# Deep learning techniques for graph embedding at different scales

Alexis Galland

▶ **To cite this version:**

Alexis Galland. Deep learning techniques for graph embedding at different scales. Machine Learning [cs.LG]. Université Paris sciences et lettres, 2020. English. NNT : 2020UPSLE083 . tel-03690033

**HAL Id: tel-03690033**

**https://theses.hal.science/tel-03690033v1**

Submitted on 7 Jun 2022

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

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

PSL
UNIVERSITÉ PARIS

THÈSE DE DOCTORAT
DE L'UNIVERSITÉ PSL

Préparée à l'École Normale Supérieure

# Deep learning techniques for graph embedding at different scales


Soutenue par
**Alexis Galland**
Le 17 Décembre 2020


École doctorale n°386
**Sciences Mathématiques de Paris Centre**

Spécialité
**Mathématiques**

Composition du jury :

| | |
|---|---|
| Nicolas Vayatis<br>École Normale Supérieure Paris-Saclay | *Rapporteur* |
| Pierre Borgnat<br>École Normale Supérieure de Lyon | *Rapporteur* |
| Oana Balalau<br>Inria Saclay & École polytechnique | *Examinateur* |
| Nicolas Tremblay<br>CNRS, Laboratoire GIPSA-lab | *Examinateur* |
| Marc Lelarge<br>École Normale Supérieure & Inria | *Directeur de thèse* |
| Konstantin Avrachenkov<br>INRIA Sophia Antipolis | *Président* |

ENS | PSL

# Abstract

In many scientific fields, studied data have an underlying graph or manifold structure such as communication networks (whether social or technical), knowledge graphs or molecules. A graph is composed of nodes, also called vertices, connected together by edges. Recently, deep learning algorithms have become state-of-the-art models in many fields and in particular in natural language processing and image analysis. It led the way to a great line of studies to generalize deep learning models to graphs. In particular, several formulations of convolutional neural networks were proposed and research is carried to develop new layers and network architectures to graphs. Those models aim at solving different tasks such as node classification, link prediction or graph classification. In this work, we study node, subgraph or graph embeddings produced by graph neural networks. These embeddings at different scales encode hierarchical representations of graphs. Based on these embedding techniques, we propose new deep learning architectures to tackle node classification or graph classification tasks.

First, we introduce Permutation Invariant Neural Network (PINN), a neural network model invariant by node permutation, designed for graph classification. We also present a handcrafted graph embedding from spectral analysis.

Second, we introduce StructAgg, a structural aggregation model that aims at improving the accuracy on graph classification tasks by bringing more information to the final graph embedding. Moreover, by identifying structural roles for nodes in graphs, this model brings interpretability to the field of graph neural networks.

Third, we propose a pooling layer based on edge cuts. Coarsening in graphs is used to obtain several versions of a graph at different scales and to groupe nodes that are topologically close into clusters. The novelty of this approach resides in the fact that we focus on edges and not nodes to design the pooling layer.

Finally, we work on a clustering algorithm to uncover hierarchical communities. By modifying the modularity, we are able to design a procedure to find communities at different scales in graphs.

# List of publications

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

**Invariant embedding for graph classification.**
Galland, A. and Lelarge, M. (2018).
*ICML 2019 Workshop on Learning and Reasoning with Graph-Structured Representations.*
Reference [Galland and Lelarge, 2019]

**Graph structural aggregation for explainable learning.**
Galland, A. and Lelarge, M. (2020).
*Under review at ICLR 2021.*

**Graph neural network pooling by edge cut.**
Galland, A. and Lelarge, M. (2020).
*Under review at ICLR 2021.*

# Acknowledgement

First, I would like to express my gratitude to my advisor Marc Lelarge for his support all along these three years. I would like to thank you for accepting me as your student, for your useful insights and for your great contribution to this work.
I am extremely gratful to my collegues and friends at the french Department of Defense. It was great working together during this period. I would particularly like to thank Alexandre and Pierre who were my internship advisors and who helped a lot to make this PhD happen. I would also like to thank Adrien and Joseph who were of great advises and who supported me during these three years. I would also thank Matthieu and Edouard for their moral support during these three years.
I would like to thank my committee members, professor Nicolas Vayatis, professor Pierre Borgnat, professor Oana Balalau, professor Nicolas Tremblay and professor konstantin Avrachenkov for serving as my committee members and taking some time in their busy schedules and during this troubled time because of the COVID-19.
I would like to thank all my friends for supporting and encouraging me during the last three years. You were of great help for clearing my head during week-ends, holidays and diners.
I would like to thank Nicole and Leo for their support and Carlos for his great help and for taking some valuable time to read and correct all of my work.
Last but not least, I would thank my father and my mother, Anne, Abel and Alma my brother and sisters and Jules for their encouragements and support. I would finally like to thank Chloe for her love and her help all along these three years. You have always succeeded in motivating me and chearing me up. This experience would not have been possible without you.

# Contents

# Chapter 1

# Introduction

## 1.1 Motivations

### 1.1.1 Graphs

A graph is a mathematical structure used to model pairwise relations between objects. It is composed of a set of entities, called nodes (or vertices) and a set of connections between pairs of nodes, called edges. We can distinguish different types of graphs. Graphs can be undirected, which means that edges link two vertices symmetrically, or directed, in which edges link two vertices asymmetrically. More formally, a graph $G$ is an ordered pair $(V, E)$ where $V$ is the set of vertices and $E \subseteq V \times V$ is the set of edges. Additional information can be attached to graph's entities. We can have attributes on edges or attributes on nodes. In all this work, we will consider all graphs to be undirected and most graphs to be attributed graphs with attributes on nodes. We can see an example of a graph in figure 1.1.

The structure of graph is very powerful to model a great variety of data. For example, it is natural to model social networks by graphs where nodes are users that are linked by edges that represent a type of relationship that characterizes the network [Ugander et al., 2011]. But graphs can be used to model many different types of data. We can find graphs in molecules, where nodes are atoms that are linked together by a chemical bond. In chemical graphs, we often have attributes available on nodes that reprensent the type of atom or on edges to encode the type of chemical bond [Stark et al., 2010]. More generally, it is possible to represent any set of objects that lie in a metric space by a $k$-nearest neighbor graph [Connor and Kumar, 2010].

This multitude of cases in which graphs have a great importance makes it a field of growing interest. In the $18^{th}$ century, graph analysis focused on understanding

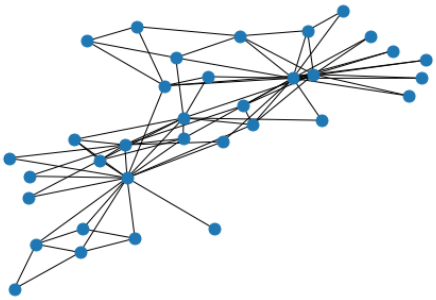


Figure 1.1: Example of a graph, Zachary's karate club dataset [Zachary, 1977]. It is a social network of a karate club. The network captures 34 members of a karate club, documenting links between pairs of members who interacted outside the club.

and analyzing the graph structure by solving combinatorial and algorithmical problems. Recently, with the progresses in deep learning on images, time series or textual data, there has been a lot of research on how to generalize those approaches to graph structured data. Those methods allow one to learn embeddings on graphs for different tasks (graph classification, node classification, link prediction ...) and to develop algorithms tailored for these objectives.

In the two following sections, we will introduce the main challenges that are tackled in graph analysis and different ways to answer them.

### 1.1.2 Embedding graphs at different scales

One of the main objectives adressed by deep learning (DL) is to perform representation learning [LeCun et al., 1995, 2015, Goodfellow et al., 2016]. The goal is to find a mapping from studied objects into a vector space of fixed dimensionality $\mathbb{R}^d$. Some fields of study focus on finding other spaces in which we can embed objects in order to have a better discriminative power and other distances more suited for the data. In this work, we will only study embeddings of data in $\mathbb{R}^d$, $d$ being most of the time fixed as a hyperparameter. In DL algorithms, the mapping is optimized for the task. For example in the case of word2vec [Mikolov et al., 2013a], embeddings of words are initialized at random and optimized so that the distance between the embeddings of two random words that are often seen in the same sentence is smaller than the distance of the embeddings of two words that do not appear in the same neighborhood. In the case of graphs there are several different objectives that are listed bellow:

**Embedding nodes.** By analogy with images, we are interested in finding a representation for nodes just like we are interested in finding a pixel representation in the case of images. An image can be considered as a special case of a graph that has a grid like structure and in which nodes are pixels connected together if they are close enough to each others. In this case, this can be used to perform image segmentation by classifying pixels in different categories. Node embedding has given rise to many applications. It can be used for link prediction [Zhang and Chen, 2018, Al Hasan et al., 2006], the links predicted are the pairs of nodes that aren't connected and that have the lowest distance (the distance being the distance defined by our objective to compute node embeddings). For node classification, node embeddings are learned in a semi-supervised or an unsupervised fashion to perform graph segmentation [Kipf and Welling, 2016, Veličković et al., 2017, Bhagat et al., 2011]. Node embedding can also be used to perform segmentation in clouds of points [Qi et al., 2017].

**Embedding graphs.** Node embeddings can also be used at a different scale to analyse networks at the graph level. This allows to tackle other tasks such as graph classification or regression on graphs [Gao and Ji, 2019, Ying et al., 2018, Xu et al., 2018]. For example, graph classification can allow one to classify molecules according to their topology and identify molecules with similar properties. Another use case can be to regress a molecular property such as the constrained solubility between different graphs representing molecules by embeddings graphs in a vector [Irwin et al., 2012, Dwivedi et al., 2020].

**Challenges.** In this emerging field of graph embedding with neural networks the main challenges in the past few years were the transposition of neural network models defined on images or textual data to graphs. One of the most effective models on grid-like structured data that was generalized to graphs is convolutional neural networks (CNNs). The translation invariance, the locality of filters and the hierarchical feature representation make them a very powerful tool to analyse images. However, there are several aspects that do not generalize from images to graphs:

- **Nodes are not ordered** in graphs compared to pixels in images. This leads the way to several issues. The algorithm must be independant of the ordering of nodes which is not the case for most objects defined on graphs (such as the adjacency matrix). The mapping from a single graph must be unique and musn't depend on the order in which we read the nodes.

- **Graphs have irregular neighborhoods.** In images, filters are defined at a certain scale and at this scale the number of neighbors of each pixel

is fixed. It is thus easy to apply the same weight to all pixels. In graphs, nodes can have different degrees, it is thus necessary to define a convolution operator that enables a weight sharing strategy between nodes.

- **Include features available on graphs.** In most cases we have features available on nodes and/or edges in addition to the topological structure of the graph. Works on graph signal processing make it possible to apply filters and to process this type of data along the topological structure of the graph. Including all the information into the algorithm is a great challenge and may improve its performance by a lot.

Finally, a challenge that applies to graphs and that can be solved by many different methods including graph embedding is the problem of time evolving graphs. Time evolving graphs can be encountered in evolving social communities or in traffic networks in which weights or edges vary in time. One goal can be to predict edges at a time $T$ knowing the set of nodes and edges at times $t = 1, .., T - 1$. This problem can be tackled by feature based matrix completion [Richard et al., 2010] or by learning node states based on previous observations [Yu et al., 2019]. The problem can also be to perform anomaly detection with observations of evolving graphs [Akoglu et al., 2015, Henderson et al., 2012]. In this work, we do not work on time dependent graphs. However, we tackled the problem of finding structural roles in graphs which can be used for anomaly detection as Rolx [Henderson et al., 2012]. Moreover, we worked on edge scoring to perform pooling in graphs. This can be generalized to learn node feature with a time dependancy to predict links in time evolving graphs. In future work, we plan to generalize those methods to time evolving graphs for anomaly detection or link prediction.

## 1.2 Contributions

Many algorithms were proposed to tackle the problems defined above. Even so, there remain many aspects in graph neural networks that need answers and we need novel methods to improve the field and existing algorithms. In this work we study different aspects of graph neural networks to tackle graph classification and node classification tasks.

- First, we study a way to tackle the isomorphism problem in graphs. In order to aleviate the problem of node ordering, we develop an embedding invariant by node permutation. We try two ways to develop such an algorithm.

  - We first develop a handcrafted graph representation with classical spectral graph embeddings.
  - The second one is a neural network designed to be invariant by node permutation.

- Second, we focus on an aggregation process that is a mapping from node embeddings to a graph representation to perform graph classification. The aggregation process brings interpretability to a class of models that are difficult to interpret.

- Third, we study a pooling layer. As in the case of images in which the pooling step aims at identifying hierarchical structures, we develop a pooling layer based on edge cuts in graphs. The pooling layer allows us to find similarities between nodes at different scales and to improve the classification accuracies of both graph classification and node classification tasks.

- Finally, we study a hierarchical clustering algorithm based on the optimization of a measure derived from the modularity.

## 1.3 Notations

Unless mentioned otherwise, we consider that we are given an undirected graph $G = (V, E)$, where $V$ is the set of nodes and $E$ the set of edges. The size of the graph is denoted by $n = |V|$ and $m = |E|$ denotes the number of edges. In most cases we use information available on nodes and represented as a feature matrix $X \in \mathbb{R}^{n \times f}$ where $f$ is the dimensionality of each node feature vector. We use $A$ to denote the adjacency matrix.

$$A_{ij} = \begin{cases} w_{ij} & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

We denote by $d_i$ the degree of node $i$

$$d_i = \sum_{j=1}^{n} A_{ij}$$

We use $D = diag(d_1, ..., d_n)$ to denote the degree diagonal matrix.
In all this work we use $i$ and $j$ to denote the $i^{th}$ and $j^{th}$ nodes of the graph. We denote by $\mathcal{N}(i)$ the neighborhood of node $i$. $\mathcal{N}(i) = \{j \in V | (i,j) \in E\}$. Moreover, we define the $k$-hop neighborhood of node $i$ by $\mathcal{N}_k(i)$.

$$\mathcal{N}_k(i) = \{j \in V | d(i,j) \leq k\}$$

where $d(i,j)$ is the shortest-path distance between nodes $i$ and $j$.

## 1.4 Datasets description

To evalutate the efficiency of our algorithms, we use a series of benchmark datasets for node classification or graph classification.

### 1.4.1 Node classification

For node classification we use three datasets: *cora*, *citeseer* and *pubmed* [Getoor, 2005].

**Cora.** The Cora dataset contains 2708 scientific publications classified into one of seven classes. The citations network consists of 5429 links. Each publication in the dataset is described by a 0/1-valued word vector indicating the absence/presence of the corresponding word from the dictionary. The dictionary consists of 1433 unique words.

**CiteSeer.** The CiteSeer dataset contains 3227 machine learning documents with 3703 authors references to 165 author entities. For this dataset, the only attribute information available is the author name. The full last name is always given, and in some cases the author's full first name and middle name are given and in other times only the initials are given.

**PubMed Diabetes.** The Pubmed dataset consists of 19717 scientific publications from PubMed database pertraining to diabetes classified into one of three classes. The citations network consists of 44338 links. Each publication in the dataset is described by a TF/IDF weighted word vector from a dictionary which consists of 500 unique words.

| Dataset | Nodes | Features | Classes | Training | Validation | Testing | Degree |
|---|---|---|---|---|---|---|---|
| Cora | 2708 | 1433 | 7 | 140 | 500 | 1000 | 4 |
| Citeseer | 3327 | 3703 | 6 | 120 | 500 | 1000 | 5 |
| Pubmed | 19717 | 500 | 3 | 60 | 500 | 1000 | 6 |

Table 1.1: Node classification datasets.

### 1.4.2 Graph classification

For graph classification, we use two kinds of datasets.

**Bioinformatics datasets [Yanardag and Vishwanathan, 2015].** The 7 bioinformatics datasets are: Mutag, PRC, Proteins, NCI1, NCI109, D&D. All datasets contain two classes of molecules, for example, in NCI1, molecules are

either active or inactive against a cerain type of cancer. The aim is to classify the molecules according to their anti-cancer activity.

| **Dataset** | MUTAG | PTC | PROTEINS | NCI1 | NCI109 | D&D |
|---|---|---|---|---|---|---|
| Average #Nodes per Graph | 18 | 14 | 39 | 30 | 30 | 284 |
| #Graphs | 188 | 344 | 1113 | 4110 | 4127 | 1178 |
| #Classes | 2 | 2 | 2 | 2 | 2 | 2 |

Table 1.2: Bioinformatics datasets. *# Nodes per Graph* is the average number of nodes per graph in each dataset.

**OGB datasets [Hu et al., 2020].** OGB stands for Open Graph Benchmark. It is a recent database of datasets proposed by Hu et al. [2020] to address several prediction tasks in graphs. It is composed of large-scale and diverse benchmark datasets. It addresses tasks of node classification, link prediction and graph classification. For all these tasks several datasets of different scales are available. We use a dataset of graph classification to expand our experiments in the later parts of the thesis. Indeed the bioinformatics datasets are composed of a small number of graphs and using datasets with several thousands of graphs makes the classification accuracy more reliable. The ogbg-molhiv dataset is a molecular property prediction dataset. It is adopted from the MoleculeNet, and are among the largest of the MoleculeNet dataset. Each graph represents a molecule, where nodes are atoms, and edges are chemical bonds. Input node features are 9-dimensional, containing atomic number and chirality, as well as other additional atom features such as formal charge and whether the atom is in the ring or not.

| **Name** | **#Graphs** | **Average #Node per Graph** | **Average #Edges per Graph** | **#Tasks** | **Task** | **Metric** |
|---|---|---|---|---|---|---|
| ogbg-molhiv | 41127 | 25.5 | 27.5 | 1 | Binary classification | ROC-AUC |

Table 1.3: OGB dataset for graph classification.

# Chapter 2

# Background and related work

## 2.1 Graph Signal Processing

We find examples of graph signals in many different fields. In chemistry when studying graphs represented by molecules the graph signal may be the type of atom that represents each node. In transportation networks, one may be interested in studying the variation of temperature between different cities. Graph signal processing is an emerging field that aims at bridging the gap between signal processing techniques like wavelet analysis [Hammond et al., 2011, Mallat, 2008] and graph theory such as spectral graph analysis [Von Luxburg et al., 2008, Von Luxburg, 2007]. It is at the basis of spectral convolutions presented in section 2.2.1 and of other concepts of section 4.3.2.

### 2.1.1 Spectrum Analysis

We consider in this section a weighted and undirected graph $G = (V, E, X)$ where $V$ is the set of nodes, $E$ the set of edges and $X$ is a matrix of node features. Let $n = |V|$ be the number of nodes of $G$. Let $A$ be the weighted adjacency matrix of $G$:

$$A_{ij} = \begin{cases} w_{ij} & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

Let $D$ be the degree diagonal matrix of $G$, $D_{ii} = \sum_j A_{ij}$. From the degree diagonal matrix and the adjacency matrix we define the Laplacian $L$:

$$L = D - A$$

The Laplacian is the basis of numerous algorithms such as the spectral clustering which is an unsupervised clustering algorithm. The Laplacian is a difference operator, as, for any signal $X \in \mathbb{R}^n$ it satisfies:

$$(LX)_i = \sum_{j \in \mathcal{N}(i)} A_{ij}[X_i - X_j]$$

Where the neighborhood $\mathcal{N}(i)$ is the set of vertices connected to vertex $i$ by an edge as defined previously. The Laplacian is a real symmetric matrix, it has a complete set of orthonormal eigenvectors, which we denote by $\{u_l\}_{l=0,1,...,n}$ satisfying $Lu_l = \lambda_l u_l$ for $l = 0, 1, ..., n$. These eigenvectors have associated real, non-negative eigenvalues $\{\lambda_l\}_{l=0,1,...,n}$. We can also write:

$$L = U \Lambda U^T$$

Where $u_l$ is the $l^{th}$ column of $U$.
Studying these eigenvalues allows us to characterize the graph and to cluster nodes in the case of the spectral clustering algorithm. Zero is an eigenvalue with multiplicity equal to the number of connected components of the graph. Small eigenvalues are associated with a smoothly varying signal and high eigenvalues with a signal varying a lot along nodes since we have:

$$(Lu_l)_i = \sum_{j \in \mathcal{N}(i)} A_{ij}[u_l(i) - u_l(j)] = u_l(i)\lambda_l$$

Where $u_l(i)$ is the value of the vector $u_l$ on node $i$. Since $\{u_l\}_{l=0,1,...,n}$ is an orthonormal basis we have:

$$u_l^T L u_l = \sum_{(i,j) \in E} A_{ij}[u_l(i) - u_l(j)]^2 = \lambda_l$$

So the signal associated with small eigenvalues is smooth since $[u_l(i) - u_l(j)]^2 \leq \frac{\lambda_l}{A_{ij}}$ is small for any $(i, j) \in E$ as illustrated in figure 2.1. This understanding of eigenvalues and eigenvectors is at the basis of the understanding of graph fourier transform and signal filtering.
There exist other possible choices regarding the Laplacian of a graph $G$ [Tremblay and Borgnat, 2014b, Tremblay et al., 2018]:
**The normalized Laplacian** $L_{norm}$ defined by:

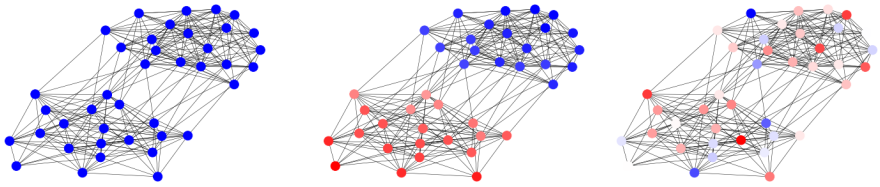
$$L_{norm} = I - D^{-\frac{1}{2}} A D^{-\frac{1}{2}}$$

$L_{norm}$ is interesting because its eigenvalues are non-negative and are bounded by 2 because of the normalization of the adjacency matrix. This is quite useful in many cases and specially when working with functions that have a bounded definition domain.

**The random walk Laplacian** $L_{rw}$ defined by:

$$L_{rw} = D^{-1}L = I - D^{-1}A$$

$D^{-1}A$ is a random walk operator that is not symmetric and that is used mostly for algorithms that find their roots in random walks such as the PageRank algorithm. $L_{rw}$ is not symmetric but is diagonalizable in $\mathbb{R}$ because if $u$ is an eigenvector of $L_{norm}$, $D^{-\frac{1}{2}}u$ is an eigenvector of $L_{rw}$. Thus $L_{rw}$ has the same eigenvalues as $L_{norm}$ but its Fourier basis is not orthonormal.



(a) $\lambda_0 = 0$: constant signal (b) $\lambda_1 = 2,05$: smooth signal (c) $\lambda_{12} = 10,97$: non-smooth signal

Figure 2.1: Graph signals associated with successive eigenvalues.

### 2.1.2 Graph Fourier Transform

The classical Fourier transform on a signal $X$ is defined by:

$$\hat{X}(\zeta) =< X, e^{2i\pi\zeta t} >= \int_{\mathbb{R}} X(t)e^{-2i\pi\zeta t}dt$$

The complex exponentials $e^{2i\pi\zeta t}$ are eigenfuntions of the one-dimensional Laplacian operator $\frac{d}{dt^2}$. The Fourier transform can thus be seen as an expansion of the signal $X$ in terms of the eigenfunctions of the Laplacian operator.
The graph Fourier transform is defined in analogy with the Fourier transform [Defferrard et al., 2016, Hammond et al., 2011, Ortega et al., 2017]. It is defined via a choice of reference operator admitting a spectral decomposition. Let us denote by $R$ the reference operator admitting a spectral decomposition in $\mathbb{R}$. As seen in the previous section, we have

$$R = U\Lambda U^{-1}$$

where $\Lambda \in \mathbb{R}^{n\times n}$ is a diagonal matrix and $U \in \mathbb{R}^{n\times n}$.
For a given operator $R$, the graph Fourier transform of a signal $X \in \mathbb{R}^n$ on the vertices of $G$ is the expansion of $X$ in terms of the eigenvectors of this operator:

$$\hat{X}(\lambda_l) =< X, u_l >= \sum_{i=1}^{n} X_i {u_l}^T(i) \tag{2.1}$$

Where $\hat{X}(\lambda_l)$ is a notation that corresponds to the projection of signal $X$ on the eigenvector associated to eigenvalue $\lambda_l$. The inverse graph Fourier transform is then given by:

$$X_i = \sum_{l=0}^{n-1} \hat{X}(\lambda_l) u_l(i) \tag{2.2}$$

We can decompose the signal $X$ on the vertices of a graph in terms of Fourier coefficients and eigenvectors of the operator $R$.

**Choice of reference operator.** In all our work we consider only undirected graphs and we use as reference operator $R$ the graph Laplacian. There are two reasons that motivate this choice. First, the Laplacian is the discretized version of the continuous Laplacian which admits the Fourier modes as eigenmodes in classical Fourier decomposition. By analogy with the classical Fourier Transform, it is fair to use the Laplacian $L$. Second, the graph Laplacian has interesting properties. As seen previously, since the graphs are undirected, the Laplacian is symmetric and its eigenvalues are all real and positive by definition of the operator.

### 2.1.3 Signal Filtering

We want to generalize the fundamental operation of signal filtering to graphs. This operation will then be used to define several algorithms and in particular convolutions on graphs. Graphs can be represented either in the vertex domain or in the spectral domain. This allows us to define filtering either in the spatial domain (vertices) or in the spectral domain and see how those filtering operations are linked.

**Graph filtering in the vertex domain** is simply the fact of writing the output $X_{out}(i)$ at vertex $i$ as the linear combination of the components of the input signal at vertices within a $k$-hop neighborhood of vertex $i$:

$$X_{out}(i) = b_{i,i} X_{in}(i) + \sum_{j \in \mathcal{N}_k(i)} b_{i,j} X_{in}(j) \tag{2.3}$$

Where $\{b_{i,j}\}_{(i,j)\in V^2}$ are filter parameters and $\mathcal{N}_k(i)$ is the $k$-hop neighborhood of vertex $i$.

**Graph spectral filtering.** The Laplacian can be written as a sum of projectors on its eigenspaces:

$$L = \sum_{\lambda} \lambda Pr_\lambda = \sum_{k} \lambda_k u_k u_k^T \tag{2.4}$$

as defined by Tremblay et al. [2018]:

**Definition 1.** *The most general definition of a graph filter is an operator that acts separately on all the eigenspaces of $L$, depending on their eigenvalue $\lambda$. Mathematically, any function*

$$\begin{aligned} h :&\mathbb{R} \mapsto \mathbb{R} \\ &\lambda \mapsto h(\lambda) \end{aligned} \tag{2.5}$$

*defines a graph filter $H$ such that*

$$H = \sum_k h(\lambda_k) u_k u_k^T \tag{2.6}$$

The filtering weight $h(\lambda)$ attenuates or increases the importance of each eigenspace in the decomposition of the signal. For an input signal $X_{in}$ on the graph, the action of the filter $H$ is written as:

$$X_{out} = HX_{in} = \sum_k h(\lambda_k) Pr_{\lambda_k} X_{in} \tag{2.7}$$

It can also be written as:

$$X_{out} = HX_{in} = Uh(\Lambda)U^T X_{in} \tag{2.8}$$

where $h(\Lambda) = diag(h(\lambda_1), ...h(\lambda_n))$.

A non-parametric filter is a filter whose parameters are all free, i.e. that can be written like $h(\Lambda) = diag(\theta)$ where the parameter $\theta \in \mathbb{R}^n$ is a vector of Fourier coefficients.
The main limitation with non-parametric filtering is that filters are not localized in space. A solution to overcome this issue is to use a Laplacian-based polynomial spectral filter that would take the following form:

$$h(\Lambda) = \sum_{k=0}^{K-1} \theta_k \Lambda^k \tag{2.9}$$

Where $\theta \in \mathbb{R}^K$ is a vector of polynomial coefficients. This type of filter is also localized because $h(L) = \sum_{k=0}^{K-1} \theta_k L^k$ and $L^k$ is localized in a $k$-hop neighborhood of each node. Since $U$ is othonormal, we have $h(U\Lambda U^T) = Uh(\Lambda)U^T$. Moreover they have a limited number of free parameters independant of the size of the graph $n$, which is very useful for scalability to train learning algorithms. The cost to filter a signal $X_{in}$ as $X_{out} = Uh(\Lambda)U^T X_{in}$ is still high with $\mathcal{O}(n^2)$ operations

because of the multiplication with the Fourier basis $U$. To overcome this problem, a solution is to parametrize $h(L)$ as a polynomial funtion that can be computed recursively from $L$. This can be done with Chebyshev expansion [Hammond et al., 2011, Shuman et al., 2011]. The graph spectral filtering is illustrated in figure 2.2.

**Link between spectral and spatial filtering.** We can make the link between spatial and spectral filtering by taking a specific choice of parameters $b_{i,j} = \sum_{k=0}^{K-1} \theta_k L^k_{i,j}$ [Ortega et al., 2017, Shuman et al., 2013]. This way we have:

$$\begin{aligned} X_{out}(i) &= \sum_{l=0}^{n-1} \hat{X}_{in}(\lambda_l) h(\lambda_l) u_l(i) \\ &= \sum_{j=1}^{n} X_{in}(j) \sum_{k=0}^{K} \theta_k \sum_{l=0}^{n-1} {u_l}^*(j) u_l(i) \\ &= \sum_{j=1}^{n} X_{in}(j) \sum_{k=0}^{K} \theta_k (L^k)_{i,j} \\ &= b_{i,i} X_{in}(i) + \sum_{j \in \mathcal{N}(i,K)} b_{i,j} X_{in}(j) \end{aligned} \tag{2.10}$$

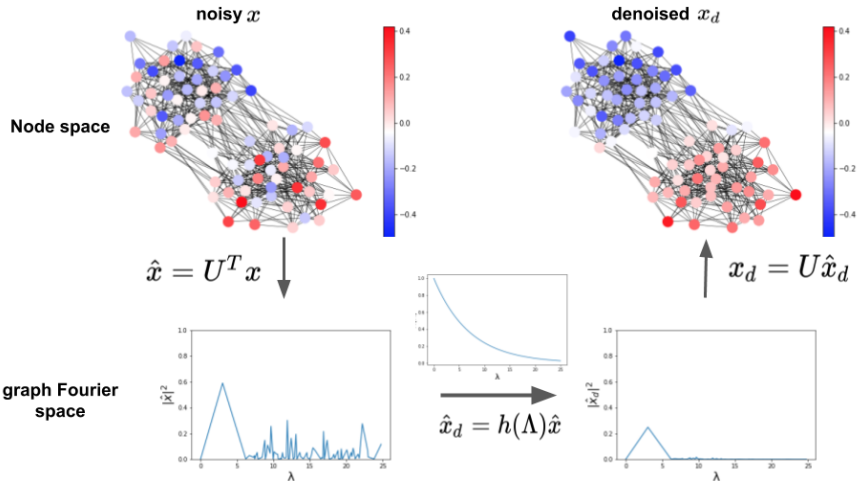



Figure 2.2: Illustration of the filtering of a noisy signal on a graph. We have a noisy signal on the graph on the left-hand side. By applying a low-pass filter we are able to denoise the signal that we visualize on the graph on the right hand side. The figure is a reproduction of results from Tremblay et al. [2018]

### 2.1.4 Graph Coarsening

When working with images and more generally with euclidean data, coarsening is a powerful tool to obtain representations of our original data at different scales. In the case of images, applying successive CNNs followed by coarsening steps were a great breakthrough for image classification tasks. However, the task of coarsening is quite intuitive when working on images (pixels that are merged are adjacent pixels) but not as simple when it comes to applying it on graph structured data. Indeed, in graphs, there is no order of nodes and no regularity regarding the number of neighbors of each node. Many different works proposed different models to solve this problem.

The first kind of coarsening methods are deterministic methods that partition the graph into sets of nodes that form clusters that represent nodes at the next coarsened level. This kind of coarsening method is used by Defferrard et al. [2016]. They use the algorithm Graclus [Dhillon et al., 2007], built on Metis [Karypis and Kumar, 1998] that greedily identifies pairs of nodes to be merged by optimizing a graph metric. This algorithm reduces the size of the graph by approximately 2 after each coarsening step.

Another line of work tries to learn the coarsening step during there learning of the algorithm. The learning strategy can be done in three different ways. The first two are node based coarsening layers. First by learning an assignment matrix for nodes to clusters [Ying et al., 2018, Tsitsulin et al., 2020, Bianchi et al., 2019]. This is done by projecting node representation on a learnable weight. The second node based coarsening layer is based on node deletion. By computing importance scores on nodes, we are able to keep only few nodes that are the most representative of the graph [Gao and Ji, 2019]. On the opposite Diehl [2019] developped a pooling strategy based on edge representation. They develop an edge contraction pooling strategy that aims at identifying edges to be contracted which means pairs of nodes to be merged. Like for the Graclus method, the size of the coarsened graph is divided by 2 after each coarsening step.
In section 5 we'll introduce a coarsening algorithm that aims at calculating scores on edges in order to approach the minCUT problem. We studied a pooling layer based on the deletion of edges in graphs. Deleting edges produces a sparser graph with possibly multiple connected components that were seperated in the process of edge cut. Those connected components are strongly connected groups of nodes that represent nodes in the coarsened level.

## 2.2 Convolutional Graph Neural Networks

In this section we introduce a model of graph neural network that we call convolutional neural networks. Those models that are used in chapters 3, 4 and 5 are built upon signal processing algorithms in graphs presented in the previous section. Graph convolutional network falls into two categories, spectral-based and spatial-based. Spectral-based approaches define convolutions with spectral filters introduced in Section 2.1.3, where the convolution is interpreted as removing noise from a graph signal. Spatial-based approaches define convolutions by information propagation. We will see in this section how both of these methods are defined and the different models proposed over the years.

### 2.2.1 Spectral Graph Convolutions

In this section we review the different ways to define spectral graph convolutions that are built upon the spectral signal processing methods.

Spectral networks were first introduced by Bruna et al. [2013] in order to learn convolutional filters for graph classification problems. Their layers are defined by $X_j^{(l+1)} = \rho(U \sum_{i=1}^{f_{l-1}} F_{l,i,j} U^T X_i^{(l)})$, where $F_{l,i,j}$ is a diagonal matrix of filters of layer $l$, $\rho$ is a non linearity and $X^{(l)} \in \mathbb{R}^{n \times f_{l-1}}$ is the input feature vector, with features of dimension $f_{l-1}$ for each node. Often, only the first $d$ eigenvectors of the Laplacian carry the smooth geometry of the graph, $d$ depending on the topology and the regularity of the graph. It is thus possible to replace $U$ (the eigenvector matrix of the Laplacian) by $U_d$, in which we have kept only the first $d$ eigenvectors. This allows us to have faster computation for this propagation rule. Moreover in oder to be able to scale graph convolutions to large graphs, it is preferable to have filters that don't depend on the size of the graph. To this end, it is possible to define filters as sum of $K$ predefined function, such as spline, and this way, instead of learning $N$ values, we learn $K$ coeffectients $\alpha_k$ of the sum $F = \sum_{k=1}^{K} \alpha_k g_k$ where $g_k$ are spline functions.

This approach was then simplified by Defferrard et al. [2016] by defining filters from the Chebyshev polynomial decomposition [Hammond et al., 2011, Shuman et al., 2011] defined in 2.1.3 to obtain filters that are strictly localized in a ball of radius $K$. The propagation layer can be expressed by $X^{(l+1)} = \sum_{k=0}^{K-1} \theta_k T_k(\tilde{L}) X^{(l)}$ where $\tilde{L} = \frac{2L}{\lambda_{max}} - I$ is the scaled Laplacian. The advantages of this approach is the fact that filters are localized in a ball of radius $K$, which means $K$-hops from the central vertex and this layer is more efficient computationally because of the recursive formulation of the Chebyshev polynomials. Moreover, Defferrard et al. [2016] use a graph coarsening algorithm to pool the graph between

successive layers of convolution in order to identify features at different scales. The algorithm used is Graclus [Dhillon et al., 2007] built upon Metis Karypis and Kumar [1998] because it reduces the size of the graph by a factor two at each level and offers a precise conrol on the coarsening and pooling size.
This method was then simplified by Kipf and Welling [2016] who proposed a first order decomposition with Chebyshev polynomials, which results in a linear formulation of filters in the Laplacian, which gives, for a single channel input $X^{(l+1)} = \theta(I + \tilde{L})X^{(l)}$, where $\theta$ is a learnable parameter. The formulation for a multi-channel input is then $X^{(l+1)} = \tilde{D}^{\frac{-1}{2}}\tilde{A}\tilde{D}^{\frac{-1}{2}}X^{(l)}\Theta$, where $\tilde{A} = A + I$ is the adjacency with added self-loops and $\tilde{D}$ is the corresponding degree diagonal matrix. This model corresponds to an average sum of neighbors' features weighted by the Laplacian. In their work they do not use pooling layer to obtain versions of the graph at different scales but they only stack several layers of Graph Convolutional Network (GCN) in order to tackle a node classification task.
Several recent works made incremental modifications over GCN by exploring alternative symmetric matrices such as Adaptative Graph Convolutional Network (AGCN) [Li et al., 2018] or Dual Graph Convoluional Network (DGCN) [Zhuang and Ma, 2018]. Let's note that all these methods based on the spectral decomposition of the graph are defined on undirected graphs and do not generalize easily to directed graphs because of the asymmetry induced in the Laplacian.

### 2.2.2 Spatial Graph Convolutions

Analogous to the convolutional operation of a CNN on an image, spatial-based methods define graph convolutions based on a node's topological relation. An image can be considered as a special case of a graph that has a grid like structure. Similarly to images, spatial graph convolutions convolve the central node's representation with its neighbors' features. The spatial-based convolutional operation propagates node information along edges.
One of the first work towards spatial-based graph convolutions is Neural Network for Graphs (NN4G) [Micheli, 2009]. NN4G learns mutual dependencies with independant parameters at each layers, by learning two sets of parameters at each layers, $W$ that transforms central nodes' features and $\Theta$ that transforms neighbors features, ${h_i}^{(l)} = \rho({W^{(l)}}^T x_i + \sum_{k=0}^{l} \sum_{j \in \mathcal{N}(i)} {\Theta^{(k)}}^T {h_j}^{(k)})$ where $\rho$ is a non linear function.
Recently many variants have been proposed and Gilmer et al. [2017] proposed Message Passing Neural Network (MPNN) family fitting the vast majority of models of that time. In their work, Gilmer et al. [2017] treat graph convolutions as a message passing process in which information can be passed from one node

to another along edges directly. MPNN runs $K$-step message passing iterations to let information propagate further. The propagation rule is defined in terms of a message function $m$ and an update function $u$ as:

$$M_i{}^{(t+1)} = \sum_{j \in \mathcal{N}(i)} m_t(h_i{}^{(t)}, h_j{}^{(t)}, E_{i,j})$$
$$h_i{}^{(t)} = u_t(h_j{}^{(t)}, M_i{}^{(t)})$$

In this formulation $h_i{}^{(t)}$ defines the hidden representation of node $i$ after $t$ steps of message passing. The update function $u_t$ takes as input $M_i^{t+1}$ which is the output of aggregation of the messages $m_t$ from each neighbor. The final layer representation can be passed to an output layer to perform node-level prediction tasks or to a readout function that aggregates these node representations to tackle a graph-level prediction task. The readout function $r$ generates a representation of the entire graph based on node hidden representation, $h_G = r(h_i{}^{(T)} | i \in G)$. Many algorithms that follow this construction were proposed by the graph community to answer node classification tasks or graph classification tasks [Veličković et al., 2017, Xu et al., 2018, Kipf and Welling, 2016, Battaglia et al., 2016, Kearnes et al., 2016].

## 2.3 Node Embedding and Kernel Methods

In the machine learning community, graph convolutional networks are not the only algorithms that allow us to embed nodes in a trainiable fashion. In this section we present several methods that have an interest because of their theoretical results and links with convolutional networks or because of their efficiency and coherence in node embeddings.

### 2.3.1 Embedding Nodes by Topological Proximities

Graph convolutional networks produce node embeddings by aggregating the information of nodes that reside in a $K$-hop neighborhood of the central node. Previous to works on convolutions, some algorithms tried to produce a node embedding $Z_i$ for node $i$ as only a function of the node's attributes $X_i$. The learning strategy residing mainly in the objective function to be minimized.
These algorithms were inspired by works from the natural language processing (NLP) community. In NLP, multiple techniques for unsupervised learning are based on the assumption that similar words tend to appear in similar word neighborhoods. In particular, Skip-gram [Mikolov et al., 2013b] aims at predicting a word's embedding based on the embeddings of its contextual words.
This idea has been transposed to graphs by generating random walks in graphs and by making an analogy between random walks and sentences and words

and nodes. A pioneer work in this field is DeepWalk [Perozzi et al., 2014] that uses the Skip-gram algorithm to compute node embeddings based on random walks generated uniformally on the graph. This method was improved by Grover and Leskovec [2016] by generating random walks with two different strategies, *Breadth-first Sampling* (BFS) and *Depth-first Sampling* (DFS). BSF samples nodes that are not to far from the central node. Whereas DFS samples nodes iteratively at increasing distances from the source node. The objectives behind those two sampling strategies are different, one explores neighborhoods of nodes whereas the other explores node that are far from the source node, the strategy of Grover and Leskovec [2016] was to develop a neighborhood sampler that interpolates between those two strategies. Node embeddings are then optimized by gradient descend by maximizing the log-probability of observing a network neighborhood $N_s(i)$ for node $i$.
In the same spirit, [Tang et al., 2015] compute direct node embeddings by minimizing the distance between embeddings of nodes that are at first-order proximity (direct neighbors) or second-order proximity (at 2-hops in the graph). In all those methods, negative sampling is used in order to separate embeddings of nodes that don't lie in the same part of the graph.

### 2.3.2 Kernel methods

Kernel methods are algorithms that learn by comparing pairs of objects using similarity measures defined by kernels. Graph kernels were first proposed in 2003 [Gärtner et al., 2003, Kashima et al., 2003]. Graph kernels are particularly useful to compute graph-level representation for graph classification tasks and to approach the graph isomorphism problem. We introduce two classical graph kernel methods.
The Weisfeiler-Lehman kernel is the most well known algorithm of this type [Shervashidze et al., 2011, Weisfeiler and Lehman, 1968]. The Weisfeiler-Lehman test is an iterative algorithm that aims at comparing two graph. The key idea is to augment the node labels by the sorted set of node labels of neighboring nodes and then to hash these new node representation to produce new node labels. At each step we compare the sets of node labels. If they differ, graphs are non-isomorphic, otherwise we continue until we have reached a predefined number of steps.
A second well known kernel is graphlet-based kernel [Shervashidze et al., 2009]. In order to compare two graphs, we compare the occurences of motives present in each graph, motives being drawn from a predefined bank of motives. These motives are patterns of connections between nodes. They can be 2 nodes motives (which means the number of edges), 3 nodes motives and so long. There exists

four types of graphlets of size 3: disconnected, connected by only one edge, connected by two edges and complete graphlet. There is a trade off to find between the computationnal efficiency and the classification accuracy. If we use a large bank of graphlets, the classification will be accurate but it will be very costly to compute graph features.

# Chapter 3

# Graph Invariant Embedding

In this section we design a model to tackle a graph classification. First, we design a handcrafted embedding derived from graph spectral analysis and study its properties. We then use this embedding as input of a graph neural network designed to capture correlations between nodes of different graphs in order to classify graphs. We design this neural network to be invariant by node permutation to make it as suited as possible to discriminate graphs and to approach the isomorphism problem. To this end we adapted PointNet [Qi et al., 2017], a neural network model on point sets for 3D classification and segmentation to a setting of graphs. This model made for classification allows us to discriminate graphs while being invariant by node permutation. This chapter is based on the work presented in [Galland and Lelarge, 2019].

## 3.1 Introduction

As stressed in chapter 1, the main objective when using graph neural network models is to classify nodes according to the topology of the graph and to features that are available on nodes. However, in analogy with the field of computer vision in which image classification has a predominant importance, graph classification has a growing interest in the graph community. Indeed, graph classification has many applications in diverse fields: bioinformatics, chemoinformatics or social networks. To solve this problem, one needs to compute features that help discriminate between graphs of different classes.

Graph classification is related to the fundamental problem of testing isomorphism of graphs, i.e. being able to tell whether two graphs are the same or not up to a permutation of their nodes. Testing isomorphisms of graphs is a notorious problem which is not known to be solvable in polynomial time nor

to be NP-complete. This fundamental problem triggered a lot of efforts in the theoretical computer science community. The last developments are: Babai [2016] announced a quasipolynomial time algorithm for all graphs. Helfgott discoverd a flaw in the proof, fixed by Babai with a confirmation by Helfgott claiming a running time of $2^{\mathcal{O}((logn)^3)}$ [Helfgott, 2017]. In this chapter, we are interested in telling if two graphs are similar (as opposed to identical in the graph isomorphism problem). This relaxation allows for potentially much faster algorithms and for the ability to quantify the topological similarity of two graphs in order to solve a graph classification problem.

Methods to learn representations that encode structure information about the graph can be split in two categories: kernel based methods or graph neural networks (GNN). Interestingly, in both cases, best performing methods are closely related to the Weisfeiler-Lehman (WL) graph isomorphism test [Weisfeiler and Lehman, 1968], see Shervashidze et al. [2011] for its connection with graph kernels and Xu et al. [2018] for its connection with GNN. The WL's test iteratively updates a given node's features by aggregating feature vector of its neighbors. In their work, Babai et al. [1982] proposed an alternative algorithm for the graph isomorphism problem based on spectral graph theory. Our aim in this chapter is to demonstrate the relevance of ideas from spectral graph theory Spielman [2007] to the graph representation learning problem. Moreover, we introduce a novel and powerful graph feature representation called Spectral Graph Embedding (SGE). We show that SGE is invariant for a large family of graphs and also give intuition on why SGE has a great discriminative power. We also build a permutation invariant neural network (PINN) to learn graph embeddings from a given dataset.
The remainder of this chapter is organized as follows. In section 3.3 we introduce four handcrafted embedding with invariance properties and derived from spectral graph theory. In section 3.4 we introduce a neural network that has a high discriminative power and that aims at identifying correlations between node embeddings in order to solve a graph classification task. Finally, in section 3.5 we present experimental results on benchmark datasets.

## 3.2 Notations and Problem Formulation

We consider connected undirected graphs without self-loops, $G = (V, E)$ where $V$ is the vertex set and $E$ the edge set. The size of the graph is the number of nodes and is denoted by $n = |V|$. The adjacency matrix is denoted by $A$. Let $d = A\mathbf{1}$ be the $n$-dimensional vector of degrees. Let $D = diag(d)$ be the degree diagonal matrix. From $D$ and $A$ we can define the Laplacian $L = D - A$ and

the transition matrix for the random walk of the graph $P = D^{-1}A$.

Two graphs $G = (V, E)$ and $H = (V, F)$ are isomorphic if there exists a permutation $\pi$ of $V$ such that $(i, j) \in E \Rightarrow (\pi(i), \pi(j)) \in F$. We can express this relation in terms of matrices associated with the graphs.

Every permutation may be realized by a permutation matrix. For a permutation $\pi$, this is the matrix $\Pi$ with entries given by

$$\Pi(i, j) = \begin{cases} 1 & \text{if } \pi(i) = j \\ 0 & \text{otherwise} \end{cases}$$

For a vector $x$, we see that $(\Pi x)(i) = x(\pi(i))$. Let $A$ be the adjacency matrix of $G$ and let $B$ be the adjacency matrix of $H$. We see that $G$ and $H$ are isomorphic if and only if there exists a permutation $\Pi$ such that $\Pi A \Pi^T = B$. Note that the same relation can be written for the respective Laplacians. In this case, we will write $\pi(G) = H$.

A graph embedding is a function $\mathcal{F}$ mapping graphs to vectors in $\mathbb{R}^d$, where $d$ is the dimension of the embedding. A graph representation is invariant if for any two isomorphic graphs $G$ and $H$, we have $\mathcal{F}(G) = \mathcal{F}(H)$.

An embedding of nodes is a function $\mathcal{E}$, mapping a graph of size $n$ to a vector $(\mathcal{E}(G)(i))_{i \in V} \in \mathbb{R}^{d \times n}$. For a graph $G$, $\mathcal{E}(G)(i) \in \mathbb{R}^d$ is the embedding of node $i$. An embedding of nodes is equivariant if for any two isomorphic graphs $G$ and $H$ such that $\pi(G) = H$ for a permutation $\pi$, we have $\mathcal{E}(H)(i) = \mathcal{E}(G)(\pi(i))$ for all $i \in V$.

From an embedding of nodes it is straightforward to create a graph embedding. A very simple choice is just to sum the nodes' representations:

$$\mathcal{F}(G) = \sum_{i \in V} \mathcal{E}(G)(i)$$

Now, if the embedding of nodes $\mathcal{E}$ is equivariant, this will produce an invariant graph embedding $\mathcal{F}$, but the invariance property would be preserved if instead of the sum, we apply any symmetric function as defined below:

**Definition 2.** *A function $f : \mathcal{S}^n \to \mathbb{R}^m$ is symmetric if for any permutation $\sigma$ and any $x \in \mathcal{S}^n$, we have $f(x(1), ..., x(n)) = f(x(\sigma(1)), ..., x(\sigma(n)))$*

In all the following we will use this generic construction to build graph represen-

tations from nodes' embeddings.

## 3.3 Handcrafted embedding

In this section we introduce four handcrafted embeddings, three of them are classical embeddings from spectral graph theory, the last one encodes the topology of the graph with a spatial representation. Our final embedding is the concatenation of those four graph representations. Moreover, we define a new aggregation function that is a differentiable histogram to go from node level to graph level embeddings.

### 3.3.1 Eigenvalues

We present an embedding based on eigenvalues of the graph Laplacian. When dealing with graph embeddings that have invariance properties, it is trivial to consider features such as the number of nodes, the number of edges, the diameter of the graph or the number of connected components. We could concatenate all these graph invariants to get an invariant embedding.

However, we also care about the discriminative power of the embedding, i.e. if $G$ and $H$ are non-isomorphic, then we would like $\mathcal{F}(G) \neq \mathcal{F}(H)$, $\mathcal{F}$ being the embedding mapping. An invariant graph embedding with a perfect discriminative power would solve the graph isomorphism problem. In order to get an easily computable graph embedding, we need either to relax the invariant property or lower its discriminative power. We will do both in the sequel.

As presented in chapter 2, a powerful tool for graph analysis is graph spectral decomposition. The graph Laplacian of a graph with $n$ nodes is positive semi-definite with eigenvalues $\lambda_1 = 0 < \lambda_2 \leq ... \leq \lambda_n$. Note that $\lambda_2 > 0$ because we assume that the graph is connected. Otherwise the number of eigenvalues equal to 0 is equal to the number of connected components in the graph. The embedding that we define must be of fixed size in order to be able to compare it between graphs of different sizes. We fix $k_1$ the size of this embedding. We define $\mathcal{F}_1(G) = (\lambda_2, ..., \lambda_{k_1+1})$ if $n \geq k_1 + 1$ and $\mathcal{F}_1(G) = (0, ..., 0, \lambda_2, ..., \lambda_n)$ otherwise. This embedding is constructed by padding the eigenvalue vector with 0 or by truncating it in order to obtain in the end a vector of size $k_1$. Note that padding the eigenvalue vector with 0 if the size of the graph is smaller then $k_1$ boils down to adding single nodes to the graph until the graph is of size $k_1$.

### 3.3.2 Spatial Embedding

We now introduce a spatial embedding that encodes the topology of the graph and that takes into account potentiel features that can be available on nodes.
We construct an embedding capturing the property of a random walk on the graph. Remember that $P^k_{ij}$ is the probability for a random walk started in node $i$ to be in node $j$ after $k$ steps, hence $1^T P^k = (x^k(1), ..., x^k(n))$ is a vector where $x^k(j)/n$ is the probability for a random walk started uniformaly at random on the graph to be in position $j$ after $k$ steps. Clearly, the vector $1^T P^k$ is an equivariant embedding of nodes so that for any symmetric function $f : \mathbb{R}^n \to \mathbb{R}^m$, $f(x^k(1), ..., x^k(n))$ is an invariant feature of the graph.
If we have node features available, we use this propagation process on features [Atwood and Towsley, 2016], we have $P^k X = (x^k(1), ..., x^k(n))$. This vector is also an equivariant embedding of nodes and the application of any symmetric function $f$ is an invariant feature of the graph. To compute features that capture hierarchical patterns in the graph, we stack those embeddings for different values of $k$.

**Aggregation function.** After having defined node embeddings we need an aggregation function in order to create a graph representation from those node representations. In most cases, the function used is a *max* or a *mean* of node embeddings component-wise. In this work we wanted to define a more complexe function that could take into account all node values. Ideally we would like to use a histogram to aggregate node informations. But for works that will be presented in section 3.4 we need this function to be differentiable. Moreover this function must be symmetric. We thus introduce a novel function that approximates histograms in a differentiable way.
We introduce

$$f(x_1, ..., x_n) = (l(x_1, .., x_n; ti))_{1 \leq i \leq m} \tag{3.1}$$

where $t_1 < t_2 < ... < t_m$ are fixed parameters and

$$l(x_1, ..., x_n; t) = \frac{\sum_i x_i e^{t x_i}}{\sum_i e^{t x_i}} \in \mathbb{R} \tag{3.2}$$

Note that $l(.; t)$ is a symmetric function for any $t$ so $f$ is also a symmetric function. Moreover, we have: $l(x_1, ..., x_n; 0) = \frac{1}{n} \sum_i x_i$ and $\lim_{t->\infty} l(x_1, ..., x_n; t) = \max_i x_i$ and $\lim_{t->-\infty} l(x_1, ..., x_n; t) = \min_i x_i$.

**Proposition 1.** *For any $(x_1, ..., x_n) \in \mathbb{R}^n$, the function $t \mapsto l(x_1, ..., x_n; t)$ characterizes the multiset $\{x_1, ..., x_n\}$*

*Proof.* An argument from probability theory gives a simple proof. Consider a random variable $X$ with probability distribution on $\mathbb{R}$ consisting in picking an element in the multiset $\{x_1, ..., x_n\}$ with density $\mathbb{P}_t = \sum_i \frac{e^{tx_i}}{\sum_j e^{tx_j}} \delta(x_i)$. Then we have:

$$\frac{dl(x;t)}{dt} = \mathbb{E}_t[X^2] - \mathbb{E}_t[X]^2$$

and taking the following derivatives, will give the next moment of the distribution $\mathbb{P}_t$. Since $\mathbb{P}_t$ is a discrete distribution, it is characterized by its moment, hence we can characterize $\mathbb{P}_t$ for any $t$ and in particular $\mathbb{P}_0$ which is the uniform distribution over the multiset $\{x_1, ..., x_n\}$. □

In summary, our second graph embedding $\mathcal{F}_2(G) \in \mathbb{R}^{k_2 m}$ is obtained by concatenating the features defined in (3.1) and applied to the vector $1^T P^k$ or $P^k X$, if we have features on nodes, for values of $k \in \{1, \ldots, k_2\}$ for a fixed parameter $k_2$.

### 3.3.3 Commute Times

We define a classical spectral embedding for nodes. Let $X = \sqrt{\Lambda^+} U^T$, where $\Lambda^+ = diag(0, \frac{1}{\lambda_2}, ..., \frac{1}{\lambda_n})$ denotes the pseudo-inverse of $\Lambda$. The columns $x(1), ..., x(n)$ of $X$ define an embedding of nodes in $\mathbb{R}^n$. Each dimension corresponds to an eigenvector of the Laplacian. The first component of each vector $x(1), ...x(n)$ is equal to 0 by construction. This corresponds to the first eigenvector of the Laplacian which is constant and thus is not informative. Since $Xu_1 = 0$, the centroid of the $n$ vectors is the origin:

$$\sum_{i=0}^{n} x(i) = 0 \tag{3.3}$$

The Gram matrix of matrix $X$ is the pseudo-inverse of the Laplacian:

$$X^T X = U \Lambda^+ U^T = L^+$$

In particular we can recover the Laplacian matrix by taking the pseudo-inverse of $X^T X$ and hence the adjacency matrix and the graph itself. In words, the graph is completely encoded in nodes' embedding $X$, i.e. without any loss of information.

We now give a random walk interpretation of this embedding. We consider the random walk with transition rate $A_{ij}$ from node $i$ to $j$: the walker stays at node $i$ an exponential time with parameter $d_i$, then moves from node $i$ to node $j$ with probability $P_{ij} = \frac{A_{ij}}{d_i}$. This defines a continuous-time Markov chain with

generator matrix $L$ and uniform stationary distribution. The sequence of nodes visited forms a discrete-time Markov chain with transition $P$. Let $H_{ij}$ be the mean hitting time of node $j$ from node $i$. We can note that $H_{ii} = 0$ for all node $i$ of the graph. The following results are standard, see [Lovász et al., 1993], we follow the notation in [Bonald et al., 2018b]: $H_{ij} = n(x(j) - x(i))^T x(j)$, hence the mean commute time between nodes $i$ and $j$ is:

$$C_{ij} = H_{ij} + H_{ji} = n||x(i) - x(j)||^2$$

The matrix of commute times caracterizes the graph:

**Proposition 2.** *We can reconstruct the adjacency matrix from the matrix of commute times.*

*Proof.* Since $\frac{C_{ij}}{n} = ||x(i)||^2 = 2x(i)^T x(j) + ||x(j)||^2$, we can recover the matrix $X^T X = L^+$ from $C$ if we know $||x(i)||^2$. But, thanks to 3.3, we have $\sum_j C_{ij}/n = n||x(i)||^2 + \sum_j ||x(j)||^2$ and $\sum_{ij} C_{ij}/n^2 = 2\sum_j ||x(j)||^2$, hence the proof of theorem 2. □

The information of the graph is contained in the intrinsic geometry of vectors $x(1), ..., x(n)$. The intrinsic values of those vectors also carry relevant information. According to Fiedler's nodal domain theorem [Fiedler, 1975], for any $k \geq 2$, let $W_k = \{i \in V, x_k(i) \geq 0\}$. Then the graph induced by $G$ on $W_k$ has at most $k - 1$ connected components. This implies that for low values of $k$, nodes $i$ with non-negative entries $x_k(i)$ tend to be well connected.

### 3.3.4 From Node Embeddings to Graph Embedding

In order to incorporate the geometric information in the node embeddings $x(1), \ldots, x(n)$, we can proceed as above and construct a graph embedding thanks to the symmetric functions $\ell(.; t)$ of Proposition 1 applied to the vector of $x(i)$'s componentwise.

Note however that this will not produce an invariant graph embedding. Indeed as soon as an eigenvalue has multiplicity larger than 2, the associated eigenvectors are defined up to a rotation. Even if all eigenvalues have multiplicity one, the eigenvector is only defined up to a sign.

We now show how to construct an invariant graph embedding from the embedding of the nodes under the assumption that all eigenvalues of the Laplacian have multiplicity 1.

Consider two graphs $G$ and $H$ with respective Laplacians $L_G = U\Lambda U^T$ and $L_H = V\Lambda V^T$, where $\Lambda$ is the common diagonal matrix of eigenvalues. If $\Pi$ is

the matrix of an isomorphism from $G$ to $H$, then, we have

$$\Pi U \Lambda U^T \Pi^T = V \Lambda V^T.$$

As each entry of $\Lambda$ is distinct, this looks like it would imply $\Pi U = V$. But, the eigenvectors are only determined up to sign, hence we only have the following result:

**Lemma 1.** *Assume $L_G$ an $L_H$ have the same distinct eigenvalues $0 = \lambda_1 < \lambda_2 < \cdots < \lambda_n$. A permutation matrix $\Pi$ satisfies $\Pi L_G \Pi^T = L_H$ if and only if there exists a diagonal $\pm 1$ matrix $S$ such that:*

$$\Pi U = VS.$$

*Proof.* Let $U_1, \ldots, U_n$ be the columns of $U$ and $V_1, \ldots, V_n$ the columns of $V$. The equality $\Pi L_G \Pi^T = L_H$ gives:

$$V \Lambda V^T = \sum_{i=1}^{n} V_i \lambda_i V_i^T = \sum_{i=1}^{n} (\Pi U_i) \lambda_i (U_i^T \Pi^T),$$

which implies that for all $i$:

$$V_i V_i^T = (\Pi U_i)(\Pi U_i)^T.$$

This in turn implies that $V_i = \pm \Pi U_i$.
To go the other direction, assume $\Pi U = VS$, so that

$$\begin{aligned} \Pi L_G \Pi^T &= \Pi U \Lambda U^T \Pi^T \\ &= VS\Lambda SV^T \\ &= V\Lambda SSV^T = V\Lambda V^T = L_H, \end{aligned}$$

as $S$ and $\Lambda$ are diagonal and thus commute. □

From this lemma, we see that for any $k \in [1, n]$ and $t \geq 0$, the embedding given by:

$$\frac{\ell(x_k(1), \ldots, x_k(n); t) + \ell(-x_k(1), \ldots, -x_k(n); t)}{2} \in \mathbb{R},$$

where the function $\ell$ is defined by (3.2), is an invariant embedding of the graph, provided the eigenvalues of this graph have all multiplicity one.
In practice, we will fix the parameters $t_1 < \cdots < t_m$ as in Section 3.3.2 as well as a parameter $k_3 \geq 2$ and pick the first informative eigenvectors of the Laplacian $k \in \{2, \ldots, k_3\}$. Note that our embedding loses the notion of Fiedler's

nodal domain because we do not have access anymore to the sign of the $x_k(i)$'s. However, we explain in the next section how to incorporate the more accurate information contained in the commute times. At this stage, we created a graph embedding $\mathcal{F}_3(G)$ with dimension $m(k_3-1)$ from the matrix $(x_k(i))_{1\leq i\leq n, 2\leq k\leq k_3}$.

**Graph embeddings from commute times.** Moreover, in order to incorporate the information of the commute times matrix, we proceed in a very straightforward manner. Instead of the Euclidean distances $||x(i)-x(j)||^2$, we compute the dot products $x(i)^T x(j)$ for $1 \leq i,j \leq n$. Then, we flatten this matrix to obtain a vector of size $n^2$ and pass this vector through our symmetric function $l$ defined in 3.2.
Our embedding is thus defined by:

$$\mathcal{F}_4(G) = (l(x_1, ..., x_n; t_k))_{1\leq k\leq m} \in \mathbb{R}^m \tag{3.4}$$

which is clearly an invriant embedding of the graph. Note that even if we can reconstruct the original graph from the matrix $(x(i)^T x(j))_{1\leq i,j\leq n}$, our embedding $\mathcal{F}_4$ defined in 3.4 will not allow to reconstruct the graph. Indeed, even if we had access to the function $t \mapsto l((x(i)^T x(j))_{1\leq i,j\leq n}; t)$, this function characterizes only the multiset $\{x(i)^T x(j), 1 \leq i,j \leq n\}$ which does not a priori characterize the graph up to isomorphisms.

### 3.3.5 Final Graph Embedding

So far, we defined four different graph embeddings: $\mathcal{F}_1(G) \in \mathbb{R}^{k_1}$ defined in Section 3.3.1; $\mathcal{F}_2(G) \in \mathbb{R}^{k_2 m_2}$ defined in Section 3.3.2; $\mathcal{F}_3(G) \in \mathbb{R}^{m_3(k_3-1)}$ defined in Section 3.3.4 and $\mathcal{F}_4(G) \in \mathbb{R}^{m_4}$ defined in Section 3.3.4. Our final graph embedding is the concatenation of these embeddings for fixed parameters $k_1, k_2, k_3$ and $m_2, m_3, m_4$: $\mathcal{F}_{SGE}(G) = (\mathcal{F}_1(G), \mathcal{F}_2(G), \mathcal{F}_3(G), \mathcal{F}_4(G))$ which is a vector of size $k_1 + m_2 k_2 + m_3(k_3-1) + m_4$, independent of the size of the graph. All these embeddings are invariant graph embeddings except $\mathcal{F}_3$ which has been shown to be invariant on the familly of graphs with Laplacian eigenvalues of multiplicity one. In particular, for two graphs to give the same embedding $\mathcal{F}_{SGE}$, they need to be cospectral, i.e. have the same eigenvalues, with at least one eigenvalue with muliplicity at least 2. This is only a necessary condition. Finding the non-isomorphic graphs for which our embedding is not discriminative is out of the scope of this paper. In Section 3.5, we will assess the discriminative power of our embedding on benchmark datasets for graph classification.
At this stage, we should stress that our general method for computing graph embedding is rather generic and several variations are possible. For example Zhang et al. [2018b] constructs a graph kernel based on return probabilities of

random walks. We can easily adapt this idea into our space embedding in Section 3.3.2 by replacing the vector $1^T P^k$ by the return probability vector defined by $\text{diag}(P^k) = (P^k_{11}, \ldots, P^k_{nn})$. Our analysis is still valid as this vector is still an invariant embedding of nodes. In the spectral domain, Verma and Zhang [2017] introduces an embedding very similar to our embedding $\mathcal{F}_4$ based on the node embedding $\sqrt{g(\Lambda)}U^T$ where $g(\Lambda) = \text{diag}(0, g(\lambda_2), \ldots, g(\lambda_n))$ for a monotonic function $g$. In their experiments, they choose $g(\lambda) = 1/\lambda$ corresponding to our choice in Section 3.3.3 and then constructed an histogram of the commute times to get a graph embedding.

We should stress that using a general function $g$ in this context was already proposed in Hammond et al. [2011] for the spectral construction of graph wavelets. Indeed, the graph Fourier transform is defined in term of the graph Laplacian $L$. Indeed for a signal $S = (S_1, \ldots S_n)$ on the graph, the Fourier transform of $S$ is defined by $\hat{S} = U^T S$ and for a so-called wavelet kernel $g$ (i.e. a function with $g(0) = 0$ and $g(x) \to 0$ as $x \to \infty$), the wavelet operator at scale $t$, $T^t_g$ is defined by:

$$T^t_g S = g(tL)S = \sum_i U_i g(t\lambda_i) U_i^T S, \text{ for } t > 0.$$

Hence a natural generalization of our embedding $\mathcal{F}_4$ would be to include various scale parameters by considering the node embedding $\sqrt{g(t\Lambda)}U^T$ for predefined values of $t$.

## 3.4 Permutation Invariance Neural Network (PINN)

So far, our aim in constructing our graph embedding has been to get the best possible embedding in term of disciminative power. As we did not make any assumption on the families of graphs to disciminate, we used the graph isomorphism problem as a proxy for the quality of our embedding. In other words, our graph embedding is constructed in order to be able to distinguish graphs in a worst case scenario. As explained in previous section, several extensions of our embedding are possible. However, our embedding should remain in a low dimensional space in order to be useful. Moreover, our embedding will not depend on the dataset but some features might be much more useful on say social graphs than for bioinformatics datasets. We now explore possible algorithms able to learn embeddings from the data.

### 3.4.1 Invariance properties

As explained in the introduction, this problem in the specific context of graph classification is not new and GNN have been devised exactly for this task. Our

approach is rather orthogonal to the GNN approach also we will discuss this point in more details in Section 3.4.2. In order to learn our graph embedding, we assume that we start from an embedding of the nodes $(\mathcal{E}(G)(i))_{i\in V}$ and need to learn the symmetric function $f$ able to create the best graph embedding $\mathcal{F}(G) = f(\mathcal{E}(G)(i)_{i\in V})$ for the classification task on the given dataset.
If the embedding of the nodes is equivariant, then we know that the resulting graph embedding is invariant. This is a desirable property of our embedding which would ensure that the classifier gives always the same class for two isomorphic graphs. The embedding of the nodes could be otained thanks to a GNN or using the node embedding presented in Section 3.3.2 or 3.3.3. We do not make any assumption on the embedding of nodes and now concentrate on the learning of a symmetric function. Our main structure Theorem 1 shows that the learning of a symmetric function $f : \mathbb{R}^k \to \mathbb{R}$ reduces to the learning of a function $h : \mathbb{R}^k \to \mathbb{R}^K$ and a function $g : \mathbb{R}^K \to \mathbb{R}$ where $K$ decreases with the smoothness of the funtion $f$.

**Theorem 1.** *Let $f : \mathcal{S}^n \to \mathbb{R}^m$ be a symmetric function. There exists a function $\varphi$ from the multisets with elements of $\mathcal{S}$ and cardinality $n$ to $\mathbb{R}^m$ such that*

$$f(x) \quad = \quad \varphi(\{x_1, \ldots, x_n\}).$$

*Let consider the case $\mathcal{S} = [0,1]^k$ and that the function $f$ is smooth as follows: for all $x, y \in S^n$ such that $\inf_\sigma \sup_i |x_{\sigma(i)} - y_i| \leq \delta$, we have $|f(x) - f(y)| \leq \epsilon$. In this case, for $K = \lceil \frac{3}{\delta} \rceil^k$, there exists a function $g : \mathbb{R}^K \to \mathbb{R}$ and a function $h : \mathbb{R}^k \to \mathbb{R}^K$ such that for all $x_i \in [0,1]^k$,*

$$\left| f(x_1, \ldots, x_n) - g\left( \sum_{i=1}^{n} h(x_i) \right) \right| \leq \epsilon. \tag{3.5}$$

*If moreover all the $x_i$'s are distant by at least $\delta$ (i.e. $min_{i\neq j}|x_i - x_j| \geq \delta$), then with the same functions $g$ and $h$, we have:*

$$\left| f(x_1, \ldots, x_n) - g\left( \max_{i=1}^{n} h(x_i) \right) \right| \leq \epsilon. \tag{3.6}$$

*In the case where $\mathcal{S}$ is countable, for any symmetric function $f$, there exists $h : \mathcal{S} \to \mathbb{R}$ and $g : \mathbb{R} \to \mathbb{R}^m$ such that $f(x_1, \ldots, x_n) = g\left(\sum_{i=1}^{n} h(x_i)\right)$.*

The proof is given below but some particular cases of our result appeared in the literature. The countable case was already proved in Zaheer et al. [2017] for sets and extended in Xu et al. [2018] for multisets. The uncountable case is stated as an open problem in Zaheer et al. [2017]. Theorem 1 in Qi et al. [2017] proves (3.6) for sets.

*Proof.* We define by $\mathcal{N}_n(\mathcal{S})$ the set of point measures $\nu$ on $\mathcal{S}$ such that $\nu(\mathcal{S}) = n$. If $x = (x_1, \ldots, x_n) \in \mathcal{S}^n$ then the type of $x$ is the element $\epsilon_x$ of $\mathcal{N}_n(\mathcal{S})$ defined by

$$\epsilon_x := \sum_{i=1}^{n} \delta_{x_i} = \sum_{y \in \{x_1, \ldots, x_n\}} \delta_y,$$

where the last sum is over the multiset $\{x_1, \ldots, x_n\}$. The set $\mathcal{S}^n(\nu) \subset \mathcal{S}^n$ is defined by, for $\nu \in \mathcal{N}_n(\mathcal{S})$:

$$\mathcal{S}^n(\nu) = \{y \in S^n : \epsilon_y = \nu\}.$$

In words, this is the set of all vectors such that the multiset obtained from their individual components is descrived by $\nu$. In particular if $\nu = \sum_{\ell=1}^{k} \nu_\ell \delta_{x_\ell}$ for $\sum_\ell \nu_\ell = n$ and $x_1, \ldots x_k \in \mathcal{S}$, then the cardinality of $S^n(\nu)$ is $|\mathcal{S}^n(\nu)| = \frac{n!}{\prod_\ell \nu_\ell!}$. By definition, for all $x = (x_1, \ldots, x_n) \in \mathcal{S}^n$,

$$\begin{aligned} f(x) &= \frac{1}{n!} \sum_{\sigma \in \mathcal{S}_n} f(x_{\sigma(1)}, \ldots, x_{\sigma(n)}) \\ &= \frac{1}{|\mathcal{S}^n(\epsilon_x)|} \sum_{y \in \mathcal{S}^n(\epsilon_x)} f(y). \end{aligned}$$

In particular, we can write:

$$f(x) = \varphi(\{x_1, \ldots, x_n\}),$$

where $\varphi$ is a function of the multiset $\{x_1, \ldots, x_n\}$ defined by the line just above. We now prove (3.5). Let $(P_{\mathbf{j}})_{\mathbf{j} \in [K]}$ be a partition of $[0,1]^k$ in squares with length $\lceil \frac{3}{\delta} \rceil$ and let $(y_{\mathbf{j}})_{\mathbf{j} \in [K]}$ be the centers of these squares. We then define $h(x)_{\mathbf{j}} = \mathbf{1}(x \in P_{\mathbf{j}})$ for $\mathbf{j} \in [K]$. Hence we have for all $\mathbf{j} \in [K]$,

$$\sum_{i=1}^{n} h(x_i)_{\mathbf{j}} = \epsilon_x(P_{\mathbf{j}}).$$

Now we define $g$ as follows: for any $\nu = (\nu_1, \ldots, \nu_K) \in \mathbb{R}^K$ with $\sum_{\mathbf{j}=1}^{K} \nu_{\mathbf{j}} = n$ and $\nu_{\mathbf{j}} \in \mathbb{N}$, there exists a (unique) multiset $\{y_{\mathbf{j}_1}, \ldots, y_{\mathbf{j}_n}\}$ of n centers such that $\nu_{\mathbf{j}} = \sum_{y \in \{y_{\mathbf{j}_1}, \ldots, y_{\mathbf{j}_n}\}} \mathbf{1}(y \in P_{\mathbf{j}})$ and we define $g(\nu) = \varphi(\{y_{\mathbf{j}_1}, \ldots, y_{\mathbf{j}_n}\})$. Then (3.5) follows from the fact that for a given $x$, the multiset of centers associated to the vector $(\epsilon_x(P_{\mathbf{j}}))_{\mathbf{j} \in [K]}$ is such that $\inf_\sigma \sup_i |x_{\sigma(i)} - y_{\mathbf{j}_i}| \leq \delta$. If all the $x_i$'s are distant then we have $\epsilon_x(P_{\mathbf{j}}). \in \{0, 1\}$ for all $\mathbf{j}$ so $\max_{i=1}^{n} h(x_i)_{\mathbf{j}} = \epsilon_x(P_{\mathbf{j}})$ and (3.6) follows.

The case where $\mathcal{S}$ is countable is simple: consider an injective mapping $c : S \to \mathbb{N}$

and then define $h(x) = n^{-c(x)}$. The result then follows from the fact that $\sum h(x_i)$ characterizes the multiset. □

### 3.4.2 Algorithm

Theorem 1 hints at a general strategy for inference over permutation invariant objects, like sets or multisets. We can use deep neural networks to model and learn the functions $h$ and $g$ in (3.5) in order to produce permutation invariant networks. This idea was used in Qi et al. [2017] where an architecture based on a multi-layer perceptron with shared weights was used in combination with a max pooling layer (accross the batch) to aggregate information in order to give a symmetric function acting on point clouds. Since point clouds are considered as sets in $\mathbb{R}^3$, we see that using a max pooling layer instead of an averaging layer is a valid option in view of (3.6). This approach was extended in Zaheer et al. [2017] to deal with general sets. The architecture is explicited in figure 3.1.
Equivariant models are also introduced in Zaheer et al. [2017]. We recall the definition of equivariant function:

**Definition 3.** *A function $f : \mathcal{X}^n \to \mathcal{Y}^n$ is equivariant if for any permutation $\sigma$ and any $x \in \mathcal{X}^n$, we have $f(x_{\sigma(1)}, \dots, x_{\sigma(n)}) = (f_{\sigma(1)}(x), \dots, f_{\sigma(n)}(x))$.*

Note that the notion of equivariance can be made more general by replacing the group of permutations by any other group Cohen and Welling [2016] (but this will not be of interest in our setting).
Clearly, if a function is symmetric, then first passing its input through an equivariant function from $\mathcal{S}^n$ to $\mathcal{S}^n$ will leave the function symmetric. In particular as suggested in Zaheer et al. [2017], stacking equivariant layers will produce an equivariant network and hence if a final symmetric layer is applied, then we again obtain a symmetric network. This idea is explored further in Lucas et al. [2018], where Theorem 1 in Lucas et al. [2018] shows that such architectures with equivariant linear layers and with a final summation layer can approximate any symmetric function.
From a practical perspective, it seems appropriate to consider more general architectures using both ideas. Starting from the aproximation $g(\sum_i h(x_i))$, we see that the function $(x_1, \dots, x_n) \mapsto h(x_1), \dots h(x_n)$ is equivariant and can be replaced by any equivariant function $(x_1, \dots, x_n) \mapsto h(x_1, \mu), \dots h(x_n, \mu)$ where $\mu$ is an invariant function of $(x_1, \dots, x_n)$. Hence, we can stack such layers, where invariant information of the previous layers can be incorporated in the next equivariant layer. From a theoretical perspective, this modification will not increase the expressiveness of the network.
We feed the algorithm with input features that are arbitrary node embeddings.

The algorithm is not dependent on the size of the graph since we apply a shared fully connected layer to all nodes' input representation and we then aggregate the outputs over all nodes. The output of this method is thus of size the size of the output linear layer and is thus independant of the size of the graph. We chose node embeddings defined in section 3.3 as input of the premutation invariant neural network, since the input must be node level embeddings, we use only $\mathcal{F}_2$ and $\mathcal{F}_4$.

**Comparison with GNN:** in contrast with our approach, GNN computes a state at each node of a given graph by aggregating the information from the neighboring nodes. In order to be able to transfer the learning process from one graph to another, only symmetric functions can be learned Xu et al. [2018]. In particular, our Theorem 1 applies and gives all the possible aggregation function at a given node. Our method is rather orthogonal as our symmetric function is applied on the whole graph instead of local pattern of the graph. In particular, we can combine both methods by first using GNN to construct embedding of the nodes and then use a permutation invariant neural network (PINN). Both GNN and PINN can be learned simultanously.

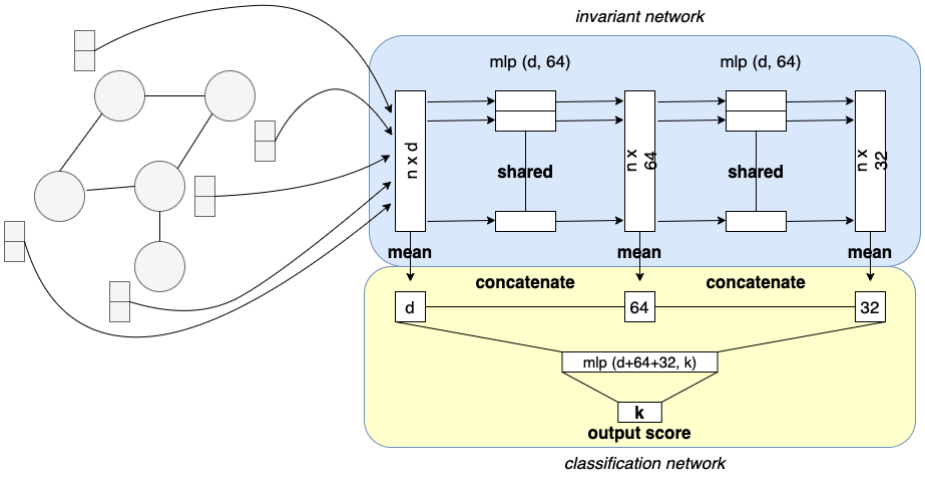



Figure 3.1: PINN architecture. Node embeddings are collected in order to generate an input matrix of size $n \times d$. Successive shared multi-layer perceptrons (MLP) are applied to input features are node features are aggregated after each step and then concatenated to obtain a hierarchical representation of the graph. A MLP is finally applied to these concatenated features to compute a score of belonging to each labeled class. This network is trained in a supervised fashion.

## 3.5 Experiments

### 3.5.1 Graph Classification

**Datasets:** We use the 6 bioinformatics datasets presented in chapter 1 and 5 social network datasets to compare our results with Verma and Zhang [2017]: REDDIT-BINARY and REDDIT-MULTI, IMDB-BINARY and IMDB-MULTI and COLLAB. The social network datasets are composed of ego-networks the labels of the graphs are the nature of the entity from which we have generated the ego-network. More details can be found in [Yanardag and Vishwanathan, 2015].

**Experimental setup:** After having computed the features for each graph, we use a Random Forest for the classification. We set the parameters to 1000 for the number of trees and we choose a max depth of 20 (we ran experiments with max depth varying between 20 and 50 to quantify the importance of this parameter and the results were the same). We use cross-validation by splitting each dataset into 10 folds and by preserving the class proportion in each fold. We train our algorithm on 9 folds and then test it on the remaining one. We then average the accuracy over all testing sets. We compare our method with 6 state-of-art algorithms: Family of Graph Spectral Distances Verma and Zhang [2017], Diffusion CNNs (DCNN) Atwood and Towsley [2016], Deep Graph Kernel (DGK) Yanardag and Vishwanathan [2015]. Most methods do not use node labels in their experiments, in order to compare ours to these methods we didn't use node labels in our algorithm. Concerning the parameters, we set $m_2 = m_3 = 11$ and $m_4 = 21$ for all experiments. We adapt parameters $k_1, k_2, k_3$ to the size of the graph. These parameters represent the number of eigenvalues, the hopes at which we investigate the graph and the dimension of the eigenvectors selected. We use three settings: for smaller graphs (PTC, MUTAG) we set $k_1 = 5, k_2 = 2, k_3 = 5$, for PROTEINS, NCI1, NCI109 and IMDB-B and IMDB-M we set $k_1 = 50, k_2 = 5, k_3 = 50$ and for bigger graphs (DD, REDDIT-B, REDDIT-M, COLLAB) we set $k_1 = 100, k_2 = 5, k_3 = 100$. We built these three groups according to the number of nodes of the largest graph in each dataset.

**Results:** From the results of Table 5.1 we can observe that SGE is challenging with state-of-the-art methods. Indeed, on most datasets, SGE has a score very close to the ones obtained by Verma and Zhang [2017] and SGE outperforms FSGD on PTC dataset. From Table 3.2, SGE is challenging with all other algorithms on social network datasets and outperforms FSGD on both REDDIT

datasets.
For the permutation invariant neural network (PINN), we can see that this models performs well on certain datasets such as PROTEINS but is less effective on most datasets. This can be due to the fact that those datasets are quite small in terms of number of graphs which makes it difficult to learn a neural network model. Moreover, a limitation of PINN is that the topology of the graph is taken into account into the input features used for the algorithm. But most neural network models such as GCN [Kipf and Welling, 2016, Veličković et al., 2017] aggregate features of neighbors into the central node which adds a lot of information to the model. This model that is very effective on 3D cloud of points may be less effective on graph datasets and may be dependant on the features that are set as inputs.

**Datasets overview:** On IMDB-MULTI a low accuracy by all methods can be explained by the fact that more than half of the graphs of each class are complete graphs. We counted 789 complete graphs accross all classes which is more than half of the dataset since IMDB-MULTI is composed of 1500 graphs. Two complete graphs of same size $N$ are not distinguishable by an embedding of their structure. We compared the repartition of complete graphs by size in each class. We observed that complete graphs of a given size $N$ are not all in the same class which makes it impossible for the classifier to classify properly all complete graphs of size $N$. A good classifier would attribute all complete graphs of size $N$ to the class in which we find the largest number of complete graphs of size $N$. This way we can count the number of mis-classification for complete graphs of size $N$. We repeated the process over all unique sizes of complete graphs present in our dataset and we obtained that 389 graphs are miss-classified over the dataset composed of 1500 graphs. The best classifier would thus have a classification of 74% and not 100%.
In order to distinguish those graphs we should use features on nodes which are not available on this dataset.
Moreover for small datasets, namely PTC and MUTAG, a significant fraction of graphs have an identical topology and are thus not distinguishable without labels on nodes. In order to be able to differentiate them we should use node labels in our methods which we plan to do in future works.

### 3.5.2 Discussion

In this chapter, we introduced two algorithms, one based on handcrafter embeddings and a learnable approach to develop an invariant neural network for a graph classification task.

| **Dataset** | MUTAG | PTC | PROTEINS | NCI1 | NCI109 | D&D |
|---|---|---|---|---|---|---|
| Avg | 18 | 14 | 39 | 30 | 30 | 284 |
| #Graphs | 188 | 344 | 1113 | 4110 | 4127 | 1178 |
| DCNN | 66.51 | 55.79 | 65.22 | 63.10 | 60.67 | - |
| DGK | 86.17 | 59.88 | 71.69 | 64.40 | 67.14 | 72.95 |
| FSGD | **92.12** | 62.80 | **73.42** | **79.80** | **78.84** | **77.10** |
| **SGE** | $89.37 \pm 0.93$ | $\mathbf{63.77 \pm 0.31}$ | $\mathbf{73.85 \pm 0.11}$ | $76.64 \pm 0.10$ | $75.38 \pm 0.22$ | $\mathbf{77.10 \pm 0.21}$ |
| **PINN** | $81.22 \pm 0.69$ | $61.48 \pm 0.29$ | $\mathbf{75.65 \pm 0.17}$ | $74.63 \pm 0.05$ | $73.33 \pm 0.26$ | $72.31 \pm 0.14$ |

Table 3.1: Classification accuracy on bioinformatics datasets

| **Dataset** | IMDB-B | IMDB-M | REDDIT-B | REDDIT-M | COLLAB |
|---|---|---|---|---|---|
| Avg | 20 | 13 | 430 | 509 | 74 |
| #Graphs | 1000 | 1500 | 2000 | 4999 | 5000 |
| DGK | 66.96 | 44.55 | 78.04 | 41.27 | 73.09 |
| FSGD | **73.62** | **52.41** | 86.50 | 47.76 | **80.02** |
| **SGE** | $\mathbf{73.7 \pm 3.2}$ | $49.60 \pm 2.1$ | $\mathbf{89.30 \pm 2.2}$ | $\mathbf{51.13 \pm 2.4}$ | $78.71 \pm 2.0$ |
| **PINN** | $71.02 \pm 0.23$ | $45.40 \pm 0.22$ | $\mathbf{87.88 \pm 0.16}$ | $48.46 \pm 0.12$ | $74.64 \pm 0.04$ |

Table 3.2: Classification accuracy on social network datasets

The handcrafted embedding presented in 3.3 is based on classical spectral graph representations.

In the work on PointNet, Qi et al. [2017] developped an invariant neural netwrok to perform classification and segmentation of 3D data points. Inspired by the invariance property of PointNet, we developped PINN that has a strong discriminative power because it is able to identify correlations between node representation by its weight sharing architecture. Moreover, this model, originally developped to classify cloud of points, is particularly adapted to graph structured data. Indeed, when trying to classify graphs, we need to embed each graph into a vector invariant by node permutation. This is precisely what PINN allows us to do. Nevertheless, there are two limitations to this kind of architectures.

- First, the neighborhood structure of graphs is not taken into account in the network architecture but only in the initial representation of nodes that is given as input to the algorithm. This limitation is overcame by Ying et al. [2018] by aggregating features of neighbors of each nodes after having performed fully connected layers. This allows to obtained embeddings that will be much closer for nodes that are topologically close.

- The second limitation concerns the features that need to be set as inputs of the algorithm. The choice of the input narrows us in a direction instead

of learning all parts of the algorithm.

# Chapter 4

# Node Structural Aggregation

In this Chapter we continue working on the problem of graph classification using a different perspective. We develop an aggregation layer to identify structural roles for nodes in graphs and to bring explainability to graph classification algorithms. We design an algorithm trainable in an end-to-end fashion. The work from this chapter is currently under review at ICLR 2021.

## 4.1 Introduction

Convolution neural networks [LeCun et al., 1995] have proven to be very efficient at learning meaningful patterns for many articificial intelligence tasks. They convey the ability to learn hierarchical information in data with Euclidean grid-like structures such as images and text. Convolutional Neural Networks (CNNs) have rapidly become state-of-the art methods in the fields of computer vision [Russakovsky et al., 2015] and natural language processing [Devlin et al., 2018]. However in many scientific fields, studied data have an underlying graph or manifold structure such as communication networks (whether social or technical) or knowledge graphs. Recently there have been many attempts to extend convolutions to those non-Euclidean structured data [Hammond et al., 2011, Kipf and Welling, 2016, Defferrard et al., 2016]. In these new approaches, the authors propose to compute node embeddings in a semi-supervised fashion in order to perform node classification. Those node embeddings can also be used for link prediction by computing distances between each node of the graph [Hammond et al., 2011, Kipf and Welling, 2016].
Graph classification is studied in many fields. Whether for predicting the chemical

activity of a molecule or to cluster authors from different scientific domains based on their ego-networks [Freeman, 1982]. However when trying to generalize neural network approaches to the task of graph classification there are several aspects that differ widely from image classification. When trying to perform graph classification, we can deal with graphs of different sizes. To compare them we first need to obtain a graph representation that is independant of the size of the graph. Moreover, for a fixed graph, nodes are not ordered. The graph representation obtained with neural network algorithms must be independant of the order of nodes and thus be invariant by node permutation. In Chapter 3 we persented a graph embedding invariant by node permutation for a graph classification task. In this Chapter, we present an aggregation function that is also invariant by node permutation but that aims at identifying structural roles for nodes in graphs.

Aggregation functions are functions that operate on node embeddings to produce a graph representation. When tackling a graph classification task, the aggregation function used is usually just a mean or a max of node embeddings as illustrated in figure 4.1b. But when working with graphs of large sizes, the mean over all nodes does not allow us to extract significant patterns with a good discriminating power. In order to identify patterns in graphs, some methods try to identify structural roles for nodes. Donnat et al. [2018] define structural role discovery as the process of identifying nodes which have topologically similar network neighborhoods while residing in potentially distant areas of the network as illustrated in figure 4.1a. Those structural roles represent local patterns in graphs. Identifying them and comparing them among graphs could improve the discriminative power of graph embeddings obtained with graph neural networks. In this work, we build an aggregation process based on the identification of structural roles, called StructAgg.

The main contributions of this work are summarized bellow:

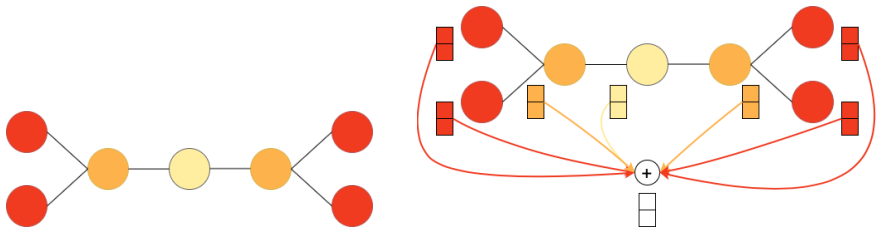
1. **Learned aggregation process.** A differentiable aggregation process that learns how to aggregate node embeddings in order to produce a graph representation for a graph classification task.

2. **Identification of structural roles.** Based on the definition of structural roles from Donnat et al. [2018], our algorithm learns structural roles during the aggregation process. This is innovative because most algorithms that learn structural roles in graphs are not based on graph neural networks.

3. **Explainability of selected features for a graph classification task.** The identification of structural roles enables us to understand and explain what features are selected during training. Graph neural networks often

lack explainability and there are only few works that tackle this issue. One contribution of this work is the explainability of the approach. We show how our end-to-end model provides interpretability to a graph classification task based on graph neural networks.

4. **Experimental results.** Our method achieves state-of-the-art results on benchmark datasets. We compare it with kernel methods and state-of-the-art message passing algorithms that use pooling layers as aggregation processes.



(a) nodes with the same structural role are classified together (same color).

(b) aggregation process to create a graph embedding, the node features are summed to produce a representation of the graph.

Figure 4.1: Identification of structural roles and aggregation of node features over the whole graph.

## 4.2 Related Work

Recently there has been a rich line of research, inspired by deep models in images, that aims at redefining neural networks in graphs and in particular convolutional neural networks [Defferrard et al., 2016, Kipf and Welling, 2016, Veličković et al., 2017, Hamilton et al., 2017, Bronstein et al., 2017, Bruna et al., 2013, Scarselli et al., 2009]. Those convolutions can be viewed as message passing algorithms that are composed of two phases [Gilmer et al., 2017]. A message passing phase that runs for T steps is first defined in terms of message functions and vertex update functions. A readout phase then computes a feature vector for the whole graph using some readout function. In this work we will see how to define a readout phase that is learnable and that is representative of meaningful patterns in graphs.

Graph neural networks are used for a wide variety of tasks. For node classification Hamilton et al. [2017] and Kipf and Welling [2016] learn node features that are representative of the modular topology of the graph. Those features can also

be used for link prediction [Zhang and Chen, 2018]. For both of these tasks, the size of the graph does not matter since a single graph is set as input and a feature vector is computed for each node. When addressing graph classification problems the setting is not the same. We have a set of graphs of different sizes and we need to learn a representation for the whole graph in order to address the classification task. The readout phase of the message passing algorithm is highly important in this case since we need to aggregate the information of nodes to compute a graph representation that must be as discriminative as possible, invariant to node permutation and independant of the size of the input graph. There are several ways to develop a readout function that goes from node embeddings to a representation of the entire graph. The simplest way to answer this, is to use a simple aggregation function like the sum or the average [Xu et al., 2018] of the node embeddings although more complex functions can also be learned. Another way to address this issue is to apply a pooling layer to the graph in order to obtain a smaller graph of fixed size [Gao and Ji, 2019, Ying et al., 2018, Defferrard et al., 2016]. This process can be applied iteratively to the input graph to obtain a hierarchical representation of the graph.

In this work we will see how to define a new aggregation process that is based on the identification of structural roles for nodes in graphs [Gilpin et al., 2013, Henderson et al., 2012, Ribeiro et al., 2017]. The identication of these roles provides key insight into the organisation of a network and is a way to identify meaningful patterns in graphs. A way to identify these structural roles is by creating handcrafted features that encode local topology of nodes [Donnat et al., 2018]. In this work, we will first see how to learn those roles during the training of our classification algorithm. We will then see how to develop an aggregation function that is based on these roles. We will finally compare this end-to-end algorithm with state-of-the-art models on benchmark datasets. We will also see how the identification of structural roles provides explainability to graph neural network algorithm. More precisely, we will see that those structural roles allow us to understand local topological patterns that are selected during training.

## 4.3 Wavelets for structural node embedding

In this section we present related works on node structural embeddings. We present more specifically handcrafted embeddings based on wavelets [Donnat et al., 2018] and we relate it to our work on node structural embedding based on graph neural networks.

### 4.3.1 Wavelets in Graphs for Node Structural Embedding

Wavelets are a powerful tool for signal processing [Mallat, 1999, Bruna and Mallat, 2013]. There has been numerous works that aimed at redefining wavelets on graphs [Hammond et al., 2011, Gama et al., 2019, Tremblay and Borgnat, 2014b]. With the appropriate scaling function, wavelets are useful to decompose the signal into a hierarchical one that can be localized in space [Kipf and Welling, 2016, Hammond et al., 2011]. In their work, Donnat et al. [2018] use wavelets to build node representations that encode neighborhood topologies in order to embed nodes according to the structural role they play in graphs.

We recall that $U$ is the eigenvector decomposition of the Laplacian $L = D - A = U\Lambda U^T$ and let $\lambda_1 < \lambda_2 \leq ... \leq \lambda_n$ be the eigenvalues of $L$, $\Lambda = Diag(\lambda_1, ..., \lambda_n)$. Let $g_s$ be a filter kernel wih scaling parameter $s$. Spectral graph wavelets associated to function $g_s$ at scale $s$ are defined as the signal resulting from the modulation in the spectral domain of a Dirac signal centered in node $v$. The spectral graph wavelet $\Psi_v$ is then given by:

$$\Psi_v = UDiag(g_s(\lambda_1), ..., g_s(\lambda_n))U^T\delta_v \tag{4.1}$$

Where $\delta_v$ is the one-hot vector for node $v$. The $m^{th}$ wavelet coefficient of the vector can thus be given by:

$$\Psi_{uv} = \sum_{l=1}^{n} g_s(\lambda_l)U_{ul}U_{vl} \tag{4.2}$$

Eigenvectors associated with smaller eigenvalues carry slow varying signal as explained in 2. Thus signals associated with smaller eigenvalues encourage nodes that are neighbors to share similar values. In contrast, eigenvectors associated with greater eigenvalues carry erratic signals. The functions $g_s$ allows us to modulate the filters applied to the eigenvalues to select low pass and high pass signals in order to separate the graph signal and detect meaningful patterns.

### 4.3.2 Node Structural Embedding

From wavelets in graphs, Donnat et al. [2018] design an embedding to encode structural roles in graphs.
Spectral graph wavelets are first applied to the graph to obtain a diffusion pattern for every node. For a given scale $s$, coefficients in each wavelet represent the amount of energy that node $v$ received from node $u$ ($\Psi_{uv}$). We use $u$ and $v$ to denote nodes to avoid confusion with the $i$ of the characteristic function. In all the rest of the thesis we use $i$ and $j$ to denote the $i^{th}$ and $j^{th}$ nodes.

The basis assumption, that is demonstrated in this work, is that two nodes that have similar neighborhoods have similar spectral wavelet coefficients. They treat the wavelet coefficients as a probability distribution and characterize it via empirical caracteristic function. For a probability distribution $X$, the characteristic function is $\phi_X(t) = \mathbb{E}[e^{itX}]$. This function captures information about all the moments of probability distribution $X$ [Lukacs, 1970]. This way, for a node $v$ and a scale $s$, the caracteristic function of the graph wavelets is:

$$\phi_v(t) = \sum_{u=1}^{n} e^{it\Psi_{uv}} \tag{4.3}$$

The embedding they define is the real and imaginary parts of this characteristic function evaluated at $d$ evenly spaced points $t_1, ..., t_d$ concatenated:

$$\chi_v = [Re(\phi_v(t)), Im(\phi_v(t))]_{t_1,...,t_d} \tag{4.4}$$

Moreover, this embedding is evaluated at $J$ different scales $s$ to incorporate more information on the propagation processes of the graphs. The final embedding $\chi_v = [Re(\phi_v^{s_j}(t_i)), Im(\phi_v^{s_j}(t_i))]_{s_j,t_i}$ is of size $2dJ$.

After having computed node embeddings with diffusion wavelets, they apply an agglomerative clustering algorithm to cluster nodes according to their structural role in graphs.

In analogy to GraphWave, in this work we compute node embeddings with diffusion processes in graphs. We build hierarchical embeddings to encode topological nodes' neighborhoods. Moreover, our method allows us to include features available on nodes in node embeddings in order to identifiy roles that represent local topological patterns. We finally cluster nodes in an end-to-end trainable fashion to identify structural roles for each node of graphs in our dataset.

## 4.4 Method

In this Section we introduce the structural aggregation layer (StructAgg). We show how we identify structural classes for nodes in graphs; how those classes are used in order to develop an aggregation layer; and how this layer allows us to compare significant structural patterns in graphs for a supervised classification task.

### 4.4.1 Structural Embedding

**Graph neural networks.** We build our work upon graph neural networks (GNNs). Several architectures of graph neural networks have been proposed by Defferrard et al. [2016], Kipf and Welling [2016], Veličković et al. [2017] or Bruna and Li [2017]. Those graph neural network models are all based on propagation mechanisms of node features that follow a general neural message passing architecture [Ying et al., 2018, Gilmer et al., 2017]:

$$X^{(l+1)} = MP(A, X^{(l)}; W^{(l)}) \tag{4.5}$$

where $X^{(l)} \in \mathbb{R}^{n \times f_l}$ are the node embeddings computed after $l$ steps of the GNN, $X^{(0)} = X$, and $MP$ is the message propagation function, which depends on the adjacency matrix. $W^{(l)}$ is a trainable weight matrix that depends on layer $l$. Let $f_l$ be the dimension of the node vectors after $l$ steps of the GNN, $f_0 = f$.
The aggregation process that we introduce next can be used with any neural message passing algorithm that follows the propagation rule 4.5. In all the following of our work we denote by $MP$ the algorithm. For the experiments, we consider Graph Convolutional Network (GCN) defined by [Kipf and Welling, 2016]. This model is based on an approximation of convolutions on graphs defined by [Defferrard et al., 2016] and that use spectral decompositions of the Laplacian. The popularity of this model comes from its computational efficiency and the state-of-the-art results obtained on benchmark datasets. This layer propagates node features to 1-hop neighbors. Its propagation rule is the following:

$$X^{(l+1)} = MP(A, X^{(l)}; W^{(l)}) = GCN(A, X^{(l)}) = \rho(\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}X^{(l)}W^{(l)}) \tag{4.6}$$

Where $\rho$ is a non-linear function (a $ReLU$ in our case), $\tilde{A} = A + I_n$ is the adjacency matrix with added self-loops and $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$.
This propagation process allows us to obtain a node representation representing its $l$-hop neighborhood after $l$ layers of GCN. We build a hierarchical representation for nodes by concatenating their embeddings after each step of GCN. The final representation $X_{struct\,i}$ of a node $i$ is given by:

$$X_{struct\,i} = \Big\|_{l=1}^{L} X_i^{(l)} \tag{4.7}$$

Where $L$ is the total number of GCN layers applied.

**Identifying structural classes.** Embedding nodes with $MP$ creates embed-

dings that are close for nodes that are structurally equivalent. Some use a handcrafted node embedding based on propagation processes with wavelets in graphs to identify structural clusters based on hierarchical representation of nodes [Donnat et al., 2018]. By analogy, we learn hierarchical node embeddings and an aggregation layer that identifies structural roles for node in graphs. Those structural roles are consistent along graphs of a dataset which allows us to bring interpretability to our graph classification task.
Node features $X_{struct}$ contain the information of their $L$-hop neighborhood decomposed into $L$ vectors each representing their $l$-hop neighborhood for $l$ varying between 1 and $L$. We will show next that nodes that have the same $L$-hop neighborhood are embedded into the same vector.
To identify structural roles, we thus project each node embedding on a matrix $p \in \mathbb{R}^{f_{struct} \times c}$ where $c$ is the number of structural classes and $f_{struct} = \sum_{l=1}^{L} f_l$ is the dimensionality of $X_{struct\,i}$ for each node $i$. We obtain a soft assignment matrix:

$$C = softmax(X_{struct}p) \in \mathbb{R}^{n \times c} \tag{4.8}$$

Where the softmax function is applied in a row-wise fashion. This way, $C_{ij}$ represents the probability that node $i$ belongs to cluster $j$.

**Definition 4.** *Let $i$ and $j$ be two nodes of a graph $G = (V, E, X)$. Let $\mathcal{N}_l(i) = \{i' \in N | d(i, i') \leq l\}$ be the $l$-hop neighborhood of $i$, which means all the nodes that are at distance lower or equal to $l$ of $i$, $d$ being the shortest-path distance. Let $X_{N_l(i)}$ be the feature matrix of the $l$-hop neighborhood of $u$. Let $G_{i,l}$ be the subgraph of $G$ composed of the $l$-hop neighborhood of $i$.*
*We say that $u$ and $v$ are $l$-structurally equivalent if there exists an isomorphism $\psi$ from $\mathcal{N}_l(j)$ to $\mathcal{N}_l(i)$ such that the two following conditions are verified:*

- $G_{i,l} = \psi(G_{j,l})$
- $\forall j' \in \mathcal{N}_l(j), X_{\psi(j')} = X_{j'}$

**Theorem 2.** *Two nodes $i$ and $j$ that are $L$-structurally equivalent have the same final embedding, $X_{struct\,i} = X_{struct\,j}$.*

*Proof.* Let $i$ and $j$ be two nodes of a graph $G = (V, E, X)$ that are $L$-structurally equivalent.
Let $\mathcal{P}(l)$ be the following proposition:
Two nodes that are $l$-structurally equivalent for some $l$, have the same embedding after $l$ steps of GCN.

Let's prove this proposition by induction.
$\mathcal{P}(0)$ is true, two nodes that have the same embedding are 0-structurally equivalent after 0 step of GCN.
Let $\mathcal{P}(l)$ be true and let's prove $\mathcal{P}(l+1)$.
Let $i$ and $j$ be two nodes that are $l$-structurally equivalent. After a step of GCN we have:

$$X^{(l+1)} = GCN(A, X^{(l)})$$

So we have:

$$(\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}X^{(l)})_i = \sum_{i' \in \mathcal{N}(i)} \frac{a_{ii'}}{\sqrt{d_i d_{i'}}} X_{i'}^{(l)}$$

Since $i$ and $j$ are $l$-structurally equivalent, there exists an isomorphism $\psi$ such that:

$$\forall i' \in \mathcal{N}_l(i), \exists j' \in \mathcal{N}_l(j) \text{ such that } X_{i'}^{(l)} = X_{\psi(j')}^{(l)} = X_{j'}^{(l)}$$

$$\Rightarrow \sum_{i' \in \mathcal{N}(i)} \frac{a_{ii'}}{\sqrt{d_i d_{i'}}} X_{i'}^{(l)} = \sum_{j' \in \mathcal{N}(j)} \frac{a_{\psi(j)\psi(j')}}{\sqrt{d_{\psi(j)} d_{\psi(j')}}} X_{\psi(j')}^{(l)} = \sum_{j' \in \mathcal{N}(j)} \frac{a_{jj'}}{\sqrt{d_j d_{j'}}} X_{j'}^{(l)}$$

$$\Rightarrow (\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}X^{(l)})_i = (\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}X^{(l)})_j$$

$$\begin{aligned} \Rightarrow X_i^{(l+1)} &= f(\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}X^{(l)}W^{(l)})_i \\ &= f(\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}X^{(l)}W^{(l)})_j \\ &= X_j^{(l+1)} \end{aligned}$$

So two nodes $i$ and $j$ that are $L$ structurally equivalent, are $l$ structurally equivalent for all $l$ between 0 and $L$ and thus, $X_{struct\,i} = X_{struct\,j}$ because $X_{struct}$ is the concatenation of embeddings after each layer. □

### 4.4.2 Structural Aggregation

After having identified structural classes, we aggregate node embeddings over those classes, as illustrated in figure 4.2. The goal is to obtain an embedding that discriminates graphs that do not belong to the same class and that selects similar patterns in graphs of the same class. Graphs that have similar properties and thus similar node patterns should have nodes with similar roles and similar embeddings.

**Performing a structural aggregation.** The structural aggregation aims at comparing embeddings of nodes that have similar roles in the graph. When computing the distance between two graphs, if those two graphs have the same distribution of structural roles, they will have embeddings that are close. Nodes that are leaves or nodes that are central in the graph should be compared separately. Mathematically, the variance over all nodes of the graph may be high and decomposing nodes per structural roles aims at decreasing the variance per cluster and thus at bringing more information to the final graph embedding.

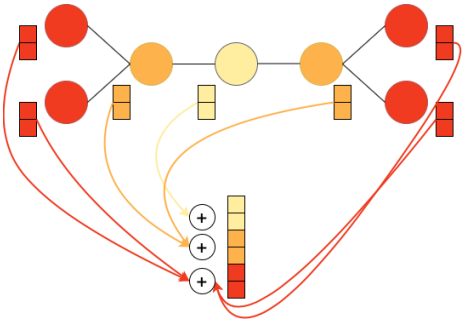


Figure 4.2: Aggregation of node features over structural classes

The structural aggregation layer performs an aggregation per structural role. This aggregation is a mean of embeddings of nodes that belong to the same cluster. The final embedding is a concatenation of the mean embeddings of the nodes per cluster of structural role.

$$Z_{graph} = C^T X^{(L)} \in \mathbb{R}^{c \times f_L}$$

**Proposition 3.** *The embedding $Z_{graph}$ is invariant by node permutation.*

*Proof.* Let $P \in 0, 1^{n \times n}$ be any permutation matrix. Since $P$ is a permutation matrix we have $PP^T = I$. We have $PX^{(l+1)} = GCN(PAP^T, PX^{(l)})$.

$$\begin{aligned} Z_{graph} &= C^T X^{(L)} \\ &= (softmax(X_{struct}p))^T X^{(L)} \\ &= (softmax(P^T P X_{struct}p))^T P^T P X^{(L)} \end{aligned}$$

Since the softmax is applied in a row-wise fashion, we have:

$$\begin{aligned} Z_{graph} &= (P^T softmax(P X_{struct}p))^T P^T P X^{(L)} \\ &= (softmax(P X_{struct}p))^T P X^{(L)} \\ &= Z_{Pgraph} \end{aligned}$$

Where $Z_{Pgraph}$ is the embedding of our graph to which we have applied the permutation $P$ □

Moreover when performing soft classification, the output cluster assignment for each node should generally be close to a one-hot vector so that each node has a clear structural role identified. We therefore regularize the entropy of the cluster assigmnent matrix by minimizing $L_{reg} = \frac{1}{n}\sum_{i=1}^{n} H(C_{i.})$ where $H(C_{i.})$ is the entropy of the assignment vector $C_{i.}$ of node $i$. This is often done when identifying classes for nodes. The same regularization is applied in [Ying et al., 2018] to identify communities in graphs to develop a pooling layer.

**Remarks.**

- The structural classes identified by our algorithm are local. They contain the information of the $L$-hop neighborhood of each node.
- Compared to the structural embedding presented in Donnat et al. [2018], our algorithm needs multiple graphs and is trained in a supervised fashion. Donnat et al. [2018] introduced a node embedding that allows us to identify structural classes in a single graph. It is not straightforward how to generalize this procedure to the case of a set of graphs. Indeed, when dealing with a single graph, this procedure is very efficient at identifying structural roles. However, those structural roles are defined per graph and thus two nodes that have the same structural role but that lie in two different graphs can have embeddings that differ and can thus be classified in two different classes.

## 4.5 Experiments

### 4.5.1 Graph Classification

**Datasets:** We choose a wide variety of benchmark datasets for graph classification to evaluate our model. The datasets can be separated in two types. 2 bioinformatics datasets: PROTEINS and D&D; and a social network dataset: COLLAB. In the bioinformatics datasets, graphs represent chemical compounds. Nodes are atoms and edges represent connections between two atoms. D&D and PROTEINS contain two classes of molecules that represent the fact that a molecule can be either active or inactive against a certain type of cancer. The aim is to classify molecules according to their anti-cancer activity. COLLAB is composed of ego-networks. Graphs' labels are the nature of the entity from which we have generated the ego-network. In Table 4.1 we report some information on these datasets such as the maximum number of nodes in graphs, the average number of nodes per graph, the number of graphs, the size of node features $f$ (if available) and the number of classes. More details can be found in [Yanardag and Vishwanathan, 2015]. We also use the new database Open Graph Benchmark

(OBG) to test our method on larger datasets [Hu et al., 2020]. We use the ogb-molhiv dataset whose properties are listed bellow. Each graph represents a molecule, where nodes are atoms, and edges are chemical bonds. Input node features are 9-dimensional, containing atomic number and chirality, as well as other additional atom features such as formal charge and whether the atom is in the ring or not.

**Experimental setup:** We perform a 10-fold cross validation split which gives 10 sets of train, validation and test data indices in the ratio 8:1:1. We use stratified sampling to ensure that the class distribution remains the same across splits. We fine tune hyperparameters $n_{roles}$ the number of structural roles, $lr$ the learning rate and finally the dimensions of the successive layers respectively chosen from the sets $\{2, 5, 10, 20\}$, $\{0.01, 0.001, 0.0001\}$, $\{8, 16, 32, 64, 128, 256\}$. We fix $l_1 = 1$ and $l_2 = 2$. The sets from which our hyperparameters are selected vary according to the sizes of graphs in each dataset. We do not set a maximum number of epochs but we perform early stopping to stop the training which means that we stop the training when the validation loss has not improved for 20 epochs. We report the mean accuracy and the standard deviation over the 10 folds on the validation set. We compare our method with kernel methods and with a graph neural network that uses pooling layers [Ying et al., 2018]. We should note that kernel methods do not use node features that are available on bioinformatics datasets. For COLLAB, we don't have any features available on nodes. We compute the one-hot encoding of node degrees that we use as node features for our algorithm.

**Results:** From the results of Table 4.1 we can observe that StructAgg is competing with state-of-the-art methods. Indeed, on most datasets, StructAgg has a score very close to those obtained by Ying et al. [2018] and Gao and Ji [2019]. From Table 4.1, StructAgg outperforms all algorithms on COLLAB. Moreover, StructAgg allows us to improve classification results from GCN on ogb-molhiv dataset as illustrated in figure 4.2.

| **Name** | **#Graphs** | **#Node per Graph** | **#Edges per Graph** | **GCN** | **strcutAgg** |
|---|---|---|---|---|---|
| ogbg-molhiv | 41127 | 25.5 | 27.5 | $0.7606 \pm 0.0097$ | $0.7701 \pm 0.0102$ |

Table 4.2: OGB dataset for graph classification. The score reported is the ROC-AUC score.

| **Dataset** | D&D | PROTEINS | COLLAB |
|---|---|---|---|
| Max #Nodes | 5748 | 620 | 492 |
| Avg #Nodes | 284.32 | 39.06 | 74.49 |
| #Graphs | 1178 | 1113 | 5000 |
| $f$ | 89 | 3 | - |
| classes | 2 | 2 | 3 |
| Graphlet [Shervashidze et al., 2009] | 74.85 | 72.91 | 64.66 |
| Shortest-Path [Borgwardt and Kriegel, 2005] | 78.86 | 76.43 | 59.10 |
| 1-WL [Shervashidze et al., 2011] | 74.02 | 73.76 | 78.61 |
| WL-OA [Kriege et al., 2016] | 79.04 | 75.26 | 80.74 |
| GraphSage [Hamilton et al., 2017] | 75.42 | 70.48 | 68.25 |
| DGCNN [Zhang et al., 2018a] | 79.37 | 76.26 | 73.76 |
| DIFFPOOL [Ying et al., 2018] | 80.64 | 76.25 | 75.48 |
| g-U-Nets [Gao and Ji, 2019] | **82.43** | **77.68** | 77.56 |
| **StructAgg** | 78.42 ± 0.97 | 76.72 ± 2.53 | **80.26 ± 2.73** |

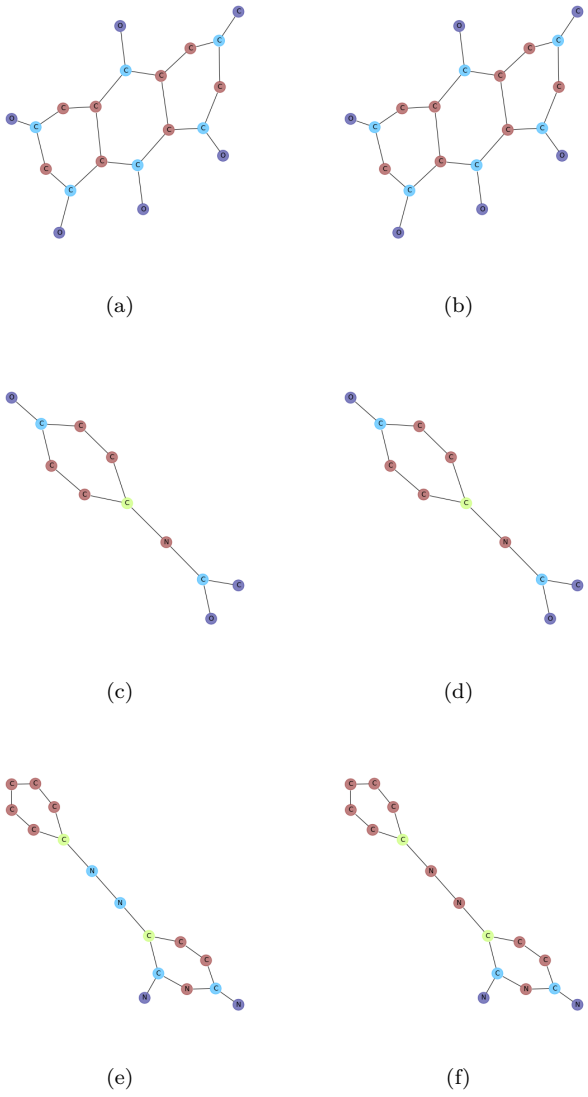
Table 4.1: Classification accuracy on bioinformatics datasets

### 4.5.2 Structural Role

We show some examples of molecules of a molecular dataset and the roles identified by our algorithms. From figure 4.3 we can see that there is some consistence in the assignment of nodes to structural roles. Moreover, a great advantage of our method is that it takes into account node features that are in our case a one-hot encoding of the atom type. Compared to Donnat et al. [2018] whose embeddings do not include node features, our method identifies roles in a macroscopic fashion by identifying similar roles in the whole dataset and not in a single graph. Most methods rely on computed features that correspond to a single graph and do not generalize to multiple graphs because of scales issues. It is the case for GraphWave for which features computed in order to cluster nodes per roles depend on the size of the graph. To compare our role assignment with GraphWave, we computed assignments per graph and not along the whole dataset. We can note that compared to GraphWave, our algorithm uses node features to select structal roles.

### 4.5.3 Pattern Identification

Identifying roles in graphs boils down to identifying significant patterns. As shown in subsection 4.5.2, the roles that we were able to identify represent



(a) (b)

(c) (d)

(e) (f)

Figure 4.3: Molecules drawn for a bioinformatics dataset. The colors represent structural classes identified by our algorithm in figures 4.3a, 4.3c and 4.3e compared to the output of GraphWave in figures 4.3b, 4.3d and 4.3f. To compare the structural classes output by the two algorithms, 4.3a and 4.3b are the same molecule, 4.3c and 4.3d also and 4.3e and 4.3f are the same molecule. To obtain comparable results, we run our algorithm and output the structural classes for each molecule. For each molecule, we then run GraphWave with the number of classes from StructAgg being set as input of GraphWave.

coherent patterns in graphs. Graphs' final embeddings are made of two parts. Nodes' embeddings that are the result of successive GCNs and the allocation of

all nodes to different structural classes. We would like to quantify how much information is contained in the structural roles and if the decomposition of the graph in $c$ classes improves our classification accuracy. To this end, from our trained algorithm, we compute all allocation matrices of all graphs of a dataset. In order to validate the fact that roles themselves have a high discriminative power, we need to identify combinations of roles that are specific to different classes. We thus want to identify the structural roles that compose each graph. Let's consider a graph $G$ and an allocation matrix $C \in \mathbb{R}^{n \times c}$. We obtain a soft assignment matrix. We compute on it a histogram per class over each node of the graph. The feature vector is now a concatenation of all histograms per class. We have:

$$Feat_G = \Big|\Big|_{j=1}^{c} hist(C_{.j}, bins) \in \mathbb{R}^{bins*c}$$

Where $bins$ is the number of bins for the histogram. The histogram upper and lower bounds are 0 and 1 because the values of $C$ are the output of a softmax function.

Let $(G_1, ..., G_d)$ be $d$ graphs of labels $(y_1, ..., y_d)$. After having computed all graphs' histograms we obtain $d$ vectors $(Feat_{G_1}, ..., Feat_{G_d})$ that we use as imputs of a classifier. We use a SVM and we display the results in table 4.3 in order to compare the information that comes from the strutural roles identification and the information that comes from node embeddings.

We can see that the accuracy is lower when we don't consider node embeddings which is consistent with the fact that a lot of information is contained in the embeddings of nodes that are the outputs of GCNs. But the accuracy is significantly higher than a random model which proves that we can identify some patterns in the distribution of structural roles among graphs and that those patterns are a good first approximation to separate classes in a dataset.

| **Dataset** | D&D | PROTEINS | COLLAB |
|---|---|---|---|
| **StructAgg** | 78.42 ± 0.97 | 76.72 ± 2.53 | 80.26 ± 2.73 |
| **StructHist** | 74.54 ± 2.87 | 73.68 ± 2.03 | 70.94 ± 2.08 |

Table 4.3: Classification accuracy based on the histogram of the assignment matrix (StructHist) compared to our algorithm (StructAgg).

## 4.6 Discussion

In this work we developped an aggregation process based on node structural role identification. In this section we will discuss the similarity of those roles with

those output by GraphWave [Donnat et al., 2018] and presented in section 4.2.

Our algorithm is designed for a task of graph classification. We embed each node with a message passing algorithm. Node embeddings include the information of their $L$-hop neighborhood after $L$ layers of GCN (we use a GCN for experiments. However, as said previously, we could use any algorithm following the message passing rule 4.5). We identify structural roles, which are topological patterns in graphs, by clustering each node into a predefined number of roles. This is done in a trainable fashion by learning an assignment matrix for nodes in graphs. We feed several graphs to our algorithm and solve a graph classification task, the structural roles are useful to improve the classification accuracy and to possibly interpret the patterns selected during the classification process. It is thus not obvious how to modify it in order to embed a single graph's nodes. Indeed, we could use a U-Net architecture in order to tackle problematics on a single graph such as node classification or link prediction. However, since structural role identification is made during the aggregation phase, it is not obvious how to generalize it.

In the opposite direction, Donnat et al. [2018] define a node embedding that is designed for a single graph. Roles identifed are consistent for a unique graph and it is not obvious in this case how to define a node embedding that would be consistent for different graphs and among them. This can be done by creating a graph which is the concatenation of all graphs of our dataset. Applying GraphWave would boil down to embedding all nodes of all graphs at the same time. Identifying strucutral roles by clustering node embeddings would produce structural roles that are consistent among all graphs. However, there are some limitations to this methodology. Node embedding are scaled regarding the size of the graph. This means that nodes that have the same role in graphs but that belong to graphs of different sizes have embedding that differ with GraphWave. This is thus not enough to identify roles that are consistent among graphs with GraphWave.
To compare structural role embedding with GraphWave in Section 4.5.2, we simply embedded single graphs with GraphWave and compared the results with our structural role identification. To set the number of clusters as input, we used the minimum between the number of clusters that we set as input of our algorithm and the number of roles identified for the graph with StructAgg. This isn't optimal for GraphWave because it constraints the choice for the number of structural roles.

Finally, StructAgg allows us to use features available on nodes to classify nodes into structural roles. When dealing with grah structured data, we often have

additional information on nodes and/or on edges. This information modeled by feature vectors is to be taken into account when trying to identify structural roles. Indeed in the present case, we are in a graph classification setting and we want to classify molecules. We have node feature available that represent the type of atom each node is in datasets DD and PROTEINS. For dataset *ogbg-molhiv*, we also have chemical features on nodes such as the atomic number of the chirality. These features characterize each node and play a great deal in the identification of structural roles. Indeed, two nodes that have the same topological connections but different atom types may play different roles in the graph. One other advantage of our method compared to graphWave is to take into account node features in node embeddings in order to be more accurate in the classification of nodes into structural roles.

# Chapter 5

# Graph Pooling by Edge Cut

In this Chapter we introduce a novel pooling layer for graphs. In analogy with Convolutional Neural Networks on images for which pooling plays a great role, we want to design a pooling layer to learn hierarchical features in graphs. We design an algorithm trainable in an end-to-end fashion. The work presented in this Chapter is currently under review at ICLR 2021.

## 5.1 Introduction

Convolution neural networks [LeCun et al., 1995] have been proven to be very efficient at learning meaningful patterns for many articificial intelligence tasks. They convey the ability to learn hierarchical informations in data with Euclidean grid-like structures such as images and text. Convolutional Neural Networks (CNNs) have rapidly become state-of-the art methods in the fields of computer vision [Russakovsky et al., 2015] and natural language processing [Devlin et al., 2018].

However in many scientific fields, studied data have an underlying graph or manifold structure such as communication networks (whether social or technical) or knowledge graphs. Recently there has been many attempts to extend convolution to those non-Euclidean structured data [Hammond et al., 2011, Kipf and Welling, 2016, Defferrard et al., 2016]. In these new approaches, the authors propose to compute node embeddings in a semi-supervised fashion in order to perform node classification. Those node embeddings can also be used for link prediction by computing distances between each node of the graph [Hammond et al., 2011, Kipf and Welling, 2016].

Images can be seen as a special case of graph that lie on a 2D grid and where nodes are pixels and edges are weighted according to the difference of intensity and set between pairs of pixels that are close enough [Zhang et al., 2015, Achanta

and Susstrunk, 2017, Van den Bergh et al., 2012, Stutz et al., 2018]. In the emerging field of graph analysis based on convolutions and deep neural networks models, it is appealing to try to apply models that worked best in the field of computer vision to graphs. In this effort, several ways to perform convolutions in graphs have been proposed [Hammond et al., 2011, Kipf and Welling, 2016, Defferrard et al., 2016, Gilmer et al., 2017, Veličković et al., 2017, Xu et al., 2018, Battaglia et al., 2016, Kearnes et al., 2016]. Moreover, when dealing with image classification, an important step is pooling [Gao and Ji, 2019, Ying et al., 2018, Defferrard et al., 2016, Diehl, 2019]. It allows us to extract hierarchical features in images in order to make the classification more accuracte. While it is easy to extract a coarsened image from an input image, it isn't obvious how to coarsen a graph since nodes are not ordered like pixels in an image. In this work we present a novel pooling layer based on edge scoring and related to the minCUT problem.
The main contributions of this work are summarized bellow:

1. **Learned pooling layer.** A differentiable pooling layer that learns how to aggregate node in clusers to produce a pooled graph of reduced size.

2. **Learned edge scores.** During the pooling layer, we learn edge scores. Scores are obtained by a function applied to all nodes couples that are linked by an edge. This score outputs the topological proximity of two nodes.

3. **Approximation of the problem of minCUT.** We develop a pooling layer based on edge cuts. We regularize our problem by a term that corresponds to the problem of Ncut in order to learn edge scores and clusters that are consistent with the topology of the graph.

4. **Experimental results.** Our method achieves state-of-the-art results on benchmark datasets. We compare it with kernel methods and state-of-the-art message passing algorithms that use pooling layers as aggregation processes.

## 5.2 Related Work

Recently there has been a rich line of research, inspired by deep models in images, that aims at redefining neural networks in graphs and in particular convolutional neural networks [Defferrard et al., 2016, Kipf and Welling, 2016, Veličković et al., 2017, Hamilton et al., 2017, Bronstein et al., 2017, Bruna et al., 2013, Scarselli et al., 2009]. Those convolutions can be viewed as message passing algorithms that are composed of two phases Gilmer et al. [2017]. A message passing phase

that runs for T steps and is defined in terms of message functions and vertex update functions. A readout phase that computes a feature vector for the whole graph using some readout function. In this work we will see how to define a readout phase that is learnable and that is representative of meaningful patterns in graphs.
In the field of graph neural networks, along the generalization of convolutions and other deep neural models to graphs, the scientific community has proposed several approaches to perform pooling in graphs. We can distinguish four types of coarsening algorithms:

- **Top-k**: Like Gao and Ji [2019], the objective is to score nodes according to their importance in graphs and then to keep only the nodes with the top-k scores. Because this approach can produce sparse coarsen graphs, a step of edge by adding edges at 2hops is necessary.

- **Cluster identification**: This is usually done by projecting nodes features on a matrix to obtain an assignment matrix. Nodes that have close embeddings are projected on the same cluster. After having obtained the assignment, super nodes at the coarsened level can be computed by aggregating all nodes that belong to the same cluster [Ying et al., 2018].

- **Edge based pooling**: An edge contraction pooling layer has recently been proposed by Diehl [2019]. They compute edge scores in order to perform contracting, which means that they merge two by two nodes that are linked by the edges of the highest scores.

- **Deterministic coarsening strategies**: Finally, a way to perform pooling in graphs can simply be to apply a deterministic clustering algorithm in order to identify cluste s of nodes that will represent nodes in the coarsened level [Defferrard et al., 2016, Ying et al., 2018]. The main drawback of it is that the strategy isn't learned and thus may not be the best suited for the graph classification task.

In this work we define a new pooling layer that is based on edge cuts. Like Diehl [2019] we focus our pooling method on edges instead of nodes. In their work, Diehl [2019] calculate scores on edges to perform contraction pooling. This means that at each pooling step, they merge pairs of nodes that are associated with the highest edge scores, without merging nodes that were already involved in a contracted edge. This methods results in pooled graphs of size divided by 2 compared to the previous graph.
There are several differences with the pooling layer that we propose in this work. We want our pooling layer not to be constrained by a number of communities

or by a predefined size of pooled graph. Moreover, our pooling layer works by edge cuts and the goal is to remove edges that minimize the minCUT problem. Once edges cut, the graph is no longer connected and is composed of several connected components. These connected components correspond to nodes in the coarsened level. In this work, we will first introduce the pooling architecture based on edge scoring in section 5.4.1. We will then relate this pooling layer to the minCUT problem in section 5.4.2. We will finally compare this pooling layer to state-of-the-art methods on benchmark dataset on a graph classification task and a node classification task in section 5.5.

## 5.3 Pooling layers

In this section we present related work on learnable pooling layers. Moreover, we relate our edge scoring function and our pooling layer to other recent works on graph neural networks.

### 5.3.1 Top k Pooling [Gao and Ji, 2019]

We first introduce the top-$k$ pooling layer and more specifically the layer proposed by Gao and Ji [2019]. Since there is no locality information among nodes in graphs, it is impossible to apply classical pooling strategies that work on grid-like structured data. The key idea proposed in their work is to score nodes in order to obtain an order for nodes in graphs and to keep the $k$ most important ones.

To this end, they employ a trainable projection vector $\mathbf{p}$. By projecting all node features to 1D, they can perform $k$ max-pooling for node selection [Kalchbrenner et al., 2014]. Given a node $i$ with its feature vector $x_i$, the scalar projection of $x_i$ on $p$ is $y_i = x_i\mathbf{p}/||\mathbf{p}||$. Here, $y_i$ measures how much information of node $i$ can be retained when projected onto the direction of $\mathbf{p}$. By sampling nodes, they wish to preserve as much information as possible. They thus decide to keep the nodes with the highest projection values on $\mathbf{p}$. Let $A^{(l)}$ be the adjacency matrix of the coarsened graph after layer $l$, $X^{(l)}$ be the feature matrix and $p^{(l)}$ be the projection vector. The layer-wise propagation rule of the graph pooling layer $l$ is

$$
\begin{aligned}
\mathbf{y} &= X^{(l)}\mathbf{p}^{(l)}/||\mathbf{p}^{(l)}|| \\
idx &= rank(\mathbf{y}, k) \\
\tilde{\mathbf{y}} &= sigmoid(\mathbf{y}(idx)) \\
\tilde{X}^{(l)} &= X^{(l)}(idx, :) \\
A^{(l+1)} &= A^{(l)}(idx, idx) \\
X^{(l+1)} &= \tilde{X}^{(l)} \odot (\tilde{y}\mathbf{1}^T)
\end{aligned}
$$

where $k$ is the number of nodes selected in the new graph. $rank(\mathbf{y}, k)$ is the operation of node ranking, which returns indices of the $k$-largest values in $\mathbf{y}$. The $idx$ returned by $rank(\mathbf{y}, k)$ contains the indices of nodes selected for the new graph. $A^{(l)}(idx, idx)$ and $X^{(l)}(idx, :)$ perform the row and/or column extraction to form the adjacency matrix and the feature matrix for the new graph. $\mathbf{y}(idx)$ extracts values in $\mathbf{y}$ with indices $idx$ followed by a sigmoid operation. $\mathbf{1}$ is a vector with all components being 1, and $\odot$ represents the element-wise matrix multiplication.

**Remarks.**

- The pooling algorithm is illustrated in figure 5.1. The element-wise multiplication between the components of $\tilde{X}$ and $\tilde{\mathbf{y}}$ makes the node selection trainable by back-propagation.
- The node deletion implies that we can remove important connections and more importantly, the coarsened graph can be disconnected. A way to counter this effect can be to recreate lost connections in the graph by adding edges that were two hops edges in the original graph.

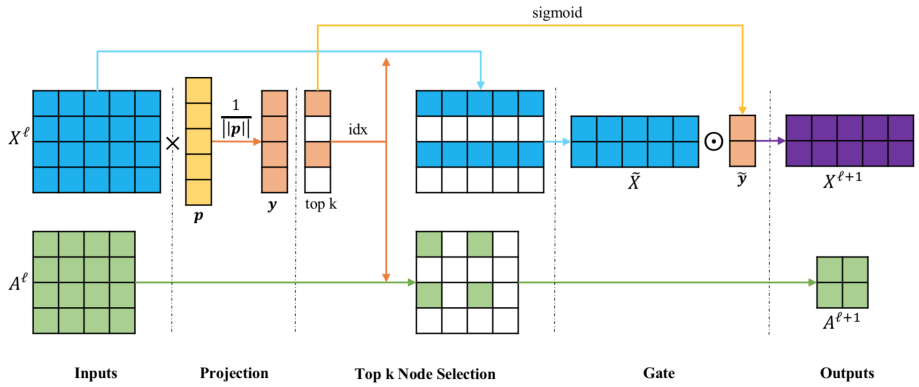



Figure 5.1: gPool

### 5.3.2 Cluster Assignment [Ying et al., 2018]

The second pooling strategy is to obtain a cluster assignment matrix from nodes embeddings. In their work, Ying et al. [2018]. They develop an end-to-end learnable algorithm based on successive convolutions and pooling layers. They use GNNs to embed nodes and to compute the assignment matrices.

**Learning the assignment matrix.** They use generic GNNs to embed nodes. The output node embedding after layer $l$ is

$$Z^{(l)} = GNN_{l,embed}(A^{(l)}, X^{(l)})$$

Where $GNN_{l,embed}$ denotes the GNN model at layer $l$ to embed nodes, in contrast with $GNN_{l,pool}$ that is the GNN model used at layer $l$ to compute the assignment matrix. In parallel of computing node embeddings, the node assignment matrix $S^{(l)}$ to clusters is defined by node embeddings $X^{(l)}$:

$$S^{(l)} = softmax(GNN_{l,pool}(A^{(l)}, X^{(l)})) \in \mathbb{R}^{n_l \times n_{l+1}}$$

where the $softmax$ is applied in a row-wise fashion and $n_l$ and $n_{l+1}$ denote the sizes of the input graph of layer $l$ and of the coarsened graph. The output dimension of $GNN_{l,pool}$ is a hyperparameter and corresponds to a predefined maximum number of cluster in layer $l$.

**Pooling with an assignment matrix.** After having computed the assignment matrix $S^{(l)}$ and node embeddings $Z^{(l)}$ at layer $l$, they define the coarsening strategy. The coarsened adjacency matrix $A^{(l+1)}$ and the new matrix of embeddings $X^{(l+1)}$ are defined as follow

$$X^{(l+1)} = {S^{(l)}}^T Z^{(l)}$$
$$A^{(l+1)} = {S^{(l)}}^T A^{(l)} S^{(l)}$$

### 5.3.3 Edge based pooling [Diehl, 2019]

The two previous pooling strategies focused on nodes whether by deleting the less imformative nodes or by calculating an assignment matrix in order to regroup nodes by clusters. A third way to perform pooling is by an agglomerative way. In their work, Diehl [2019] compute edge scores and perform contraction. They identify pairs of nodes that are linked by edges with the highest scores and merge them into a single node. They only merge nodes once so that the size of the coarsened graph is approximately half of the size of the original graph.

**Edge score.** From node embeddings $X_i$ and $X_j$ for nodes $i$ and $j$, the edge raw score is

$$r_{ij} = W(X_i||X_j) + b$$

where $W$ and $b$ are learnable parameters and $(X_i||X_j)$ denotes the concatenation of vector $X_i$ and $X_j$. The final score $s_{ij}$ of edge $(i,j) \in E$ is

$$s_{ij} = 0.5 + softmax_{r_{\cdot j}}(r_{ij})$$

The 0.5 term is here to rescale to score so that the mean lies in the neighborhood of 1 for better gradient flow and better performance in the classification.

**Edge contraction.** Given the edge score, they then iteratively select edges ranked by scores and contract nodes two by two. This roughly pools half of the total nodes. New features $\hat{X}$ are then computed by summing node features and by rescaling it according to the edge score.

$$\hat{X}_{ij} = s_{ij}(X_i + X_j)$$

Moreover, we can add edge features to compute edge scores by concatenating them with node features during the edge scoring procedure.

**Remark.** This is an agglomerative procedure in which each node starts in its own cluster and at each step we merge two nodes into a same cluster and thus reduce the size of the graph by 1. This procedure can be applied until all nodes are in the same cluster and the remaining graph is composed of a single node. In contrast, the pooling strategy that we will introduce next is a based on a divisive approach (or top-down approach) where the graph is composed of a single cluster that we successively split.

### 5.3.4 Attention Mechanisms [Veličković et al., 2017]

In this section we introduce the Graph Attention Networks (GAT) [Veličković et al., 2017] that closely follows the work of Bahdanau et al. [2014]. The input of this layer is a set of node features $\mathbf{X} = \{X_1, ..., X_n\}$ where $X_i \in \mathbb{R}^f$. The layer produces a new set of features $\mathbf{X'} = \{X_1^{'}, ..., X_n^{'}\}$.
In order to obtain sufficient expressive power to transform the input features into higher-level features, at least one learnable linear transformation is required. To that end, as an initial step, a shared linear transformation, parametrized by a weight matrix, $W \in \mathbb{R}^{f^{'} \times f}$ , is applied to every node. We then perform self-attention on the nodes—a shared attentional mechanism $a : \mathbb{R}^{f^{'}} \times \mathbb{R}^{f^{'}} \mapsto \mathbb{R}$ computes attention coefficients

$$e_{ij} = a(WX_i, WX_j)$$

That indicates the importance of node $j$'s feature to node $i$. In its most general formulation, the model allows every node to attend on every other node, dropping all structural information. We inject the graph structure into the mechanism by performing masked attention—we only compute $e_{ij}$ for nodes $j \in \mathcal{N}_i$, where $\mathcal{N}_i$ is some neighborhood of node $i$ in the graph. To make coefficients easily comparable with one another, they are normalized with a softmax that is applied allong all nodes of a neighborhood

$$\alpha_{ij} = softmax_j(e_{ij}) = \frac{exp(e_{ij})}{\sum\limits_{k \in \mathcal{N}_i} exp(e_{ij})}$$

This mechanism scores edges according to their relative importance to the central node as illustrated in figure 5.2. The weights are then used to rescale the importance of each node feature in the graph neural network formulation. The output feature of node $i$ is then

$$X_i^{'} = \sigma(\sum_{j \in \mathcal{N}_i} \alpha_{ij} W X_j)$$

To stabilize the learning process of self-attention, they have found extending their mechanism to employ multi-head attention to be beneficial, similarly to Vaswani et al. [2017]. Their output feature is thus the concatenation of $K$ independant attention mechanisms

$$X_i^{'} = \Big\Vert_{k=1}^{K} \sigma(\sum_{j \in \mathcal{N}_i} \alpha_{ij}^{(k)} W^{(k)} X_j)$$

Where $\alpha_{ij}^{(k)}$ are the attention coefficients of head $k$ and $W^{(k)}$ is the corresponding input linear transformation's weight matrix.

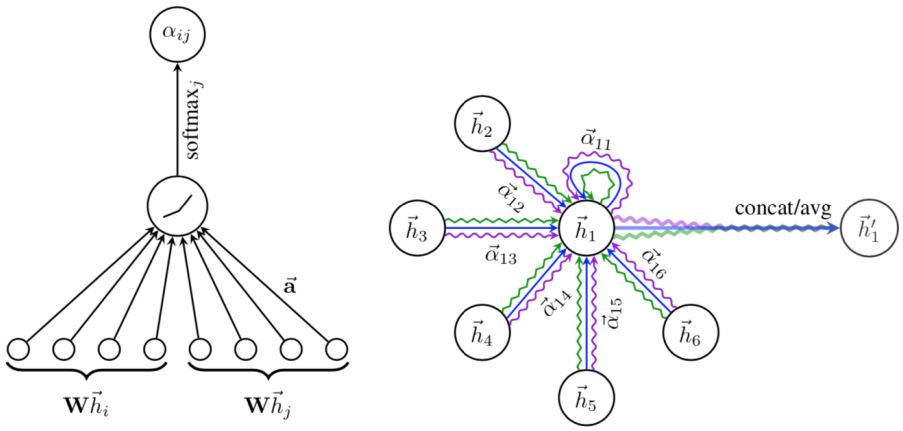



Figure 5.2: Graph attention networks.

## 5.4 Pooling architecture

When designing a pooling layer, most algorithms need a number of classes for the pooling layer that is usually set as a hyperparameter. This is very restrictive especially when working on graphs of different sizes. Indeed, the pooling layer should cluster nodes according to the topology of the graph without being constrained by a number of classes. In this section we present our pooling layer that is based on edge cutting and that does not necessitate any *a priori* on the number of classes that needs to be found.

### 5.4.1 GNNs

Let $G = (V, E, X)$ be a graph composed of a set of nodes $V$, a set of edges $E$ and a node feature matrix $X \in \mathbb{R}^{n \times f_0}$ where $f_0$ is the dimensionality of node features. We denote by $A$ the adjacency matrix.

**Graph neural networks.** We build our work upon graph neural networks (GNNs). There are several architectures of graph neural networks that have been proposed by Defferrard et al. [2016], Kipf and Welling [2016], Veličković et al. [2017] or Bruna and Li [2017]. Those graph neural network models are all based on propagation mechanisms of node features that follow a general neural message passing architecture [Ying et al., 2018, Gilmer et al., 2017]:

$$Z^{(l+1)} = MP(A, Z^{(l)}; W^{(l)}) \tag{5.1}$$

where $Z^{(l)} \in \mathbb{R}^{n \times f_l}$ are node embeddings computed after $l$ steps of $MP$, $Z^{(0)} = X$, and $MP$ is the message propagation function, which depends on the adjacency matrix. $W^{(l)}$ is a trainable weight matrix that depends on layer $l$ and $f_l$ is the dimensionality of node embeddings.

The pooling layer that we introduce next can be used with any neural message passing algorithm that follows the propagation rule 5.1. In all the following of our work we denote by $MP$ the algorithm. For the experiments, we consider the Graph Convolutional Network (GCN) defined by [Kipf and Welling, 2016]. This model is based on an approximation of convolutions on graphs defined by [Defferrard et al., 2016] and that use spectral decompositions of the Laplacian. It is very popular because it is very efficient computationally and obtains state-of-the-art results on benchmark datasets. This layer propagates node features to 1-hop neighbors. Its propagation rule is the following:

$$Z^{(l+1)} = MP(A, Z^{(l)}; W^{(l)}) = GCN(A, Z^{(l)}) = \rho(\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}Z^{(l)}W^{(l)}) \tag{5.2}$$

Where $\rho$ is a non-linear function (a $ReLU$ in our case), $\tilde{A} = A + I_n$ is the adjacency matrix with added self-loops and $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$ is the degree diagonal matrix associated with adjacency matrix $\tilde{A}$.

**Scoring edges.** After layer $l$, each node $i$ in the graph has an embedding $Z_i^{(l)}$. To simplify notations, we consider all matrices to be associated to layer $l$ and we do not keep the exponent $l$. For example, we write feature of node $i$ at layer $l$, $Z_i$ and its dimensionality is denoted by $f$. Based on these embeddings, we develop a scoring function that characterizes the importance of each edge of the graph. The input of our scoring algorithm is a set of node features, $\{Z_1, ..., Z_n\} \in \mathbb{R}^{n \times f}$. The scoring function produces a matrix $S \in \mathbb{R}^{n \times n}$ associated with layer $l$,

$S_{ij} = \mathbf{1}_{(i,j)\in E} * s_{ij}$ where $s_{ij}$ is the score of edge $(i, j)$.
In order to compute the score of each edge of the graph, we apply a shared linear transformation, parametrized by a weight matrix $W_{pool} \in \mathbb{R}^{f\times d}$, to each node of the graph, $d$ being the output size of the linear transformation. We then perform self-attention on nodes, as used in the Graph Attention Network (GAT) [Veličković et al., 2017], by applying a shared weight $a : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ to obtain a score on edge $(i, j) \in E$:

$$s_{ij} = \sigma(a[W_{pool}Z_i||W_{pool}Z_j]) \tag{5.3}$$

Where $\sigma$ is the sigmoid function, $W_{pool}$ and $a$ are trainable matrices associated with layer $l$ and $[W_{pool}Z_i||W_{pool}Z_j] \in \mathbb{R}^{2d}$ is a vector that is the concatenation of $W_{pool}Z_i$ and $W_{pool}Z_j$. Let's note that this scoring function isn't symmetric and depends on the order of nodes. We can symmetrize this function by computing

$$s_{ij} = \frac{1}{2}\left(\sigma(a[W_{pool}Z_i||W_{pool}Z_j]) + \sigma(a[W_{pool}Z_j||W_{pool}Z_i])\right)$$

By applying the sigmoid function to the attention mechanism we compute an importance of edges. The goal is to obtain a distribution on edges for whose nodes that are close topologically have an edge which value is close to 1. In the opposite case, we would like an edge to have a weight close to 0 if it links two nodes that do not lie in the same community. By doing so we would like to solve the minimum cut problem in graphs.

After having computed the edge score matrix, we keep a ratio $r$ of edges that correspond to edges with the $r\%$ higher scores. We obtain a threshold $s_{threshold}$ that corresponds to the $r^{th}$ percentile of the distribution of edge scores. This way, we cut edges which scores are close to 0 in the graph. Edges with the smallest scores represent edges that link nodes that aren't in the same community and thus by cutting those edges, we separate the graph into several clusters. We denote by $S_{cut}$ the score matrix with values under $s_{threshold}$ truncated to 0. Each row is renormalized by the number of positive components. This renormalization is useful in the following to compute node features in the coarsened level.

$$\forall (i, j) \in V^2, S_{cutij} = \frac{1}{\sum\limits_{j\in\mathcal{N}(i)} \mathbf{1}_{s_{ij}\geq s_{threshold}}} s_{ij}\mathbf{1}_{s_{ij}\geq s_{threshold}}$$

We then extract the connected components of the new graph with cut edges. Those connected components represent super nodes in the pooled graph. We obtain a cluster assignment matrix $C \in \mathbb{R}^{n\times c}$, $c$ being a free parameter that isn't fixed and that can vary during the training of the algorithm. After layer $l$, the pooled adjacency matrix and the pooled feature matrix are thus:

$$A^{(l+1)} = A^{(l)}_{pool} = {C^{(l)}}^T A^{(l)} C^{(l)}$$
$$Z^{(l+1)} = Z^{(l)}_{pool} = {C^{(l)}}^T S^{(l)}_{cut} Z^{(l)}$$

**Remark.** The multiplication by $S_{cut}Z$ makes the weights $W_{pool}$ and $a$ trainable by back-propagation. Otherwise it wouldn't be the case because the function that outputs the matrix $C$ by finding connected components from matrix $S_{cut}$ is not differentiable.
Moreover, this multiplication weights the importance of each node feature in the super node of the coarsened level. In order to compute the feature $Z_k$ of cluster (or super node) $k$, we compute a node importance score $s_{cut\,i}$ at layer $l$ for each node $i$ of the graph:

$$s_{cut\,i} = \frac{1}{\sum\limits_{j \in \mathcal{N}(i)} \mathbf{1}_{{s_{cut}}^{(l)}_{ij} > 0}} \sum_{j \in \mathcal{N}(i)} s_{cut\,ij}$$

The feature $Z_k$ of cluster $k$ is then a weighted mean of the features of nodes that belong to cluster $k$:

$$Z_k = \sum_{i \in k} s_{cut\,i} Z_i$$

Moreover, for edge scores to be consistent with the minCUT algorithm, we add a regularization term that we define in the next section.

### 5.4.2 Min cuts

**Graph Cuts.** In community detection, the min cut problem aims at finding the separation, between disjoint subsets, that minimizes the cut. For two disjoint subsets $V_1, V_2 \subset V$ we define

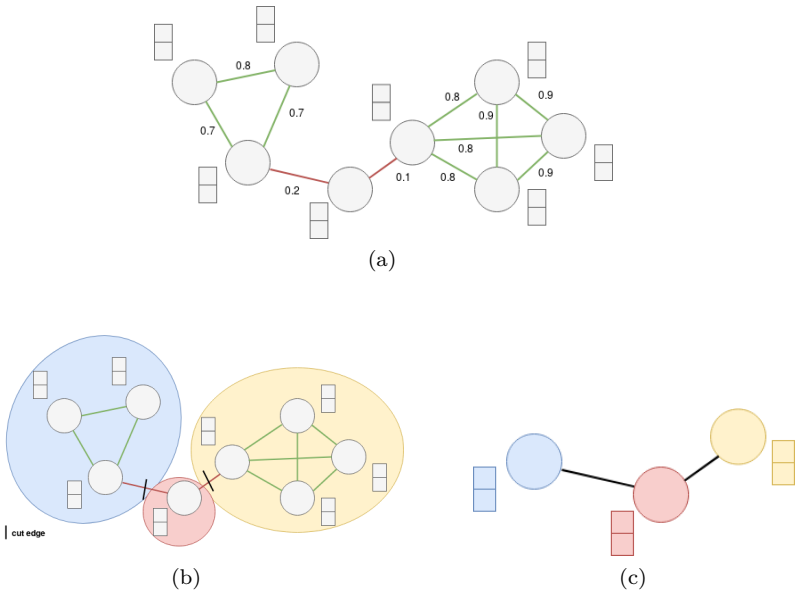
$$cut(V_1, V_2) = \sum_{i \in V_1, j \in V_2} A_{ij}$$

Given a similarity graph with adjacency matrix $A$, the simplest and most direct way to construct a partition is to solve the mincut problem. This consists of choosing the partition $V_1, ..., V_c$ of $V$ which minimizes

$$cut(V_1, ..., V_c) = \sum_{k=1}^{c} cut(V_k, \bar{V_k})$$

When solving this problem, the solution often boils down to separating one indivual vertex from the rest of the graph. In order to circumvent this problem, other objective functions can be defined such as RatioCut [Hagen and Kahng, 1992] or the normalized cut (NCut) [Shi and Malik, 2000]. Those objective functions are defined as follow:

$$RatioCut(V_1, ..., V_c) = \sum_{k=1}^{c} \frac{cut(V_k, \bar{V_k})}{|V_k|}$$




(a) (b) (c)

Figure 5.3: In figure 5.3a, scores are computed and edges with the smallest scores (in red) are cut. After having cut edges, we identify connected components in figure 5.3b (black lines on edges mean that those edges have been removed). We finally merge nodes that belong to the same cluster and reconstruct the edges between clusters as shown in figure 5.3c.

$$Ncut(V_1, ..., V_c) = \sum_{k=1}^{c} \frac{cut(V_k, \bar{V}_k)}{vol(V_k)}$$

Where $|V_k|$ is the number of vertices in $V_k$ and $vol(V_k) = \sum_{i \in V_k} d_i$. These new objectives tend to produce balanced communities, in terms of number of edges or in terms of weights inside the communities.

It can be proved that solving these objectives is equivalent to solving optimization problems derived from spectral decompositions of graphs [Von Luxburg et al., 2008]. An approximation of the Ratio Cut can be obtained by the minimazation of the following problem:

$$\min_{H \in \mathbb{R}^{n \times c}} Tr(H^T L H) \text{ s.t. } H^T H = I \tag{5.4}$$

An approximation of the Normalized Cut can be obtained by minimizing the following problem:

$$\min_{U \in \mathbb{R}^{n \times c}} Tr(U^T D^{-1/2} L D^{-1/2} U) \text{ s.t. } U^T U = I \tag{5.5}$$

In their work, Bianchi et al. [2019] develop a pooling layer that aims at minimizing the minCUT problem. By calculating an assignment matrix $C$ as in [Ying

et al., 2018] based on the projection of node embeddings on a cluster matrix, they are able to minimize problem 5.4 by adding a regularization term that depends on the assignation matrix $C$.

In our work, the assignment matrix $C$ does not include the information of learnable parameters since it is computed by looking at the connected components of the graph after having cut the less informative edges. But the Normalized Cut of the partition of the graph can be computed easily from the edge score matrix $S$. If we compute the matrix $M_{cut} = C^T SC \in \mathbb{R}^{c \times c}$, we obtain the super node matrix in which each diagonal parameter represents the sum of the edge weights inside each cluster (super node). Moreover, $\sum_{j=1}^{c} M_{cut\,ij} 1_{i \neq j}$ represents the sum of the weights between cluster $j$ and the rest of the graph. With those two matrices, we can easily compute $Ncut(V_1, ..., V_c)$ at layer $l$ for a graph $G$. We thus add a regularization term equal to:

$$L_{reg} = Ncut(V_1, ..., V_c) = \sum_{i=1}^{c} \frac{\sum_{j=1}^{c} M_{cut\,ij} 1_{i \neq j}}{M_{cut\,ii}} \tag{5.6}$$

### 5.4.3 U-Nets

U-Net was first developped for biomedical image segmentation [Ronneberger et al., 2015]. U-Net is a convolutional neural network that contains two paths. The first path is the contraction path (also called encoder) which is used to capture the context in the image. The encoder is just a traditional stack of convolutional and pooling layers. The second path is the symmetric expanding path (also called decoder) which is used to enable precise localization using transposed convolutions as illustrated in figure 5.4. Thus it is an end-to-end fully convolutional network.

In order to perform node classification with a pooling architecture we use a Graph U-Net model proposed by Gao and Ji [2019] and inspired by the works of [Ronneberger et al., 2015]. Considering that images can be seen as special cases of graphs that lie on regular 2D lattices, we can have a correspondance between image segmentation and node classification in graphs. By using graph convolutional layers, node information is propagated in order to identify clusters of nodes that are topologically close and that are pooled together during the training of the algorithm.

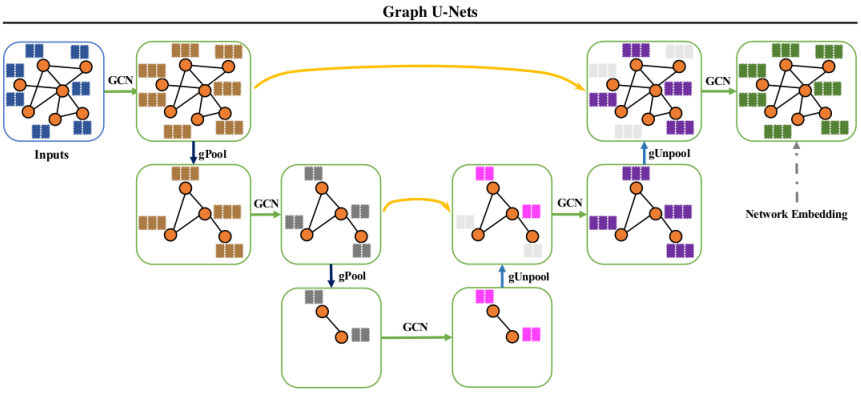



Figure 5.4: Graph U-Nets model. It is composed of successive convolution and pooling layers. After this encoder part, the decoder is composed of convolutions and unpooling layers. Unpooling layers reconstruct the graph based on the pooling layer of the encoder.

## 5.5 Experiments

### 5.5.1 Graph Classification

**Datasets:** We choose a wide variety of benchmark datasets for graph classification to evaluate our model. The datasets can be separated in two types. 2 bioinformatics datasets: PROTEINS and D&D; and a social network dataset: COLLAB. In the bioinformatics datasets, graphs represent chemical compounds. Nodes are atoms and edges represent connections between two atoms. D&D and PROTEINS contain two classes of molecules that represent the fact the a molecule can be either active or inactive against a certain type of cancer. The aim is to classify the molecules according to their anti-cancer activity. COLLAB is composed of ego-networks. Graphs' labels are the nature of the entity from which we have generated the ego-network. More details can be found in [Yanardag and Vishwanathan, 2015].

**Experimental setup:** We perform a 10-fold cross validation split which gives 10 sets of train, validation and test data indices in the ratio 8:1:1. We use stratified sampling to ensure that the class distribution remains the same across splits. We fine tune hyperparameters $f_l$ and $d_l$ the dimensions of features in each layer, $r$ the cut ratio, $lr$ the learning rate respectively chosen from the sets $\{256, 512, 1024\}$, $\{32, 64, 128\}$, $\{10\%, 30\%, 50\%, 70\%, 90\%\}$ and $\{0.01, 0.001\}$. We do not set a maximum number of epochs but we perform early stopping to stop the training which means that we stop the training when the validation loss has not improved for 50 epochs. We report the mean accuracy and the standard

deviation over the 10 folds on the test set. We compare our method with kernel methods and with graph neural networks that use pooling layers [Ying et al., 2018, Gao and Ji, 2019]. We should note that kernels methods do not use node features that are available on bioinformatics datasets.

**Results:** From the results of Table 5.1 we can observe that our pooling layer is challenging with state-of-the-art methods. Indeed, on most datasets, the score of our model is very close to those obtained by Gao and Ji [2019] and our model outperforms Ying et al. [2018] on these datasets. From Table 5.1, EdgeCut competes with all algorithms on COLLAB.

| **Dataset** | D&D | PROTEINS | COLLAB |
|---|---|---|---|
| Max | 5748 | 620 | 492 |
| Avg | 284.32 | 39.06 | 74.49 |
| #Graphs | 1178 | 1113 | 5000 |
| Graphlet | 74.85 | 72.91 | 64.66 |
| Shortest-Path | 78.86 | 76.43 | 59.10 |
| 1-WL | 74.02 | 73.76 | 78.61 |
| WL-OA | 79.04 | 75.26 | 80.74 |
| GraphSage [Hamilton et al., 2017] | 75.42 | 70.48 | 68.25 |
| DGCNN [Zhang et al., 2018a] | 79.37 | 76.26 | 73.76 |
| DIFFPOOL [Ying et al., 2018] | 80.64 | 76.25 | 75.48 |
| g-U-Nets [Gao and Ji, 2019] | **82.43** | **77.68** | **77.56** |
| **EdgeCut** | 80.33 ± 0.05 | 76.42 ± 0.23 | **77.23 ± 0.11** |

Table 5.1: Classification accuracy on bioinformatics datasets

### 5.5.2 Node Classification

**Datasets:** For node classification we conduct our experiments on three real-world datasets, Cora, citeseer and Pubmed. They are three citations networks where nodes are articles that are linked together by an edge if there exists a citation between them. All datasets contain attributes on nodes that are extracted from the title and the abstract. Attributes represent sparse bag-of-word vectors.

| **Dataset** | **Nodes** | **Features** | **Classes** | **Training** | **Validation** | **Testing** | **Degree** |
|---|---|---|---|---|---|---|---|
| Cora | 2708 | 1433 | 7 | 140 | 500 | 1000 | 4 |
| Citeseer | 3327 | 3703 | 6 | 120 | 500 | 1000 | 5 |
| Pubmed | 19717 | 500 | 3 | 60 | 500 | 1000 | 6 |

Table 5.2: Statistics on node classification datasets. The classification is made with 20 nodes per class in the training set.

**Experimental setup:** We split the edges into a training, a test and a validation set according to the splits used by Kipf and Welling [2016]. We fine tune hyperparameters $f_l$ and $d_l$ the dimensions of features in each layer, $r$ the cut ratio, $lr$ the learning rate respectively chosen from the sets $\{256, 512, 1024\}$, $\{32, 64, 128\}$, $\{10\%, 30\%, 50\%, 70\%, 90\%\}$ and $\{0.01, 0.001\}$. We do not set a maximum number of epochs but we perform early stopping to stop the training which means that we stop the training when the validation loss has not improved for 50 epochs. We report the mean accuracy and the standard deviation after several iterations of the algorithm on each set of hyperparameters. We compare our method with graph neural networks that uses pooling layers and with other graph neural networks referenced in table 5.3.
We denote by EdgeCut without regularization the version of our algorithm that isn't regularized by the minCUT term introduced in equation 5.6. We denote by EdgeCut the version regularized. We conduct an ablation study and we show results in table 5.3 to show the effects of the regularization term.
**Architecture for node classification:** In order to perform node classification with a pooling architecture we use a Graph U-Net model proposed by Gao and Ji [2019] and introduced in Section 5.4.3.

| Models | Cora | Citeseer | Pubmed |
|---|---|---|---|
| DeepWalk [Perozzi et al., 2014] | 67.2% | 43.2% | 65.3% |
| Planetoid [Yang et al., 2016] | 75.7% | 64.7% | 77.2% |
| Chebyshev [Defferrard et al., 2016] | 81.2% | 69.8% | 74.4% |
| GCN [Kipf and Welling, 2016] | 81.5% | 70.3% | 79.0% |
| GAT [Veličković et al., 2017] | 83.0 ± 0.7% | 72.5 ± 0.7% | 79.0 ± 0.3% |
| EdgeCut without regulazization | 81.9 ± 0.8% | 69.8 ± 0.7% | 78.7 ± 0.3% |
| EdgeCut | 82.3 ± 0.6% | 70.9 ± 0.5% | 79.1 ± 0.4% |

Table 5.3: Classification accuracy on node classification datasets.

## 5.6 Discussion

In this work we studied a new pooling layer based on edge cuts in graphs. To this end we compute scores on edges in order to quantify the importance of each edge regarding the propagation of information on the graph. By cutting a certain percentage of edges, we can uncover communities of topologically close nodes in order to pool the graph. There are many advantages to this approach:

- By deleting edges, we are not constraint by a hyperparameter that wourld represent the number of nodes of the coarsened level. The number of nodes in the pooled graph only depends on the topology of the graph and may

vary during the training.

- Moreover, scoring and ordering of edges can be controled by a regularization term introduced in Section 5.4.2. The regularization term that aims at minimizing the Ncut problem allows us to controle edge scores.
- Finally, by combining the regularization term and by gating edge scores, we are able to identify less informative edges in order to remove them from the graph to uncover strongly connected components.

Nevertheless, there are also limitation to this approach:

- Even if we don't need to specify a number of nodes to find in the coarsened level, we need to specify a percentage of edges to cut. This hyperparameter is directly linked to the number of nodes of the coarsened level. Indeed, if we set this ratio to 100%, we will remove all edges. All nodes will thus be in their own community and thus the pooling layer will be useless. In the opposite, if we set the ratio of cut edges to 0%, all connected components will be regrouped into one single node and the coarsened graph will be composed of single separated nodes.
- The value of this parameter is thus important even if hardly interpretable. It is difficult to predict the number of nodes that will compose the coarsened graph at the output of the pooling layer.

# Chapter 6

# Hierarchical graph clustering by node pair sampling

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]

## 6.1 Introduction

Many datasets can be represented as graphs, being the graph explicitely embedded in data (e.g., the friendship relation of a social network) or built through some suitable similarity measure between data items (e.g., the number of papers co-authored by two researchers). Such graphs often exhibit a complex, multi-scale community structure where each node is invoved in many groups of nodes, so-called communities, of different sizes.

One of the most popular graph clustering algorithm is known as Louvain in name of the university of its inventors Blondel et al. [2008]. It is based on the greedy maximization of the modularity, a classical objective function introduced in Newman and Girvan [2004]. The Louvain algorithm is fast, memory-efficient,

and provides meaningful clusters in practice. It does not enable an analysis of the graph at different scales, however Fortunato and Barthelemy [2007], Lancichinetti and Fortunato [2011]. The resolution parameter available in the current version of the algorithm [1] is not related to clustering features like the number of clusters and thus 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. 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 6.2. The notation used in the Chapter, the distance between clusters used to aggregate nodes and the clustering algorithm are presented in Sections 6.4 and 6.5. The link with modularity and the Louvain algorithm is explained in Section 6.6 and section 6.7 shows the experimental results.

## 6.2 Related Work

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 present work.

Usual hierarchical clustering techniques apply to vector data Ward [1963], Murtagh and Contreras [2012]. They do not directly apply to graphs, unless the graph is embedded in some metric space, through spectral techniques for instance Von Luxburg [2007], Donetti and Munoz [2004].

A number of hierarchical clustering algorithms have been developped specifically for graphs. A first category consists of *agglomerative* algorithms, that are mainly characterized by some distance between clusters. This distance may be based on the modularity increase Newman [2004], on a random walk in the graph Pons and Latapy [2005], on some notion of structural similarity, involving the neighborhood of each node Huang et al. [2010], or on a correlation measure between clusters Chang et al. [2011]. None of these distances has been proved to be reducible, a key property of hierarchical clustering Murtagh and Contreras

[1] See the `python-louvain` Python package.

[2012].
*Non-agglomerative* algorithms include the divisive approach of Newman and Girvan [2004], based on the notion of edge betweenness, the iterative approach of Sales-Pardo et al. [2007] and Lancichinetti et al. [2009], looking for local maxima of modularity or some fitness function, and other approaches based on statistical interence Clauset et al. [2008], replica correlations Ronhovde and Nussinov [2009] and graph wavelets Tremblay and Borgnat [2014a].
Finally, the agglomerative approach of the Louvain algorithm also provides a hierarchy Blondel et al. [2008]. This hierarchy is of limited practical interest, however, since the modularity optimization is performed at some *fixed* resolution, so that only part of the hiearchy is meaningful. We shall see that our algorithm may be seen as a modified version of Louvain using a *sliding* resolution.

## 6.3 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.

### 6.3.1 Agglomerative algorithms

We are interested in clustering a set of $n$ observations $X = \{x_1, ..., x_n\}$ 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). We assume that we are given a norm $||.||$ 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 approach (or top-down approach), we are typically given a cut function $(A, B) \mapsto cut(A, B)$ that measures the quality of a split between two clusters $A, B \subset X$ as used in chapter 5. A small value of $cut(A, B)$ corresponds to two distant or dissimilar clusters $A$ and $B$, whereas a large value of $cut(A, B)$ 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 $cut(A, B)$. More precisely, this greedy divisive algorithm can be described as follows.

1. (Initialization) $C \leftarrow \{\{1\}, \ldots, \{n\}\}$;
2. While $\mathcal{C}$ contains a cluster with at least two observations.
   - $C \leftarrow \arg\max_{C' \in \mathcal{C}} |C'|$
   - $A, B \leftarrow \arg\min_{a,b \in \mathcal{C}, a \cup b = C} cut(a, b)$
   - $\mathcal{C} \leftarrow \mathcal{C} \cup \{A, B\} \setminus \{C\}$
3. Return $A, B$ and $cut(A, B)$.

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 output 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.

1. (Initialization) $C \leftarrow \{\{1\}, \ldots, \{n\}\}$;
2. For $t = 1, ..., n - 1$
   - $A, B \leftarrow \arg\min_{a,b \in \mathcal{C}, a \neq b} d(a, b)$
   - $\mathcal{C} \leftarrow \mathcal{C} \cup \{C\} \setminus \{A, B\}$
3. Return $A, B$ 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 6.2). 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.

**Remarks.** In terms of pooling layer presented in precedent works, the distinction between the divisive approach and the agglomerative approach can be illustrated by comparing the pooling layer introduced by [Diehl, 2019] and the pooling layer the we introduced in chapter 5. Indeed, in their work, Diehl [2019] calculate edge scores in order to successively merge pairs of nodes. This is an agglomerative approach in which we start with all nodes in a different cluster and clusters are merged until we obtain a single cluster. In our work, we use a divisive approach by successively splitting clusters into smaller ones by cutting edges.

### 6.3.2 Classical 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

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

### 6.3.3 Ward's method

Another classic distance used in the agglomerative approach is Ward's distance [Ward, 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 cluster $C$:

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

We recognize the objective function of the $k$-means problem 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 $J(C) = \sum_{C \in \mathcal{C}} S(C)$, then we have

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

### 6.3.4 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

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

where $\alpha, \beta, \gamma$ and $\delta$ 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

$$d(a \cup b, c) = \frac{(|a|+|c|)d(a,c)+(|b|+|c|)d(b,c)-|c|d(a,b)}{|a|+|b|+|c|}$$

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

## 6.4 Node pair sampling

The weights induce a probability distribution on node pairs,

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

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[2]:

$$d(i,j) = \frac{p(i)p(j)}{p(i,j)}, \tag{6.1}$$

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, $i$).

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

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:

$$d(a, b) = \frac{p(a)p(b)}{p(a, b)}, \tag{6.2}$$

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 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. □

**Proposition 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 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 $c$.

## 6.5 Clustering algorithm

The 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. 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 (see Section 6.6).

**Aggregate graph.** In view of (6.2), 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) $V \leftarrow \{1, \dots, n\}$; $L \leftarrow \emptyset$
2. (Agglomeration) For $t = 1, \dots, n - 1$,
   - $i, j \leftarrow \arg\min_{i', j' \in V, i' \neq j'} d(i', j')$
   - $L \leftarrow L \cup \{\{i, j\}\}$
   - $V \leftarrow V \setminus \{i, j\}$; $V \leftarrow V \cup \{n + t\}$
   - $p(n + t) \leftarrow p(i) + p(j)$
   - $p(n + t, u) \leftarrow p(i, u) + p(j, u)$ 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 Murtagh and Contreras [2012]. Starting from an arbitrary node, a chain a nearest neighbors is formed. Whenever two nodes of the chain are mutual nearest neighbors, these two nodes are merged and the chain is updated recursively, until the initial node is eventually merged. 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).

## 6.6 Link with modularity

The modularity is a standard metric to assess the quality of a clustering $C$ (any partition of $V$). Let $\delta_C(i, j) = 1$ if $i, j$ are in the same cluster under clustering

$C$, and $\delta_C(i,j) = 0$ otherwise. The modularity of clustering $C$ is defined by Newman and Girvan [2004]:

$$Q(C) = \frac{1}{w} \sum_{i,j \in V} (A_{ij} - \frac{w_i w_j}{w}) \delta_C(i,j), \tag{6.3}$$

which 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,

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

It is clear from (6.3) that any clustering $C$ maximizing modularity has some resolution limit, as pointed out in Fortunato and Barthelemy [2007], because the second term is normalized by the total weight $w$ and thus becomes negligible for too small clusters. To go beyond this resolution limit, it is necessary to introduce a multiplicative factor $\gamma$, called the resolution. The modularity becomes:

$$Q_\gamma(C) = \sum_{i,j \in V} (p(i,j) - \gamma p(i)p(j)) \delta_C(i,j), \tag{6.4}$$

or equivalently,

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

This resolution parameter can be interpreted 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 consists, for any *fixed* resolution parameter $\gamma$, of 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 $C$

The result of step 2 depends on the order in which nodes and clusters are considered; typically, nodes are considered in a cyclic way and the target cluster of each node is that maximizing the modularity increase.
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 $\gamma$, say $\gamma_1$, that triggers a single merge between two nodes, resulting in clustering $C_1$. In view of (6.4), 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 $\gamma_2$, 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, clustering $C_t$ (after $t$ merges) is not the clustering produced by the Louvain algorithm at resolution $\gamma_t$. We shall see that the number of clusters are approximately the same, however, so that $\gamma_t$ can also be used as the resolution parameter of the Louvain algorithm to get approximately $n - t$ clusters.

## 6.7 Experiments

Our hierarchical clustering algorithm, called Paris[3], is available as a Python package[4]. 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. The jupyter notebooks and datasets used in the experiments are available online[5] so that the reader can reproduce the experiments presented here and test other parameters.

[3]Paris = Pairwise node Agglomeration with Resolution Incremental Sliding.
[4]See https://github.com/Charpenb/pyparis
[5]See https://perso.telecom-paristech.fr/bonald/mlg.zip

**Synthetic data.** We start with a simple hierarchical stochastic block model, as described in Lyzinski et al. [2017]. There are $K = 2$ levels, with 16 blocks of 10 nodes at level 1 and 4 blocks of 4 level-1 blocks at level 2, forming a total of $n = 160$ nodes (see Figure 6.1).

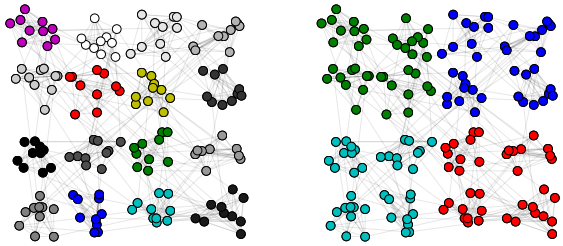


Figure 6.1: A hierachical stochastic block model.

The connectivity parameters are $\mu = 2, \alpha_1 = 0.02, \alpha_2 = 0.01$. The output of Paris algorithm is shown in Figure 6.2 as a dendrogram where the distances (on the $y$-axis) are in log-scale. The two levels of hierarchy clearly appear.

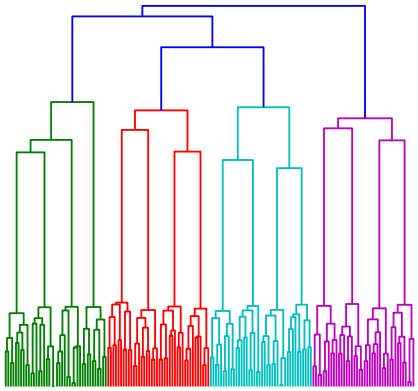


Figure 6.2: Dendrogram associated with the clustering of Paris on a hierrachical stochastic block model of 16 blocks.

Finally, we show in Figure 6.3 the number of clusters with respect to the resolution parameter $\gamma$ for Paris (top) and Louvain (bottom). 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 $\gamma$ (here 100 values ranging from 0.01 to 20), Paris is run only once, the relevant resolutions being direct outputs of the algorithm.

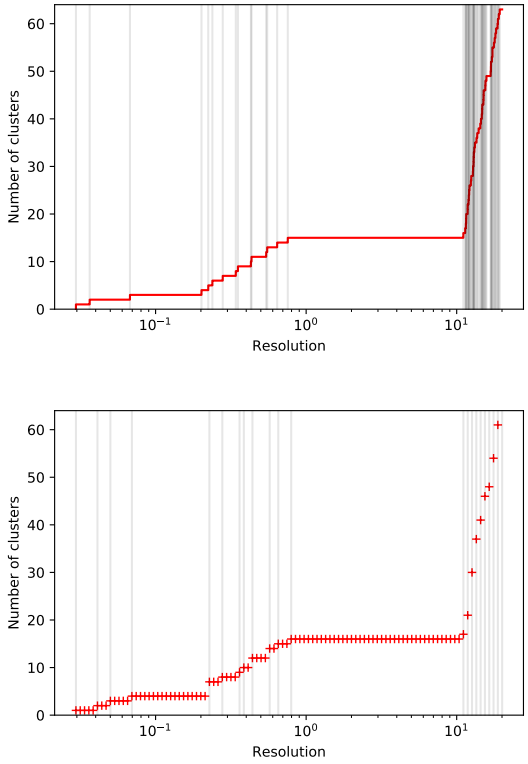



Figure 6.3: Number of clusters with respect to the resolution parameter $\gamma$ for Paris (top) and Louvain (bottom) on the hierachical stochastic block model of Figure 6.1.

**Real data.** We consider four real datasets, whose characteristics are summarized in Table 6.1.

The first dataset, extracted from OpenStreetMap[6], 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 6.4. 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 the Rive Droite cluster into 3

[6]https://openstreetmap.org

| Graph | # nodes | # edges |
|---|---|---|
| OpenStreet | 5,993 | 6,957 |
| OpenFlights | 3,097 | 18,193 |
| Wikipedia Schools | 4,589 | 106,644 |
| Wikipedia Humans | 702,782 | 3,247,884 |

Table 6.1: Summary of the datasets

sub-clusters.

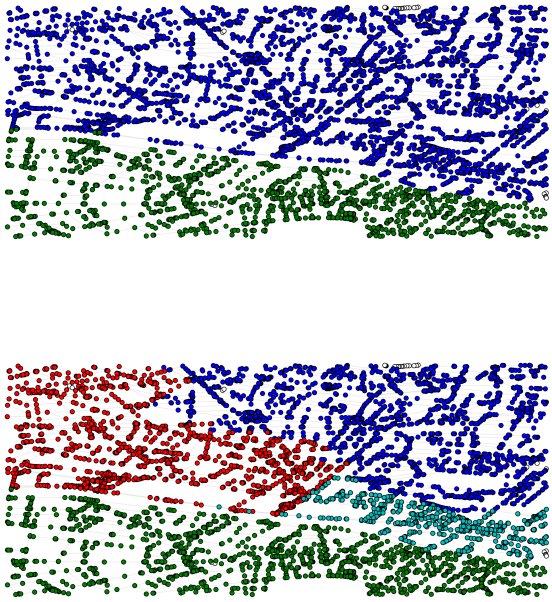


Figure 6.4: Clusterings of the OpenStreet graph by Paris.

The second dataset, extracted from OpenFlights[7], is the graph of airports with the weight between two airports equal to the number of daily flights between these airports. 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 Iceland/Groenland area and 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 6.5, with respectively 5 and 10 clusters corresponding to meaningful continental regions of the world.

[7] https://openflights.org

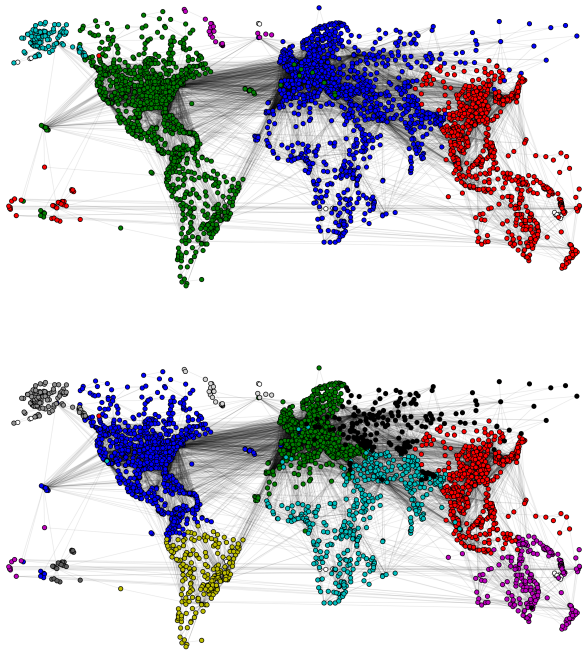


Figure 6.5: Clusterings of the OpenFlights graph by Paris.

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

Table 6.2: 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[8], see West et al. [2009], West and Leskovec [2012]. The graph is considered as undirected. Table 6.2 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 6.3 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 6.2. Again, only pages of highest degrees are shown for each cluster.

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

Table 6.3: The 10 largest subclusters of the first cluster of Table 6.2 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 6.4-6.5.
Finally, we give in Table 6.6 the running times of Louvain, Paris, and a spectral algorithm[9], for each of the 4 datasets. The experiments have been conducted

[8] https://schools-wikipedia.org
[9] The algorithm gives a hierarchical clustering using the Ward method applied to the spectral

| 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 6.4: The 10 largest clusters of Wikipedia Humans among 100 clusters found by Paris.

on a 2.7GHz Intel Core i5 CPU with 8GB of RAM. We observe that Paris is almost as fast as Louvain, while producing a much richer information on the graph (hierarchical clustering vs. simple clustering); it is faster than the spectral algorithm in most cases.

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 |
|---|---|
| 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 6.5: The 10 largest subclusters of the first cluster of Table 6.4 among 500 clusters found by Paris.

| Graph | Louvain | Paris | Spectral |
|---|---|---|---|
| OpenStreet | 0.5 | 0.5 | 1.1 |
| OpenFlights | 0.8 | 0.8 | 0.9 |
| SchoolsWikipedia | 3.1 | 3.5 | 11.2 |
| Wikipedia Humans | 92 | 110 | 340 |

Table 6.6: Mean running times (in seconds).

# Chapter 7

# Conclusion

The goal of this work was to study and propose novel algorithms in the fastly evolving field of graph neural networks in order to perform graph classification or node classification. The field of graph analysis with deep learning techniques is quite young and there remains many unexplored paths. We identified several aspects that haven't been studied yet and tried to answer them. Nevertheless, at the end of each work, we tried to remain critical and to analyse the positive and negative aspects of all our works.

Deep learning models have achieved state-of-the-art results in many scientific fields and it is quite natural to try to adapt these techniques to graph analysis, field in which there were barely no work on deep learning previous to 2014. In Chapters 1 we have introduced concepts of graphs and problematics that we tackle in this thesis. Moreover, in Chapter 2, we have presented background work on which rely current deep learning models and algorithms proposed in Chapters 3, 4 and 5. The definition of deep learning models and more specifically of filtering and convolutions on graphs is at the crossroads of several fields of research. It emerges from the application of signal processing techniques to graphs. This forces to define new concepts such as filtering on graphs which can have several formulations as seen in Chapter 2. In the next paragraphs, we will review algorithms proposed and the problems they answer.

In Chapter 3 we worked on embedding techniques for graph classification. We proposed two approaches that have the same objective: producing graph embeddings invariant by node permutation. When working on graph classification tasks by embedding graphs into finite dimensional vectors, it is important that a graph read into two different orders of nodes is embedded into one unique vector. Moreover, we want graph embeddings to contain a great discriminative power in order to be accurate in the graph classification task. To this end we developped a handcrafted graph embedding that originates from spectral

analysis and that is the concatenation of four well known embeddings. We showed through experiments on real life datasets that this embedding competes with state-of-the-art graph classification methods. Moreover, in order to improve this algorithms, we proposed a learned graph embedding invariant by node permutation for graph classification. In this second part of the work, we study two aspects of embeddings that are, the invariance by node permutation and the independance regarding the size of the graph. Indeed, in order to compare two graphs that may not be of the same size, their representation must have the same shape. We adapted an architecture, originally for cloud of points named PointNet [Qi et al., 2017] that provides a representation of points invariant by permutation, to graphs. This method allows us to compute graph embeddings invariant by node permutation. As seen in Chapter 3, the limitation of this method is that it does not take enough into account the structure of the graph. This is a limitation to find significant patterns between graphs. In the other Chapters about graph neural networks, we use message passing algorithms and more precisely GCNs to embed nodes and compare ourself with similar methods. When working on graph classification, most algorithms generate node embeddings that are then aggregated into a graph representation. The aggregation function is often just a mean of a max of node embeddings seems not optimal. Therefore, in Chapter 4, we proposed a new aggregation method that has several advantages. First, the goal of this method is to improve graph classification accuracy on benchmark dataset compared to using GCNs and an aggregation function that would be a mean or a max of node embeddings. Second, the objective is to bring interpretability to this type of algorithm that produces node and graph representations that are hardly interpretable. This is done by classifying nodes according to their roles in graphs. Indeed, nodes that lie in different parts of a graph can have similar structural role. These roles represent the local topological connectivity of nodes and thus connections patterns of nodes. We believe that, identifying patterns in graphs would improve the classification accuracy and would allow us to identify combinations of patterns that are representative of a certain group of graphs in a dataset. We showed that we were able to detect structural roles, and that these roles bring a certain amount of interpretability to a graph classification task. Nevertheless, there remains some work to be done in this line of research in order to have a better understanding of graph neural networks and possibly develop new methods more adapted to a classification task.

In the field of research of graph neural networks, instead of an aggregation layer to go from node embeddings to a graph representation, we can also think about using a pooling layer. Pooling layers allow one to detect hierarchical patterns in graph by computing representations of graphs at different scales.

Coarsening in graphs is not a simple task since nodes are not ordered and they can have irregular neighborhoods. Several formulations have been proposed to pool graphs. One of them focuses on nodes: we delete less informative nodes to decrease the size of the graph or we identify clusters of nodes to group vertices that are topologically close. Other formulations focus on edges, by performing contraction pooling by grouping nodes two by two. Finally, the alternative to learning a pooling strategy is to apply a deterministic clustering algorithm to group nodes by communities and obtain a graph of smaller size. In Chapter 5, we proposed a novel pooling layer, based on edges and more precisely on edge cuts. This method originates from edge scoring which has several formulations. Scores on edges that represent proximity of nodes allow us to detect edges that link nodes that are not topologically close and that possibly belong to different communities. Removing these edges allow us to highlight groups of nodes that are strongly connected and that we use as clusters in the coarsened level. This method has the great advantage of not being dependent of the number of clusters that we need to find and thus to output super nodes that are consistent with the topology of the graph. We validated our approach by experiments on graph classification and node classification.
Finally, the work presented in Chapter 6 was slighly different because not connected to graph neural networks. We tackled the problem of finding communities in graphs and more precisely hierarchical communities. We based our approach on the modularity which is a measure that is used to evaluate if a graph partitioning is composed of clusters that are srtongly connected or if the clusters are close to a random model. We defined an algorithm that outputs a partition of a graph at different scales by modifying the modularity and by adding a resolution parameter that controles the size of output communities. We define a distance between clusters and use the nearest-neighbor chain technique traditionally used to cluster vectors in Euclidean spaces. Future work could be done to automatically extract the most relevant clusters from the dendrogram at different scales. 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.

## 7.1 Future work

There are many applications to graph neural networks and many directions that we would like to explore. One of them is graph generation. Many models applied to images such as variational autoencoders or generative adversarial networks can be applied to graphs to generate graphs that have certain properties. Those models work well on images or textual data, however, generating graphs

represents a greater challenge. In the case of molecules, molecules must verify certain chemical properties to be valid molecules. Moreover, it is not easy to generate both nodes, edges and node features. A recursive methodology to create edges based on existing ones might work better then generating all edges at once. Many different ways exist to generate graphs with the highest probability.
Moreover, graph generation can also be applied to code. Programs in different languages can be represented by graphs. This can be used to classify graphs according to the author or to the objective of the program. This can also be used to generate code according to a task or to translate a program from one language to another.

Finally, as introduced in Chapter 1, a final line of work on which we would like to work next is time evolving graphs. There are many applications and many different ways to tackle this kind of problem. We would like to generalize our methods on edge scoring or structural role discovery to link prediction and anomaly detection and in particular to cases in which graphs are time dependent.

# Bibliography


R. Achanta and S. Susstrunk. Superpixels and polygons using simple non-iterative clustering. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4651–4660, 2017.

L. Akoglu, H. Tong, and D. Koutra. Graph based anomaly detection and description: a survey. *Data mining and knowledge discovery*, 29(3):626–688, 2015.

M. Al Hasan, V. Chaoji, S. Salem, and M. Zaki. Link prediction using supervised learning. 30:798–805, 2006.

A. Arenas, A. Fernandez, and S. Gomez. Analysis of the structure of complex networks at different resolution levels. *New journal of physics*, 10(5):053039, 2008.

J. Atwood and D. Towsley. Diffusion-convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1993–2001, 2016.

L. Babai. Graph isomorphism in quasipolynomial time. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, pages 684–697. ACM, 2016.

L. Babai, D. Y. Grigoryev, and D. M. Mount. Isomorphism of graphs with bounded eigenvalue multiplicity. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 310–324. ACM, 1982.

D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

P. Battaglia, R. Pascanu, M. Lai, D. J. Rezende, et al. Interaction networks for learning about objects, relations and physics. In *Advances in neural information processing systems*, pages 4502–4510, 2016.

J.-P. Benzécri. Construction d'une classification ascendante hiérarchique par la recherche en chaîne des voisins réciproques. *Cahiers de l'analyse des données*, 7(2):209–218, 1982.

S. Bhagat, G. Cormode, and S. Muthukrishnan. Node classification in social networks. pages 115–148, 2011.

F. M. Bianchi, D. Grattarola, and C. Alippi. Mincut pooling in graph neural networks. *arXiv preprint arXiv:1907.00481*, 2019.

V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008.

T. Bonald, B. Charpentier, A. Galland, and A. Hollocou. Hierarchical graph clustering using node pair sampling. *arXiv preprint arXiv:1806.01664*, 2018a.

T. Bonald, A. Hollocou, and M. Lelarge. Weighted spectral embedding of graphs. *arXiv preprint arXiv:1809.11115*, 2018b.

K. M. Borgwardt and H.-P. Kriegel. Shortest-path kernels on graphs. In *Data Mining, Fifth IEEE International Conference on*, pages 8–pp. IEEE, 2005.

M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst. Geometric deep learning: going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42, 2017.

J. Bruna and X. Li. Community detection with graph neural networks. *arXiv preprint arXiv:1705.08415*, 2017.

J. Bruna and S. Mallat. Invariant scattering convolution networks. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1872–1886, 2013.

J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun. Spectral networks and locally connected networks on graphs. *arXiv preprint arXiv:1312.6203*, 2013.

C.-S. Chang, C.-Y. Hsu, J. Cheng, and D.-S. Lee. A general probabilistic framework for detecting community structure in networks. In *INFOCOM, 2011 Proceedings IEEE*, pages 730–738. IEEE, 2011.

A. Clauset, C. Moore, and M. E. Newman. Hierarchical structure and the prediction of missing links in networks. *Nature*, 453(7191):98, 2008.

T. Cohen and M. Welling. Group equivariant convolutional networks. In *International conference on machine learning*, pages 2990–2999, 2016.

M. Connor and P. Kumar. Fast construction of k-nearest neighbor graphs for point clouds. *IEEE transactions on visualization and computer graphics*, 16 (4):599–608, 2010.

M. Defferrard, X. Bresson, and P. Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in Neural Information Processing Systems*, pages 3844–3852, 2016.

J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

I. S. Dhillon, Y. Guan, and B. Kulis. Weighted graph cuts without eigenvectors a multilevel approach. *IEEE transactions on pattern analysis and machine intelligence*, 29(11):1944–1957, 2007.

F. Diehl. Edge contraction pooling for graph neural networks. *arXiv preprint arXiv:1905.10990*, 2019.

L. Donetti and M. A. Munoz. Detecting network communities: a new systematic and efficient algorithm. *Journal of Statistical Mechanics: Theory and Experiment*, 2004(10):P10012, 2004.

C. Donnat, M. Zitnik, D. Hallac, and J. Leskovec. Learning structural node embeddings via diffusion wavelets. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1320–1329. ACM, 2018.

V. P. Dwivedi, C. K. Joshi, T. Laurent, Y. Bengio, and X. Bresson. Benchmarking graph neural networks. *arXiv preprint arXiv:2003.00982*, 2020.

M. Fiedler. A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory. *Czechoslovak Mathematical Journal*, 25(4): 619–633, 1975.

S. Fortunato and M. Barthelemy. Resolution limit in community detection. *Proceedings of the National Academy of Sciences*, 104(1):36–41, 2007.

L. C. Freeman. Centered graphs and the structure of ego networks. *Mathematical Social Sciences*, 3(3):291–304, 1982.

A. Galland and M. Lelarge. Invariant embedding for graph classification. In *ICML 2019 Workshop on Learning and Reasoning with Graph-Structured Representations*, 2019.

F. Gama, A. Ribeiro, and J. Bruna. Stability of graph scattering transforms. In *Advances in Neural Information Processing Systems*, pages 8038–8048, 2019.

H. Gao and S. Ji. Graph u-nets. *arXiv preprint arXiv:1905.05178*, 2019.

T. Gärtner, P. Flach, and S. Wrobel. On graph kernels: Hardness results and efficient alternatives. In *Learning theory and kernel machines*, pages 129–143. Springer, 2003.

L. Getoor. Link-based classification. In *Advanced methods for knowledge discovery from complex data*, pages 189–207. Springer, 2005.

J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1263–1272. JMLR. org, 2017.

S. Gilpin, T. Eliassi-Rad, and I. Davidson. Guided learning for role discovery (glrd) framework, algorithms, and applications. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 113–121, 2013.

I. Goodfellow, Y. Bengio, and A. Courville. Deep learning. 1, 2016.

A. Grover and J. Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864, 2016.

L. Hagen and A. B. Kahng. New spectral methods for ratio cut partitioning and clustering. *IEEE transactions on computer-aided design of integrated circuits and systems*, 11(9):1074–1085, 1992.

W. Hamilton, Z. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *Advances in neural information processing systems*, pages 1024–1034, 2017.

D. K. Hammond, P. Vandergheynst, and R. Gribonval. Wavelets on graphs via spectral graph theory. *Applied and Computational Harmonic Analysis*, 30(2): 129–150, 2011.

H. A. Helfgott. Isomorphismes de graphes en temps quasi-polynomial (d'apr\es babai et luks, weisfeiler-leman...). *arXiv preprint arXiv:1701.04372*, 2017.

K. Henderson, B. Gallagher, T. Eliassi-Rad, H. Tong, S. Basu, L. Akoglu, D. Koutra, C. Faloutsos, and L. Li. Rolx: structural role extraction & mining in large graphs. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1231–1239, 2012.

W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *arXiv preprint arXiv:2005.00687*, 2020.

J. Huang, H. Sun, J. Han, H. Deng, Y. Sun, and Y. Liu. Shrink: A structural clustering algorithm for detecting hierarchical communities in networks. In *Proceedings of the 19th ACM International Conference on Information and Knowledge Management*, CIKM '10, pages 219–228, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0099-5. doi: 10.1145/1871437.1871469. URL `http://doi.acm.org/10.1145/1871437.1871469`.

J. J. Irwin, T. Sterling, M. M. Mysinger, E. S. Bolstad, and R. G. Coleman. Zinc: a free tool to discover chemistry for biology. *Journal of chemical information and modeling*, 52(7):1757–1768, 2012.

J. Juan. Programme de classification hiérarchique par l'algorithme de la recherche en chaîne des voisins réciproques. *Cahiers de l'analyse des données*, 7(2):219–225, 1982.

N. Kalchbrenner, E. Grefenstette, and P. Blunsom. A convolutional neural network for modelling sentences. *arXiv preprint arXiv:1404.2188*, 2014.

G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1): 359–392, 1998.

H. Kashima, K. Tsuda, and A. Inokuchi. Marginalized kernels between labeled graphs. In *Proceedings of the 20th international conference on machine learning (ICML-03)*, pages 321–328, 2003.

S. Kearnes, K. McCloskey, M. Berndl, V. Pande, and P. Riley. Molecular graph convolutions: moving beyond fingerprints. *Journal of computer-aided molecular design*, 30(8):595–608, 2016.

T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

N. M. Kriege, P.-L. Giscard, and R. Wilson. On valid optimal assignment kernels and applications to graph classification. In *Advances in Neural Information Processing Systems*, pages 1623–1631, 2016.

R. Lambiotte, J.-C. Delvenne, and M. Barahona. Random walks, Markov processes and the multiscale modular organization of complex networks. *IEEE Transactions on Network Science and Engineering*, 2014.

A. Lancichinetti and S. Fortunato. Limits of modularity maximization in community detection. *Physical review E*, 84(6):066122, 2011.

A. Lancichinetti, S. Fortunato, and J. Kertész. Detecting the overlapping and hierarchical community structure in complex networks. *New Journal of Physics*, 11(3):033015, 2009.

Y. LeCun, Y. Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.

Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

R. Li, S. Wang, F. Zhu, and J. Huang. Adaptive graph convolutional neural networks. In *Thirty-second AAAI conference on artificial intelligence*, 2018.

L. Lovász et al. Random walks on graphs: A survey. *Combinatorics, Paul erdos is eighty*, 2(1):1–46, 1993.

T. Lucas, C. Tallec, J. Verbeek, and Y. Ollivier. Mixed batches and symmetric discriminators for gan training. *arXiv preprint arXiv:1806.07185*, 2018.

E. Lukacs. Characteristic functions. 1970.

V. Lyzinski, M. Tang, A. Athreya, Y. Park, and C. E. Priebe. Community detection and classification in hierarchical stochastic blockmodels. *IEEE Transactions on Network Science and Engineering*, 4(1):13–26, 2017.

S. Mallat. *A wavelet tour of signal processing*. Elsevier, 1999.

S. Mallat. A wavelet tour of signal processing. 2008.

A. Micheli. Neural network for graphs: A contextual constructive approach. *IEEE Transactions on Neural Networks*, 20(3):498–511, 2009.

T. Mikolov, K. CHen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013a.

T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013b.

F. Murtagh and P. Contreras. Algorithms for hierarchical clustering: an overview. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 2012.

M. Newman. Community detection in networks: Modularity optimization and maximum likelihood are equivalent. *arXiv preprint arXiv:1606.02319*, 2016.

M. E. Newman. Fast algorithm for detecting community structure in networks. *Physical review E*, 69(6):066133, 2004.

M. E. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical review E*, 2004.

A. Ortega, P. Frossard, J. Kovačević, J. M. Moura, and P. Vandergheynst. Graph signal processing. *arXiv preprint arXiv:1712.00468*, 2017.

B. Perozzi, R. Al-Rfou, and S. Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710, 2014.

P. Pons and M. Latapy. Computing communities in large networks using random walks. In *International symposium on computer and information sciences*, pages 284–293. Springer, 2005.

C. R. Qi, H. Su, K. Mo, and L. J. Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. *Proc. Computer Vision and Pattern Recognition (CVPR), IEEE*, 1(2):4, 2017.

J. Reichardt and S. Bornholdt. Statistical mechanics of community detection. *Physical Review E*, 74(1):016110, 2006.

L. F. Ribeiro, P. H. Saverese, and D. R. Figueiredo. struc2vec: Learning node representations from structural identity. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 385–394, 2017.

E. Richard, N. Baskiotis, T. Evgeniou, and N. Vayatis. Link discovery using graph feature tracking. In *Advances in Neural Information Processing Systems*, pages 1966–1974, 2010.

P. Ronhovde and Z. Nussinov. Multiresolution community detection for megascale networks by information-based replica correlations. *Physical Review E*, 80(1): 016109, 2009.

O. Ronneberger, P. Fischer, and T. Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.

M. Sales-Pardo, R. Guimera, A. A. Moreira, and L. A. N. Amaral. Extracting the hierarchical organization of complex systems. *Proceedings of the National Academy of Sciences*, 104(39):15224–15229, 2007.

F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1): 61–80, 2009.

N. Shervashidze, S. Vishwanathan, T. Petri, K. Mehlhorn, and K. Borgwardt. Efficient graphlet kernels for large graph comparison. In *Artificial Intelligence and Statistics*, pages 488–495, 2009.

N. Shervashidze, P. Schweitzer, E. J. v. Leeuwen, K. Mehlhorn, and K. M. Borgwardt. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12(Sep):2539–2561, 2011.

J. Shi and J. Malik. Normalized cuts and image segmentation. *IEEE Transactions on pattern analysis and machine intelligence*, 22(8):888–905, 2000.

D. I. Shuman, P. Vandergheynst, and P. Frossard. Chebyshev polynomial approximation for distributed signal processing. In *2011 International Conference on Distributed Computing in Sensor Systems and Workshops (DCOSS)*, pages 1–8. IEEE, 2011.

D. I. Shuman, S. K. Narang, P. Frossard, A. Ortega, and P. Vandergheynst. The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains. *IEEE signal processing magazine*, 30(3):83–98, 2013.

D. A. Spielman. Spectral graph theory and its applications. In *Foundations of Computer Science, 2007. FOCS'07. 48th Annual IEEE Symposium on*, pages 29–38. IEEE, 2007.

C. Stark, B.-J. Breitkreutz, A. Chatr-Aryamontri, L. Boucher, R. Oughtred, M. S. Livstone, J. Nixon, K. Van Auken, X. Wang, X. Shi, et al. The biogrid interaction database: 2011 update. *Nucleic acids research*, 39(suppl_1): D698–D704, 2010.

D. Stutz, A. Hermans, and B. Leibe. Superpixels: An evaluation of the state-of-the-art. *Computer Vision and Image Understanding*, 166:1–27, 2018.

J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei. Line: Large-scale information network embedding. In *Proceedings of the 24th international conference on world wide web*, pages 1067–1077, 2015.

N. Tremblay and P. Borgnat. Graph wavelets for multiscale community mining. *IEEE Transactions on Signal Processing*, 62(20):5227–5239, 2014a.

N. Tremblay and P. Borgnat. Graph wavelets for multiscale community mining. *IEEE Transactions on Signal Processing*, 62(20):5227–5239, 2014b.

N. Tremblay, P. Gonçalves, and P. Borgnat. Design of graph filters and filterbanks. In *Cooperative and Graph Signal Processing*, pages 299–324. Elsevier, 2018.

A. Tsitsulin, J. Palowitch, B. Perozzi, and E. Müller. Graph clustering with graph neural networks. *arXiv preprint arXiv:2006.16904*, 2020.

J. Ugander, B. Karrer, L. Backstrom, and C. Marlow. The anatomy of the facebook social graph. *arXiv preprint arXiv:1111.4503*, 2011.

M. Van den Bergh, X. Boix, G. Roig, B. de Capitani, and L. Van Gool. Seeds: Superpixels extracted via energy-driven sampling. In *European conference on computer vision*, pages 13–26. Springer, 2012.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

S. Verma and Z.-L. Zhang. Hunt for the unique, stable, sparse and fast feature learning on graphs. In *Advances in Neural Information Processing Systems*, pages 88–98, 2017.

U. Von Luxburg. A tutorial on spectral clustering. *Statistics and computing*, 17 (4):395–416, 2007.

U. Von Luxburg, M. Belkin, and O. Bousquet. Consistency of spectral clustering. *The Annals of Statistics*, pages 555–586, 2008.

J. H. Ward. Hierarchical grouping to optimize an objective function. *Journal of the American statistical association*, 1963.

B. Weisfeiler and A. Lehman. A reduction of a graph to a canonical form and an algebra arising during this reduction. *Nauchno-Technicheskaya Informatsia*, 2 (9):12–16, 1968.

R. West and J. Leskovec. Human wayfinding in information networks. In *Proceedings of the 21st international conference on World Wide Web*, pages 619–628. ACM, 2012.

R. West, J. Pineau, and D. Precup. Wikispeedia: An online game for inferring semantic distances between concepts. In *IJCAI*, pages 1598–1603, 2009.

K. Xu, W. Hu, J. Leskovec, and S. Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

P. Yanardag and S. Vishwanathan. Deep graph kernels. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1365–1374. ACM, 2015.

Z. Yang, W. Cohen, and R. Salakhudinov. Revisiting semi-supervised learning with graph embeddings. In *International conference on machine learning*, pages 40–48. PMLR, 2016.

Z. Ying, J. You, C. Morris, X. Ren, W. Hamilton, and J. Leskovec. Hierarchical graph representation learning with differentiable pooling. In *Advances in Neural Information Processing Systems*, pages 4800–4810, 2018.

B. Yu, H. Yin, and Z. Zhu. St-unet: A spatio-temporal u-network for graph-structured time series modeling. *arXiv preprint arXiv:1903.05631*, 2019.

W. W. Zachary. An information flow model for conflict and fission in small groups. *Journal of anthropological research*, 33(4):452–473, 1977.

M. Zaheer, S. Kottur, S. Ravanbakhsh, B. Poczos, R. R. Salakhutdinov, and A. J. Smola. Deep sets. In *Advances in Neural Information Processing Systems*, pages 3391–3401, 2017.

M. Zhang and Y. Chen. Link prediction based on graph neural networks. In *Advances in Neural Information Processing Systems*, pages 5165–5175, 2018.

M. Zhang, Z. Cui, M. Neumann, and Y. Chen. An end-to-end deep learning architecture for graph classification. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018a.

X. Zhang, S. E. Chew, Z. Xu, and N. D. Cahill. Slic superpixels for efficient graph-based dimensionality reduction of hyperspectral imagery. In *Algorithms and Technologies for Multispectral, Hyperspectral, and Ultraspectral Imagery XXI*, volume 9472, page 947209. International Society for Optics and Photonics, 2015.

Z. Zhang, M. Wang, Y. Xiang, Y. Huang, and A. Nehorai. Retgk: Graph kernels based on return probabilities of random walks. In *Advances in Neural Information Processing Systems*, pages 3968–3978, 2018b.

C. Zhuang and Q. Ma. Dual graph convolutional networks for graph-based semi-supervised classification. In *Proceedings of the 2018 World Wide Web Conference*, pages 499–508, 2018.

## RÉSUMÉ

Les graphes sont présents dans de nombreux domaines de recherche, que ce soit pour représenter des molécules, des réseaux sociaux ou des réseaux de transport. Un graphe est un outil mathématique utilisé pour représenter des relations entre des objets. Il est composé de nœuds reliés entre eux par des liens, appelés arêtes. Récemment, les techniques d'apprentissage profond ont prouvé leur efficacité dans de nombreux domaines tels que le traitement de texte ou l'analyse d'images. Ce constat a motivé de nombreux travaux de recherche visant à généraliser les techniques d'apprentissage profond à l'analyse de graphes. Ainsi des algorithmes se basant notamment sur des réseaux de neurones et des convolutions ont été développés afin de répondre à des problématiques de classification de nœuds et de graphes. Au cours de cette thèse nous analysons les représentations vectorielles des nœuds, des communautés ou de l'ensemble du graphe qui émerge de ces modèles. Ces représentations à différentes échelles, encodent des informations hierarchiques sur le graphe. En ce basant sur ces vectorialisations de graphes, nous proposons de nouvelles architectures afin de répondre à des tâches de classification de nœuds et classification de graphes. Nous étudions plusieurs applications de ces nouvelles techniques notamment le problème d'obtenir une représentation invariante par permutation des nœuds ou encore l'interprétabilité de ces algorithmes.

## MOTS CLÉS

graphe, réseaux, apprentissage profond, traitement du signal, classification, plongement, analyse spectrale

## ABSTRACT

In many scientific fields, studied data have an underlying graph or manifold structure such as communication networks (whether social or technical), knowledge graphs or molecules. A graph is composed of nodes, also called vertices, connected together by edges. Recently, deep learning algorithms have become state-of-the-art models in many fields and in particular in natural language processing and image analysis. It led the way to a great line of studies to generalize deep learning models to graphs. In particular, several formulations of convolutional neural networks were proposed and research is carried to develop new layers and network architectures to graphs. Those models aim at solving different tasks such as node classification, link prediction or graph classification. In this work, we study node, subgraph or graph embeddings produced by graph neural networks. These embeddings at different scales encode hierarchical representations of graphs. Based on these embedding techniques, we propose new deep learning architectures to tackle node classification or graph classification tasks. We study several applications of these new techniques. For example, we study the problem of having a graph embedding invariant by node permutation and the interpretability of graph neural networks.

## KEYWORDS

graph, network, deep learning, signal processing, classification, embedding, spectral analysis