is typically represented as a dendrogram (see Figure 4.2). A dendrogram is a special tree whose leaves correspond to individual observations and whose other nodes correspond to the clusters C observed throughout the execution of the algorithm. A and B are connected to C in the dendrogram if A and B have been merged into $C = A \cup B$ during the execution of the algorithm. Moreover the position on the vertical axis of $C = A \cup B$ corresponds to the distance between A and B, $d(A, B)$. Therefore, the greater the difference in height, the more dissimilarity between clusters. We immediately see the interest of such approaches to study a dataset at multiple scales. Indeed, different levels in the dendrogram correspond to different scales in the dataset.

4.3.2 Classic distances

The result of the greedy agglomerative approach presented above depends on the choice of the distance function $d : (A, B) \mapsto d(A, B)$ to measure the dissimilarity between clusters $A, B \subset X$. The classic distance measures that are commonly used are

- the *minimum distance*, which leads to the so-called *single-linkage* clustering, defined as

$$d(A, B) = \min_{x \in A, y \in B} \|x - y\|;$$

- the *maximum distance*, which leads to the so-called *complete-linkage* clustering, defined as

$$d(A, B) = \max_{x \in A, y \in B} \|x - y\|;$$

- the *average distance*, which leads to the so-called *average-linkage* clustering, defined as

$$d(A, B) = \frac{1}{|A||B|} \sum_{x \in A} \sum_{y \in B} \|x - y\|.$$

These three distances have an important property in common: they are *reducible*. A distance measure between clusters is said to be *reducible* if, for any pair of mutual nearest neighbors A and B merged during the execution of the algorithm, and for all cluster C, we have

$$d(A \cup B) \geq \min(d(A, C), d(B, C)).$$

In other words, a distance measure is reducible if the distance between any merged cluster $A \cup B$ and any cluster C is larger than the individual distance between A and C, or the distance between B and C. This property guarantees that the sequence of distances $d(A, B)$ found during the algorithm is non-decreasing. Indeed, if d is reducible and if A and B are merged by the algorithm, then, for any other cluster $C \in \mathcal{C}$ we have

$$d(A \cup B, C) \geq \min(d(A, C), d(B, C)) \geq d(A, B) = \min_{a,b \in \mathcal{C}, a \neq b} d(a, b).$$

4.3.3 Ward's method

Another classic distance used in the agglomerative approach is Ward's distance [Ward Jr, 1963]. This distance arises from the analysis of the minimization problem where the objective is to minimize

$$J(\mathcal{C}) = \sum_{C \in \mathcal{C}} \sum_{x \in C} \|x - \mu(C)\|^2,$$

where $\mu(C)$ is the centroid of the cluster C

$$\mu(C) = \frac{1}{|C|} \sum_{x \in C} x.$$

We recognize the objective function of the k-means problem presented in Chapter 2 when the means correspond to the centroids of clusters. Ward's distance is defined as the cost of merging A and B. In other words, it is defined as $d(A, B) = J(\mathcal{C} \setminus \{C\} \cup \{A, B\}) - J(\mathcal{C})$. If we use $S(C)$ to denote $S(C) = \sum_{x \in C} \|x - \mu(C)\|^2$ so that $J(\mathcal{C}) = \sum_{C \in \mathcal{C}} S(C)$, then we have

$$d(A, B) = S(A \cup B) - S(A) - S(B) = \frac{|A||B|}{|A| + |B|} \|\mu(A) - \mu(B)\|^2.$$

The greedy agglomerative algorithm using this distance is commonly referred to as *Ward's method*.

4.3.4 The Lance-Williams family

The Lance-Williams method is an infinite class of agglomerative algorithms where we have a recursive formula for updating distances between clusters at each step. In a naive implementation of the greedy agglomerative approach described above, it is necessary to recompute pairwise distances between clusters after each merge, whereas, with the Lance-Williams method, we do not need to recompute these distances at each step. An agglomerative algorithm belongs to the Lance-Williams family if the updated distance $d(a \cup b, c)$ between a merged cluster $a \cup b$ and any cluster c can be written as

$$d(a \cup b, c) = \alpha d(a, c) + \beta d(b, c) + \gamma d(a, b) + \delta |d(a, c) - d(b, c)|,$$

where α, β, γ and δ are parameters that may depend on the cluster sizes. The single-linkage, complete-linkage and average-linkage algorithms belong to this family. Indeed, we have

- for the minimum distance: $d(a \cup b, c) = \min(d(a, c), d(b, c)) = \frac{d(a,c)+d(b,c)-|d(a,c)-d(b,c)|}{2}$;
- for the maximum distance: $d(a \cup b, c) = \max(d(a, c), d(b, c)) = \frac{d(a,c)+d(b,c)+|d(a,c)-d(b,c)|}{2}$;
- for the average distance: $d(a \cup b, c) = \frac{|a|d(a,c)+|b|d(b,c)}{|a|+|b|}$.

Ward's method also belongs to this family. Indeed, it can be shown that we have

$$d(a \cup b, c) = \frac{1}{|a| + |b| + |c|} \left[(|a| + |c|) d(a, c) + (|b| + |c|) d(b, c) - |c| d(a, b) \right].$$

From this formula, we can easily show that Ward's distance is reducible.

4.3.5 Nearest-neighbor chain algorithm

The greedy algorithm that we present above for agglomerative clustering, that consists in placing each observation in an isolated cluster and then iteratively merging the closest pair of clusters, is too expensive in space or time for large datasets [Eppstein, 2000, Day and Edelsbrunner, 1984]. Indeed, at each step of the algorithms, even if we use the Lance-Williams method to efficiently compute the new pairwise distances, we need to find the two closest clusters to merge. Existing methods to perform this task either require superlinear space to maintain a data structure in which we can find the clusters to merge efficiently, or require superlinear time to find the closest pair of clusters at each step.

The *nearest-neighbor chain* algorithm merges the clusters in a different order than the greedy approach in a way that avoid to search for the closest pair of clusters at each step. This algorithm starts for any observation $x \in X$ and builds the chain of nearest neighbors, which is directed, until two clusters are mutually nearest neighbors. When it finds such clusters A and B, it merges them and proceeds with the rest of the chain until the chain is empty. If the chain is empty and there are at least two clusters remaining, it restarts from any cluster. More formally, the algorithm can be described as follows.

- (Initialization of clusters) $\mathcal{C} \leftarrow \{\{x_1\}, \ldots, \{x_n\}\}$
- (Initialization of the chain) $S \leftarrow$ empty stack
- While $|\mathcal{C}| \geq 2$:
  - If $|S| = 0$, choose C in $\mathcal{C}$ arbitrarily and push C onto S.
  - $C \leftarrow$ cluster on the top of S.
  - Compute the distance $d(C, D)$ for all $D \in \mathcal{C}$.
  - $D \leftarrow \underset{D' \in \mathcal{C}}{\arg\min}\, d(C, D')$
  - If $D \in S$, pop C and D from S and merge them, i.e. $\mathcal{C} \leftarrow \mathcal{C} \cup \{C \cup D\} \setminus \{C, D\}$.
  - Otherwise push D onto S.

The time complexity of the algorithm is in $O(n^2)$ because the time to find the nearest cluster at each step is at most $(n-1)$ and each cluster is added only once to the chain. The space complexity is in $O(n)$, because the length of the chain is at most n.

As previously stated, this algorithm requires the distance d to be reducible. On the one hand, the fact that the chain S remains a nearest neighbor chain after merging the last two clusters that are jointly nearest neighbors relies on this property. Indeed, if the distance d is reducible, when we merge two clusters C and D, the resulting cluster $C \cup D$ cannot be the nearest neighbor of any other cluster E if one of the cluster C and D was not already the nearest neighbor of E because $d(C \cup D, E) \geq \min(d(C, E), d(D, E)) \geq \min_F d(E, F)$. Therefore, we see that, at each step, S forms a valid chain of nearest neighbors. On the other hand, the fact that d is reducible guarantees that the result obtained with the nearest-neighbor chain algorithm is the same as the one obtained with the greedy algorithm, even if the order in which the clusters are merged is in general different. Indeed, both clustering will only merge clusters that are mutually nearest neighbors, and such pairs of clusters remain mutual nearest neighbors until they are merged.

As a consequence the nearest neighbor chain algorithm can be used with the minimum, maximum and average distances and with Ward's distance. In this chapter, we define a reducible distance for graph nodes and use the nearest-neighbor chain algorithm to perform hierarchical clustering in graphs and show that this technique is closely related to the notion of modularity. We will see that this distance relies on node pair sampling.

4.4 Node pair sampling

As seen in Chapter 1, the modularity function can be interpreted using a probability distribution on node pairs induced by edge weights,

$$\forall i, j \in V, \quad p(i, j) = \frac{A_{ij}}{w},$$

and a probability distribution on nodes,

$$\forall i \in V, \quad p(i) = \sum_{j \in V} p(i, j) = \frac{w_i}{w}.$$

Observe that the joint distribution $p(i, j)$ depends on the graph (in particular, only *neighbors* i, j are sampled with positive probability), while the marginal distribution $p(i)$ depends on the graph through the node weights only.

We define the *distance* between two distinct nodes i, j as the node pair sampling ratio[1]:

$$d(i,j) = \frac{p(i)p(j)}{p(i,j)}, \tag{4.1}$$

with $d(i,j) = +\infty$ if $p(i,j) = 0$ (i.e., i and j are not neighbors). Nodes i, j are *close* with respect to this distance if the pair i, j is sampled much more frequently through the joint distribution $p(i,j)$ than through the product distribution $p(i)p(j)$. For unit weights, the joint distribution is uniform over the edges, so that the closest node pair is the pair of neighbors having the lowest degree product.

Another interpretation of the node distance d follows from the conditional probability,

$$\forall i, j \in V, \quad p(i|j) = \frac{p(i,j)}{p(j)} = \frac{A_{ij}}{w_j}.$$

This is the conditional probability of sampling i given that j is sampled (from the joint distribution). The distance between i and j can then be written

$$d(i,j) = \frac{p(i)}{p(i|j)} = \frac{p(j)}{p(j|i)}.$$

Nodes i, j are *close* with respect to this distance if i (respectively, j) is sampled much more frequently given that j is sampled (respectively, i).

Similarly, consider a clustering C of the graph (that is, a partition of V). The weights induce a probability distribution on cluster pairs,

$$\forall a, b \in C, \quad p(a,b) = \sum_{i \in a, j \in b} p(i,j),$$

and a probability distribution on clusters,

$$\forall a \in C, \quad p(a) = \sum_{i \in a} p(i) = \sum_{b \in C} p(a,b).$$

We define the distance between two distinct clusters a, b as the cluster pair sampling ratio:

$$d(a,b) = \frac{p(a)p(b)}{p(a,b)}, \tag{4.2}$$

with $d(a,b) = +\infty$ if $p(a,b) = 0$ (i.e., there is no edge between clusters a and b). Defining the conditional probability

$$\forall a, b \in C, \quad p(a|b) = \frac{p(a,b)}{p(b)},$$

which is the conditional probability of sampling a given that b is sampled, we get

$$d(a,b) = \frac{p(a)}{p(a|b)} = \frac{p(b)}{p(b|a)}.$$

This distance will be used in the agglomerative algorithm to merge the closest clusters. We have the following key results.

Proposition 4.4.1 (Update formula)**.** *For any distinct clusters $a, b, c \in C$,*

$$d(a \cup b, c) = \left(\frac{p(a)}{p(a \cup b)} \frac{1}{d(a,c)} + \frac{p(b)}{p(a \cup b)} \frac{1}{d(b,c)} \right)^{-1}.$$

Proof. We have:

$$\begin{aligned} p(a \cup b)p(c)d(a \cup b, c)^{-1} &= p(a \cup b, c), \\ &= p(a,c) + p(b,c), \\ &= p(a)p(c)d(a,c)^{-1} + p(b)p(c)d(b,c)^{-1}, \end{aligned}$$

from which the formula follows. □

[1] The distance d is not a metric in general. We only require symmetry and non-negativity.

Proposition 4.4.2 (Reducibility)**.** *For any distinct clusters $a, b, c \in C$,*

$$d(a \cup b, c) \geq \min(d(a, c), d(b, c)).$$

Proof. By Proposition 4.4.1, $d(a \cup b, c)$ is a weighted harmonic mean of $d(a, c)$ and $d(b, c)$, from which the inequality follows. □

By the reducibility property, merging clusters a and b cannot decrease their minimum distance to any other cluster c.

4.5 Clustering algorithm

For simplicity, and without restricting the generality of the results, we assume that $V = \{1, \ldots, n\}$. As explained above, the greedy agglomerative approach consists in starting from individual clusters (i.e., each node is in its own cluster) and merging clusters recursively. At each step of the algorithm, the two *closest* clusters are merged. For our distance d, we obtain the following algorithm:

1. (Initialization) $C \leftarrow \{\{1\}, \ldots, \{n\}\}$; $L \leftarrow \emptyset$
2. (Agglomeration) For $t = 1, \ldots, n-1$,
   - $a, b \leftarrow \arg\min_{a', b' \in C, a' \neq b'} d(a', b')$
   - $C \leftarrow C \setminus \{a, b\}$; $C \leftarrow C \cup \{a \cup b\}$
   - $L \leftarrow L \cup \{\{a, b\}\}$
3. Return L

The successive clusterings $C_0, C_1, \ldots, C_{n-1}$ produced by the algorithm, with $C_0 = \{\{1\}, \ldots, \{n\}\}$, can be recovered from the list L of successive merges. Observe that clustering C_t consists of $n - t$ clusters, for $t = 0, 1, \ldots, n-1$. By the reducibility property, the corresponding sequence of distances $d_0, d_1, \ldots, d_{n-1}$ between merged clusters, with $d_0 = 0$, is non-decreasing, resulting in a regular dendrogram (that is, without inversions) [Murtagh and Contreras, 2012].

It is worth noting that the graph G does not need to be connected. If the graph consists of k connected components, then the clustering C_{n-k} gives these k connected components, whose respective distances are infinite; the $k-1$ last merges can then be done in an arbitrary order. Moreover, the hierarchies associated with these connected components are independent of one another (i.e., the algorithm successively applied to the corresponding subgraphs would produce exactly the same clustering). Similarly, we expect the clustering of weakly connected subgraphs to be approximately independent of one another. This is not the case of the Louvain algorithm, whose clustering depends on the whole graph through the total weight w, a shortcoming related to the resolution limit of modularity underlined in Chapter 1.

Aggregate graph. In view of (4.2), for any clustering C of V, the distance $d(a, b)$ between two clusters $a, b \in C$ is the distance between two nodes a, b of the following aggregate graph: nodes are the elements of C and the weight between $a, b \in C$ (including the case $a = b$, corresponding to a self-loop) is $\sum_{i \in a, j \in b} A_{ij}$. Thus the agglomerative algorithm can be implemented by merging nodes and updating the weights (and thus the distances between nodes) at each step of the algorithm. Since the initial nodes of the graph are indexed from 0 to $n-1$, we index the cluster created at step t of the algorithm by $n + t$. We obtain the following version of the above algorithm, where the clusters are replaced by their respective indices:

1. (Initialization) $V \leftarrow \{1, \ldots, n\}$; $L \leftarrow \emptyset$
2. (Agglomeration) For $t = 1, \ldots, n-1$,
   - $i, j \leftarrow \arg\min_{i', j' \in V, i' \neq j'} d(i', j')$
   - $L \leftarrow L \cup \{\{i, j\}\}$
   - $V \leftarrow V \setminus \{i, j\}$; $V \leftarrow V \cup \{n + t\}$
   - $p(n + t) \leftarrow p(i) + p(j)$
   - $p(n + t, u) \leftarrow p(i, u) + p(j, u)$ for $u \in V \setminus \{n \cup t\}$
3. Return L

Nearest-neighbor chain. By the reducibility property of the distance, the algorithm can be implemented through the nearest-neighbor chain scheme described above. As explained in Section 4.3, this scheme reduces the search of a global minimum (the pair of nodes i, j that minimizes $d(i, j)$) to that of a local minimum (any pair of nodes i, j such that $d(i, j) = \min_{j'} d(i, j') = \min_{i'} d(i', j)$), which speeds up the algorithm while returning exactly the *same* hierarchy. It only requires a consistent tie-breaking rule for equal distances (e.g., any node at equal distance of i and j is considered as closer to i if and only if $i < j$). Observe that the space complexity of the algorithm is in $O(m)$, where m is the number of edges of G (i.e., the graph size).

4.6 Link with modularity

As described in our introductory chapter, the modularity measure can be written in terms of probability distributions,

$$Q(C) = \sum_{i,j \in V} (p(i, j) - p(i)p(j))\delta_C(i, j).$$

Thus the modularity is the difference between the probabilities of sampling two nodes of the same cluster under the joint distribution $p(i, j)$ and under the product distribution $p(i)p(j)$. It can also be expressed from the probability distributions at the cluster level,

$$Q(C) = \sum_{a \in C} (p(a, a) - p(a)^2).$$

As stressed in Chapter 1, the modularity has some resolution limit, because its second term is normalized by the total weight w and thus becomes negligible for too small clusters. To go beyond this resolution limit, we have seen that it is necessary to introduce a multiplicative factor γ, called the resolution to define a *generalized* modularity. The generalized modularity in the node pair sampling interpretation becomes:

$$Q_\gamma(C) = \sum_{i,j \in V} (p(i, j) - \gamma p(i)p(j))\delta_C(i, j), \tag{4.3}$$

or equivalently,

$$Q_\gamma(C) = \sum_{a \in C} (p(a, a) - \gamma p(a)^2).$$

We have seen that this resolution parameter can be interpreted in different ways: through the Potts model of statistical physics [Reichardt and Bornholdt, 2006], random walks [Lambiotte et al., 2014], or statistical inference of a stochastic block model [Newman, 2016]. For $\gamma = 0$, the resolution is minimum and there is a single cluster, that is $C = \{\{1, \ldots, n\}\}$; for $\gamma \to +\infty$, the resolution is maximum and each node has its own cluster, that is $C = \{\{1\}, \ldots, \{n\}\}$.

The Louvain algorithm can be easily adapted to optimize the generalized modularity for any *fixed* resolution parameter γ with the following steps:

1. (Initialization) $C \leftarrow \{\{1\}, \ldots, \{n\}\}$
2. (Iteration) While modularity $Q_\gamma(C)$ increases, update C by moving one node from one cluster to another.
3. (Aggregation) Merge all nodes belonging to the same cluster, update the weights and apply step 2 to the resulting aggregate graph while modularity is increased.
4. Return C

Our algorithm can be viewed as a modularity-maximizing scheme with a *sliding* resolution. Starting from the maximum resolution where each node has its own cluster, we look for the first value of the resolution parameter γ, say γ_1, that triggers a single merge between two nodes, resulting in clustering C_1. In view of (4.3), we have:

$$\gamma_1 = \max_{i,j \in V} \frac{p(i, j)}{p(i)p(j)}.$$

These two nodes are merged (corresponding to the aggregation phase of the Louvain algorithm) and we look for the next value of the resolution parameter, say γ_2, that triggers a single merge between two nodes, resulting in clustering C_2, and so on. By construction, the resolution at time t (that triggers the t-th merge) is $\gamma_t = 1/d_t$ and the corresponding clustering C_t is that of our algorithm. In particular, the sequence of resolutions $\gamma_1, \ldots, \gamma_{n-1}$ is non-increasing.

To summarize, our algorithm consists of a simple but deep modification of the Louvain algorithm, where the iterative step (step 2) is replaced by a single merge, at the best current resolution (that resulting in a single merge). In particular, unlike the Louvain algorithm, our algorithm provides a full hierarchy. Moreover, the sequence of resolutions $\gamma_1, \ldots, \gamma_{n-1}$ can be used as an input to the Louvain algorithm. Specifically, the resolution γ_t provides exactly $n - t$ clusters in our case, and the Louvain algorithm is expected to provide approximately the same number of clusters at this resolution.

4.7 Experiments

Our hierarchical clustering algorithm, called Paris (Pairwise node Agglomeration with Resolution Incremental Sliding), is available as a Python package[2]. The experiments presented below are presented for illustrative purpose only as, to the best of our knowledge, there is no standard approach or benchmark to compare hierarchical clustering algorithms.

Synthetic data. We start with a simple hierarchical stochastic block model, as described in [Lyzinski et al., 2017]. There are $n = 160$ nodes structured in 2 levels, with 4 blocks of 40 nodes at level 1, each block of 40 nodes being divided into 4 blocks of 10 nodes at level 2 (see Figure 4.1).

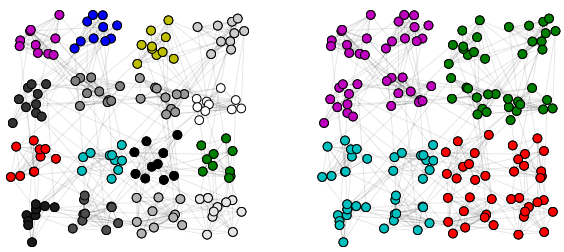


Figure 4.1: A hierachical stochastic block model.

The output of Paris algorithm is shown in Figure 4.2 as a dendrogram where the distances (on the y-axis) are in log-scale. The two levels of hierarchy clearly appear.

We give in Figure 4.3 the number of clusters with respect to the resolution parameter γ for Paris (left) and Louvain (right). The results are very close, and clearly show the hierarchical structure of the model (vertical lines correspond to changes in the number of clusters). The key difference between both algorithms is that, while Louvain needs to be run for *each* resolution parameter γ (here 100 values ranging from 0.01 to 20), Paris is run only once, the relevant resolutions being direct outputs of the algorithm.

Real data. We now consider four real datasets, whose characteristics are summarized in Table 4.1.

Graph	# nodes	# edges
OpenStreet	5,993	6,958
OpenFlights	3,097	18,193
Wikipedia Schools	4,589	106,644
Wikipedia Humans	702,782	3,247,884

Table 4.1: Summary of the datasets

[2]See https://github.com/Sharpenb/python_paris

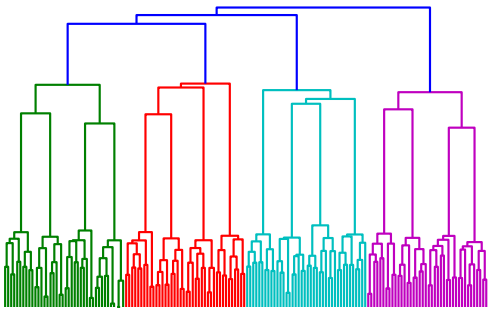


Figure 4.2: Dendrogram associated with the clustering of Paris on a hierachical stochastic block model of 16 blocks.

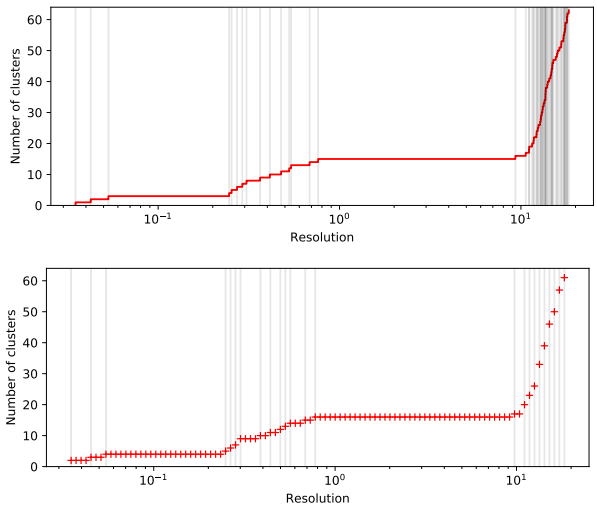


Figure 4.3: Number of clusters with respect to the resolution parameter γ for Paris (left) and Louvain (right) on the hierachical stochastic block model of Figure 4.1.

The first dataset, extracted from OpenStreetMap, is the graph formed by the streets of the center of Paris. To illustrate the quality of the hierarchical clustering returned by our algorithm, we have extracted the two "best" clusterings, in terms of ratio between successive distance merges in the corresponding dendrogram; the results are shown in Figure 4.4. The best clustering gives two clusters, Rive Droite (with Ile de la Cité) and Rive Gauche, the two banks separated by the river Seine; the second best clustering divides these two clusters into sub-clusters.

The second dataset, extracted from OpenFlights, is the graph of airports with the weight between two airports equal to the number of daily flights between them. We run Paris and extract the best clusterings from the largest component of the graph, as for the OpenStreet graph. The first two best clusterings isolate the Island/Groenland area and the Alaska from the rest of the world, the corresponding airports forming dense clusters, lightly connected with the other airports. The following two best clusterings are shown in Figure 4.5, with respectively 5 and 10 clusters corresponding to meaningful continental regions of the world.

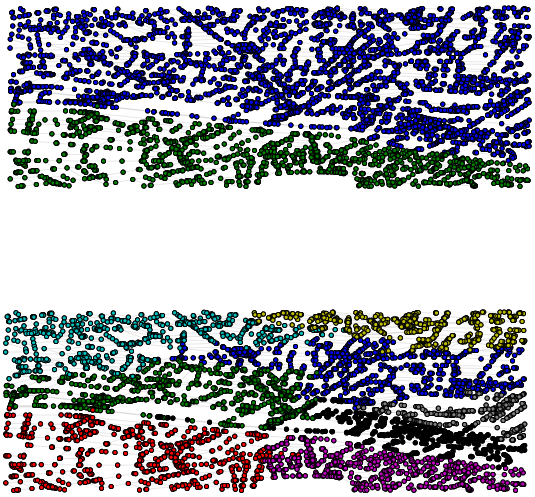


Figure 4.4: Clusterings of the OpenStreet graph by Paris.

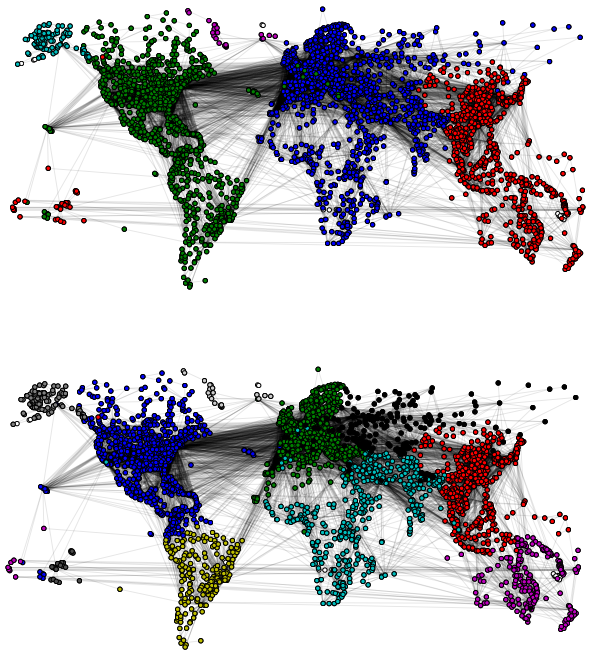


Figure 4.5: Clusterings of the OpenFlights graph by Paris.

Size	Main pages
288	Scientific classification, Animal, Chordate, Binomial nomenclature, Bird,...
231	Iron, Oxygen, Electron, Hydrogen, Phase (matter),...
196	England, Wales, Elizabeth II of the United Kingdom, Winston Churchill, William Shakespeare,...
164	Physics, Mathematics, Science, Albert Einstein, Electricity,...
148	Portugal, Ethiopia, Mozambique, Madagascar, Sudan,...
139	Washington, D.C., President of the United States, American Civil War, Puerto Rico, Bill Clinton,...
129	Earth, Sun, Astronomy, Star, Gravitation,...
127	Plant, Fruit, Sugar, Tea, Flower,...
104	Internet, Computer, Mass media, Latin alphabet, Advertising,...
99	Jamaica, The Beatles, Hip hop music, Jazz, Piano,...

Table 4.2: The 10 largest clusters of Wikipedia for Schools among 100 clusters found by Paris.

The third dataset is the graph formed by links between pages of Wikipedia for Schools[3], see [West et al., 2009, West and Leskovec, 2012]. The graph is considered as undirected. Table 4.2 shows the 10 largest clusters of C_{n-100}, the 100 last clusters found by Paris. Only pages of highest degrees are shown for each cluster.

Observe that the ability of selecting the clustering associated with some target number of clusters is one of the key advantage of Paris over Louvain. Moreover, Paris gives a full hiearchy of the pages, meaning that each of these clusters is divided into sub-clusters in the output of the algorithm. Table 4.3 gives for instance, among the 500 clusters found by Paris (that is, in C_{n-500}), the 10 largest clusters that are subclusters of the first cluster of Table 4.2, related to animals / taxonomy. The subclusters tend to give meaningful groups of animals, revealing the multi-scale structure of the dataset.

Size	Main pages
71	Dinosaur, Fossil, Reptile, Cretaceous, Jurassic,...
51	Binomial nomenclature, Bird, Carolus Linnaeus, Insect, Bird migration,...
24	Mammal, Lion, Cheetah, Giraffe, Nairobi,...
22	Animal, Ant, Arthropod, Spider, Bee,...
18	Dog, Bat, Vampire, George Byron, 6th Baron Byron, Bear,...
16	Eagle, Glacier National Park (US), Golden Eagle, Bald Eagle, Bird of prey,...
16	Chordate, Parrot, Gull, Surtsey, Herring Gull,...
15	Feather, Extinct birds, Mount Rushmore, Cormorant, Woodpecker,...
13	Miocene, Eocene, Bryce Canyon National Park, Beaver, Radhanite,...
12	Crow, Dove, Pigeon, Rock Pigeon, Paleocene,...

Table 4.3: The 10 largest subclusters of the first cluster of Table 4.2 among 500 clusters found by Paris.

The fourth dataset is the subgraph of Wikipedia restricted to pages related to humans. We have done the same experiment as for Wikipedia for Schools. The results are shown in Tables 4.4-4.5.

Finally, we give in Table 4.6 the running times of Louvain, Paris, and a spectral algorithm[4], for each of the 4 datasets. The experiments have been conducted on a 2.7GHz Intel Core i5 CPU with 8GB of RAM. We observe that Paris is faster than Louvain, while producing a much richer information on the graph (hierarchical clustering vs. flat clustering); it is faster than the spectral algorithm in most cases as well, especially for large graphs (the spectral algorithm did not complete on Wikipedia Humans).

[3]https://schools-wikipedia.org

[4]The algorithm gives a hierarchical clustering using the Ward method applied to the spectral embedding of the graph, based on the 20 leading eigenvectors of the Laplacian matrix; the implementation is based on the Python package `scipy`.

Size	Main pages
41363	George W. Bush, Barack Obama, Bill Clinton, Ronald Reagan, Richard Nixon,...
34291	Alex Ferguson, David Beckham, Pelé, Diego Maradona, José Mourinho,...
25225	Abraham Lincoln, George Washington, Ulysses S. Grant, Thomas Jefferson, Edgar Allan Poe,...
23488	Madonna, Woody Allen, Martin Scorsese, Tennessee Williams, Stephen Sondheim,...
23044	Wolfgang Amadeus Mozart, Johann Sebastian Bach, Ludwig van Beethoven, Richard Wagner, Giuseppe Verdi,...
22236	Elvis Presley, Bob Dylan, Elton John, David Bowie, Paul McCartney,...
20429	Queen Victoria, George III of the UK, Edward VII, Charles Dickens, Charles, Prince of Wales,...
19105	Sting, Jawaharlal Nehru, Rabindranath Tagore, Indira Gandhi, Gautama Buddha,...
18348	Edward I of England, Edward III of England, Henry II of England, Charlemagne, Henry III of England,...
14668	Jack Kemp, Brett Favre, Peyton Manning, Walter Camp, Tom Brady,...

Table 4.4: The 10 largest clusters of Wikipedia Humans among 100 clusters found by Paris.

Size	Main pages
2722	Barack Obama, John McCain, Dick Cheney, Newt Gingrich, Nancy Pelosi,...
2443	Arnold Schwarzenegger, Jerry Brown, Ralph Nader, Dolph Lundgren, Earl Warren,...
2058	Osama bin Laden, Hamid Karzai, Alberto Gonzales, Janet Reno, Khalid Sheikh Mohammed,...
1917	Dwight D. Eisenhower, Harry S. Truman, Douglas MacArthur, George S. Patton, Charles Lindbergh,...
1742	George W. Bush, Condoleezza Rice, Colin Powell, Donald Rumsfeld, Karl Rove,...
1700	Bill Clinton, Thurgood Marshall, Mike Huckabee, John Roberts, William Rehnquist,...
1559	Ed Rendell, Arlen Specter, Rick Santorum, Tom Ridge, Mark B. Cohen,...
1545	Theodore Roosevelt, Herbert Hoover, William Howard Taft, Calvin Coolidge, Warren G. Harding,...
1523	Ronald Reagan, Richard Nixon, Jimmy Carter, Gerald Ford, J. Edgar Hoover,...
1508	Rudy Giuliani, Michael Bloomberg, Nelson Rockefeller, George Pataki, Eliot Spitzer,...

Table 4.5: The 10 largest subclusters of the first cluster of Table 4.4 among 500 clusters found by Paris.

Graph	Louvain	Paris	Spectral
OpenStreet	1.25	0.69	1.55
OpenFlights	1.77	1.15	0.55
Wikipedia Schools	13.1	5.8	10.8
Wikipedia Humans	1438	433	-

Table 4.6: Mean running times over 10 runs (in seconds).

Chapter 5

Streaming approaches

In this chapter, we study streaming approaches for graph clustering. We introduce a novel algorithm to perform graph clustering in the edge streaming setting. In this model, the graph is presented as a sequence of edges that can be processed strictly once. Our streaming algorithm has an extremely low memory footprint as it stores only three integers per node and does not keep any edge in memory. We provide a theoretical justification of the design of the algorithm based on the *modularity* function. We perform experiments on massive real-life graphs ranging from one million to more than one billion edges and we show that this new algorithm runs more than ten times faster than existing algorithms. This chapter is based on the work presented in [Hollocou et al., 2017b].

5.1 Introduction

A major challenge for graph clustering algorithms is their ability to process very large graphs that are commonly observed in numerous fields. For instance, social networks have typically millions of nodes and billions of edges (e.g. Friendster [Mislove et al., 2007]). Most of the existing graph clustering approaches presented in Chapter 1 fail to scale to such large real-life graphs [Prat-Pérez et al., 2014] and require the whole graph to be stored in memory, which often represents a heavy constraint in practice. Streaming the edges is a natural way to handle such massive graphs. In this setting, the entire graph is not stored but processed edge by edge [McGregor, 2014]. Note that the streaming approach is particularly relevant in most real-life applications where graphs are fundamentally dynamic and edges naturally arrive in a streaming fashion. This is the case for instance of the Twitter Retweet graph presented in our introductory chapter.

In this chapter, we introduce a novel approach based on *edge streams* to detect communities in graphs. The algorithm processes each edge strictly once. When the graph is a multi-graph, in the sense that two nodes may be connected by more than one edge, these edges are streamed independently. The algorithm only stores three integers for each node: its current community index, its current degree (i.e. the number of adjacent edges that have already been processed), and the current community volume (i.e. the sum of the degrees of all nodes in the community). Hence, the time complexity of the algorithm is linear in the number of edges and its space complexity is linear in the number of nodes. In the experimental evaluation of the algorithm, we show that this streaming algorithm is able to handle massive graphs [Yang and Leskovec, 2015] with low execution time and memory consumption.

The algorithm takes only one integer parameter $v_{\max}$ and, for each arriving edge (i, j) of the stream, it uses a simple decision strategy based on this parameter and the volumes of the communities of nodes i and j. We provide a theoretical analysis that justifies the form of this decision strategy using the *modularity* measure. In this analysis, we show that, under certain assumptions, the processing of each new edge by our algorithm leads to an increase in modularity.

The chapter is organized as follows. First, we present some related work and give an introduction to graph streaming algorithms in Section 5.2. Then, we describe our streaming algorithm in Section 5.3. A theoretical analysis of this algorithm is presented in Section 5.4. In Section 5.5, we evaluate experimentally the performance of our approach on real-life graphs and compare it to state-of-the-art algorithms.

5.2 Related work

As underlined above, most of the state-of-the art techniques for graph clustering have proven to be efficient on graphs with millions of edges, but are often time-consuming and fail to scale to larger graphs, as shown in the thorough evaluation performed in [Prat-Pérez et al., 2014]. The streaming approach has drawn considerable interest in network analysis over the last decade. Within the data stream model, massive graphs with potentially billions of edges can be processed without being stored in memory [McGregor, 2014]. A lot of algorithm have been proposed for different problems that arise in large graphs, such as counting subgraphs [Bar-Yossef et al., 2002, Buriol et al., 2006], computing matchings [Goel et al., 2012, Feigenbaum et al., 2005], finding the minimum spanning tree [Elkin and Zhang, 2006, Tarjan, 1983] or graph sparsification [Benczúr and Karger, 1996], which can be used to solve the min-cut problem and iteratively cluster a graph. In this section, we introduce the most common streaming frameworks and present classic streaming algorithms

5.2.1 Edge streaming models

There exist different frameworks for streaming algorithms in graphs [McGregor, 2014]. The most common models consider that nodes are given in advance (i.e. the set V is known beforehand) and that edges arrive in a streaming fashion. The edge stream S can be an *insert-only stream*, i.e. a simple sequence of edges $S = (e_1, \ldots, e_m)$ such that $E = \{e_1, \ldots, e_m\}$. Note that we can either assume that elements in S are distinct from each other, or consider that edges can appear multiple times in S. In the later case, we generally consider that $G = (V, E)$ is a multi-graph with a multi-set of edge E (i.e. and edge (i, j) can appear multiple times in E). Otherwise, the edge stream S can be an *insert-delete stream*, i.e. a sequence of edge insertion or edge deletion actions, $S = (a_1, \ldots, a_M)$ where $a_i = (e_i, \Delta_i)$ with $e_i \in E$ and $\Delta_i \in \{-1, +1\}$. An action a_i with $\Delta_i = 1$ corresponds to an edge insertion whereas an action such that $\Delta_i = -1$ corresponds to an edge deletion. In this chapter, we only consider *insert-only edge streams*.

In this framework, a *streaming algorithm* is an algorithm that considers the edges of S sequentially (i.e. that processes the edges $e_1, \ldots, e_m$ one by one, or k by k) and computes its result without storing the whole graph structure (V, E) in memory. Generally, we consider that $\mathcal{A}$ is a *good* streaming algorithm if its memory complexity is much smaller that $|E| = m$ and if it has limited processing time per edge. Note that, in many cases, streaming algorithms can only compute approximations of the solutions of the problems that they want to solve. In order to compute its result, a streaming algorithm needs to store and maintain a *summary* of the graph structure. This data structure is referred to as a *sketch* of the data stream and is updated at each edge arrival [Ahn et al., 2012].

For instance, consider the problem of computing the node degrees in the insert-only edge steaming model, with a stream $S = (e_1, \ldots, e_m)$. We see that this task can be performed by an algorithm that only stores the updated degree of each node d_i, $i \in V$, and that updates them at each step t with $d_i \leftarrow d_i + 1$ and $d_j \leftarrow d_j + 1$ if $e_t = (i, j)$. It is straightforward to see that d_i corresponds to the degree of i in G at the end of the algorithm. The size of the sketch $\{d_i, i \in V\}$ used by this simple algorithm is n and the time complexity of each step is in $O(1)$.

5.2.2 Connected components and pairwise distances

Here, we consider the classic problem of determining the connected components of G and the pairwise distances $d(i, j)$ between nodes, where $d(i, j)$ is the length of the shortest path in G between i and j, in the insert-only streaming framework.

A simple algorithm to compute the connected components in graph G from the stream S uses a graph sketch H that corresponds to a minimal subgraph of G whose connected components are the same as G. At each step t, this algorithm considers the edge $e_t = (i, j)$ and adds it to H if there is no path in H between i and j. It is easy to see that there is no cycle in H. Therefore the number of edges in H is in $O(n)$ [Bollobás, 2004]. This simple algorithm can be extended to approximate the pairwise distance between nodes in G, by adding an edge $e_t = (i, j)$ to the subgraph H not only if there is no path in H between i and j but also if the distance between i and h is greater than a given threshold α. More formally, this algorithm can be described as follows.

- (Initialization) $H = (V_H, E_H) \leftarrow (V, \emptyset)$ (same nodes as G and no edges)

- For each step t, if $e_t = (i, j)$,
  - If $d(i, j) > \alpha$ then $E_H \leftarrow E_H \cup \{e_t\}$,

where $d_H(i, j)$ is the length of the shortest path between i and j in H (with $d_H(i, j) = \infty$ if i and j belong to distinct connected components). At the end of the algorithm, the graph H is an *α-spanner* of G, i.e. it verifies, for all $i, j \in V$,

$$d_G(i, j) \leq d_H(i, j) \leq \alpha d_G(i, j).$$

Indeed, H is a subgraph of G, so we have $d_G(i, j) \leq d_H(i, j)$. Moreover, if $e_t = (i, j)$ is not added to H by the algorithm, then there must already be a path of length $\leq \alpha$ in H between i and j. Based on this observation, we see that, for any path in G between two nodes i and j of length D, we can build a path in H of length at most αD, which proves the second inequality. Therefore, this algorithm can be used to obtain an approximation of the pairwise distance between nodes in G.

In contrast to the algorithm for connected components, the subgraph H built by the later algorithm can contain cycles. However, we see that the length of these cycles is greater than $\alpha + 1$. Therefore, if $\alpha = 2t - 1$, the length of such a cycle is greater than $2t$. This implies that the graph sketch H has at most $O(n^{1+1/t})$ edges [Bollobás, 2004].

5.2.3 Minimum cut

In Chapter 1, we have presented the minimum cut problem that consists in finding the subset $C \subset V$, such that $C \neq \emptyset$ and $C \neq V$, that minimizes

$$\text{cut}(C, V \setminus C) = 2 \sum_{i \in C} \sum_{j \notin C} A_{ij}.$$

A solution to this problem is a clustering of the graph nodes with two clusters. In this section, we study approaches to approximate $\text{cut}(C, V \setminus C)$ in the insert-only streaming setting. More precisely, we are interested in algorithms that build and maintain subgraphs H of G such that

$$\forall C \in V, \quad |\text{cut}_H(C, V \setminus C) - \text{cut}_G(C, V \setminus C)| < \epsilon \text{cut}_G(C, V \setminus C),$$

with $\epsilon > 0$. We call such a graph H an *ϵ-sparsification* of G.

We have seen in Chapter 1 that, if $v \in \{-1, +1\}^n$ is the indicator vector associated to the subset $C \subset V$ (i.e. the vector such that $v_i = 1$ if $i \in C$ and $v_i = -1$ otherwise) then we have

$$v^T L_G v = 4\text{cut}(C, V \setminus C),$$

where $L_G = D - A$ is the (unnormalized) Laplacian of the graph G. Therefore, a strategy to approximate the cut in our streaming framework consists in approximating the Laplacian L_G by building a subgraph H that verifies

$$\forall v \in \mathbb{R}^n, \quad |v^T L_H v - v^T L_G v| < \epsilon v^T L_G v.$$

Such a subgraph H is referred to as an ϵ-spectral sparsification of G. In can be easily shown that an ϵ-spectral sparsification verifies the two following properties.

- (Mergability) If H_1 and H_2 are respectively ϵ-spectral sparsifications for two graphs G_1 and G_2, then $H_1 \cup H_2$ is an ϵ-spectral sparsification for $G_1 \cup G_2$.
- (Composability) If H_3 is an α-spectral sparsification for H_2, and H_2 is a β-spectral sparsification for H_1, then H_3 is an $\alpha\beta$-spectral sparsification for H_1.

These two properties can be used to design a streaming algorithm to compute an ϵ-spectral sparsification for G using any external algorithm $\mathcal{A}$ that can compute γ-spectral sparsifications for small graphs [McGregor, 2014]. This method consists in alternatively applying the algorithm $\mathcal{A}$ on small chunks of the stream S and merging the results until we obtain a spectral sparsification of G. This strategy starts to divide the edge stream S into K chunks $S_1, \ldots, S_K$ such that each chunk has a size $|S|/K$ and is

small enough to be processed by $\mathcal{A}$. The algorithm defines for all $k \in [\![1, K]\!]$ H_k^0 as the result of the sparsification algorithm $\mathcal{A}$ applied to G_k^0, where G_k^0 corresponds to the graph defined by the chunk S_k,

$$\forall k, \quad H_k^0 = \mathcal{A}(G_k^0).$$

Then, the algorithm iteratively merges these subgraphs by pair and re-apply the algorithm $\mathcal{A}$ on these merged subgraphs. This method assume that the initial number of chunk is a power of 2 so that these chunks can be iteratively merged until we obtain the whole graph G. More formally, the algorithm proceeds as follows.

- (Initialization) $H_k^0 \leftarrow \mathcal{A}(G_k^0)$, for all k.
- For $t \in \{1, \ldots, \log_2 K\}$, for all k,

$$G_k^t = G_{2k-1}^{t-1} \cup G_{2k}^{t-1} \quad \text{and} \quad H_k^t = \mathcal{A}(H_{2k-1}^{t-1} \cup H_{2k}^{t-1}).$$

It is easy to verify that, for $T = \log_2 k$, we have $G_1^T = G$. Then using the mergeability and composability properties of spectral sparsifications, we see that H_1^T is an γ^T-spectral sparsification for $G = G_1^T$. Therefore, if $\gamma^T \leq \epsilon$, we have the wanted result.

This algorithm can be used to approximate graph cuts. Therefore, it can be used to cluster graphs by iteratively splitting the graph into two clusters corresponding to the minimum cut. However, solving the minimum cut problem typically relies on the spectral decomposition of L_H, which is expensive and does not scale to large graphs.

5.3 A streaming algorithm for graph clustering

In the rest of the chapter, we study a novel streaming algorithm for graph clustering in the insert-only edge streaming framework. More precisely, we assume that we are given an undirected and unweighted multi-graph $G(V, E)$ where E is a multi-set of edges (i.e. an edge (i, j) can appear multiple times in E), and that there is no self-loop in G, that is $A_{ii} = 0$ for all $i \in V$. We use $S = (e_1, \ldots, e_m)$ to denote the edge stream, which is an order sequence of the multi-set E. Note that each edge $e = (i, j) \in E$ appears exactly A_{ij} times in S.

5.3.1 Intuition

As explained in Chapter 1, although there is no universal definition of what a community is, most existing algorithms rely on the principle that nodes tend to be more connected within a community than across communities. Hence, if we pick uniformly at random an edge e in E, this edge is more likely to link nodes of the same community (i.e., e is an *intra-community* edge), than nodes from distinct communities (i.e., e is an *inter-community* edge). Equivalently, if we assume that edges arrive in a random order, we expect many intra-community edges to arrive before the inter-community edges.

This observation is used to design a streaming algorithm. For each arriving edge (i, j), the algorithm places i and j in the same community if the edge arrives *early* (intra-community edge) and splits the nodes in distinct communities otherwise (inter-community edge). In this formulation, the notion of an *early* edge is of course critical. In the proposed algorithm, we consider that an edge (i, j) arrives *early* if the current volumes of the communities of nodes i and j, accounting for previously arrived edges only, is low.

More formally, the algorithm considers successively each edge of the stream $S = (e_1, e_2, \ldots, e_m)$. Each node is initially in its own community. At time t, a new edge $e_t = (i, j)$ arrives and the algorithm performs one of the following actions: (a) i joins the community of j; (b) j joins the community of i; (c) no action.

The choice of the action depends on the *updated* community volumes $\mathrm{Vol}(C(i))$ and $\mathrm{Vol}(C(j))$ of the communities of i and j, $C(i)$ and $C(j)$, i.e., the volumes computed using the edges $e_1, ..., e_t$. If $\mathrm{Vol}(C(i))$ or $\mathrm{Vol}(C(j))$ is greater than a given threshold $v_{\max}$, then we do nothing; otherwise, the node belonging to the smallest community (in volume) joins the community of the other node and the volumes are updated.

5.3.2 Algorithm

We define our streaming algorithm in Algorithm 3. It takes the list of edges of the graph and one integer parameter $v_{\max} \geq 1$. The algorithm uses three dictionaries d, c and v, initialized with default value 0. At the end of the algorithm, d_i is the degree of node i, c_i the community of node i, and v_k is the volume of community k. When an edge with an unknown node arrives, let say i, we give this node a new community index, $c_i \leftarrow k$, and increment the index variable k (which is initialized with 1). For each new edge $e = (i, j)$, the degrees of i and j and the volumes of communities c_i and c_j are updated. Then, if these volumes are both lower than the threshold parameter $v_{\max}$, the node in the community with the lowest volume joins the community of the other node. Otherwise, the communities remain unchanged.

Algorithm 3 Streaming algorithm for clustering graph nodes

Require: Stream of edges S and parameter $v_{\max} \geq 1$
$d, v, c \leftarrow$ dictionaries initialized with default value 0
$k \leftarrow 1$ (*new community index*)
for $(i, j) \in S$ **do**
  if $c_i = 0$ **then** $c_i \leftarrow k$ and $k \leftarrow k + 1$
  end if
  if $c_j = 0$ **then** $c_j \leftarrow k$ and $k \leftarrow k + 1$
  end if
  $d_i \leftarrow d_i + 1$ and $d_j \leftarrow d_j + 1$ (*update degrees*)
  $v_{c_i} \leftarrow v_{c_i} + 1$ and $v_{c_j} \leftarrow v_{c_j} + 1$ (*update community volumes*)
  if $v_{c_i} \leq v_{\max}$ and $v_{c_j} \leq v_{\max}$ **then**
    if $v_{c_i} \leq v_{c_j}$ **then** (*i joins the community of j*)
      $v_{c_j} \leftarrow v_{c_j} + d_i$
      $v_{c_i} \leftarrow v_{c_i} - d_i$
      $c_i \leftarrow c_j$
    else (*j joins the community of i*)
      $v_{c_i} \leftarrow v_{c_i} + d_j$
      $v_{c_j} \leftarrow v_{c_j} - d_j$
      $c_j \leftarrow c_i$
    end if
  end if
end for
return $(c_i)_{i \in V}$

Observe that, in case of equality $v_{c_i} = v_{c_j} \leq v_{\max}$, j joins the community of i. Of course, this choice is arbitrary and can be made random (e.g., i joins the community of j with probability 1/2 and j joins the community of i with probability 1/2).

5.3.3 Complexity

The main loop is linear in the number of edges in the stream. Thus, the time complexity of the algorithm is linear in m.

Concerning the space complexity, we only use three dictionaries of integers d, c and v, of size n. Hence, the space complexity of the algorithm is $3n \cdot \texttt{sizeOf(int)} = O(n)$. Note that the algorithm does not need to store the list of edges in memory, which is the main benefit of the streaming approach. To implement dictionaries with default value 0 in practice, we can use classic dictionaries or maps, and, when an unknown key is requested, set the value relative to this key to 0. Note that, in Python, the `defaultdict` structure already allows us to exactly implement dictionaries with 0 as a default value.

5.3.4 Parameter setting

Note that the algorithm can be run once with multiple values of parameter $v_{\max}$. In this case, only arrays c and v need to be duplicated for each value of $v_{\max}$. In this multi-parameter setting, we obtain multiple results $(c^a)_{1 \leq a \leq A}$ at the end of the algorithm, where A is the number of distinct values for the parameter

$v_{\max}$. Then, the best c^a can be selected by computing quality metrics that only use dictionaries c^a and v^a. In particular, we do not want to use metrics that requires the knowledge of the input graph. For instance, common metrics [Yang and Leskovec, 2015] like entropy $H(v) = -\sum_k \frac{v_k}{w} \log\left(\frac{v_k}{w}\right)$ or average density $D(c, v) = \sum_{k:C_k \neq \emptyset} \frac{1}{|P|} \frac{v_k}{|C_k|(|C_k|-1)}$, where C_k is the set of nodes in community k and P is the set of all non-empty communities, can be easily computed from each pair of dictionaries (v^a, c^a), and be used to select the best result c^a for $a = 1, \ldots, A$. Note that modularity cannot be used here as its computation requires the knowledge of the whole graph.

5.4 Theoretical analysis

In this section, we analyze the modularity optimization problem in the edge-streaming setting and qualitatively justify the conditions on community volumes used in Algorithm 3.

5.4.1 Modularity optimization in the streaming setting

The modularity can be rewritten as

$$Q = \frac{1}{w}\left[\sum_{i\in V}\sum_{j\in V} A_{ij}\delta(i,j) - \sum_{C\in P}\frac{\mathrm{Vol}(C)^2}{w}\right].$$

In our streaming setting, we are given a stream $S = (e_1, \ldots, e_m)$ of edges such that edge (i, j) appears A_{ij} times in S. We consider the situation where t edges $e_1, \ldots, e_t$ from the stream $S = (e_1, \ldots, e_m)$ have already arrived, and where we have computed a partition $P_t = (C_1, \ldots, C_K)$ of the graph. We define S_t as $S_t = \{e_1, \ldots, e_t\}$, and Q_t as

$$Q_t = \sum_{C\in P_t}\left[2\mathrm{Int}_t(C) - \frac{(\mathrm{Vol}_t(C))^2}{w}\right]$$

where $\mathrm{Int}_t(C) = \sum_{(i,j)\in S_t} 1_{i\in C}1_{i\in C}$ and $\mathrm{Vol}_t(C) = \sum_{(i,j)\in S_t}(1_{i\in C} + 1_{j\in C})$. Note that there is no normalization factor $1/w$ in the definition of Q_t as it has no impact on the optimization problem.

We do not store the edges of S_t but we assume that we have kept updated node degrees $w_t(i) = \sum_{(i',j')\in S_t}(1_{i'=i} + 1_{j'=i})$ and community volumes $\mathrm{Vol}_t(C_k)$ in a streaming fashion as shown in Algorithm 3. We consider the situation where a new edge $e_{t+1} = (i, j)$ arrives. We want to make a decision that maximizes Q_{t+1}.

5.4.2 Streaming decision

We can express Q_{t+1} in function of Q_t as stated in Lemma 5.4.1.

Lemma 5.4.1. *If $e_{t+1} = (i, j)$ and if $P_{t+1} = P_t$, Q_{t+1} can be expressed in function of Q_t as follows*

$$Q_{t+1} = Q_t + 2\left[\delta(i,j) - \frac{\mathrm{Vol}_t(C(i)) + \mathrm{Vol}_t(C(j)) + 1 + \delta(i,j)}{w}\right]$$

where $C(v)$ denotes the community of v in P_t, and $\delta(i, j) = 1$ if i and j belongs to the same community of P_t and 0 otherwise.

Proof. Given a new edge $e_{t+1} = (i, j)$, we have the following relation between quantities $\mathrm{Int}(C)$ and $\mathrm{Vol}(C)$ at times t and $t + 1$.

$$\mathrm{Int}_{t+1}(C) = \mathrm{Int}_t(C) + 1_{i\in C}1_{j\in C}$$

and

$$\mathrm{Vol}_{t+1}(C) = \mathrm{Vol}_t(C) + 1_{i\in C} + 1_{j\in C}.$$

This gives us the following equation for $(\mathrm{Vol}_{t+1}(C))^2$

$$(\mathrm{Vol}_{t+1}(C))^2 = (\mathrm{Vol}_t(C))^2 + \begin{cases} 0 & \text{if } C \neq C(i) \text{ and } C \neq C(j) \\ 2\mathrm{Vol}_t(C(i)) + 1 & \text{if } C = C(i) \\ 2\mathrm{Vol}_t(C(j)) + 1 & \text{if } C = C(j) \end{cases}$$

in the case $C(i) \neq C(j)$, and

$$(\mathrm{Vol}_{t+1}(C))^2 = (\mathrm{Vol}_t(C))^2 + \begin{cases} 0 & \text{if } C \neq C(i) \text{ and } C \neq C(j) \\ 4\mathrm{Vol}_t(C(i)) + 4 & \text{if } C = C(i) = C(j) \end{cases}$$

in the case $C(i) = C(j)$.

Finally, the definition of Q_{t+1}

$$Q_{t+1} = \sum_{C \in P_{t+1}} \left[2\mathrm{Int}_{t+1}(C) - \frac{(\mathrm{Vol}_{t+1}(C))^2}{w} \right]$$

gives us the wanted result. □

We want to update the community membership of i or j with one of the following actions: (a) i joins the community of j; (b) j joins the community of i; (c) i and j stays in their respective communities. We consider the case where nodes i and j belongs to distinct communities of P_t, since all three actions are identical if i and j belong to the same community. We want to choose the action that maximizes Q_{t+1}, but we have a typical streaming problem where we cannot evaluate the impact of action (a) or (b) on Q_t but only on the term that comes from the new edge e_{t+1}.

Let us consider action (a), where i joins the community of j. We can assume that $\mathrm{Vol}_t(C(i)) \leq \mathrm{Vol}_t(C(j))$ without loss of generality (otherwise we can swap i and j). We are interested in $\Delta Q_{t+1} = Q_{t+1}^{(a)} - Q_{t+1}^{(c)}$, which is the variation of Q_{t+1} between the state where i and j are in their own communities and the state where i has joined $C(j)$. We have $\Delta Q_{t+1} = \Delta Q_t + 2[1 - (\mathrm{Vol}_t(C(j)) - \mathrm{Vol}_t(C(i)) + 2w_i(t) + 1)/w]$ where ΔQ_t is the variation of Q_t. Lemma 5.4.2 gives us an expression for this variation.

Lemma 5.4.2.

$$\Delta Q_t = Q_t^{(a)} - Q_t^{(c)} = 2\left[L_t(i, C(j)) - L_t(i, C(i)) - \frac{(w_t(i))^2}{w} \right]$$

where

$$L_t(i, C) = \sum_{(i',j') \in S_t} \left[1_{i' \in C} \left(1_{j'=i} - \frac{w_t(i)}{w} \right) + 1_{j' \in C} \left(1_{i'=i} - \frac{w_t(i)}{w} \right) \right].$$

Proof. Q_t is defined as a sum over all communities of partition P_t. Only terms depending on $C(i)$ and $C(j)$ are modified by action (a). Thus, we have:

$$\begin{aligned} \Delta Q_t = {} & 2\left[\mathrm{Int}_t(C(i) \setminus \{i\}) + \mathrm{Int}_t(C(j) \cup \{i\}) - \mathrm{Int}_t(C(i)) - \mathrm{Int}_t(C(i))\right] \\ & - \frac{(\mathrm{Vol}_t(C(i)) - w_t(i))^2 + (\mathrm{Vol}_t(C(j)) + w_t(i))^2 - (\mathrm{Vol}_t(C(i)))^2 - (\mathrm{Vol}_t(C(j)))^2}{w}. \end{aligned}$$

This leads to:

$$\begin{aligned} \Delta Q_t = {} & 2 \sum_{(i',j') \in S_t} \left[1_{j'=i}(1_{i' \in C(j)} - 1_{i' \in C(i)}) + 1_{i'=i}(1_{j' \in C(j)} - 1_{j' \in C(i)}) \right] \\ & - 2\frac{w_t(i)\mathrm{Vol}_t(C(j)) - w_t(i)\mathrm{Vol}_t(C(i)) + (w_t(i))^2}{w}. \end{aligned}$$

Using the definition of Vol_t, we obtain the wanted expression for ΔQ_t. □

We define $l_t(i, C)$ as $l_t(i, C) = L_t(i, C)/\mathrm{Vol}_t(C)$. Then, we can easily show that $l_t(i, C) \in [-1, 1]$ and $\mathbb{E}[l_t(i, C)] = 0$ if edges of S_t follow the null model presented above. $L_t(i, C)$ measures the difference between the number of edges connecting node i to community C in the edge stream S_t, and the number of edges that we would observe in the null model. It can be interpreted as a degree of attachment of node i to community C. Thus, $l_t(i, C)$ can be seen as a *normalized degree of attachment of node i to community C* in the edge stream S_t.

Lemma 5.4.1 and 5.4.2 gives us a sufficient condition presented in Theorem 5.4.3 in order to have a positive variation ΔQ_{t+1} of the modularity when i joins $C(j)$.

Theorem 5.4.3. *If* $\mathrm{Vol}_t(C(i)) \leq \mathrm{Vol}_t(C(j))$*, then:*

$$\mathrm{Vol}_t(C(j)) \leq v_t(i,j) \quad \Longrightarrow \quad \Delta Q_{t+1} \geq 0$$

where

$$v_t(i,j) = \begin{cases} \frac{1-(w_t(i)+1)^2/w}{l_t(i,C(i))-l_t(i,C(j))} & \text{if } l_t(i,C(i)) \neq l_t(i,C(j)) \\ +\infty & \text{otherwise.} \end{cases}$$

Proof. From Lemma 5.4.1, we obtain

$$\begin{aligned} \Delta Q_{t+1} = Q_t^{(a)} + 2\left[1 - \frac{(\mathrm{Vol}_t(C(j)) + w_i(t)) + (\mathrm{Vol}_t(C(j)) + w_i(t)) + 2)}{w}\right] \\ - Q_t^{(b)} - 2\left[0 - \frac{(\mathrm{Vol}_t(C(i)) - w_i(t)) + (\mathrm{Vol}_t(C(j)) + w_i(t)) + 1)}{w}\right], \end{aligned}$$

which gives us:

$$\Delta Q_{t+1} = \Delta Q_t + 2\left[1 - \frac{\mathrm{Vol}_t(C(j)) - \mathrm{Vol}_t(C(i)) + 2w_i(t) + 1}{w}\right] \tag{5.1}$$

Then, Equation (5.1) and Lemma 5.4.2 gives us the following expression for ΔQ_{t+1}

$$\begin{aligned} \Delta Q_{t+1} = 2\Bigg[1 + \left(l_t(i,C(j)) - \frac{1}{w}\right) Vol_t(C(j)) - \left(l_t(i,C(i)) - \frac{1}{w}\right) Vol_t(C(i)) \\ - \frac{(w_t(i)+1)^2}{w}\Bigg]. \end{aligned}$$

Thus, $\Delta Q_{t+1} \geq 0$ is equivalent to

$$\left(l_t(i,C(i)) - \frac{1}{w}\right) Vol_t(C(i)) - \left(l_t(i,C(j)) - \frac{1}{w}\right) Vol_t(C(j)) \leq 1 - \frac{(w_t(i)+1)^2}{w}. \tag{5.2}$$

We use $u_t(i,j)$ to denote the left-hand side of this inequality. If $Vol_t(C(i)) \leq Vol_t(C(j))$, then we have

$$u_t(i,j) \leq [l_t(i,C(i)) - l_t(i,C(j))]Vol_t(C(j))$$

Thus, the following inequality

$$[l_t(i,C(i)) - l_t(i,C(j))]Vol_t(C(j)) \leq 1 - \frac{(w_t(i)+1)^2}{w}$$

implies inequality (5.2), which proves the theorem.

□

Thus, we see that, if we have $v_{\max} \leq v_t(i,j)$, then the strategy used by Algorithm 3 leads to an increase in modularity. In the general case, we cannot control terms $l_t(i,C(i))$ and $l_t(i,C(j))$, but, in most cases, we expect the degree of attachment of i to $C(i)$ to be upper-bounded by some constant $\tau_1 < 1$ and the degree of attachment of i to $C(j)$ to be higher than some $\tau_2 > 0$. Indeed, the fact that we observe an edge (i,j) between node i and community $C(j) \neq C(i)$ is likely to indicate that the degree of attachment between i and $C(j)$ is greater than what we would have in the null model, and that the degree of attachment between i and $C(i)$ is below maximum. Moreover, since in real-world graphs the degree of most nodes is in $O(1)$ whereas w is in $O(m)$, we expect the term $(w_t(i)+1)^2/w$ to be smaller than a constant $\epsilon \ll 1$. Then, the condition on $v_{\max}$ becomes:

$$v_{\max} \leq \frac{1-\epsilon}{\tau_1 - \tau_2}.$$

This justifies the design of the algorithm, with the decision of joining one community or the other based on the community volumes.

5.5 Experimental results

5.5.1 Datasets

We use real-life graphs provided by the Stanford Social Network Analysis Project for the experimental evaluation of our new algorithm. As explained in Chapter 1, these datasets include ground-truth community memberships that we use to measure the quality of the detection. We consider datasets of different natures.

- **Social networks**: The YouTube, LiveJournal, Orkut and Friendster datasets correspond to social networks [Backstrom et al., 2006, Mislove et al., 2007] where nodes represent users and edges connect users who have a friendship relation. In all these networks, users can create groups that are used as ground-truth communities in the dataset definitions.
- **Co-purchasing network**: The Amazon dataset corresponds to a product co-purchasing network [Leskovec et al., 2007]. The nodes of the graph represent Amazon products and the edges correspond to frequently co-purchased products. The ground-truth communities are defined as the product categories.
- **Co-citation network**: The DBLP dataset corresponds to a scientific collaboration network [Backstrom et al., 2006]. The nodes of the graph represent the authors and the edges the co-authorship relations. The scientific conferences are used as ground-truth communities.

The size of these graphs ranges from approximately one million edges to more than one billion edges. It enables us to test the ability of our algorithm to scale to very large graphs. The characteristics of these datasets can be found in Table 5.1.

5.5.2 Benchmark algorithms

For assessing the performance of our streaming algorithm we use a wide range of state-of-the-art but non-streaming algorithms that are based on various approaches: **SCD** (S), **Louvain** (L), **Infomap** (I), **Walktrap** (W), and **OSLOM** (O), that were all briefly introduced in Chapter 1. In the data tables, we use **STR** to refer to our streaming algorithm.

5.5.3 Performance metrics and benchmark setup

We use the *average* F_1*-score* and the *Normalized Mutual Information* for the performance evaluation of the selected algorithms. The experiments were performed on EC2 instances provided by Amazon Web Services of type `m4.4xlarge` with 64 GB of RAM, 100 GB of disk space, 16 virtual CPU with Intel Xeon Broadwell or Haswell and Ubuntu Linux 14.04 LTS.

Our algorithm is implemented in C++ and the source code can be found on GitHub[1]. For the other algorithms, we used the C++ implementations provided by the authors, that can be found on their respective websites. Finally, all the scoring functions were implemented in C++. We used the implementation provided by the authors of [Lancichinetti et al., 2009] for the NMI and the implementation provided by the authors of SCD [Prat-Pérez et al., 2014] for the F1-Score.

5.5.4 Benchmark results

Execution time

We compare the execution times of the different algorithms on SNAP graphs in Table 5.1. The entries that are not reported in the table corresponds to algorithms that returned execution errors or algorithms with execution times exceeding 6 hours. In our experiments, only SCD, except from our algorithm, was able to run on all datasets. The fastest algorithms in our benchmarks are SCD and Louvain and we observe that they run more than ten times slower than our streaming algorithm. More precisely, our streaming algorithm runs in less than 50ms on the Amazon and DBLP graphs, which contain millions of edges, and less than 5 minutes on the largest network, Friendster, that has more than one billion edges.

[1] https://github.com/ahollocou/graph-streaming

In comparison, it takes seconds for SCD and Louvain to detect communities on the smallest graphs, and several hours to run on Friendster. Table 5.1 shows the execution times of all the algorithms with respect to the number of edges in the network. We remark that there is more than one order of magnitude between our algorithm and the other algorithms.

In order to compare the execution time of our algorithm with a minimal algorithm that only reads the list of edges without doing any additional operation, we measured the run time of the Unix command `cat` on the largest dataset, Friendster. `cat` reads the edge file sequentially and writes each line corresponding to an edge to standard output. In our expermiments, the command `cat` takes 152 seconds to read the list of edges of the Friendster dataset, whereas our algorithm processes this network in 241 seconds. That is to say, reading the edge stream is only twice faster than the execution of our streaming algorithm.

	$\|V\|$	$\|E\|$	S	L	I	W	O	STR
Amazon	334,863	925,872	1.84	2.85	31.8	261	1038	**0.05**
DBLP	317,080	1,049,866	1.48	5.52	27.6	1785	1717	**0.05**
YouTube	1,134,890	2,987,624	9.96	11.5	150	-	-	**0.14**
LiveJournal	3,997,962	34,681,189	85.7	206	-	-	-	**2.50**
Orkut	3,072,441	117,185,083	466	348	-	-	-	**8.67**
Friendster	65,608,366	1,806,067,135	13464	-	-	-	-	**241**

Table 5.1: SNAP dataset sizes and execution times in seconds

Memory consumption

We measured the memory consumption of streaming algorithm and compared it to the memory that is needed to store the list of the edges for each network, which is a lower bound of the memory consumption of the other algorithms. We use 64-bit integers to store the node indices. The memory needed to represent the list of edges is 14,8 MB for the smallest network, Amazon, and 28,9 GB for the largest one, Friendster. In comparison, our algorithm consumes 8,1 MB on Amazon and only 1,6 GB on Friendster.

Detection scores

Table 5.2 shows the Average F_1-score and NMI of the algorithms on the SNAP datasets. Note that the NMI on the Friendster dataset is not reported in the table because the scoring program used for its computation [Lancichinetti et al., 2009] cannot handle the size of the output on this dataset. While Louvain and OSLOM clearly outperform our algorithm on Amazon and DBLP (at the expense of longer execution times), our streaming algorithm shows similar performance as SCD on YouTube and much better performance than SCD and Louvain on LiveJournal, Orkut and Friendster (the other algorithms do not run these datasets). Thus our algorithm does not only run much faster than the existing algorithms but the quality of the detected communities is also better than that of the state-of-the-art algorithms for very large graphs.

	F1-Score						**NMI**					
	S	L	I	W	O	STR	S	L	I	W	O	STR
Ama.	0.39	**0.47**	0.30	0.39	**0.47**	0.38	0.16	0.24	0.16	**0.26**	0.23	0.12
DBLP	0.30	0.32	0.10	0.22	**0.35**	0.28	**0.15**	0.14	0.01	0.10	**0.15**	0.10
YT	0.23	0.11	0.02	-	-	**0.26**	0.10	0.04	0.00	-	-	**0.13**
LiveJ.	0.19	0.08	-	-	-	**0.28**	0.05	0.02	-	-	-	**0.09**
Orkut	0.22	0.19	-	-	-	**0.44**	0.22	0.19	-	-	-	**0.24**
Friend.	0.10	-	-	-	-	**0.19**	-	-	-	-	-	-

Table 5.2: Average F1 Scores and NMI

Chapter 6

Local approaches

In this chapter, we study local graph clustering methods that focus on finding one or multiple clusters in the neighborhood of a given set of nodes, the *seed set*. While existing approaches typically recover only one community around the seed set, most nodes belong to multiple communities in practice. In this paper, we introduce a new algorithm for detecting multiple local communities, possibly overlapping, by expanding the initial seed set. The new nodes are selected by some local clustering of the graph embedded in a vector space of low dimension. We validate our approach on real graphs, and show that it provides more information than existing algorithms to recover the complex graph structure that appears locally. This chapter is based on the work presented in [Hollocou et al., 2017a].

6.1 Introduction

As explained in Chapter 1, most of graph clustering algorithms focus on clustering the entire graph G, even if we are only interested in local regions of this graph in many applications. A practically interesting problem consists in detecting the clusters containing some given set of nodes, the so-called *seed set*. This problem, known as *local* community detection, *local graph clustering*, *seed set expansion* or *community-search*, is particularly relevant in large datasets where the exploration of the whole graph is computationally expensive, if not impossible.

The problem of local community detection is generally stated as follows: the objective to recover some unknown community C in a graph given some subset $S \subset C$ of these nodes. The implicit assumption is that the communities form a partition of the graph. However, nodes typically belong to multiple, overlapping communities in practice [Reid et al., 2013, Xie et al., 2013]. The problem is then to recover all communities containing the seed set S.

In the present chapter, we first present some existing approaches that focus on finding a unique cluster given a set of seed nodes. Then, we propose a novel approach to this problem by introducing the `MULTICOM` algorithm, which is able to detect multiple communities nearby a given seed set S. This algorithm uses local scoring metrics to define an embedding of the graph around the seed set. Based on this embedding, it picks new seeds in the neighborhood of the original seed set, and uses these new seeds to recover multiple communities.

The rest of the chapter is organized as follows. Existing approaches for local community detection are presented in Section 6.2. Our approach to *multiple* local community detection is presented in Section 6.3. The algorithm itself is presented in Section 6.4 and tested on real graphs in Section 6.5.

6.2 Related work

Local community detection has drawn the attention of many researchers, motivated both by the ever increasing size of the datasets and the complex local structure of real graphs [Leskovec et al., 2008, Jeub et al., 2015]. In this section, we present some classic approaches to this problem.

6.2.1 Scoring and sweeping

The classical approach to local community detection is based on two steps [Andersen and Lang, 2006]: first, nodes are scored according to their proximity to the seed set; then, nodes are considered in decreasing order of their score and added to the community, with a stopping rule based on some goodness metric.

Formally, the algorithm is based on some scoring function f such that, for any seed set $S \subset V$, f_S is a vector in $\mathbb{R}_+^V$ whose component $f_S(v)$ characterizes the attachment of node v to S. We expect that the higher the score $f_S(v)$, the higher the probability that v is in the same community as the nodes of S. Examples of scoring functions are the Personalized PageRank, the heat kernel diffusion and the local spectral score, described below.

Given some seed set S, the score $f_S(v)$ of each node $v \in V$ is computed. The nodes are then numbered in decreasing order of their score, so that $f_S(v_1) \geq f_S(v_2) \geq \ldots \geq f_S(v_J)$, where J is the number of nodes with non-zero scores. This numbering defines a sequence of nested sets $S_1, ..., S_J$, with $S_j = \{v_i,\ i \leq j\}$.

The second step consists in finding the set S_j that defines the best community. To measure the quality of a community C, the conductance is typically used. Recall that it is defined as

$$\Phi(C) = \frac{\sum_{u \in C} \sum_{v \notin C} A_{uv}}{\min(\mathrm{Vol}(C), \mathrm{Vol}(V \setminus C))}.$$

A community of good quality is supposed to have low conductance. Indeed, the conductance corresponds to the ratio between the weight of edges *leaving* the set C (i.e. $\mathrm{cut}(C, \bar{C})$) and the volume of C (or the volume of $V \setminus C$ if $\mathrm{Vol}(C)$ is larger than half of the total weight w, i.e. $\mathrm{Vol}(C) > w/2$). We expect a *good* cluster to have a lot more intra-cluster edges than connections with $\bar{C}$, and thus we expect its conductance to be small. The conductance of each set S_{j+1} can be computed from the conductance of S_j in time proportional to $d_{v_{j+1}}$. This step is called the *sweep* process [Andersen and Lang, 2006]. The outcome of the algorithm is the set S_j having the lowest conductance. This set is called the *sweep cut*. Other goodness metrics like modularity or density can also be used for the sweep cut [Yang and Leskovec, 2015].

We study some scoring functions f_S that can be used in the *sweep* process below.

6.2.2 Personalized PageRank

First, we introduce the PageRank algorithm and its local variant, the Personalized PageRank, that is classically used to define the scoring function f_S.

Random walk with restart and PageRank

First, consider the random walk where we go from a node i to one of its neighbor j with probability $p_{ij} = A_{ij}/w_i$. We use $X_0, X_1, \ldots$ to denote the successive nodes visited by the random walk. Recall that $(X_t)_{t \geq 0}$ is a Markov chain on V. In order to be able to apply the Perron-Frobenius theorem, which guarantees the uniqueness of the stationary distribution and the convergence in distribution of X_t towards this stationary distribution when $t \to \infty$, we need the Markov chain to satisfy two properties:

- *irreducibility*: $(X_t)_{t \geq 0}$ is irreducible if any state j is accessible from any state i, i.e. if

$$\forall t \geq 0, \forall i, j, \quad \exists t' \geq t, \quad \mathbb{P}(X_{t'} = j | X_t = i) > 0;$$

- *aperiodicity*: $(X_t)_{t \geq 0}$ is said to be aperiodic if all states have period 1, i.e. if

$$\forall i, \quad \gcd\{t > 0 : \mathbb{P}(X_t = i | X_0 = i) > 0\} = 1.$$

For the random walk defined above, if G is undirected, these properties are satisfied if and only if the graph is *connected* and *not bipartite*. However, these properties are generally not satisfied for a generic directed graph.

This motivates us to consider a slightly modified random walk where, at each step, the walk can either move from the current node i to one of its neighbor j with probability αp_{ij}, or move to a node picked uniformly at random in the graph with probability $(1 - \alpha)$, where $0 \leq \alpha \leq 1$ is a given parameter. Such

Figure 6.1: Open Flights: measure of importance of the nodes with the PageRank algorithm

a random is called *random walk with restart*. If we note $X_0, X_1, \ldots$ the successive nodes visited by this random walk, $(X_t)_{t\geq 0}$ is still a random walk and we have

$$\forall t \geq 0, \forall i, \quad \mathbb{P}(X_{t+1} = i) = \frac{1-\alpha}{n} + \alpha \sum_{j \in V} \mathbb{P}(X_t = j) \frac{A_{ji}}{w_j}.$$

Writing the distribution of $(X_t)_{t\geq 0}$ as a vector $\pi(t)$, this equation becomes

$$\pi(t+1)^T = \pi(t)^T \left(\frac{1-\alpha}{n} 11^T + \alpha D^{-1} A \right).$$

A stationary distribution π of the Markov chain satisfies

$$\pi^T = \pi^T \left(\frac{1-\alpha}{n} 11^T + \alpha D^{-1} A \right).$$

It is easy to see that, if $\alpha < 1$, then the Markov chain $(X_t)_{t\geq 0}$ is aperiodic and irreducible. Therefore, we can apply the Perron-Frobenius theorem, which proves that the stationary distribution π is unique and that

$$\lim_{t \to \infty} \pi(t) = \pi.$$

The stationary distribution can be used to rank the nodes of the graph. Indeed, we can consider that the importance of a node i is represented by the probability π_i for the walk to be at this node when $t \to \infty$, and rank the nodes by decreasing value of this probability π_i. This is the idea behind the *PageRank* algorithm [Page et al., 1999] which is at the origin of the Google search engine.

Local random walks and Personalized PageRank

The random walk described above can be generalized as follows. At each step, instead of restarting form a node chosen uniformly at random in V with probability $(1-\alpha)$, we can restart from a node chosen uniformly at random in a subset S of V. The transition probability for this modified random walk

$$\forall t \geq 0, \forall i, \quad \mathbb{P}(X_{t+1} = i) = \frac{1-\alpha}{|S|} \mathbb{1}_{i \in S} + \alpha \sum_{j \in V} \mathbb{P}(X_t = j) \frac{A_{ji}}{w_j}.$$

or, in the vectorial form

$$\pi(t+1)^T = \pi(t)^T \left(\frac{1-\alpha}{|S|} 11_S^T + \alpha D^{-1} A \right).$$

where 1_S is the indicator vector for S, i.e. the vector whose i^{th} component is equal to 1 if $i \in S$ and equal to 0 otherwise.

The set S is referred to as the *seed set* and we call the nodes of S the *seed nodes*. This Markov chain is aperiodic and irreducible on the subgraph induced by the connected components of the seed

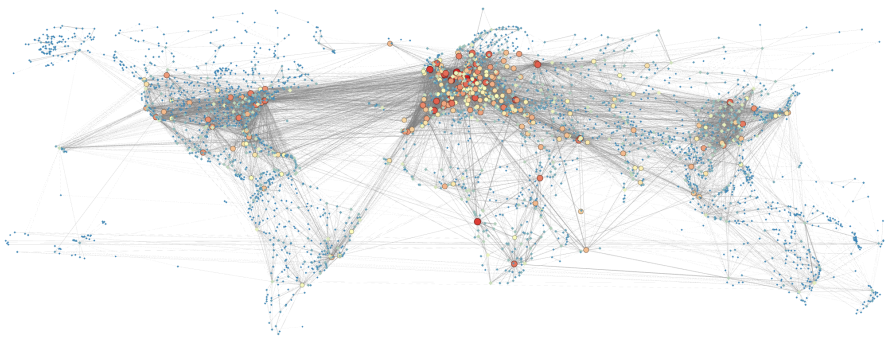


Figure 6.2: Open Flights: measure of the relative importance of the nodes with respect to the Frankfurt airport with the Personalized PageRank algorithm

nodes. Therefore, by Perron-Frobenius, it admits a unique stationary distribution π_S. Whereas the stationary distribution π for the classic random walk with restart that is used in the PageRank measures the *global* importance of a node in the graph, the stationary distribution π_S of this local random walk measures the *relative importance* of a node with respect to the seed nodes of S. We can rank the nodes by decreasing value of this probability to measure the attachment to S. This method is often referred to as the *Personalized PageRank* [Gleich, 2015].

We illustrate the difference between PageRank and Personalized PageRank in Figure 6.1 and Figure 6.2. In Figure 6.1, the color and the size of each node code for its PageRank. We see that important nodes according to this measure correspond to large hubs across the world. In Figure 6.2, the node colors and sizes code for the Personalized PageRank of the nodes, where the seed set has been chosen to contain a single seed node corresponding to the *Frankfurt airport*. We see that the important nodes according to this score mostly correspond to European airports.

Local clustering

The Personalized PageRank probability can be used as a local scoring function f_S in the sweep process described above. Indeed, if we are interested in finding a node cluster C from which we already know some nodes S ($S \subset C$), we expect the nodes i of C to have a higher probability $\pi_S(i)$ than the nodes outside the cluster. In order not to favor nodes with high degrees we can normalize the probability $\pi_S(i)$ with the degree of the node i. This leads to the following algorithm.

- Compute the Personalized PageRank π_S from the seed set S.
- Order the nodes $i \in V$ by decreasing value of $\pi_S(i)/d_i$.
- For every k, compute the score $\Phi(S_k)$ of the sweep set with the first k nodes.
- Detect the index k^* corresponding to the first local optima of $\Phi(S_k)$.
- Return the set S_{k^*}.

Variant and theoretical guarantees

Whereas the local clustering method described above is merely an heuristic, theoretical guarantees for a very similar approach have been established in the work of Spielman and Teng [Spielman and Teng, 2004] for unweighted graphs. They consider the random walk with the initial distribution

$$\pi(0) = \begin{cases} \frac{d_i}{\mathrm{Vol}(S)} & \text{if } i \in S, \\ 0 & \text{otherwise,} \end{cases}$$

and the transition equation

$$\pi(t+1)^T = \pi(t)^T \left(\frac{1}{2} I + \frac{1}{2} D^{-1} A \right).$$

Note that this random walk is different from the one used to defined the Personalized PageRank (even with $\alpha = 1/2$). Indeed, at step t, this new random walk restarts with probability 1/2 from a random node i in the graph picked with a probability proportional to $\pi(t)_i$ (instead of a random node picked uniformly at random). If the graph is connected, $\pi(t)$ converges towards a stationary distribution π when $t \to \infty$. Instead of studying this limiting distribution, they consider $\pi(t)$ after a small number of steps.

We look at the degree-normalized distribution $\frac{\pi(t)_i}{d_i}$. As in the method described above, we can order and renumber the nodes by decreasing value of $\pi(t)_i/d_i$ and define the sweep set S_j^t as $S_j^t = \{i | i \leq j\}$. If we view each vertex $i \in V$ as consisting of d_i mini-vertices $x_{(i,j)}$ with $j \in \mathrm{Nei}(i)$, we can interpret the degree-normalized distribution $\frac{\pi(t)_i}{d_i}$ as a probability distribution on mini-vertices. We use $q_t(x_{(i,j)}) = \frac{\pi(t)_i}{d_i}$ to denote this distribution on mini-vertices. At each step, we can also order the mini-vertices in a sequence $x_1^t, \ldots, x_{2m}^t$, so that

$$q_t(x_i^t) \geq q_t(x_{i+1}^t),$$

Using the notation $q_t(i) = q_t(x_i^t)$, we can define the quantities

$$P_t(k) = \sum_{i=1}^{k} q_t(i) \quad \text{and} \quad H_t(k) = P_t(k) - \frac{k}{2m}.$$

$P_t(k)$ represents the maximum amount of probability on any set of mini-vertices of size k at time t, whereas $H_t(k)$ is the difference between this maximum amount of probability and the probability to pick a set of k mini-vertices uniformly at random, which corresponds to the stationary distribution for this walk [Andersen and Lang, 2006]. The intuition behind $H_t(k)$ is the following. If $H_t(k)$ is high for some k, it means that the random walk is *localized* around the seed set and stays with high probability in the set of k mini-vertices $\{x_1^t, \ldots, x_k^t\}$. On the contrary, if $H_t(k)$ is low, it corresponds to the case where the node distribution of the walk is *close* to its stationary distribution. In this situation, we say that the random walk *mixes*.

The following Lemma is due to Spielman and Teng [Spielman and Teng, 2004].

Lemma 6.2.1. *For all $\Phi > 0$, $\alpha > 0$ and $T > 0$, one of the following statement is verified*

1. $\forall k$, $H_t(k) \leq H_0(k)\left(1 - \frac{\Phi^2}{8}\right)^T + \alpha T$
2. *there exists a sweep cut S_j^t with $t \leq T$ such that*
   - $\Phi(S_j^t) \leq \Phi$, *where $\Phi(S_j^t)$ is the conductance of S_j^t, and*
   - $q_t(k_0 - \Phi\bar{k_0}) - q_t(k_0 + \Phi\bar{k_0}) \geq \frac{2\alpha}{\Phi\bar{k_0}}$ *where $k_0 = \mathrm{Vol}(S_j^t)$ and $\bar{k_0} = \min(k_0, 2m - k_0)$.*

In other words, this lemma states that either

1. we have a strong bound on $H_T(k)$ which means that the random walk *mixes* well, or
2. one of the sweep sets S_j^t has a small conductance and we observe a large drop in degree-normalized probability between the vertices inside and outside S_j^t.

Hence, if the second statement is verified, S_j^t is a *good* cluster in the sense of the conductance.

6.2.3 Heat kernel diffusion

As seen above, the Personalized PageRank is defined as

$$\pi_S^T = \pi_S^T \left(\frac{1-\alpha}{|S|} 1 1_S^T + \alpha D^{-1} A \right),$$

which leads to the following expression for π_S,

$$\pi_S^T = \frac{(1-\alpha)}{|S|} 1_S^T (I - \alpha P)^{-1} = \frac{(1-\alpha)}{|S|} \sum_{k=0}^{\infty} \alpha^k 1_S^T P^k$$

where $P = D^{-1}A$.

Another form of diffusion has been introduced by Chung in [Chung, 2007, Chung, 2009]. It is called the heat kernel diffusion and is defined as

$$h_S^T = e^{-t}\left(\sum_{k=0}^{\infty}\frac{t^k}{k!}1_S^T P^k\right) = 1_S^T \exp\{-t(I-P)\},$$

where t is a parameter capturing the spread of the diffusion. A method of approximation of h_S is proposed in [Kloster and Gleich, 2014] and used in the sweep method to detect local communities.

6.2.4 Local spectral analysis

Another class of algorithms applies spectral techniques to detect local communities [Mahoney et al., 2012, Li et al., 2015]. In [Li et al., 2015] for instance, the `LEMON` algorithm is based on the extraction of a sparse vector y in the span of the so-called local spectral subspace of the graph around the seed set S. This vector y is then used as the scoring function. Unlike the previous algorithms, the `LEMON` algorithm is *iterative*: the nodes of highest scores are used to expand the seed set S and to find a new vector y, and so on. The iteration stops when the conductance starts increasing.

6.2.5 Other approaches

A number of other approaches have been proposed for local community detection. These include greedy algorithms [Clauset, 2005, Chang et al., 2015, Mehler and Skiena, 2009], flow-based algorithms [Orecchia and Zhu, 2014], degree-based techniques [Sozio and Gionis, 2010], motif detection [Huang et al., 2014, Yin et al., 2017] and subgraph extraction [Tong and Faloutsos, 2006]. None recover multiple communities.

6.2.6 Finding seed sets

All the approaches described above are particularly well-suited for the so-called *semi-supervised learning problems* where we already know a small portion of the cluster that we want to recover. However, in many applications, we do not have any prior knowledge on the clusters, or this prior knowledge is very limited, and a challenge consists in finding *good* seed sets. This problem has been studied in [Kloumann and Kleinberg, 2014] where the authors study the impact of the choice of the seed set on the quality of the cluster found with a local method similar to the ones studied above.

6.3 Multiple Community Detection

Let $v_1, \ldots, v_J$ be the nodes numbered in decreasing order of their score, as described in §6.2.1, and $S_1, \ldots, S_J$ be the corresponding nested sets. For a strong community, the scoring function should give high values to nodes belonging to the community and small values for nodes outside the community. As a result, we expect the existence of some $a > 0$ such that

$$\begin{aligned} &\max\left(f_S(v_2) - f_S(v_1), \ldots, f_S(v_j) - f_S(v_{j-1})\right) \leq a, \\ &\text{while } f_S(v_{j+1}) - f_S(v_j) > a, \end{aligned} \tag{6.1}$$

where j is the index of the target set S_j. The parameter a is called the *scoring gap*. As seen in Section 6.2, we can prove the existence of such a scoring gap for a variant of the Personalized PageRank [Spielman and Teng, 2004, Andersen and Lang, 2006]. Experimental studies showing the presence of a scoring gap in real graphs can be found in [Andersen and Lang, 2006, Danisch et al., 2014].

To illustrate this, we use the DBLP dataset from the Stanford Social Network Analysis Project (SNAP) website. DBLP is a database that collects the main publications in computer science. The graph we consider is a collaboration network: nodes correspond to authors and there is an edge between two authors if they have co-authored a paper. This dataset comes with ground-truth communities that correspond to journals and conferences (i.e., all authors having published in the same journal or conference form a community).

In our experiment, we pick a ground-truth community C in the graph and a seed node $s \in C$. We compute the attachment scores f_s to s with three different scoring functions: Personal PageRank p, Heat

Kernel score h and LEMON score y, introduced in Section 6.2. We compare the corresponding sets $S_1, S_2, \ldots$ to the ground-truth community C with the F1-Score.

6.3.1 Scoring gap

In Figure 6.3, we jointly plot for a given target community C and seed node $s \in C$ the values of the score $f_s(v_j)$ and the F1 score $\mathrm{F1}(S_j, C)$, for each scoring function. We observe in each case the existence of a clear scoring gap. Moreover we observe that the drop in the scoring function corresponds each time to a peak in the F1 Score, which means that the associated set is actually the best candidate for a community among all the nested sets. In order to find multiple communities, we elaborate on this idea as described in the next section.

6.3.2 Local embedding

The main idea of our algorithm is to use the scoring function to get a local embedding of the graph around the seed set S. Specifically, each node v is embedded into the vector $(f_s(v))_{s \in S}$. Note that for a node far from the seed set S, this vector will be the all zero vector and it can be safely removed since we are only interested in local communities. We then cluster nodes with respect to their mutual distances in the embedding space.

To motivate the proposed embedding, let us consider the case of a *perfect* scoring function f such that $f_s(v) = c_s$ if v is in the same community as s and 0 otherwise, where c_s is some positive constant that depends on s. Let us assume that we have disjoint communities $C_1, \ldots, C_K$ and seed nodes $s_1, \ldots, s_K$ in each of these communities ($s_k \in C_k$). Then, we see that the embedding $(f_{s_k}(v))_{1 \leq k \leq K}$ associated with these seed nodes and this perfect scoring function takes exactly K distinct non-zero values and that each of these values corresponds to a community C_k. Thus, by using a clustering algorithm on the vectors in the embedding space, we can exactly recover the communities $(C_k)_{k=1,\ldots,K}$. The scoring gap property presented above for Personalized PageRank, LEMON and Heat Kernel guarantees that these three functions are close to the *perfect* scoring function defined above. Therefore, applying clustering on the embedding $(f_s(v))_{s \in S}$ should lead to results similar to these perfect setup.

To illustrate the behavior of such embeddings, we show in Figure 6.4 the result of a local embedding using the Personal PageRank scoring function on a random graph generated from a mixture of two gaussian vectors in $\mathbb{R}^2$ by putting an edge between two points if they are within distance r from each other. We choose a seed node in each gaussian. We observe that the local embedding from these seed nodes clearly separates the nodes from the different groups and that a clustering algorithm can be used in order to recover each gaussian.

6.3.3 Finding new seeds

We use the local embedding introduced in the previous section to iteratively find new seeds $\mathcal{S}$ around the original seed set S. The main idea of the algorithm is simple: we start with $\mathcal{S} = S$ and we grow this new seed set by repeating the following three steps:

1. Perform the local embedding using $\mathcal{S}$: $(f_s(v))_{s \in \mathcal{S}}$.

2. Cluster the nodes in the embedding space.

3. Pick a new seed node in each *unexplored* cluster and add it to $\mathcal{S}$.

The new seed nodes of $\mathcal{S}$ are then used to detect multiple communities in the neighborhood of the initial seed set S. We define more formally the algorithm in the next section, and, in particular, we clarify the notion of *unexplored* cluster.

6.4 Algorithm

We now present our algorithm, named `MULTICOM`, that recovers multiple local communities.

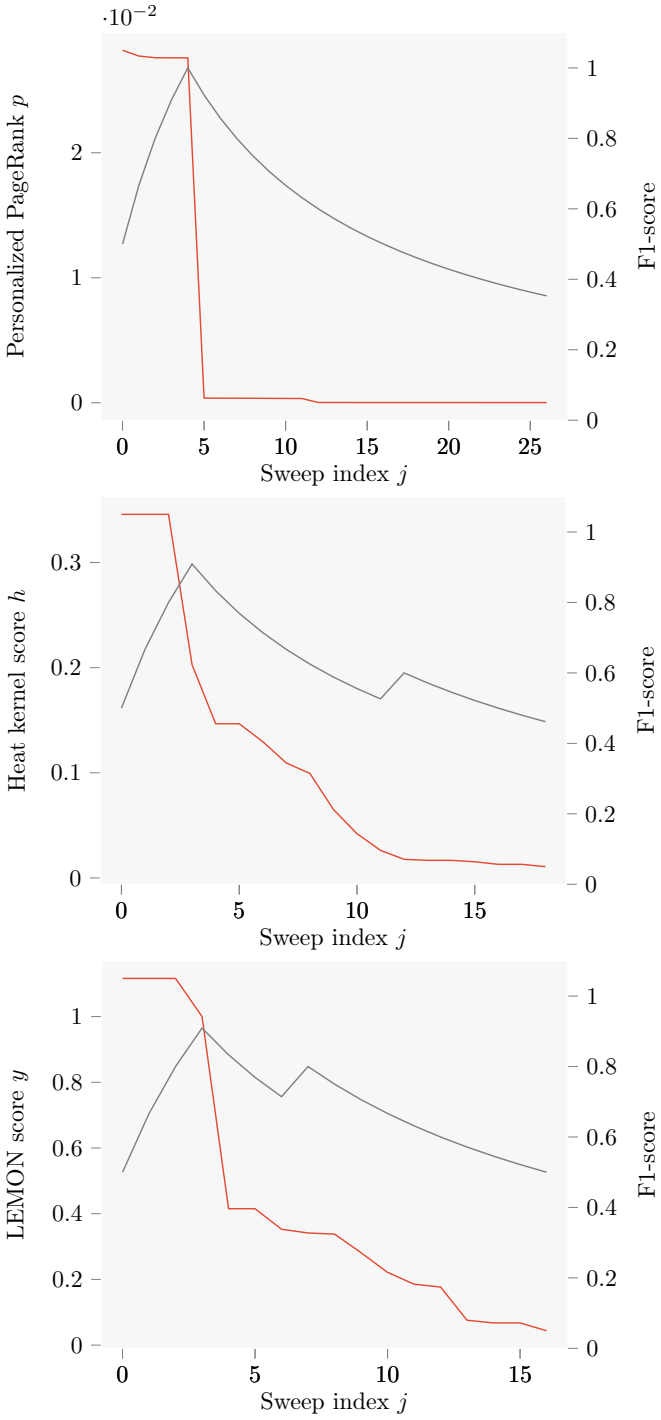
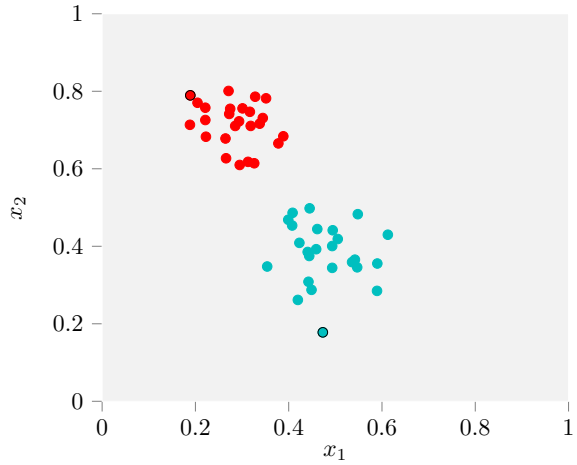
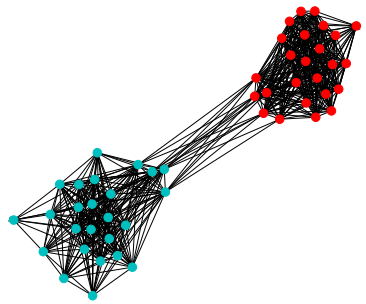


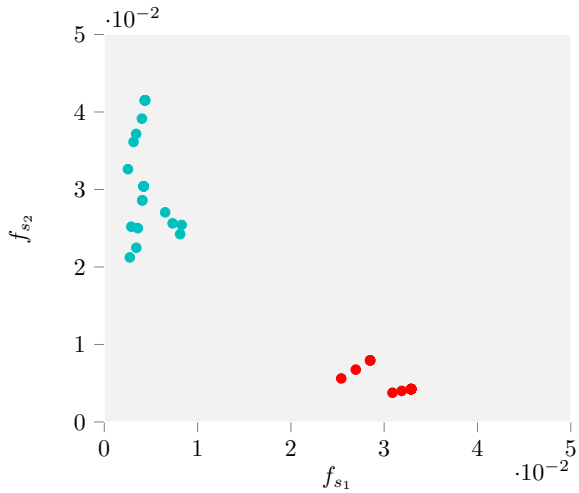
Figure 6.3: Attachment scores and F1 scores for the sweep sets in a ground-truth community of DBLP. We consider three scoring functions to compute the attachment scores: Personalized PageRank, Heat Kernel and LEMON. The attachment score curves are drawn in orange and the F1 Score curves in grey.



(a) Mixture of two gaussian vectors in $\mathbb{R}^2$
(one seed is picked in each cluster)



(b) Corresponding graph for $r = 0.2$



(c) Graph embedding using PPR

Figure 6.4: Example of local graph embedding for a random graph. The graph is generated from a mixture of two gaussians in $\mathbb{R}^2$ by putting an edge between two points if they are within a given distance r from each other. The local embedding is performed using the Personalized PageRank scoring function and two seed nodes s_1 and s_2 picked in each gaussian.

6.4.1 Inputs

The inputs of `MULTICOM` are a graph $G = (V, E)$ and a seed set $S \subset V$. Our algorithm also takes a scoring function f, which is typically one of the functions presented in Section 6.2, and a local community detection function `local` that, given a seed node s and a score f_s, returns a local community around S. For instance, this `local` function may return the sweep set with the lowest conductance by applying the sweep process described in §6.2.1. Finally the algorithm takes an integer parameter I that controls the number of detected communities and a *re-seeding threshold* $\beta \in [0, 1]$ that controls the number of new seeds at each iteration of the algorithm.

6.4.2 Description

In order to find multiple communities in the neighborhood of the initial seed set S, the algorithm uses a larger seed set $\mathcal{S}$ that contains S at the end of the algorithm. The new seed nodes found during a step of the algorithm are stored in $\mathcal{S}_{\text{new}}$. Initially, $\mathcal{S}_{\text{new}}$ is initialized with S.

Step 1: Local community detection for new seeds

The algorithm starts by computing the score vectors f_s for each new seed node $s \in \mathcal{S}_{\text{new}}$. Then, for each $s \in \mathcal{S}_{\text{new}}$ we use the `local` algorithm to extract a local community C_s around s. Thus, at the end of this step of the algorithm, we obtain one community C_s per seed node s. In particular, if the set $\mathcal{S}_{\text{new}}$ is a singleton, we only recover one community at this stage.

Step 2: Embedding and clustering

We use the scores f_s for $s \in \mathcal{S}$ to define an embedding which maps each node $v \in V$ to a vector $x_v = (f_s(v))_{s \in \mathcal{S}}$ in $\mathbb{R}_+^{|\mathcal{S}|}$. We then apply the popular clustering algorithm DBSCAN [Ester et al., 1996] to the non-zero vectors x_v and obtain K clusters of nodes, $D_1, ..., D_K$.

Step 3: Picking new seeds

The second step of the algorithm applies the ideas of Section 6.3 to find new seeds. We expect to obtain two types of clusters in Step 2:

- clusters with significant overlap with the communities $C_s, s \in \mathcal{S}$, already detected;
- clusters with low overlap with these communities.

We want to select seed nodes in the later clusters in order to detect new communities. To identify these clusters, we compute for each cluster D_k the ratio

$$\frac{|D_k \cap \cup_{s \in \mathcal{S}} C_s|}{|D_k|}.$$

Clusters with a ratio lower that the threshold β are considered as new directions that are worth exploring: a new seed node is picked in each of them. In order to have a central node in each cluster, we simply choose the node with the highest degree.

Loop

The seeds selected at the end of step 3 form the new seed set $\mathcal{S}_{\text{new}}$ to which we apply the same three steps. We stop the algorithm when the number of communities is greater than I or if there is no new seed.

Finally we output all the communities C_s found from the seed nodes $s \in \mathcal{S}$. The pseudo-code of our algorithm is given below.

Algorithm 4 `MULTICOM`

Require: Graph $G = (V, E)$, seed set $S \subset V$, score function f, function `local`, parameters I, β.

$\mathcal{S} \leftarrow \emptyset$ (all seed nodes)
$\mathcal{S}_{\text{new}} \leftarrow S$ (new seed nodes)
$\mathcal{C} \leftarrow []$ (list of communities)
while $\mathcal{S}_{\text{new}} \neq \emptyset$ and $|\mathcal{C}| \leq I$ **do**
    # **Detecting communities from new seeds**
    for $s \in \mathcal{S}_{\text{new}}$ **do**
        $f_s \leftarrow$ compute score function
        $C_s \leftarrow$ `local`(f_s) (local community detection)
        $\mathcal{C}$.`push`(C_s)
    end for
    $\mathcal{S} \leftarrow \mathcal{S} \cup \mathcal{S}_{\text{new}}$ (add new seeds)
    # **Embedding and clustering**
    $\forall v \in V,\ x_v \leftarrow (f_s(v))_{s \in \mathcal{S}}$
    $D_1, ..., D_K \leftarrow$ clustering of $(x_v)_{v \in V}$ in $\mathbb{R}_+^{|S|} \setminus \{0\}$
    # **Picking new seeds**
    $\mathcal{S}_{\text{new}} \leftarrow \emptyset$
    $\mathcal{E} \leftarrow \cup_{C \in \mathcal{C}} C$ (explored nodes)
    for $k = 1, ..., K$ **do**
        if $|D_k \cap \mathcal{E}| < \beta |D_k|$ **then**
            $s_{\text{new}} \leftarrow$ node with highest degree in $D_k \setminus \mathcal{E}$
            $\mathcal{S}_{\text{new}} \leftarrow \mathcal{S}_{\text{new}} \cup \{s_{\text{new}}\}$
        end if
    end for
end while
return $\mathcal{C}$

6.4.3 Post-processing

Note that the communities returned by `MULTICOM` might intersect. This is consistent with the fact that nodes often belong to multiple communities in practice [Reid et al., 2013, Xie et al., 2013]. However, we might want to limit the number of nodes that the communities have in common, and consider that if two communities C_i, C_j share too many nodes, then they form only one community $C_i \cup C_j$. To do so, we apply a post-processing step, called `MERGE`, at the end of `MULTICOM` that merges two communities C_i and C_j if their F1 score $\text{F1}(C_i, C_j)$ is greater than a given threshold $\gamma \in [0, 1]$. In the following experiments we use `MERGE` with parameter $\gamma = \frac{1}{2}$.

6.5 Experiments

We now analyse the performance of our algorithm, both qualitatively and quantitatively, on real graphs.

6.5.1 Case study: Wikipedia

First, we illustrate the interest of our algorithm on an extract of Wikipedia presented in [West et al., 2009]. The dataset is built from a selection of articles from the English version of Wikipedia that matches the UK National Curriculum[1]. The nodes of the graph corresponds to Wikipedia articles, and we put an edge between two articles a and a' if there is an hyperlink to article a' in article a. We apply `MULTICOM` with a Personalized PageRank scoring function and using a cut based on conductance on the seed set $S = \{\text{Albert Einstein}\}$. With the parameter $I = 5$, the algorithm returns 6 communities, for a total of 153 nodes. For each of these communities, we list the top-5 nodes according to their PageRank:

- $C_1 = \{$Albert Einstein, Special relativity, Euclid, Wave, String theory$\}$

[1] `http://schools-wikipedia.org`

- C_2 = {Light, Electric field, Contact lens, Maxwell's equations, Semiconductor device}
- C_3 = {Gottfried Leibniz, Algebra, Pi, Game theory, Thermo- dynamics}
- C_4 = {Star, Red dwarf, Open cluster, Orion Nebula, Gliese 876}
- C_5 = {Quantum mechanics, Photon, Electromagnetic radiation, Electric charge, Linus Pauling}
- C_6 = {Atom, Renormalization, Mechanical work, Quark, Ununpentium}

The top node of each community $C_2, \ldots, C_6$ turns out to be a seed found by `MULTICOM`. We see that each community corresponds to a different facet of Einstein's work. Note that the state-of-the-art algorithms like Personalized PageRank only recover community C_1, as it corresponds to the first step of the `MULTICOM` algorithm. We see that we gain precious information by considering additional communities around the Albert Einstein article.

6.5.2 Real-world data

Datasets

For a quantitative evaluation of our algorithm on real-world graphs, we use the datasets available on the SNAP website [Yang and Leskovec, 2015]. All these datasets include ground-truth community memberships. We consider graphs of different types: the social network YouTube [Backstrom et al., 2006], the product co-purchasing graph built from Amazon data [Leskovec et al., 2007], and the DBLP dataset described in Section 6.4.

Algorithms

We compare `MUTLTICOM` to the three state-of-the-art algorithms presented in Section 6.2: Personalized PageRank (`PPR`), Heat Kernel (`HK`) and LEMON (`LEMON`). For `MULTICOM` we use a function `cut` that finds the local minimum for the conductance $\Phi(S_j)$, and we take Personalized PageRank for the scoring function f. We have implemented `MULTICOM` in Python and made it available on GitHub[2].

Performance evaluation

In order to evaluate the performance of the algorithms, we measure for each returned community $\hat{C}$ its conductance $\Phi(\hat{C})$ and the maximum F1-score between $\hat{C}$ and any ground-truth community.

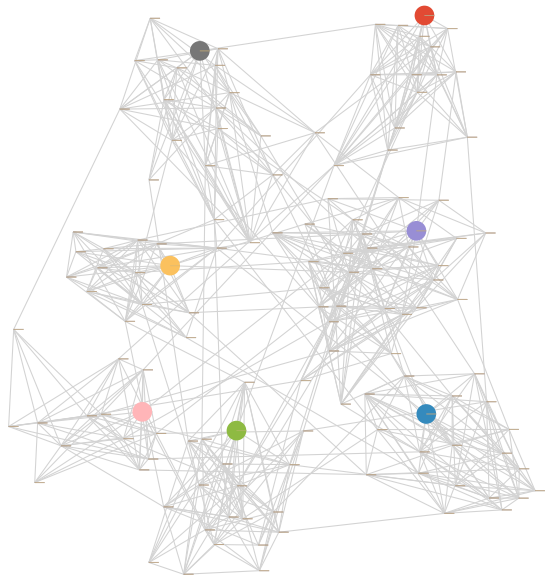
The state-of-the-art algorithms return only one community so the computation of these scores is straightforward. `MUTLTICOM` generally outputs several communities, so we consider the average values of conductance and F1-score for these communities.

We also compute the total number of nodes $|\mathcal{E}|$ *labeled* or *explored* by each algorithm, i.e. the total number of nodes that belong to a community at the end of each algorithm.

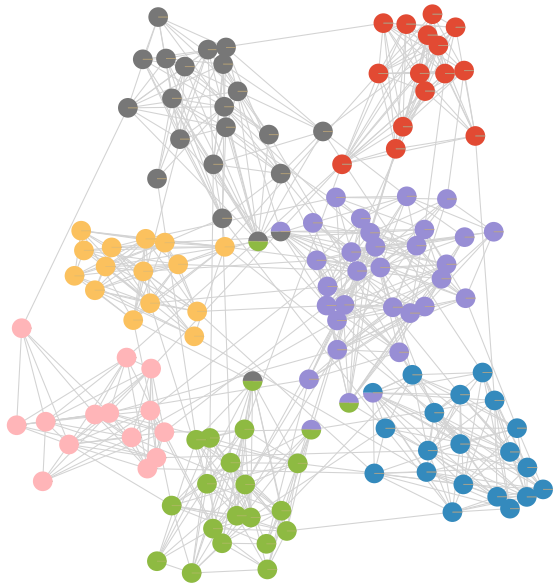
Benchmark

For each dataset, we pick 100 seed nodes uniformly at random and run the algorithms on each of these seed nodes. We use the parameter $I = 5$ and $\beta = 0.8$ for `MULTICOM`. The results are shown in Table 6.1.

Observe that the total number of nodes found by `MULTICOM`, corresponding to multiple communities, is much higher than those found by the other algorithms. In other words, the volume of information that `MULTICOM` collects in the neighborhood of the seed set is much more important. Moreover, the quality of this information is not too much downgraded by the multi-directional approach. Indeed, the average F1-score measured on the multiple communities returned by `MULTICOM` is essentially the same as the F1-scores of the other algorithms. The much higher F1-score of `LEMON` for the YouTube dataset is due to the much smaller communities detected by this algorithm.



(a) Detected seed nodes
(initial seed in red)



(b) Detected communities

Figure 6.5: Seeds and communities detected by `MULTICOM` in a SBM. The algorithm has been applied to a SBM random graph with an initial seed displayed **in red** in Figure (a). Figure (a) shows the new seeds detected by `MULTICOM`. Figure (b) shows the communities detected from these seeds. Note that some nodes belong to several communities (bi-colored nodes).

DBLP			
Algo.	$\lvert\mathcal{E}\rvert$	Φ	F1
MULTICOM	103	0.45 ± 0.14	0.23 ± 0.15
PPR	21	0.41 ± 0.17	0.23 ± 0.22
HK	15	0.32 ± 0.14	0.23 ± 0.26
LEMON	18	0.43 ± 0.20	0.26 ± 0.24
Amazon			
Algo.	$\lvert\mathcal{E}\rvert$	Φ	F1
MULTICOM	80	0.35 ± 0.14	0.44 ± 0.19
PPR	26	0.31 ± 0.16	0.46 ± 0.24
HK	24	0.27 ± 0.16	0.49 ± 0.26
LEMON	21	0.40 ± 0.27	0.48 ± 0.25
YouTube			
Algo.	$\lvert\mathcal{E}\rvert$	Φ	F1
MULTICOM	284	0.69 ± 0.11	0.05 ± 0.03
PPR	95	0.56 ± 0.25	0.05 ± 0.11
HK	100	0.55 ± 0.33	0.04 ± 0.11
LEMON	5	0.96 ± 0.12	0.12 ± 0.20

Table 6.1: Benchmark results on SNAP datasets

6.5.3 Synthetic data

We evaluate our algorithm on synthetic graphs generated with the Stochastic Block Model (SBM) [Decelle et al., 2011]. We illustrate the results of MULTICOM on such a synthetic graph in Figure 6.5. In the sub-figure (a), we display the new seeds found by the algorithm when initialized with an initial seed colored in red in the figure. In the sub-figure (b), we display the corresponding communities returned by the algorithm. Note that some communities are overlapping i.e. some nodes belong to more than one community.

In order to numerically evaluate the results of MULTICOM on the SBM model, we generate 100 graphs with 100 nodes and 5 communities of equal size. We pick a random seed node in each of these graphs and run MULTICOM starting from this seed with the same parameters as in §6.5.2. We use the same performance metrics (average conductance, average F1-score and number of *explored* nodes) to evaluate the performance of MULTICOM and we compare it to PPR. The results are shown in Table 6.2. We observe that MULTICOM recovers almost perfectly all 5 communities in each generated graph, whereas PPR recover only one of these communities.

Algo.	$\lvert\mathcal{E}\rvert$	Φ	F1
MULTICOM	98	0.11	0.98
PPR	20	0.11	0.97

Table 6.2: Benchmark results on a SBM model. Graph generated have 100 nodes and 5 equal-sized communities.

[2] https://github.com/ahollocou/multicom

Chapter 7

Conclusion

We presented the problem of graph clustering in Chapter 1 with its multiple facets, and gave numerous examples of applications in various fields. We have seen that there is no universal approach to this question and that the problem can be tackled from different angles. We also presented classic graph clustering methods that are cornerstones in all our work. In particular, we introduced *modularity*, a quality measure for clusters that can be defined and interpreted in multiple ways. We also gave an introduction to *spectral approaches* and *random walks* whose properties are tightly related to the cluster structure in graphs. The rest of our work was then dedicated to studying different facets of the graph clustering problem.

In Chapter 2, we studied *soft clustering* approaches to graph clustering, with the aim of determining degrees of membership to each cluster instead of performing a hard partitioning of graph nodes. We introduced a relaxation of the popular modularity maximization problem. In order to efficiently solve this relaxation, we introduced a novel algorithm, called *MODSOFT*, that is based on an alternating projected gradient descent. By diving into the specific form of the gradient descent updates, we were able to guarantee the locality of each algorithm step, and to make good use of the sparsity of the solutions. As a result, unlike existing methods, our approach is both local and memory efficient. Furthermore, we proved that our algorithm includes the main routine of the popular Louvain algorithm for a learning rate above a certain threshold, $t > w/\delta$. We illustrated on real-life examples that our algorithm has an execution time and a memory consumption comparable to the Louvain algorithm, but goes further than Louvain by identifying nodes with mixed memberships that represent important bridges between clusters, even if this does not always translate into a large modularity increase. Note that, in practical cases, the learning rate t is sometimes hard to tune. It would be interesting for future work to find a way to choose a correct value for t. Also note that our algorithm tends to be trapped in local optima of the soft modularity function. A solution to counter this would be to use an adaptive learning rate or to combine the use of Modsoft with another algorithm, for instance the softmax approach proposed by Chang et al. In future works, it would also be interesting to compare our approach to a non-negative matrix factorization method applied to the adjacency matrix A, which can be performed with a comparable alternating projected gradient descent [Lin, 2007]. Moreover, in some of our experiments, it proves more efficient to consider the more general update rule

$$p_{i\cdot} \leftarrow \pi_{\mathcal{Y}} \left(b p_{i\cdot} + t' \sum_{j \sim i} A_{ij} (p_{j\cdot} - \bar{p}) \right)$$

where $b \geq 0$ is a bias parameter. We think that it might be worth understanding the impact of a $b \neq 1$ on the solution for future work.

In Chapter 3, we proposed a novel agglomerative algorithm for the edge clustering problem that maximizes a variant of Newman's modularity that emerges by considering a random walk on the graph edges instead of the graph nodes. This algorithm does not require the construction of a weighted line graph unlike the existing approaches. It is based on a special aggregation operation that conserves the edge modularity as proven in Theorem 3.5.3. Thanks to this aggregation step, the size of the graph decreases at each iteration step of the algorithm which enables it to handle large graphs. We showed on both synthetic and real-world datasets that our algorithm significantly outperforms Evans and Lambiotte's methods in terms of memory consumption and time execution. Besides, we illustrated on the OpenFlights dataset

that the aggregated graph produced by our algorithm can be used as a powerful tool for visualization. For future work, we would like to speed up the first greedy maximization step of the algorithm before any aggregation takes place. This first step is indeed the most expensive one and could be accelerated using heuristics based on standard modularity maximization techniques for instance. Besides, note that our edge clustering approach is closely related to approaches using second-order Markov models, like the one introduced in [Rosvall et al., 2014], that uses the map equation framework [Rosvall and Bergstrom, 2008]. In such models, the transitions can be interpreted as movements between links instead of movements between nodes, just as the the random walk on edges $(Y_t)_{t\geq 0}$ that we introduced in Chapter 3. Such higher-order Markov models have recently regained interests because of their connection to higher-order structures [Benson et al., 2016]. It would be interesting for future work to study in details the links between these techniques and our approach.

In Chapter 4, we have proposed an agglomerative clustering algorithm for graphs, called *Paris*, based on a reducible distance between clusters. The algorithm is parameter-free and uses the nearest-neighbor chain technique traditionally used to cluster vector in Euclidean spaces. We showed that the distance we introduce is tightly linked to modularity and that Paris can be seen as modified version of Louvain using a *sliding* resolution. Unlike Louvain, our algorithm is able to capture the multi-scale structure of real graphs. We showed through experiments both on synthetic graphs (with the HSBM) and on real datasets that our algorithm is fast and memory-efficient. For future work, we plan to work on the automatic extraction of the most relevant clusterings from the dendrogram at different scales, which must be done manually with the existing version. For instance, a way to perform this task would be to extract the levels that correspond to the largest gaps between successive distances in the dendrogram, or to take the levels corresponding to the largest distance *ratios*. This is equivalent to consider the largest gaps in the dendrogram where distances are represented in log scale. With this technique, we could obtain theoretical gurantees on the hierarchical stochastic block model that we consider in our experiments.

In Chapter 5, we introduced a new algorithm for the problem of graph clustering in the edge streaming setting. In this setting, the input data is presented to the algorithm as a sequence of edges that can be examined only once. Our algorithm only stores three integers per node and requires only one integer parameter $v_{\max}$. It runs more than 10 times faster than state-of-the-art algorithms such as Louvain and SCD and shows better detection scores on the largest graphs. Such an algorithm is extremely useful in many applications where massive graphs arise. For instance, the web graph contains around 10^{10} nodes which is much more than in the Friendster dataset. We analyzed the adaptation of the popular modularity problem to the streaming setting. Theorem 5.4.3 justifies the nature of the condition on the volumes of the communities of nodes i and j for each new edge (i, j), which is the core of Algorithm 3. It would be interesting for future work to perform further experiments. First, we could perform additional experiments on synthetic datasets using stochastic block models or the LFR model introduced in Chapter 1. Then, the ability of the algorithm to handle evolving graphs could be evaluated on dynamic datasets [Panzarasa et al., 2009] and compared to existing approaches [Gauvin et al., 2014, Epasto et al., 2015]. Note that, in the dynamic network settings, modifications to the algorithm design could be made to handle events such as edge deletions. Another important subject for future work is the influence of the ordering of the edges on the final result obtained with our algorithm. In practice, we qualitatively observe that the result does not depend much on the edge order in the stream, but it would be interesting to obtain quantitative results on this stability. Finally, our algorithm only returns disjoint communities. An important research direction would consist in adapting approach to overlapping community detection and compare it to existing approaches [Xie et al., 2013, Yang and Leskovec, 2013].

In Chapter 6, we have presented an algorithm for multiple local community detection. The approach relies on the local embedding of the graph near the seed set, using a scoring function such as Personal PageRank. We have seen that the target communities are typically well separated in the embedding space. Building on this observation, we have proposed a clustering method to identify new seeds in the neighborhood of the initial seed set, so as to recover different communities in various directions of the graph. For future work, we would like to compare the local embedding obtained with `MULTICOM` with the existing local spectral embedding, and use it for other tasks such as network visualization and link prediction.

As explained in the introduction, there is no universal definition of what a good cluster is. We have addressed some of the many facets of the complex problem of graph clustering by introducing a range of different approaches. However it is not necessarily clear at this point which technique should be used

for specific applications. We think that a hierarchical approach such as the one proposed in Paris is an appropriate way to start any real-life graph analysis. Indeed, it is crucial to determine what are the relevant scales for the problem we are interested in, and an algorithm like Paris offers a solution for identifying these scales. Besides, Paris helps us to detect the relevant resolutions for modularity-based approach. The values of these resolutions can therefore be used in any modularity-based algorithm that we have introduced. The big disadvantage of a hierarchical technique such as Paris is that each level in the dendrogram returned by the algorithm corresponds a node partition, so that each node belongs to one and only cluster, which is very limiting in practice. To overcome this problem, a natural second step in the analysis of a real-life dataset would be to apply a soft-clustering approach like Modsoft or an edge clustering approach such as the one introduced in Chapter 3 at the relevant resolutions determined during the first step of the analysis. The advantage of the edge clustering approach is that it is easy to interpret, as we simply cluster the relations between the entities rather than the entities themselves. However, such a result is often hard to visualize in practice as there could be many edge clusters connected to a single node (see the example of OpenFlights presented in Chapter 3 for instance). The results of Modsoft are easier to visualize as there are few nodes with mixed membership in practice, however the parameter of the algorithm is harder to tune, which can make the analysis more difficult.

Note that we have only discussed cases where we have access to the entire graph and where we can handle it in one batch. If these conditions are not satisfied, which might be the case in real-life big data applications, one should turn to streaming or local approaches. The disadvantage of such approaches is that they are today much less developed and refined that the techniques we have mentioned earlier. This is especially the case for streaming approaches that only output node partitions.

As the variety of graph clustering approaches that we have introduced is very wide, it would be interesting to compare these methods with each other for future work. However, such a comparison would be a difficult task. Indeed, the evaluation of traditional graph clustering methods on real datasets is already troublesome as noted in Chapter 1, mainly because the definition of ground-truth clusters is highly subjective. Therefore, it is even more complex to compare techniques that perform tasks as different as soft node clustering, edge partitioning and hierarchical clustering on such real datasets. A perfect dataset would contain for each node all the ground-truth clusters to which this node can possibly belong at all possible scales, which is a pipe dream. Of course, synthetic datasets that model all these different aspects could certainly be generated using, for instance, a stochastic block model that combine overlapping blocks and a multi-scale structure as in the HSBM introduced in Chapter 4. But such a model would certainly have many parameters, which would probably make the experimental comparison of the different algorithms intractable.

Besides, if it would be useful to compare the different algorithms presented in this work as explained above, it would also be very interesting to study novel methods that combine these different techniques to tackle several aspects of the graph clustering problem at the same time. For instance, the hierarchical algorithms studied in Chapter 4 consider that clusters form a partition of the graph nodes at all level of the dendrogram, whereas it would be interesting to define a new type of hierarchies such that we could have at each level a soft clustering of the nodes or a partition of the graph edges. Considering that the methods presented in our work to perform these different tasks are all based on modularity, we believe that they could be *mixed* together to perform such hybrid tasks with a reasonable effort. For instance, we see that it is easy to generalize the reducible distance between nodes introduced in Chapter 4 to define a distance between edges, using the logic presented in Chapter 3. It seems however more challenging to combine the relaxation of modularity introduced in Chapter 2 with the hierarchical agglomerative approach used in the Paris algorithm.

Finally, note that the list of problems we have studied in the present work is far from being comprehensive. Indeed, we focused on some crucial aspects of the complex problem of graph clustering and meanwhile turned a blind eye on some other important facets of the question [Schaub et al., 2017]. For instance, we could have studied the problem of clustering *annotated graphs* [Newman and Clauset, 2016, Yang et al., 2013, Chang et al., 2018a], i.e. graphs whose nodes and/or edges can carry *attributes* or *features*. Indeed, in many real-life applications, we have access to rich information about nodes and edges in addition to the graph structure. This is the case, for instance, in a social network such as Facebook, where we have a great amount of data about nodes representing users, including their age, their gender, and their nationality. In the Spotify or Deezer user-artist graph, we also have a lot of supplementary information about the edges of the graph. For example, we know how many times a user has listened

to an artist, how regularly and to how many songs. Another important issue for which we only have scratched the surface is the problem of *dynamic graph clustering*, which consists in finding clusters in graphs that evolve over time. The streaming algorithm proposed in Chapter 5, which is mostly designed to handle massive graphs, can also be applied to dynamic graphs where edges are inserted. However, in many real-life applications, we must consider that nodes and edges can be inserted and deleted over time [Gauvin et al., 2014]. This is the case for instance of the Wikipedia graph where nodes representing articles can be created and removed, and where the modification of articles can lead to the formation of new edges or the demise of existing ones. In this framework, an interesting problem would be to adapt the different algorithms that we presented in this work so that they could be applied to time-evolving graphs without having to run the algorithm on the whole graph at each time step.

Bibliography

[Abbe, 2017] Abbe, E. (2017). Community detection and stochastic block models: recent developments. *arXiv preprint arXiv:1703.10146*.

[Ahn et al., 2012] Ahn, K. J., Guha, S., and McGregor, A. (2012). Graph sketches: sparsification, spanners, and subgraphs. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI symposium on Principles of Database Systems*, pages 5–14. ACM.

[Ahn et al., 2010] Ahn, Y.-Y., Bagrow, J. P., and Lehmann, S. (2010). Link communities reveal multiscale complexity in networks. *nature*, 466(7307):761.

[Airoldi et al., 2008] 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.

[Andersen and Lang, 2006] Andersen, R. and Lang, K. J. (2006). Communities from seed sets. In *Proceedings of the 15th international conference on World Wide Web*, pages 223–232. ACM.

[Arenas et al., 2008] Arenas, A., Fernandez, A., and Gomez, S. (2008). Analysis of the structure of complex networks at different resolution levels. *New Journal of Physics*, 10(5):053039.

[Arthur and Vassilvitskii, 2007] Arthur, D. and Vassilvitskii, S. (2007). k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics.

[Backstrom et al., 2006] Backstrom, L., Huttenlocher, D., Kleinberg, J., and Lan, X. (2006). Group formation in large social networks: membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 44–54. ACM.

[Balakrishnan, 1997] Balakrishnan, V. (1997). *Schaum's Outline of Graph Theory: Including Hundreds of Solved Problems*. McGraw Hill Professional.

[Bar-Yossef et al., 2002] Bar-Yossef, Z., Kumar, R., and Sivakumar, D. (2002). Reductions in streaming algorithms, with an application to counting triangles in graphs. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 623–632. Society for Industrial and Applied Mathematics.

[Barabási and Albert, 1999] Barabási, A.-L. and Albert, R. (1999). Emergence of scaling in random networks. *science*, 286(5439):509–512.

[Barabási et al., 2016] Barabási, A.-L. et al. (2016). *Network science*. Cambridge university press.

[Benczúr and Karger, 1996] Benczúr, A. A. and Karger, D. R. (1996). Approximating st minimum cuts in õ (n 2) time. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 47–55. ACM.

[Benson et al., 2016] Benson, A. R., Gleich, D. F., and Leskovec, J. (2016). Higher-order organization of complex networks. *Science*, 353(6295):163–166.

[Benzécri, 1982] Benzécri, J.-P. (1982). Construction d'une classification ascendante hiérarchique par la recherche en chaîne des voisins réciproques. *Les cahiers de l'analyse des données*, 7(2):209–218.

[Bezdek, 1981] Bezdek, J. C. (1981). Objective function clustering. In *Pattern recognition with fuzzy objective function algorithms*, pages 43–93. Springer.

[Blondel et al., 2008] 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.

[Bollobás, 2004] Bollobás, B. (2004). *Extremal graph theory*. Courier Corporation.

[Bonald et al., 2018a] Bonald, T., Charpentier, B., Galland, A., and Hollocou, A. (2018a). Hierarchical graph clustering by node pair sampling. *KDD Workshop*.

[Bonald et al., 2018b] Bonald, T., De Lara, N., and Hollocou, A. (2018b). Forward and backward embedding of directed graphs. *Preprint*.

[Bonald et al., 2018c] Bonald, T., Hollocou, A., and Lelarge, M. (2018c). Weighted spectral embedding of graphs. In *Communication, Control, and Computing (Allerton), 2018 56th Annual Allerton Conference on*.

[Brandes et al., 2007] Brandes, U., Delling, D., Gaertler, M., Görke, R., Hoefer, M., Nikoloski, Z., and Wagner, D. (2007). On finding graph clusterings with maximum modularity. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 121–132. Springer.

[Breitkreutz et al., 2007] Breitkreutz, B.-J., Stark, C., Reguly, T., Boucher, L., Breitkreutz, A., Livstone, M., Oughtred, R., Lackner, D. H., Bähler, J., Wood, V., et al. (2007). The biogrid interaction database: 2008 update. *Nucleic acids research*, 36(suppl_1):D637–D640.

[Buriol et al., 2006] Buriol, L. S., Frahling, G., Leonardi, S., Marchetti-Spaccamela, A., and Sohler, C. (2006). Counting triangles in data streams. In *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 253–262. ACM.

[Chang et al., 2018a] Chang, C.-H., Chang, C.-S., Chang, C.-T., Lee, D.-S., and Lu, P.-E. (2018a). Exponentially twisted sampling for centrality analysis and community detection in attributed networks. *IEEE Transactions on Network Science and Engineering*.

[Chang et al., 2015] Chang, C.-S., Chang, C.-J., Hsieh, W.-T., Lee, D.-S., Liou, L.-H., and Liao, W. (2015). Relative centrality and local community detection. *Network Science*, 3(4):445–479.

[Chang et al., 2011] Chang, C.-S., Hsu, C.-Y., Cheng, J., and Lee, D.-S. (2011). A general probabilistic framework for detecting community structure in networks. In *INFOCOM, 2011 Proceedings IEEE*, pages 730–738. IEEE.

[Chang et al., 2018b] Chang, C.-S., Lee, D.-S., Liou, L.-H., Lu, S.-M., and Wu, M.-H. (2018b). A probabilistic framework for structural analysis and community detection in directed networks. *IEEE/ACM Transactions on Networking (TON)*, 26(1):31–46.

[Chang and Chang, 2017] Chang, C.-T. and Chang, C.-S. (2017). A unified framework for sampling, clustering and embedding data points in semi-metric spaces. *arXiv preprint arXiv:1708.00316*.

[Chung, 2007] Chung, F. (2007). The heat kernel as the pagerank of a graph. *Proceedings of the National Academy of Sciences*, 104(50):19735–19740.

[Chung, 2009] Chung, F. (2009). A local graph partitioning algorithm using heat kernel pagerank. *Internet Mathematics*, 6(3):315–330.

[Chung, 1997] Chung, F. R. (1997). *Spectral graph theory*. Number 92. American Mathematical Soc.

[Clauset, 2005] Clauset, A. (2005). Finding local community structure in networks. *Physical review E*, 72(2):026132.

[Clauset et al., 2008] Clauset, A., Moore, C., and Newman, M. E. (2008). Hierarchical structure and the prediction of missing links in networks. *Nature*, 453(7191):98.

[Danisch et al., 2014] Danisch, M., Guillaume, J.-L., and Le Grand, B. (2014). Learning a proximity measure to complete a community. In *Data Science and Advanced Analytics (DSAA), 2014 International Conference on*, pages 90–96. IEEE.

[Day and Edelsbrunner, 1984] Day, W. H. and Edelsbrunner, H. (1984). Efficient algorithms for agglomerative hierarchical clustering methods. *Journal of classification*, 1(1):7–24.

[Decelle et al., 2011] Decelle, A., Krzakala, F., Moore, C., and Zdeborová, L. (2011). Asymptotic analysis of the stochastic block model for modular networks and its algorithmic applications. *Physical Review E*, 84(6):066106.

[Delvenne et al., 2010] Delvenne, J.-C., Yaliraki, S. N., and Barahona, M. (2010). Stability of graph communities across time scales. *Proceedings of the national academy of sciences*, 107(29):12755–12760.

[Donetti and Munoz, 2004] Donetti, L. and Munoz, M. A. (2004). Detecting network communities: a new systematic and efficient algorithm. *Journal of Statistical Mechanics: Theory and Experiment*, 2004(10):P10012.

[Duchi et al., 2008] Duchi, J., Shalev-Shwartz, S., Singer, Y., and Chandra, T. (2008). Efficient projections onto the l 1-ball for learning in high dimensions. In *Proceedings of the 25th international conference on Machine learning*, pages 272–279. ACM.

[Dunn, 1973] Dunn, J. C. (1973). A fuzzy relative of the isodata process and its use in detecting compact well-separated clusters.

[Elkin and Zhang, 2006] Elkin, M. and Zhang, J. (2006). Efficient algorithms for constructing $(1+\epsilon, \beta)$-spanners in the distributed and streaming models. *Distributed Computing*, 18(5):375–385.

[Epasto et al., 2015] Epasto, A., Lattanzi, S., and Sozio, M. (2015). Efficient densest subgraph computation in evolving graphs. In *Proceedings of the 24th International Conference on World Wide Web*, pages 300–310. ACM.

[Eppstein, 2000] Eppstein, D. (2000). Fast hierarchical clustering and other applications of dynamic closest pairs. *Journal of Experimental Algorithmics (JEA)*, 5:1.

[Ester et al., 1996] Ester, M., Kriegel, H.-P., Sander, J., Xu, X., et al. (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231.

[Evans and Lambiotte, 2009] Evans, T. and Lambiotte, R. (2009). Line graphs, link partitions, and overlapping communities. *Physical Review E*, 80(1):016105.

[Feigenbaum et al., 2005] Feigenbaum, J., Kannan, S., McGregor, A., Suri, S., and Zhang, J. (2005). On graph problems in a semi-streaming model. *Theoretical Computer Science*, 348(2-3):207–216.

[Fortunato, 2010] Fortunato, S. (2010). Community detection in graphs. *Physics reports*, 486(3):75–174.

[Fortunato and Barthelemy, 2007] Fortunato, S. and Barthelemy, M. (2007). Resolution limit in community detection. *Proceedings of the National Academy of Sciences*, 104(1):36–41.

[Fortunato and Castellano, 2012] Fortunato, S. and Castellano, C. (2012). Community structure in graphs. In *Computational Complexity*, pages 490–512. Springer.

[Fowlkes et al., 2004] Fowlkes, C., Belongie, S., Chung, F., and Malik, J. (2004). Spectral grouping using the nystrom method. *IEEE transactions on pattern analysis and machine intelligence*, 26(2):214–225.

[Frieze and Karoński, 2016] Frieze, A. and Karoński, M. (2016). *Introduction to random graphs*. Cambridge University Press.

[Garey et al., 1982] Garey, M., Johnson, D., and Witsenhausen, H. (1982). The complexity of the generalized lloyd-max problem (corresp.). *IEEE Transactions on Information Theory*, 28(2):255–256.

[Gauvin et al., 2014] Gauvin, L., Panisson, A., and Cattuto, C. (2014). Detecting the community structure and activity patterns of temporal networks: a non-negative tensor factorization approach. *PloS one*, 9(1):e86028.

[Girvan and Newman, 2002] Girvan, M. and Newman, M. E. (2002). Community structure in social and biological networks. *Proceedings of the national academy of sciences*, 99(12):7821–7826.

[Gleich, 2015] Gleich, D. F. (2015). Pagerank beyond the web. *SIAM Review*, 57(3):321–363.

[Goel et al., 2012] Goel, A., Kapralov, M., and Khanna, S. (2012). On the communication and streaming complexity of maximum bipartite matching. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, pages 468–485. SIAM.

[Goldstein, 1964] Goldstein, A. A. (1964). Convex programming in hilbert space. *Bulletin of the American Mathematical Society*, 70(5):709–710.

[Griechisch and Pluhár, 2011] Griechisch, E. and Pluhár, A. (2011). Community detection by using the extended modularity. *Acta Cybern.*, 20(1):69–85.

[Grover and Leskovec, 2016] 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. ACM.

[Guimera et al., 2004] Guimera, R., Sales-Pardo, M., and Amaral, L. A. N. (2004). Modularity from fluctuations in random graphs and complex networks. *Physical Review E*, 70(2):025101.

[Hassan et al., 2015] Hassan, M., Makkaoui, O., Coulet, A., and Toussaint, Y. (2015). Extracting disease-symptom relationships by learning syntactic patterns from dependency graphs. In *BioNLP 15*, page 184.

[Havens et al., 2013] Havens, T. C., Bezdek, J. C., Leckie, C., Ramamohanarao, K., and Palaniswami, M. (2013). A soft modularity function for detecting fuzzy communities in social networks. *IEEE Transactions on Fuzzy Systems*, 21(6):1170–1175.

[Heimlicher et al., 2012] Heimlicher, S., Lelarge, M., and Massoulié, L. (2012). Community detection in the labelled stochastic block model. *arXiv preprint arXiv:1209.2910*.

[Holland et al., 1983] Holland, P. W., Laskey, K. B., and Leinhardt, S. (1983). Stochastic blockmodels: First steps. *Social networks*, 5(2):109–137.

[Hollocou et al., 2017a] Hollocou, A., Bonald, T., and Lelarge, M. (2017a). Multiple local community detection. *ACM SIGMETRICS Performance Evaluation Review*, 45(2):76–83.

[Hollocou et al., 2019] Hollocou, A., Bonald, T., and Lelarge, M. (2019). Modularity-based sparse soft graph clustering. *AISTATS*.

[Hollocou et al., 2018] Hollocou, A., Lutz, Q., and Bonald, T. (2018). A fast agglomerative algorithm for edge clustering. *Preprint*.

[Hollocou et al., 2017b] Hollocou, A., Maudet, J., Bonald, T., and Lelarge, M. (2017b). A streaming algorithm for graph clustering. *NIPS Workshop*.

[Huang et al., 2010] Huang, J., Sun, H., Han, J., Deng, H., Sun, Y., and Liu, Y. (2010). Shrink: a structural clustering algorithm for detecting hierarchical communities in networks. In *Proceedings of the 19th ACM international conference on Information and knowledge management*, pages 219–228. ACM.

[Huang et al., 2014] Huang, X., Cheng, H., Qin, L., Tian, W., and Yu, J. X. (2014). Querying k-truss community in large and dynamic graphs. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1311–1322. ACM.

[Jeub et al., 2015] Jeub, L. G., Balachandran, P., Porter, M. A., Mucha, P. J., and Mahoney, M. W. (2015). Think locally, act locally: Detection of small, medium-sized, and large communities in large networks. *Physical Review E*, 91(1):012821.

[Juan, 1982] Juan, J. (1982). Programme de classification hiérarchique par l'algorithme de la recherche en chame des voisins réciproques. *Les Cahiers de l'Analyse des Données*, 7(2):219–225.

[Karrer and Newman, 2011] Karrer, B. and Newman, M. E. (2011). Stochastic blockmodels and community structure in networks. *Physical review E*, 83(1):016107.

[Kaufman and Rousseeuw, 1987] Kaufman, L. and Rousseeuw, P. (1987). *Clustering by means of medoids*. North-Holland.

[Kawase et al., 2016] Kawase, Y., Matsui, T., and Miyauchi, A. (2016). Additive approximation algorithms for modularity maximization. *arXiv preprint arXiv:1601.03316*.

[Kleinberg, 2003] Kleinberg, J. M. (2003). An impossibility theorem for clustering. In *Advances in neural information processing systems*, pages 463–470.

[Kloster and Gleich, 2014] Kloster, K. and Gleich, D. F. (2014). Heat kernel based community detection. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1386–1395. ACM.

[Kloumann and Kleinberg, 2014] Kloumann, I. M. and Kleinberg, J. M. (2014). Community membership identification from small seed sets. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1366–1375. ACM.

[Lambiotte et al., 2008] Lambiotte, R., Delvenne, J.-C., and Barahona, M. (2008). Laplacian dynamics and multiscale modular structure in networks. *arXiv preprint arXiv:0812.1770*.

[Lambiotte et al., 2014] Lambiotte, R., Delvenne, J.-C., Barahona, M., et al. (2014). Random walks, markov processes and the multiscale modular organization of complex networks. *IEEE Trans. Network Science and Engineering*, 1(2):76–90.

[Lancichinetti and Fortunato, 2009] Lancichinetti, A. and Fortunato, S. (2009). Community detection algorithms: a comparative analysis. *Physical review E*, 80(5):056117.

[Lancichinetti et al., 2009] Lancichinetti, A., Fortunato, S., and Kertész, J. (2009). Detecting the overlapping and hierarchical community structure in complex networks. *New Journal of Physics*, 11(3):033015.

[Lancichinetti et al., 2008] Lancichinetti, A., Fortunato, S., and Radicchi, F. (2008). Benchmark graphs for testing community detection algorithms. *Physical review E*, 78(4):046110.

[Lancichinetti et al., 2011] Lancichinetti, A., Radicchi, F., Ramasco, J. J., and Fortunato, S. (2011). Finding statistically significant communities in networks. *PloS one*, 6(4):e18961.

[Lanczos, 1950] Lanczos, C. (1950). *An iteration method for the solution of the eigenvalue problem of linear differential and integral operators*. United States Governm. Press Office Los Angeles, CA.

[Lee and Seung, 2001] Lee, D. D. and Seung, H. S. (2001). Algorithms for non-negative matrix factorization. In *Advances in neural information processing systems*, pages 556–562.

[Lelarge et al., 2015] Lelarge, M., Massoulié, L., and Xu, J. (2015). Reconstruction in the labelled stochastic block model. *IEEE Transactions on Network Science and Engineering*, 2(4):152–163.

[Leskovec et al., 2007] Leskovec, J., Adamic, L. A., and Huberman, B. A. (2007). The dynamics of viral marketing. *ACM Transactions on the Web (TWEB)*, 1(1):5.

[Leskovec et al., 2008] Leskovec, J., Lang, K. J., Dasgupta, A., and Mahoney, M. W. (2008). Statistical properties of community structure in large social and information networks. In *Proceedings of the 17th international conference on World Wide Web*, pages 695–704. ACM.

[Leskovec and Mcauley, 2012] Leskovec, J. and Mcauley, J. J. (2012). Learning to discover social circles in ego networks. In *Advances in neural information processing systems*, pages 539–547.

[Levitin and Polyak, 1966] Levitin, E. S. and Polyak, B. T. (1966). Constrained minimization methods. *USSR Computational mathematics and mathematical physics*, 6(5):1–50.

[Li et al., 2015] Li, Y., He, K., Bindel, D., and Hopcroft, J. E. (2015). Uncovering the small community structure in large networks: A local spectral approach. In *Proceedings of the 24th international conference on world wide web*, pages 658–668. International World Wide Web Conferences Steering Committee.

[Lin, 2007] Lin, C.-J. (2007). Projected gradient methods for nonnegative matrix factorization. *Neural computation*, 19(10):2756–2779.

[Lloyd, 1982] Lloyd, S. (1982). Least squares quantization in pcm. *IEEE transactions on information theory*, 28(2):129–137.

[Lovász et al., 1993] Lovász, L. et al. (1993). Random walks on graphs: A survey. *Combinatorics, Paul erdos is eighty*, 2(1):1–46.

[Lyzinski et al., 2017] Lyzinski, V., Tang, M., Athreya, A., Park, Y., and Priebe, C. E. (2017). Community detection and classification in hierarchical stochastic blockmodels. *IEEE Transactions on Network Science and Engineering*, 4(1):13–26.

[Mahajan et al., 2009] Mahajan, M., Nimbhorkar, P., and Varadarajan, K. (2009). The planar k-means problem is np-hard. In *International Workshop on Algorithms and Computation*, pages 274–285. Springer.

[Mahoney et al., 2012] Mahoney, M. W., Orecchia, L., and Vishnoi, N. K. (2012). A local spectral method for graphs: With applications to improving graph partitions and exploring data graphs locally. *Journal of Machine Learning Research*, 13(Aug):2339–2365.

[McDaid et al., 2011] McDaid, A. F., Greene, D., and Hurley, N. (2011). Normalized mutual information to evaluate overlapping community finding algorithms. *arXiv preprint arXiv:1110.2515*.

[McGregor, 2014] McGregor, A. (2014). Graph stream algorithms: a survey. *ACM SIGMOD Record*, 43(1):9–20.

[Mehler and Skiena, 2009] Mehler, A. and Skiena, S. (2009). Expanding network communities from representative examples. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 3(2):7.

[Mikolov et al., 2013] Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

[Mislove et al., 2007] Mislove, A., Marcon, M., Gummadi, K. P., Druschel, P., and Bhattacharjee, B. (2007). Measurement and analysis of online social networks. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pages 29–42. ACM.

[Molloy and Reed, 1995] Molloy, M. and Reed, B. (1995). A critical point for random graphs with a given degree sequence. *Random structures & algorithms*, 6(2-3):161–180.

[Murtagh and Contreras, 2012] 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, 2004] Newman, M. E. (2004). Fast algorithm for detecting community structure in networks. *Physical review E*, 69(6):066133.

[Newman, 2006] Newman, M. E. (2006). Finding community structure in networks using the eigenvectors of matrices. *Physical review E*, 74(3):036104.

[Newman, 2016] Newman, M. E. (2016). Equivalence between modularity optimization and maximum likelihood methods for community detection. *Physical Review E*, 94(5):052315.

[Newman and Clauset, 2016] Newman, M. E. and Clauset, A. (2016). Structure and inference in annotated networks. *Nature communications*, 7:11863.

[Newman and Girvan, 2004] Newman, M. E. and Girvan, M. (2004). Finding and evaluating community structure in networks. *Physical review E*, 69(2):026113.

[Ng et al., 2002] Ng, A. Y., Jordan, M. I., and Weiss, Y. (2002). On spectral clustering: Analysis and an algorithm. In *Advances in neural information processing systems*, pages 849–856.

[Nicosia et al., 2009] Nicosia, V., Mangioni, G., Carchiolo, V., and Malgeri, M. (2009). Extending the definition of modularity to directed graphs with overlapping communities. *Journal of Statistical Mechanics: Theory and Experiment*, 2009(03):P03024.

[Orecchia and Zhu, 2014] Orecchia, L. and Zhu, Z. A. (2014). Flow-based algorithms for local graph clustering. In *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 1267–1286. Society for Industrial and Applied Mathematics.

[Page et al., 1999] Page, L., Brin, S., Motwani, R., and Winograd, T. (1999). The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab.

[Panzarasa et al., 2009] Panzarasa, P., Opsahl, T., and Carley, K. M. (2009). Patterns and dynamics of users' behavior and interaction: Network analysis of an online community. *Journal of the American Society for Information Science and Technology*, 60(5):911–932.

[Peel et al., 2017] Peel, L., Larremore, D. B., and Clauset, A. (2017). The ground truth about metadata and community detection in networks. *Science advances*, 3(5):e1602548.

[Perozzi et al., 2014] Perozzi, B., Al-Rfou, R., and Skiena, S. (2014). Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710. ACM.

[Perron, 1907] Perron, O. (1907). Grundlagen für eine theorie des jacobischen kettenbruchalgorithmus. *Mathematische Annalen*, 64(1):1–76.

[Pons and Latapy, 2005] Pons, P. and Latapy, M. (2005). Computing communities in large networks using random walks. In *International Symposium on Computer and Information Sciences*, pages 284–293. Springer.

[Prat-Pérez et al., 2014] Prat-Pérez, A., Dominguez-Sal, D., and Larriba-Pey, J.-L. (2014). High quality, scalable and parallel community detection for large real graphs. In *Proceedings of the 23rd international conference on World wide web*, pages 225–236. ACM.

[Reichardt and Bornholdt, 2006] Reichardt, J. and Bornholdt, S. (2006). Statistical mechanics of community detection. *Physical Review E*, 74(1):016110.

[Reid et al., 2013] Reid, F., McDaid, A., and Hurley, N. (2013). Partitioning breaks communities. In *Mining Social Networks and Security Informatics*, pages 79–105. Springer.

[Ronhovde and Nussinov, 2009] Ronhovde, P. and Nussinov, Z. (2009). Multiresolution community detection for megascale networks by information-based replica correlations. *Physical Review E*, 80(1):016109.

[Rosvall and Bergstrom, 2008] Rosvall, M. and Bergstrom, C. T. (2008). Maps of random walks on complex networks reveal community structure. *Proceedings of the National Academy of Sciences*, 105(4):1118–1123.

[Rosvall et al., 2014] Rosvall, M., Esquivel, A. V., Lancichinetti, A., West, J. D., and Lambiotte, R. (2014). Memory in network flows and its effects on spreading dynamics and community detection. *Nature communications*, 5:4630.

[Saade et al., 2014] Saade, A., Krzakala, F., and Zdeborová, L. (2014). Spectral clustering of graphs with the bethe hessian. In *Advances in Neural Information Processing Systems*, pages 406–414.

[Sales-Pardo et al., 2007] Sales-Pardo, M., Guimera, R., Moreira, A. A., and Amaral, L. A. N. (2007). Extracting the hierarchical organization of complex systems. *Proceedings of the National Academy of Sciences*, 104(39):15224–15229.

[Schaub et al., 2017] Schaub, M. T., Delvenne, J.-C., Rosvall, M., and Lambiotte, R. (2017). The many facets of community detection in complex networks. *Applied Network Science*, 2(1):4.

[Shi et al., 2013] Shi, C., Cai, Y., Fu, D., Dong, Y., and Wu, B. (2013). A link clustering based overlapping community detection algorithm. *Data & Knowledge Engineering*, 87:394–404.

[Shi and Malik, 2000] Shi, J. and Malik, J. (2000). Normalized cuts and image segmentation. *IEEE Transactions on pattern analysis and machine intelligence*, 22(8):888–905.

[Simpson, 1949] Simpson, E. H. (1949). Measurement of diversity. *nature*.

[Sozio and Gionis, 2010] Sozio, M. and Gionis, A. (2010). The community-search problem and how to plan a successful cocktail party. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 939–948. ACM.

[Spielman and Teng, 2004] Spielman, D. A. and Teng, S.-H. (2004). Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 81–90. ACM.

[Sylvester, 1852] Sylvester, J. J. (1852). Xix. a demonstration of the theorem that every homogeneous quadratic polynomial is reducible by real orthogonal substitutions to the form of a sum of positive and negative squares. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 4(23):138–142.

[Tang et al., 2015] Tang, J., Qu, M., Wang, M., Zhang, M., Yan, J., and Mei, Q. (2015). Line: Large-scale information network embedding. In *Proceedings of the 24th International Conference on World Wide Web*, pages 1067–1077. International World Wide Web Conferences Steering Committee.

[Tarjan, 1983] Tarjan, R. E. (1983). *Data structures and network algorithms*. SIAM.

[Tong and Faloutsos, 2006] Tong, H. and Faloutsos, C. (2006). Center-piece subgraphs: problem definition and fast solutions. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 404–413. ACM.

[Tremblay and Borgnat, 2014] Tremblay, N. and Borgnat, P. (2014). Graph wavelets for multiscale community mining. *IEEE Transactions on Signal Processing*, 62(20):5227–5239.

[Von Luxburg, 2007] Von Luxburg, U. (2007). A tutorial on spectral clustering. *Statistics and computing*, 17(4):395–416.

[Ward Jr, 1963] Ward Jr, J. H. (1963). Hierarchical grouping to optimize an objective function. *Journal of the American statistical association*, 58(301):236–244.

[West and Leskovec, 2012] West, R. and Leskovec, J. (2012). Human wayfinding in information networks. In *Proceedings of the 21st international conference on World Wide Web*, pages 619–628. ACM.

[West et al., 2009] West, R., Pineau, J., and Precup, D. (2009). Wikispeedia: An online game for inferring semantic distances between concepts. In *IJCAI*, pages 1598–1603.

[Whang et al., 2015] Whang, J. J., Dhillon, I. S., and Gleich, D. F. (2015). Non-exhaustive, overlapping k-means. In *Proceedings of the 2015 SIAM International Conference on Data Mining*, pages 936–944. SIAM.

[Xie et al., 2013] Xie, J., Kelley, S., and Szymanski, B. K. (2013). Overlapping community detection in networks: The state-of-the-art and comparative study. *Acm computing surveys (csur)*, 45(4):43.

[Yang and Leskovec, 2013] Yang, J. and Leskovec, J. (2013). Overlapping community detection at scale: a nonnegative matrix factorization approach. In *Proceedings of the sixth ACM international conference on Web search and data mining*, pages 587–596. ACM.

[Yang and Leskovec, 2015] Yang, J. and Leskovec, J. (2015). Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, 42(1):181–213.

[Yang et al., 2013] Yang, J., McAuley, J., and Leskovec, J. (2013). Community detection in networks with node attributes. In *Data Mining (ICDM), 2013 IEEE 13th international conference on*, pages 1151–1156. IEEE.

[Yin et al., 2017] Yin, H., Benson, A. R., Leskovec, J., and Gleich, D. F. (2017). Local higher-order graph clustering.

[Zhao et al., 2012] Zhao, Y., Levina, E., Zhu, J., et al. (2012). Consistency of community detection in networks under degree-corrected stochastic block models. *The Annals of Statistics*, 40(4):2266–2292.

[Zhou et al., 2008] Zhou, H., Yuan, X., Cui, W., Qu, H., and Chen, B. (2008). Energy-based hierarchical edge clustering of graphs. In *Visualization Symposium, 2008. PacificVIS'08. IEEE Pacific*, pages 55–61. IEEE.

Résumé

Les graphes sont omniprésents dans de nombreux domaines de recherche, allant de la biologie à la sociologie. Un graphe est une structure mathématique très simple constituée d'un ensemble d'éléments, appelés nœuds, reliés entre eux par des liens, appelés arêtes. Malgré cette simplicité, les graphes sont capables de représenter des systèmes extrêmement complexes, comme les interactions entre protéines ou les collaborations scientifiques. Le partitionnement ou clustering de graphe est un problème central en analyse de graphe dont l'objectif est d'identifier des groupes de nœuds densément interconnectés et peu connectés avec le reste du graphe. Ces groupes de nœuds, appelés clusters, sont fondamentaux pour une compréhension fine de la structure des graphes. Il n'existe pas de définition universelle de ce qu'est un bon cluster, et différentes approches peuvent s'avérer mieux adaptées dans différentes situations. Alors que les méthodes classiques s'attachent à trouver des partitions des nœuds de graphe, c'est-à-dire à colorer ces nœuds de manière à ce qu'un nœud donné n'ait qu'une et une seule couleur, des approches plus élaborées se révèlent nécessaires pour modéliser la structure complexe des graphes que l'on rencontre en situation réelle. En particulier, dans de nombreux cas, il est nécessaire de considérer qu'un nœud donné peut appartenir à plus d'un cluster. Par ailleurs, de nombreux systèmes que l'on rencontre en pratique présentent une structure multi-échelle pour laquelle il est nécessaire de partir à la recherche de hiérarchies de clusters plutôt que d'effectuer un partitionnement à plat. De plus, les graphes que l'on rencontre en pratique évoluent souvent avec le temps et sont trop massifs pour être traités en un seul lot. Pour ces raisons, il est souvent nécessaire d'adopter des approches dites de streaming qui traitent les arêtes au fil de l'eau. Enfin, dans de nombreuses applications, traiter des graphes entiers n'est pas nécessaire ou est trop coûteux, et il est plus approprié de retrouver des clusters locaux dans un voisinage de nœuds d'intérêt plutôt que de colorer tous les nœuds. Dans ce travail, nous étudions des approches alternatives de partitionnement de graphe et mettons au point de nouveaux algorithmes afin de résoudre les différents problèmes évoqués ci-dessus.

Mots Clés

Graphe, Partitionnement, Détection de communautés, Analyse de données, Apprentissage statistique, Réseaux

Abstract

Graphs are ubiquitous in many fields of research ranging from sociology to biology. A graph is a very simple mathematical structure that consists of a set of elements, called nodes, connected to each other by edges. It is yet able to represent complex systems such as protein-protein interaction or scientific collaborations. Graph clustering is a central problem in the analysis of graphs whose objective is to identify dense groups of nodes that are sparsely connected to the rest of the graph. These groups of nodes, called clusters, are fundamental to an in-depth understanding of graph structures. There is no universal definition of what a good cluster is, and different approaches might be best suited for different applications. Whereas most of classic methods focus on finding node partitions, i.e. on coloring graph nodes so that each node has one and only one color, more elaborate approaches are often necessary to model the complex structure of real-life graphs and to address sophisticated applications. In particular, in many cases, we must consider that a given node can belong to more than one cluster. Besides, many real-world systems exhibit multi-scale structures and one much seek for hierarchies of clusters rather than flat clusterings. Furthermore, graphs often evolve over time and are too massive to be handled in one batch so that one must be able to process stream of edges. Finally, in many applications, processing entire graphs is irrelevant or expensive, and it can be more appropriate to recover local clusters in the neighborhood of nodes of interest rather than color all graph nodes. In this work, we study alternative approaches and design novel algorithms to tackle these different problems. The novel methods that we propose to address these different problems are mostly inspired by variants of modularity, a classic measure that accesses the quality of a node partition, and by random walks, stochastic processes whose properties are closely related to the graph structure. We provide analyses that give theoretical guarantees for the different proposed techniques, and endeavour to evaluate these algorithms on real-world datasets and use cases.

Keywords

Graph, Clustering, Community Detection, Data analysis, Machine Learning, Networks