Bridging graph and kernel spaces : a pre-image perspective

Linlin Jia

Normandie Université

THÈSE

Pour obtenir le diplôme de doctorat

Spécialité Informatique

Préparée au sein de INSA ROUEN NORMANDIE

Bridging Graph and Kernel Spaces: A Pre-image Perspective

Présentée et soutenue par

Linlin JIA

Thèse soutenue publiquement le 09/07/2021 devant le jury composé de		
Mme. Florence D'ALCHÉ-BUC	PR de Télécom Paris et Institut Polytechnique de Paris	Rapporteuse
M. Donatello CONTE	MCF HDR de Université de Tours	Rapporteur
M. Sébastien ADAM	PR de Université de Rouen Normandie	Examinateur
M. Francesc SERRATOSA	PR de Universitat Rovira i Virgili, Tarragona, Catalonia	Examinateur
M. Florian YGER	MCF de Université Paris-Dauphine	Examinateur
M. Paul HONEINE	PR de Université de Rouen Normandie	Directeur de thèse
M. Benoit GAÜZÈRE	MCF de INSA Rouen Normandie	Encadrant de thèse

Thèse dirigée par Paul HONEINE et Benoit GAÜZÈRE, laboratoire LITIS

INSA ROUEN NORMANDIE

litis

Acknowledgements

First, I would like to thank my supervisors, Professor Paul Honeine and Professor Benoit Gaüzère. They are great mentors and friends for me. They provided me not only professional tutorials on research, but also the spirit to have an interesting life.

Second, I thank my parents for trusting and supporting me to pursue my own life and dreams, even though they are so worried about me not having a wife.

Third, I thank my friends, whose names I will not mention here in case I forget someone. My colleagues in my Lab, my friends from China and the CSC projects, and all the others. We have shared many wonderful moments.

I would like to give special thanks to Brigitte, Madame Diarra, the secretary of the LITIS Lab, for helping me deal with all kinds of stuff, with such a kind attitude and melodious, unhurried French.

Fourth, I would like to thank the two countries, France and China, for providing me an opportunity to see a bigger world. The drop-dead gorgeous views across Europe let me forget the difficulties during this period.

Fifth, I would like to thank my money provider. Our research was supported by CSC (China Scholarship Council) and the French national research agency (ANR) under the grant APi (ANR-18-CE23-0014). We would like to thank the CRIANN (Le Centre Régional Informatique et d'Applications Numériques de Normandie) for providing computational resources.

Last but not least, I would like to thank the guy who finished this job, for there are so many chances to give up, but instead, he chose to try a little bit more. Thanks for still believing.

Résumé

Les graphes permettent de modéliser un large éventail de données du monde réel. Avec les progrès des méthodes d'apprentissage automatique, l'utilisation de ces méthodes sur les graphes est devenue de plus en plus intéressante au sein de la communauté scientifique. Parmi les différentes études, les problèmes de prédiction et de génération de graphes suscitent un intérêt grandissant, porté par de nombreuses applications, notamment en chemoinformatique, en analyse de réseaux sociaux et en vision par ordinateur.

La flexibilité associée aux graphes possède un inconvénient majeur. Les algorithmes d'apprentissage automatique étant définis sur des données vectorielles, leur adaptation aux graphes n'est pas triviale. Il est donc important de concevoir des outils permettant d'utiliser ces méthodes d'apprentissage automatique sur les graphes. Les noyaux sur graphes sont un outil puissant pour traiter ce problème en comblant le fossé entre un espace de graphes et un espace à noyau. Inversement, étant donné un élément dans un espace à noyau, on s'intéresse à construire son homologue dans l'espace de graphes ; c'est le problème de pré-image de graphe. La résolution de ce problème ouvre la voie à de nombreuses applications prometteuses, telles que la synthèse de molécules et la conception de médicaments.

Pour résoudre les problèmes susmentionnés, nous proposons dans cette thèse de nouvelles méthodes pour construire la pré-image de graphe. Tout d'abord, nous passons en revue les noyaux sur graphes basés sur des motifs linéaires, qu'on compare à des noyaux basés sur des motifs non linéaires. Des analyses théoriques et expérimentales approfondies sont présentées et trois stratégies sont proposées pour réduire la complexité de calcul et de stockage de ces noyaux. Ensuite, nous étudions la stabilité des plus récentes heuristiques de calcul de distance d'édition de graphes et proposons une méthode d'apprentissage de métrique pour optimiser les coûts d'édition utilisés par la distance d'édition de graphes pour des problèmes de régression. Les expériences montrent que notre méthode permet d'améliorer les performances de prédiction par rapport à l'état de l'art. Sur la base de ces deux travaux, nous proposons une méthode de calcul de pré-image de graphe basée sur l'apprentissage de métrique. Pour cela, l'espace de graphes, muni d'une distance d'édition de graphes, est aligné avec l'espace à noyau associé à un noyau particulier. Pour ce faire, la méthode proposée optimise les coûts d'édition afin de faire correspondre les distances dans les deux espaces. Ensuite, une estimation de la pré-image de graphe est obtenue par une méthode générative de graphe médian, configurée avec les coûts d'édition optimisés. Les expérimentations effectuées montrent que notre méthode est plus performante et plus générique que l'état de l'art. Enfin, pour aider à résoudre les problèmes susmentionnés, nous avons publié une bibliothèque en Python pour l'apprentissage automatique de graphes, comprenant l'implémentation de noyaux sur graphes, d'algorithmes de calcul de distances d'édition de graphes et de méthodes de calcul de la pré-image de graphes.

Mots-clés : Noyaux sur graphes, pré-images de graphes, distances d'édition de graphes, apprentissage automatique, reconnaissance de formes, chimioinformatique

Abstract

Graphs are able to represent a wide range of real-world data due to the nature of their structure and their rich expressiveness. With the advance of state-of-the-art machine learning methods, applying these methods on graph data has become more and more intriguing for academic and industry societies. Among all studies, graph prediction and graph construction problems draw a lot of interest, since they allow to address many applications such as in chemoinformatics, social network analysis, and computer vision.

However, the expressiveness of graphs is a double-edged sword. Indeed, it is difficult to apply most machine learning algorithms on graphs because they were defined for vector data. Therefore, it is important to design tools to utilize these machine learning methods on graphs. Graph kernels provide a powerful tool to deal with this problem by bridging the gap between a graph space and a kernel space. Inversely, given an element in a kernel space, one is interested in constructing its counterpart in the graph space; this is the so-called pre-image problem. Solving the pre-image problem for graphs opens the door to many interesting applications, such as molecule synthesis and drug design.

To address the aforementioned problems, we propose in this thesis novel methods to construct graph pre-images given a graph kernel. First, we provide an in-depth review of graph kernels based on linear patterns, and compare them to some kernels based on non-linear patterns. Thorough theoretical and experimental analyses are provided and three strategies are proposed to address computational complexity and memory usage of these kernels. Second, we study the stability of the state-of-the-art graph edit distance heuristics and propose a metric learning approach to estimate the graph edit costs for regression. Experiments show that the proposed method outperforms state-of-the-art methods. Based on these two successful working directions, we then propose a graph pre-image method based on metric learning. For this purpose, the graph space endowed with graph edit distances is aligned with the kernel space defined by a graph kernel. To this end, we optimize the edit costs by matching graph edit distances and distances in the kernel space. After that, the graph pre-image is estimated by a median generative method. Experiments show that our method outperforms the state of the art. Finally, to help solve the aforementioned problems, we published a Python library on graph machine learning, including the implementations of graph kernels, graph edit distances, and graph pre-image methods.

Keywords: Graph Kernels, Graph Pre-image, Graph Edit Distance, Machine Learning, Pattern Recognition, Chemoinformatics

Contents

List of Publications

During this Ph.D., the following research works have been published:

Peer-reviewed journal papers

Linlin Jia, Benoit Gaüzère, and Paul Honeine. graphkit-learn: A python library for graph kernels based on linear patterns. Pattern Recognition Letters, 2021.

Linlin Jia, Benoit Gaüzère, and Paul Honeine. Graph Kernels Based on Linear Patterns: Theoretical and Experimental Comparisons. working paper or preprint, March 2019. (submitted to Expert Systems with Applications)

Peer-reviewed international conference papers

Linlin Jia, Benoit Gaüzère, and Paul Honeine. A Graph Pre-image Method Based on Graph Edit Distances. Proceedings of IAPR Joint International Workshops on Statistical Techniques in Pattern Recognition (SPR 2020) and Structural and Syntactic Pattern Recognition (SSPR 2020), 2021.

Linlin Jia, Benoit Gaüzère, Florian Yger and Paul Honeine. A Metric Learning Approach to Graph Edit Costs for Regression. Proceedings of IAPR Joint International Workshops on Statistical Techniques in Pattern Recognition (SPR 2020) and Structural and Syntactic Pattern Recognition (SSPR 2020), 2021.

Libraries

`graphkit-learn`: A Python package on graph kernels, graph edit distances and graph pre-image problem.

List of Figures

List of Tables

List of Algorithms

Part I

Introduction and preliminaries

Chapter 1

Introduction

Contents

This chapter gives a general introduction to this thesis. It starts with the background information for this work. After that, the main contributions are presented. Finally, the structure of the monograph is presented in detail.

1.1 Background of graph representation

Graphs are powerful tools to model a wide range of real-world data. Consisting of vertices and edges connecting them, graphs are able to encode elements as well as the relationship between them, which enables to capture the underlying structural information of the data. Moreover, each vertex and edge can be equipped with discrete and/or continuous attributes, offering the ability to encode a vast variety of aspects to describe data.

Benefiting from the complexity and the expressiveness of the graph structure, broad applications in wide domains embrace its representation. In chemoinformatics and bioinformatics, molecules including biomacromolecules can be naturally represented by graphs [Trinajstic, 2018, Huber et al., 2007]. A vertex can represent an atom, which is labeled by the atom type and sometimes other attributes such as its physical position, while an edge can represent the connection between the two atoms, normally labeled by the valency. Biological and biomedical applications can be then based on it, such as drug design and discovery [Vamathevan et al., 2019], biological and chemical effect analysis, as well as biometric identification [Kisku et al., 2011]. In social media analysis, large graphs are used to model social networks [Scott, 2011, Wasserman et al., 1994]. For instance, a user profile can be encoded by a vertex labeled by its detailed information, and the relationship between user profiles can be encoded by edges. This representation can then be used for applications such as web data mining [Russell, 2013, Rettinger et al., 2012] and advertising [Guo et al., 2020]. In computer vision, both 2D and 3D images can be represented by graphs. In 2D applications, a common one is handwriting recognition, where the lines are represented by edges and the connecting points are represented by vertices labeled by position [Riesen and Bunke, 2008]. This representation allows extracting directly the information. A similar strategy can be used for applications such as object recognition [Nene et al., 1996]. Other approaches to model an image as a graph include encoding each pixel with a vertex and connecting the adjacent pixels with an edge [Wang, 2015]. As a 3D data, a point cloud can be intuitively represented by a graph, with each point as a ver-

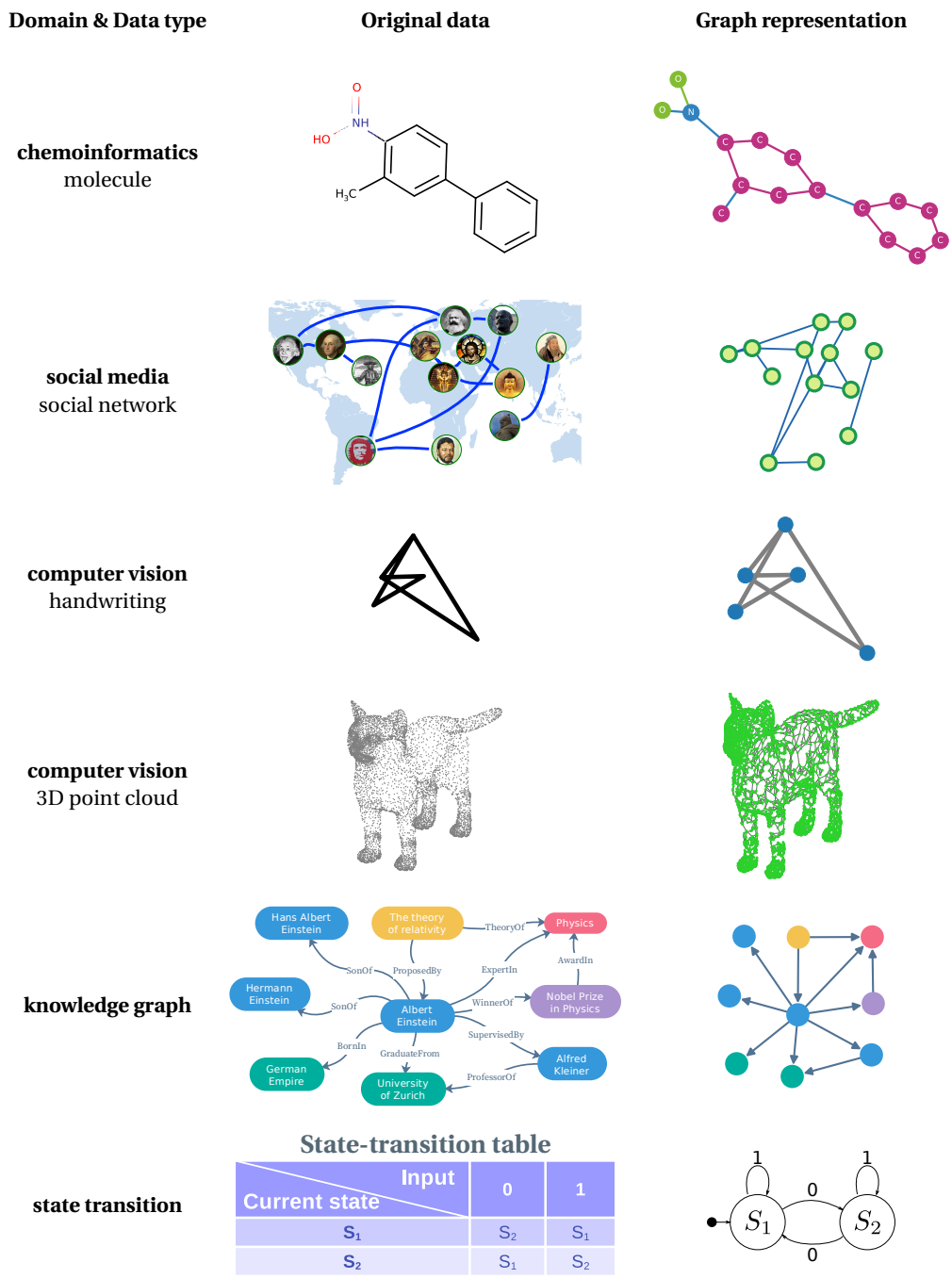

Figure 1.1 – Example data and their graph representation.

tex [de Oliveira Rente et al., 2018]. Knowledge graph, as well as its schema-oriented counterpart ontology, constitutes another domain that is founded on graphs [Ji et al., 2020]. Coherent to its name, a knowledge graph is a graph constructed by information from a domain knowledge vault. A large amount of applications

falls into this category, such as information retrieval [Reinanda et al., 2020], natural-language processing [Nastase et al., 2015], the semantic web [Ding et al., 2007], and manufacturing process modeling [Cao et al., 2018]. Other applications of graphs include state transition [Conte et al., 2004], temporal graphs [Michail, 2016], etc. The inter-domain modeling and applications are widely seen. Figure 1.1 presents several example data in these domains and their graph representations. The molecule is from the *MUTAG* dataset [Debnath et al., 1991]; the images used for the avatars in the social network are from Wikipedia[1]; the handwriting letter "A" is from the *Letter-high* dataset [Riesen and Bunke, 2008]; the point cloud is generated from a three-dimensional cat model[2] using the `Open3D` library [Zhou et al., 2018b]; the knowledge graph is built on data from [Ji et al., 2020].

All these applications are founded on the answers to two basic questions to the graph model:

- **Question 1: How to acquire intrinsic features and properties from graph datasets?**
- **Question 2: How to construct graphs endowing desired features and properties?**

For example, when a drug molecule is represented as a graph, analyzing the influence of a sub-structure on its function (e.g., its anti-cancerous ability) relates to question 1, while question 2 may involve designing new drugs for a given property. In this thesis, we aim at designing effective and efficient methods to tackle these two fundamental questions.

1.2 Graph properties and the kernel paradigm

1.2.1 From human to machine learning

As in most data science applications, extracting features and analyzing properties from graph data have been carried out by domain experts for many years. Traditional domain-specific methods may be explored to help the procedure. The situation is the same for graph construction applications. However, these methods suffer from several major issues. First, to design some methods, domain knowledge is required a priori, which may have a high influence on the performance. Second, due

[1] `https://www.wikipedia.org`.

[2] Available at `https://free3d.com/3d-model/cat-v1--522281.html`.

to their specification to domains and data, their generalization ability is not generally very good, which limits their application and expands the designing costs. Last but not least, these methods normally lack the ability to learn from (new) data, causing a loss of potential information endowed by the data [George and Hautier, 2020].

With the advent of the so-called big data era, automated methods to analyze massive amounts of data are called for, where the advantages of machine learning lie. Kevin P. Murphy [Murphy, 2012] defines machine learning as

> *a set of methods that can automatically detect patterns in data, and then use the uncovered patterns to predict future data, or to perform other kinds of decision making under uncertainty (such as planning how to collect more data!).*

Consistent with this definition, Tom M. Mitchell [Mitchell et al., 1997] provided a more formal and operational definition of machine learning algorithms:

> *A computer program is said to learn from experience* E *with respect to some class of tasks* T *and performance measure* P *if its performance at tasks in* T*, as measured by* P*, improves with experience* E.

In recent years, machine learning has been providing considerably effective and efficient tools to address multiple real-world tasks, including regression and classification problems. Relying on advanced statistics, they can extract information from data and provide more generalized results than the conventional expert-based methods. Moreover, they have the potential to be applied in different datasets and domains with minor modifications. Furthermore, expert knowledge can be integrated with them to obtain better performance. The advantages of machine learning make it intriguing to apply machine learning methods on graph data [Chami et al., 2020], including molecular machine learning [Wu et al., 2018] and drug discovery [Giguère et al., 2015]. However, machine learning algorithms have been conventionally defined on vector spaces, allowing to take advantage of the easiness in linear algebra operations. It is challenging to vectorize many data types due to their complex structures, such as strings, trees, and graphs, with graph structures being one of the toughest as they generalize strings and trees.

Despite providing rich expressiveness, the complexity lying in graph structures becomes its Achilles' heel when applying machine learning methods for

graph data. To unleash the power of these two powerful tools, it is essential to represent the graph structure in forms that are able to be accepted by most popular machine learning methods, without losing considerable information while encoding the graphs. Since most machine learning algorithms rely on (dis)similarity measures between data, the problem boils down to measuring the (dis)similarity between graphs. Within this category lies the graph matching problem [Conte et al., 2004, Foggia et al., 2014, Livi and Rizzi, 2013, Yan et al., 2016, Wills and Meyer, 2020]. Graph similarity measures can be roughly grouped into two major categories: exact similarity and inexact similarity [Conte et al., 2004]. The former requires a strict correspondence between the two graphs being matched or between their subparts, such as graph isomorphism and subgraph isomorphism [Kobler et al., 2012]. Unfortunately, the exact similarity cannot be computed in polynomial time by these methods; hence, it is not practical for real-world data. For this reason, inexact similarity measures are commonly applied for graphs. An intuitive way to tackle the graph dissimilarity problem is to define distances and metrics directly on the space of graphs.

1.2.2 On spaces of graphs

We define a *space of graphs* as all possible graphs whose vertex and edge labels are defined by a label alphabet or domain. To operate machine learning methods on a space of graphs, some distance or metric is usually assigned to it. The cornerstone is defining an appropriate distance between two graphs. Ideally, we seek distances that are valid metrics, i.e., satisfying the conditions of non-negativity, identity of indiscernibles, symmetry, positive definiteness, and triangle inequality. This is the case of the *chemical distance*, which aims at minimizing edge discrepancies between two graphs [Kvasnička et al., 1991], and the *Chartrand-Kubiki-Shultz (CKS) distance*, which uses the pairwise shortest-path distances between vertices [Chartrand et al., 1998]. However, these distances are computationally expensive [Bento and Ioannidis, 2019].

Pseudometrics relax these conditions. An intuitive and widely-applied family is the *edit distance* [Garey and Johnson, 1979, Sanfeliu and Fu, 1983], which measures the dissimilarity between two graphs by the cost of transforming one graph to the other. Heuristics are proposed to approximate it with tolerable computational complexity with the possible relaxation of the triangle inequality

[Blumenthal et al., 2020]. An extension of the chemical distance is proposed in [Jain, 2016], which can tackle edge attributes. However, it is limited to the Frobenius norm and requires further relaxations for tractability. Other distances, such as the *maximum common subgraph distance* [Bunke and Shearer, 1998, Bunke, 1997] and the *reaction distance* [Koca et al., 2012], suffer the same issues as the above two [Bento and Ioannidis, 2019].

Literature on theoretical properties of graph spaces is limited [Jain, 2016], which includes graph spaces endowed with the maximum common subgraph distance [Hurshman and Janssen, 2015] and the ones endowed with an *optimal alignment kernel* [Jain and Obermayer, 2009], the latter being extended for tree-shape spaces [Feragen et al., 2010, Feragen et al., 2011, Feragen et al., 2012]. In [Jain, 2016], the authors propose a Graph Representation Theorem that induces an orbit space in which graphs can be considered as points. A related work is carried out in [Kolaczyk et al., 2020].

The limit of metric distances is well unfolded in [Grattarola et al., 2019]:

> *The use of metric distances, like graph alignment distances [Jain, 2016], only mitigates the problem (that application-specific graph distances often do not satisfy the identity of indiscernibles or the triangular inequality [Livi and Rizzi, 2013, Wilson et al., 2014]), as they are computationally intractable and hence not useful in practical applications. Therefore, a common approach is to embed graphs on a more conventional geometric space, such as the Euclidean one.*

As corroborated in [Grattarola et al., 2019] and many others, this limit drives researchers to study the graph embedding and graph kernels methods.

1.2.3 Graph embedding and kernels on graphs

Graph embedding and graph kernels lie in the category of inexact matching. These strategies consist in embedding the graphs into a space where computations can be easily carried out, such as combining embedded graphs or performing a classification or regression task.

Classic graph embedding maps graphs to finite-dimensional spaces, in which vectors are computed (explicitly in many cases) by encoding some information of the graphs. Techniques developed for graph embedding include matrix factorization, random walk, and deep learning [Goyal and Ferrara, 2018, Cai et al., 2018].

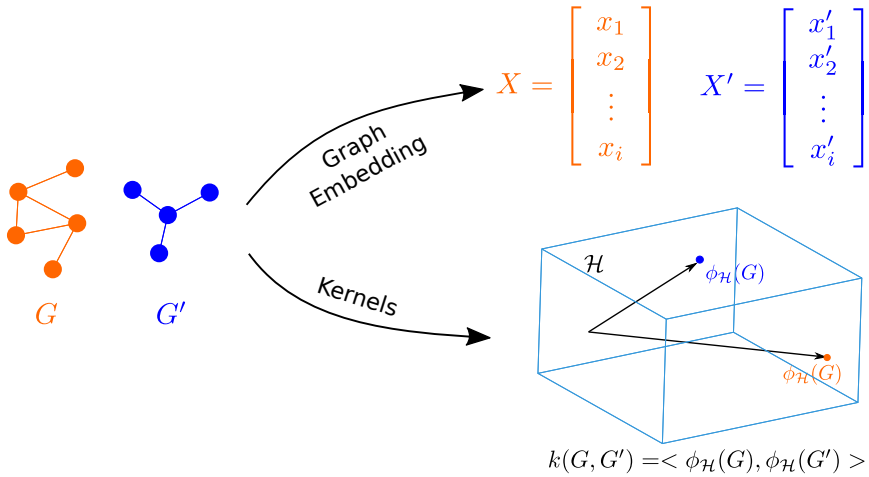

Figure 1.2 – Illustrative comparison between graph embedding and kernels for two arbitrary graphs G and G′. Through graph embedding, the two graphs are represented by two vectors, X and X′. By kernels, the two graphs are implicitly embedded by a function $\phi_{\mathcal{H}}(\cdot)$ into a Hilbert space $\mathcal{H}$, yielding $\phi_{\mathcal{H}}(G)$ and $\phi_{\mathcal{H}}(G')$; moreover, their inner product $< \phi_{\mathcal{H}}(G), \phi_{\mathcal{H}}(G') >$ is easily computed using a kernel function $k(G,G')$.

Due to the precision limitation of the graph embedding representations, a loss of information is anticipated. In contrast, kernels allow an implicit embedding by representing graphs in a possibly infinite-dimension feature space that relaxes the limitations on the encoded information. The two strategies are illustrated in Figure 1.2.

Indeed, as generalizations of the scalar inner product, kernels are natural similarity measures between data, expressed as inner products between elements in some feature space. By employing the kernel trick, one can evaluate the inner products in the feature space without explicitly describing each representation in that space [Schölkopf and Smola, 2002]. Kernels have been widely applied in machine learning with well-known popular methods, such as Support Vector Machines [Cortes and Vapnik, 1995]. Therefore, defining kernels between graphs is a powerful design to bridge the gap between machine learning and data encoded as graphs and furthermore provides solutions to answer the first question raised in Section 1.1.

Graph kernels can be constructed using both global and local information in graphs, by means of varied strategies. Among these include kernels based on sub-structures, information propagation kernels, and deep graph kernels [Ghosh et al., 2018]. Of particular interest are kernels based on sub-patterns/structures. When comparing graphs and analyzing their properties, the similarity principle has been widely investigated [Johnson and Maggiora, 1990]. It states that molecules having more common substructures turn to have more similar

properties. This principle can be generalized to other fields where data is modeled as graphs. It provides a theoretical support to construct graph kernels by studying graphs' substructures, which are also referred to as patterns.

1.3 Constructing graphs by solving the pre-image problem

Graph kernels bridge the gap between graph structures and kernel methods, allowing an implicit embedding of graphs into a kernel space. In the meantime, the counterpart of the (implicit) embedding with graph kernels is the so-called graph pre-image. The pre-image problem seeks the estimation of the mapping back, from the kernel space to the input space. This problem and its resolution continue to intrigue researchers. The graph pre-image problem consists in constructing a graph corresponding to some element in the kernel space, thus implying desired features and properties. Solving the graph pre-image problem provides answers to the second question raised in Section 1.1. Figure 1.3 illustrates the relationship between graph kernels and graph pre-image. Estimating graph pre-images lie in the interdiscipline of the pre-image problem in machine learning and the graph construction problem.

The pre-image is a non-linear reverse mapping of elements from the kernel space back to the input space. The pre-image problem has been primarily investigated on Euclidean spaces, with many applications including denoising and feature extraction with kernel principal component analysis [Honeine, 2012] and with kernel nonnegative matrix factorization [Zhu and Honeine, 2017]. It is also closely related to the dimensionality-reduction problem. The challenge of finding the pre-image lies in the fact that the reverse mapping does not exist in general and that most elements in the kernel space do not own valid pre-images in the input space. Consequently, various methods have been developed to approximate the solution, namely, to solve the pre-image problem. We refer interested readers to the tutorial [Honeine and Richard, 2011].

Solving the pre-image problem for graphs opens the door to many interesting applications, such as molecule synthesis and drug design. However, finding the pre-image as a graph inherits the difficulties of the traditional pre-image problem. Additionally, unlike inputs considered by the traditional pre-image problem (i.e.,

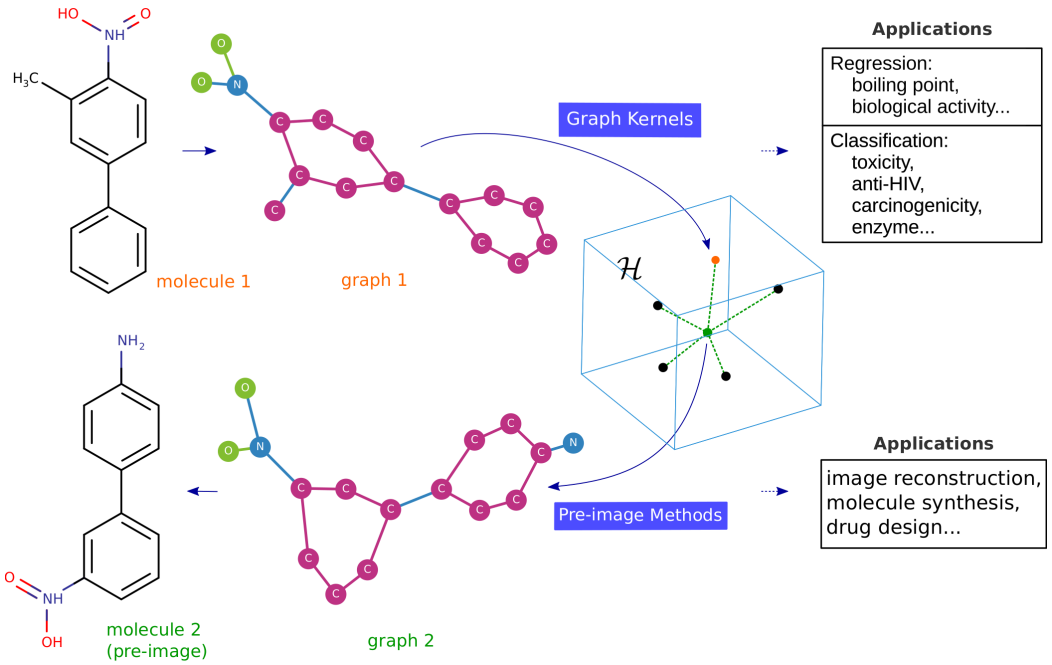

Figure 1.3 – Relationship between graph kernels and graph pre-images. The former maps graphs to a kernel space $\mathcal{H}$, while the latter provides the reverse procedure, by mapping elements from kernel space back to graphs.

vectors) usually lying in continuous spaces, graphs are structures where the numbers of vertices and edges can only be integers. The numbers of vertices and edges in a graph can be arbitrary and an edge can exist between any pair of vertices. Furthermore, multiple labels and attributes can be plugged into each vertex and edge in a graph. Given these structure features, the graph pre-image problem is more challenging to tackle. Several pioneer works to construct graph pre-images have been proposed [Bakır et al., 2004, Akutsu and Fukagawa, 2005, Nagamochi, 2009]; however, they are restricted to specific graph kernels or graph types, such as strings. Another set of algorithms is based on generative deep neural networks by modeling the distribution of implied features of the training data, such as the variational autoencoder, the recurrent long short-term memory (LSTM) network, and the generative adversarial network [Schneider et al., 2020, Hamilton, 2020]. It is thus interesting to propose generalized methods.

Metric learning for graph pre-image

Graph kernels do not directly operate in the graph space; thus, they are difficult to be used directly to construct graph pre-images. Defining tools and metrics that

operate in the graph space is necessary. In this category lie graph edit distances. Meanwhile, by measuring and adjusting the relationship between graph space and kernel space, one can construct graph pre-images according to the information in the kernel space. Metric learning fits perfectly for this purpose.

In the spirit of the no-free-lunch principle [Wolpert and Macready, 1997], metric learning consists in learning a (dis)similarity measure given a training set composed of data instances and associated targeted properties. For the classical metric learning where each data instance is encoded by a real-valued vector, the problem consists in learning a dissimilarity measure, which decreases (resp. increases) where the vectors have similar (resp. different) targeted properties. Many metric learning studies focus on Euclidean data, while only a few addresses this problem on structured data [Bellet et al., 2013]. A complete review for general structured data representation is given in [Ontañón, 2020]. When it comes to graph edit distances, a supervised metric learning strategy can help find optimized edit costs for the task of interest, with the guidance of the information lying in the target space.

In the spirit of metric learning, multidimensional scaling (MDS) seeks to embed data in a low-dimensional space by preserving pairwise distances [Cox and Cox, 2008]. Methods under the MDS framework have been proposed to solve the vector pre-image problem [Kwok and Tsang, 2004]. In Chapter 5, we propose the resolution of the graph pre-image problem considering the metric learning of distances in graphs spaces, with the graph edit distances [Jia et al., 2021].

1.4 Contributions of the research

This thesis studies metrics in graph and kernel spaces, to provide connections between these spaces in the perspective of solving the pre-image problem. The contributions of this thesis constitute these aspects.

The first contributions focus on graph kernels, with an emphasis on the ones based on linear patterns, and several ones based on non-linear patterns for comparison. Among them, the generalized random walk kernel is split into four different kernels due to the computing methods they use. A thorough investigation and comparison of these kernels are proposed, theoretically and experimentally. Considering the theoretical aspects, we examine their mathematical expressions with connections between them, and their computational complexities, as well as the strengths and weaknesses of each kernel. Moreover, we provide connections

to other kernels from the literature. In the exhaustive experimental analysis conducted in this thesis, each kernel is applied on various synthesized and well-known real-world datasets exhibiting different types of graphs, including labeled and unlabeled graphs, graphs with different numbers of vertices, graphs with different average vertex degrees, linear and non-linear graphs, etc. A thorough performance analysis including the comparison between these types of graphs is made considering both accuracy and computational time. This rigorous examination allows to provide suggestions to choose kernels according to the type of graph data at hand.

Since computational complexity is an Achilles' heel of graph kernels, we provide several strategies to address this critical issue, including parallelization, the trie data structure, and the FCSP (Fast Computation of Shortest Path) method that we extend to other kernels and edge comparison. All proposed strategies save orders of magnitudes of computing time and memory usage. Experiments are performed to demonstrate their relevance.

The second contributions concentrate on a metric in graph space, namely the graph edit distance (GED). When approximating GED by heuristics, the results may vary and additionally influence the task performance. Based on the revisiting of two representative heuristics, i.e., `bipartite` and `IPFP`, a study of the stability of the GED computation is carried out on well-known real-world datasets. A criterion to measure the stability is defined. A multi-start counterpart of `bipartite` (resp. `IPFP`) introduced in [Daller et al., 2018], namely m`bipartite` (resp. m`IPFP`), makes it possible to acquire better approximations. We examine how computation stability changes with the change of numbers of solutions used for `IPFP`. Effects of another factor on the stability are studied simultaneously, namely the ratio between edit costs on vertices and edges. The reasons that cause these influences are explained, and the instruction to choose the proper values of these factors are proposed.

A strategy to optimize edit costs of GED according to a particular task is proposed, and thus the use of predefined costs is avoided. The idea is to align the metric in the graph space (namely, the GED) to the target space in the spirit of metric learning. With this distance-preserving principle, an alternate iterative procedure is proposed, where the update on edit costs obtained by solving a constrained linear problem and a re-computation of the optimal edit paths according to the newly computed costs are performed alternately. The edit costs resulting from the optimization procedure can then be analyzed to understand how the graph space is structured. The relevance of the proposed method is demonstrated on two regres-

sion tasks, showing that the optimized costs lead to a lower prediction error compared to random generation, to expert costs, and to the state-of-the-art methods.

Our third contributions, which benefit from these studies on graph and kernel spaces, lie in the resolution of the graph pre-image problem. We propose a novel pre-image method for graphs. To this end, we bridge the gap between graph edit distances (GEDs) and graph kernels, which allows to uncover the relationship between graph space and kernel space. Borrowing the essence of the aforementioned metric learning, the edit costs of GED are tuned according to the corresponding distances in kernel space. The alternate iterative optimization strategy over edit costs and optimal edit paths is used. Consequently, the metrics of the two spaces are aligned, thus allowing to construct the graph pre-image by graph construction methods based on GEDs with the optimized edit costs. Specifically, a pre-image problem for the median graph of a graph set is addressed, based on the hypothesis that, benefiting from the alignment of the two metrics, the middle of the set of graphs corresponds to the mean of their embeddings in the kernel space, thus the median pre-image can be hereby approximated. We take advantage of recent advances in GEDs to solve this problem, where we revisit an iterative alternate minimization procedure introduced in [Boria et al., 2019] to generate median graphs. Experiments show that our method can generate better pre-images than others, and the optimized edit costs yield better results than random costs, and are competitive with expert costs.

The last contribution of this work is the implementation of an open-source Python library of machine learning tools for graphs, which is publicly available on GitHub[3]. The library mainly consists of four parts. In part one, all graph kernels based on linear patterns and two kernels based on non-linear patterns (namely the Weisfeiler-Lehman (WL) subtree kernel [Shervashidze et al., 2011, Morris et al., 2017] and the treelet kernel [Gaüzère et al., 2015b, Bougleux et al., 2012, Gaüzère et al., 2012]) are implemented, as well as the strategies to reduce the computational complexity of these kernels. The implementation of each kernel is able to tackle various types of graphs. Part two implements a framework for GED computation based on the C++ library `GEDLIB` [Blumenthal et al., 2019, Blumenthal et al., 2020]. The paradigm `LSAPE-GED`, which uses transformations to the linear sum assignment problem with error correction (LSAPE), and the heuristic `bipartite` are included. A median graph

[3]The GitHub link is `https://github.com/jajupmochi/graphkit-learn`.

estimator and a metric learning module for edit costs are coded based on them. In addition, a module that defines the Python interface of GEDLIB is integrated for the sake of computation speed, which is based on the GEDLIBPY library[4]. Part three contains methods for graph pre-image, such as our aforementioned method and the one based on random generation [Bakır et al., 2004]. Part four constitutes miscellaneous modules. One module fetches, loads, and manipulates graph datasets from public databases on the Internet; another one fits the computation of graph kernels to the pipeline of the `scikit-learn` library [Pedregosa et al., 2011] and performs the model selection and validation automatically.

1.5 Structure of the thesis

The rest of the thesis is organized as follows:

Chapter 2 introduces preliminaries for the thesis. Section 2.1 presents the basic concepts and definitions of graph theory. Section 2.2 walks through the mathematical background of kernel method, patterns-based kernels, graph kernels based on patterns, and graph pre-image. Then concepts concerning the graph edit distance are introduced in Section 2.3. Finally, Section 2.4 exhibits and categorizes graph datasets used in the thesis.

Chapter 3 presents the improvements and analyses on graph kernels. Graph kernels based on linear patterns and their connections to other kernels from the literature are first examined thoroughly in Sections 3.2, 3.3, and 3.4. Then two graph kernels based on non-linear patterns are presented in Section 3.5. Following that are three strategies to deal with the computational complexity of graph kernels, detailed in Section 3.6. Comprehensive experiments and analyses for these kernels on various types of graph datasets are performed in Section 3.7. At last, Section 3.8 concludes the work.

Chapter 4 focuses on graph edit distances. First, in Section 4.2, the paradigms to approximate GED are introduced, with the emphasis on two representative heuristics. Then, a study on the stability of the GED computation is carried out in Section 4.3. Next, a metric learning approach to estimate the graph edit costs for regression is proposed and evaluated experimentally in Section 4.4. The work is concluded in Section 4.5.

Chapter 5 proposes a graph pre-image method based on GEDs. Section 5.1

[4]The GitHub link of GEDLIBPY is: `https://github.com/Ryurin/gedlibpy`.

presents state-of-the-art pre-image methods and graph generations. Section 5.2 provides the problem formulation and Section 5.3 presents the proposed method in two folds, learning edit costs for GEDs by the distances in kernel space (Section 5.3.1) and inferring the graph pre-image (Section 5.3.2). Section 5.4 gives experiments and analyses. Finally, Section 5.5 concludes the work.

Chapter 6 reveals the implementation details of our Python library on machine learning for graphs: `graphkit-learn`. A brief introduction is given in Section 6.1 and the overall architecture is described in Section 6.2. Then from Section 6.3 to 6.7, we present details of the `graphkit-learn` library, respectively implementations of graph data processing, graph kernels, graph edit distances, graph pre-image methods, and auxiliary tools. Comparison with other libraries is performed during the presentation. Example codes to use the library are presented in corresponding sections. Section 6.8 gives the conclusion and future work.

Chapter 7 concludes the thesis and provides perspectives on future work.

Chapter 2

Preliminaries

Contents

This chapter introduces notations and terminologies that will be used in this thesis. We first define basic concepts of graph theory, and then the main notions in kernel-based machine learning, including graph kernels and the pre-image problem. After that, for the purpose of proposing a graph pre-image method, the graph edit distance is presented. Real-world graph datasets are introduced at last.

2.1 Basic concepts of graph theory

In this section, we introduce notations and terminologies of graph theory that will be used in this thesis. For more details, we refer interested readers to [West et al., 2001]. First, we clarify definitions of different types of graphs. Figure 2.1 shows the types of graphs mentioned below. The indicator function $\mathbf{1}_{\mathrm{A}} : \mathrm{X} \to \{0,1\}$ is defined as $\mathbf{1}_{\mathrm{A}}(x) = 1$ if $x \in \mathrm{A}$, and 0 otherwise. Let $|\cdot|$ denote the cardinality of a set, namely the number of its elements.

Definition 2.1 (graph)**.** *A graph* G *is defined by an ordered pair of disjoint sets* (V, E)*, such that* V *corresponds to a finite set of vertices and* $\mathrm{E} \subset \mathrm{V} \times \mathrm{V}$ *corresponds to a set of edges. A vertex* $u \in \mathrm{V}$ *is adjacent to another vertex* $v \in \mathrm{V}$ *if* $(u, v) \in \mathrm{E}$*. A vertex is also called a node in literature. We denote the number of graph vertices as* n*, i.e.,* $n = |\mathrm{V}|$*, which is also called the order of the graph, and the number of graph edges as* m*, i.e.,* $m = |\mathrm{E}|$*.*

Definition 2.2 (labeled and unlabeled graph)**.** *A labeled graph* G *is a graph that has additionally a set of labels* L *along with a labeling function* ℓ *that assigns a label to each edge and/or vertex. In edge-labeled graphs, the labeling function* $\ell_e : \mathrm{E} \to \mathrm{L}$ *assigns labels to edges only; in vertex-labeled graphs, the labeling function* $\ell_v : \mathrm{V} \to \mathrm{L}$ *assigns labels to vertices only; in fully-labeled graphs, the labeling function* $\ell_f : \mathrm{V} \cup \mathrm{E} \to \mathrm{L}$ *assigns labels to both vertices and edges. Unlabeled graphs have no such labeling function.*

Definition 2.3 (symbolic and non-symbolic labels)**.** *Labels defined in Definition 2.2 can be either symbolic or non-symbolic, for vertices and/or edges. A symbolic label* $\ell \in \{\ell_i \mid i \in \mathbb{N}_+\}$ *is a discrete symbol, such as the type of atoms or chemical bonds; a non-symbolic label* $\ell \in \mathbb{R}$ *is a continuous value. Specific similarity measures between two labels* ℓ_i *and* ℓ_j *are usually given due to this difference. For instance, two symbolic labels* ℓ_i *and* ℓ_j *are considered equal as long as they are the same and unequal otherwise (namely, the Kronecker delta function; see Definition 2.4 below),*

while non-symbolic labels are compared by continuous measures, such as the Gaussian kernel (see Definition 2.5 below). Both symbolic and non-symbolic labels can be one-dimensional or multi-dimensional vectors. A label is also referred to as an attribute.

Two similarity measures are used between labeled vertices and edges: Kronecker delta function for symbolic labels and Gaussian kernel for non-symbolic labels.

Definition 2.4 (Kronecker delta function)**.** *The Kronecker delta function between two labels ℓ_i and ℓ_j is defined as*

$$k(\ell_i, \ell_j) = \delta_{\ell_i \ell_j} = \begin{cases} 1, & \text{if} \quad \ell_i = \ell_j; \\ 0, & \text{if} \quad \ell_i \neq \ell_j. \end{cases} \tag{2.1}$$

For the sake of conciseness, it is denoted as the delta function $\delta_{\ell_i \ell_j}$.

Definition 2.5 (Gaussian kernel)**.** *The Gaussian kernel between labels ℓ_i and ℓ_j is defined as*

$$k(\ell_i, \ell_j) = \exp\left(-\frac{\|\ell_i - \ell_j\|^2}{2\sigma^2}\right), \tag{2.2}$$

where σ is the tunable bandwidth parameter.

Definition 2.6 (directed and undirected graph)**.** *A directed graph is a graph whose edges are directed from one vertex to another, where the edge set* E *consists of ordered pairs of vertices (u, v). An ordered pair (u, v) is said to be an edge directed from u to v, namely an edge beginning at u and ending at v. In contrast, a graph where the edges are bidirectional is called an undirected graph, i.e., if $(u, v) \in \mathrm{E}$, then $(v, u) \in \mathrm{E}$.*

Graph substructures, such as walks, paths, and cycles, allow to describe graphs, thus providing elegant ways to construct graph kernels. The concepts of the adjacency matrix, neighbors, and degrees of vertices are fundamental for building these kernels.

Definition 2.7 (neighbors and degree)**.** *In a graph $\mathrm{G} = (\mathrm{V}, \mathrm{E})$, a neighbor of a vertex $v \in \mathrm{V}$ is a vertex u that meets the condition $(u, v) \in \mathrm{E}$. We denote the set of all neighbors of v as $\mathcal{N}(v)$. If* G *is undirected, the degree of a vertex $v \in \mathrm{V}$ is the number of these neighbors, namely* $\deg(v) = |\{u \in \mathrm{V} \mid (u, v) \in \mathrm{E}\}|$*; if* G *is directed, then* $\deg^-(v) = |\{u^- \in \mathrm{V} \mid (u^-, v) \in \mathrm{E}\}|$ *is called the indegree of vertex v,* $\deg^+(v) = |\{u^+ \in \mathrm{V} \mid (v, u^+) \in \mathrm{E}\}|$ *is*

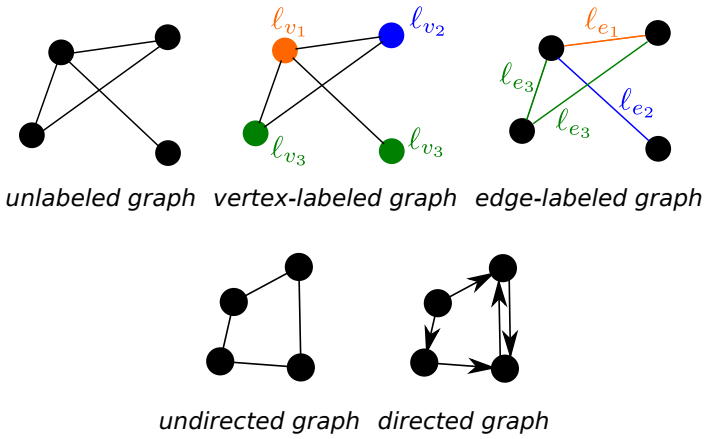

Figure 2.1 – The different types of graphs. In vertex- and edge-labeled graphs, vertices and edges with different labels are distinguished by color in the figure.

the outdegree of v, and the degree of v is the sum of its indegree and outdegree. The degree of the graph, denoted by d, is the largest vertex degree of all its vertices.

Definition 2.8 (adjacency matrix)**.** *The adjacency matrix of an n-vertex graph $\mathrm{G} = (\mathrm{V}, \mathrm{E})$ is an $n \times n$ matrix $\mathrm{A}(\mathrm{G})$ with entries $a_{ij} = \mathbf{1}_{\mathrm{E}}((v_i, v_j))$, namely $a_{ij} = 1$ if $(v_i, v_j) \in \mathrm{E}$ and 0 otherwise.*

Definition 2.9 (walks, paths and cycles)**.** *For a graph $\mathrm{G} = (\mathrm{V}, \mathrm{E})$, a walk of length h is a sequence of vertices $\mathrm{W} = (v_1, v_2, \ldots, v_{h+1})$ where $(v_i, v_{i+1}) \in \mathrm{E}$ for any $i \in \{1, 2, \ldots, h\}$. The length of a walk W is defined as its number of edges h. If each vertex appears only once in W, then W is a path. A walk with $v_1 = v_{h+1}$ is called a cycle. Note that when $h = 0$, a walk or path is a single vertex without edges.*

Definition 2.10 (label sequences)**.** *The (contiguous) label sequence of a length h walk/path W of a fully-labeled graph is defined as*

$$s = (\ell_v(v_1), \ell_e((v_1, v_2)), \ell_v(v_2), \ell_e((v_2, v_3)), \ldots, \ell_v(v_{h+1})).$$

For a vertex-labeled or edge-labeled graph, the label sequence of W is constructed by removing all edge labels $\ell_e((v_i, v_j))$ or vertex labels $\ell_v(v_i)$ in s, respectively.

2.2 Kernel methods and pre-image

In this section, we define terminologies about the graph kernel and its reverse procedure, i.e., graph pre-image. First, formal definitions of a kernel and Gram matrix

are introduced. Then the kernel trick is presented to show its ability of evaluating inner products in some feature space. To this end, two classical kernel-based machine learning methods are presented next, kernel ridge regression and support vector machines for classification, and applied in this thesis to assess the relevance of graphs. With the help of these definitions, we are able to provide formal definitions of the pre-image. Then the related techniques are introduced. A method to approximate pre-image, namely the multi-dimensional scaling-based technique, is presented, which is closely related to the methods we proposed in this thesis.

2.2.1 Kernel methods

Let $\mathcal{X}$ denote the input space. A positive semi-definite kernel is defined as follows:

Definition 2.11 (positive semi-definite kernel)**.** *A positive semi-definite kernel defined on $\mathcal{X}$ is a symmetric bilinear function $k : \mathcal{X}^2 \to \mathbb{R}$ that fulfills the condition $\sum_{i=1}^{n} \sum_{j=1}^{n} c_i\, c_j\, k(x_i, x_j) \geqslant 0$, for all $x_1, \ldots, x_n \in \mathcal{X}$ and $c_1, \ldots, c_n \in \mathbb{R}$.*

Positive semi-definite kernels have some general properties. Of particular interest, the products and sums, weighted with non-negative coefficients, of a set of positive semi-definite kernels, are also positive semi-definite kernels. Moreover, any limit $\lim_{n\to\infty} k_n$ of a sequence of positive semi-definite kernels k_n is also a positive semi-definite kernel [Schölkopf and Smola, 2002]. These properties are useful for constructing graph kernels. The first property is applied for all graph kernels discussed in this thesis, and the second one for the common walk kernel and the generalized random walk kernel.

Mercer's theorem states that any positive semi-definite kernel corresponds to an inner product in some Hilbert space [Mercer, 1909], namely, for all $(x_i, x_j) \in \mathcal{X}^2$, there exists a space $\mathcal{H}$ and an embedding function $\phi_{\mathcal{H}} : \mathcal{X} \to \mathcal{H}$ such that:

$$k(x_i, x_j) = \langle \phi_{\mathcal{H}}(x_i), \phi_{\mathcal{H}}(x_j) \rangle_{\mathcal{H}}. \tag{2.3}$$

The positive semi-definiteness of a kernel is a sufficient condition to the existence of this function. For the sake of conciseness, positive semi-definite kernels are simply denoted as *kernels* in this thesis. Table 2.1 lists several commonly used kernels on vector representations.

Kernel-based methods in machine learning take advantage of Mercer's theorem, in order to transform conventional linear models into non-linear ones, by

Table 2.1 – Commonly used kernels between vectors $\boldsymbol{x}$ and $\boldsymbol{y}$.

Kernels	**Expressions**
Kronecker delta	$k(\boldsymbol{x},\boldsymbol{y}) = \delta_{\boldsymbol{x}\boldsymbol{y}}$
Linear	$k(\boldsymbol{x},\boldsymbol{y}) = \boldsymbol{x}^\top \boldsymbol{y}$
Monomial	$k(\boldsymbol{x},\boldsymbol{y}) = (\boldsymbol{x}^\top \boldsymbol{y})^d, d \in \mathbb{N}$
Polynomial	$k(\boldsymbol{x},\boldsymbol{y}) = (\boldsymbol{x}^\top \boldsymbol{y})^d + c, c \in \mathbb{R}, d \in \mathbb{N}$
Exponential	$k(\boldsymbol{x},\boldsymbol{y}) = e^{\frac{\boldsymbol{x}^\top \boldsymbol{y}}{2\sigma^2}}, \sigma \in \mathbb{R}_+$
Cosine	$k(\boldsymbol{x},\boldsymbol{y}) = \frac{\boldsymbol{x}^\top \boldsymbol{y}}{\|\|\boldsymbol{x}\|\|\|\|\boldsymbol{y}\|\|}$
Intersection	$k(\boldsymbol{x},\boldsymbol{y}) = \sum_{i=1}^{N} \min(x_i, y_i)$
Sigmoid	$k(\boldsymbol{x},\boldsymbol{y}) = \tanh(\gamma \boldsymbol{x}^\top \boldsymbol{y} + c_0), \gamma, c_0 \in \mathbb{R}_+$
Gaussian/RBF	$k(\boldsymbol{x},\boldsymbol{y}) = e^{-\frac{\|\|\boldsymbol{x}-\boldsymbol{y}\|\|^2}{2\sigma^2}}, \sigma \in \mathbb{R}_+$
Laplacian	$k(\boldsymbol{x},\boldsymbol{y}) = e^{-\frac{\|\|\boldsymbol{x}-\boldsymbol{y}\|\|}{2\sigma^2}}, \sigma \in \mathbb{R}_+$
Multiquadratic	$k(\boldsymbol{x},\boldsymbol{y}) = \sqrt{\|\|\boldsymbol{x}-\boldsymbol{y}\|\|^2 + c}, c \in \mathbb{R}_+$
Inverse multiquadratic	$k(\boldsymbol{x},\boldsymbol{y}) = \frac{1}{\sqrt{\|\|\boldsymbol{x}-\boldsymbol{y}\|\|^2 + c}}, c \in \mathbb{R}_+$

replacing classical inner products between data with a non-linear kernel. Let $\mathrm{X} = \{x_1, x_2, \ldots, x_N\}$ be the finite dataset of N samples available for training the machine learning method, with associated data labels $\{y_1, y_2, \ldots, y_N\}$ where $y_i \in \{-1, +1\}$ for binary classification and $y_i \in \mathbb{R}$ for regression (extensions to multiclass classification and vector output are straightforward [Honeine et al., 2013]). It turns out that one does not need access to the raw data X, but only the evaluation of the kernel on all pairs of the data, namely the Gram matrix K is sufficient. This approach, called the *kernel trick*, is often computationally efficient, compared to the direct computation over the raw data. A Gram matrix K associated to a kernel k for a training set X is an $N \times N$ matrix defined as $\mathrm{K}_{i,j} = k(x_i, x_j)$, for all $(x_i, x_j) \in \mathscr{X}^2$.

Kernel-based machine learning relies on regularized cost functions, often of the form $\arg\min_{\psi \in \mathscr{H}} \sum_{i=1}^{N} c(y_i, \psi(x_i)) + \lambda \|\psi\|^2$, for some cost function c and positive regularization parameter λ. The generalized representer theorem [Schölkopf et al., 2001] states that the optimal solution has the form $\psi(x) = \sum_{i=1}^{N} \omega_i k(x_i, x)$ for any $x \in \mathscr{X}$, where k is the kernel inducing the Hilbert space $\mathscr{H}$. The coefficients ω_i's can be computed only using the Gram matrix, and can be used to represent x given the decomposition elements x_i, for $i = 1, \ldots, N$. For example, the kernel ridge regression corresponds to the square loss $c(y_i, \psi(x_i)) = (y_i - \psi(x_i))^2$, which leads to $[\omega_1 \ldots \omega_N]^\top = (\mathrm{K} + \lambda \mathrm{I})^{-1} [y_1 \ldots y_N]^\top$ [Murphy, 2012]. Support Vector Machines (SVM) for classification consider the hinge loss $c(y_i, \psi(x_i)) =$

$\max(0, 1-y_i\psi(x_i))$, and the optimal coefficients are efficiently obtained by quadratic programming algorithms [Boser et al., 1992].

Kernel-based methods provide an elegant and powerful framework in machine learning for any input space, without the need to exhibit the data or optimize in that space, as long as one can define a kernel on it. Besides conventional kernels, such as the Gaussian kernel for vector spaces, kernels can be engineered by combining other valid kernels, using additive or multiplicative rules. Of particular interest in kernel engineering are R-convolution kernels [Haussler, 1999], which provide the foundation of kernels based on bags of patterns and can be regarded as the cornerstones to engineer graph kernels using graph patterns. For more details on kernel methods, we refer interested readers to [Shawe-Taylor and Cristianini, 2004, Schölkopf and Smola, 2002].

2.2.2 Graph kernels

R-convolution kernels propose a way to measure similarity between two objects, by measuring the similarities between their substructures [Haussler, 1999]. Suppose that each sample $x_i \in \mathcal{X}$ has a composite structure, namely described by its "parts" $(x_{i1}, x_{i2}, \ldots, x_{i\mathrm{D}}) \in \mathcal{X}_1 \times \mathcal{X}_2 \times \cdots \times \mathcal{X}_\mathrm{D}$, for some positive integer D. Since multiple decompositions could exist, let $\mathrm{R}(x)$ denotes all possible decompositions of x, namely $\mathrm{R}(x) = \left\{\boldsymbol{x} \in \mathcal{X}_1 \times \mathcal{X}_2 \times \cdots \times \mathcal{X}_\mathrm{D} \;\middle|\; \boldsymbol{x} \text{ is a decomposition of } x\right\}$. For each decomposition space $\mathcal{X}_d$, let k_d be a kernel defined on this space to measure the similarity on the d-th part. Then, a generalized convolution kernel, called R-convolution kernel, between any two samples x_i and x_j from $\mathcal{X}$ is defined as:

$$k(x_i, x_j) = \sum_{\substack{(x_{i1},\ldots,x_{i\mathrm{D}})\in\mathrm{R}(x_i)\\(x_{j1},\ldots,x_{j\mathrm{D}})\in\mathrm{R}(x_j)}} \prod_{d=1}^{\mathrm{D}} k_d(x_{id}, x_{jd}). \tag{2.4}$$

Figure 2.2 illustrates an example of computing the R-convolution kernel between two bunches of bananas with respect to the calories that they contain. First, each bunch is decomposed into individual bananas, which corresponds to the x_{id}'s and x_{jd}'s in (2.4). Next, the linear kernel, namely k_d in (2.4), is used between the calories of each pair of bananas in two bunches. At last, the computed values are summed up, which serves as a similarity measure between the two bunches. Images of bananas are from the Internet.

Graph kernels are kernels defined on graphs. For a given graph kernel, $k(\mathrm{G}_i, \mathrm{G}_j)$

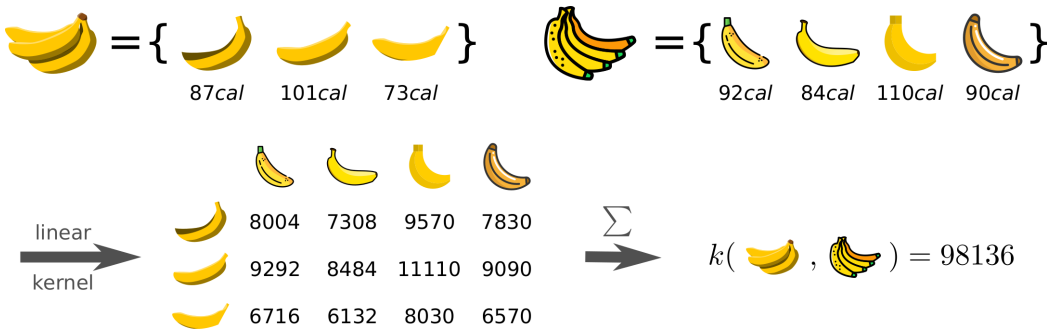

Figure 2.2 – An example of an R-convolution kernel.

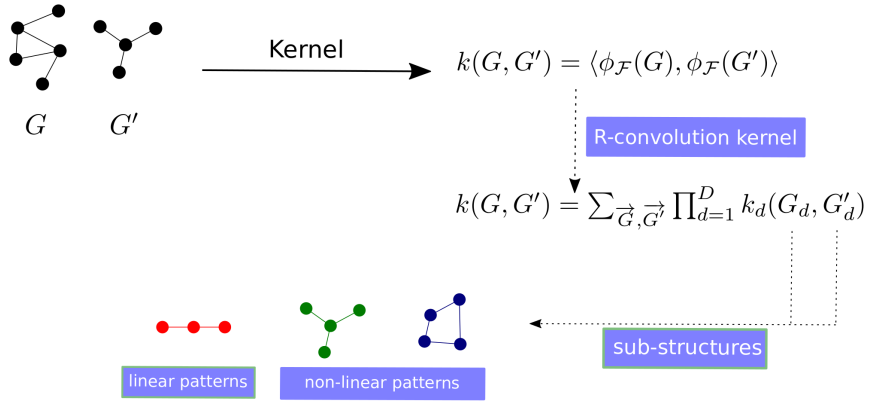

Figure 2.3 – Graph kernels based on sub-structures of graphs, where G_d is the d-th sub-structure in G and k_d is a kernel defined between sub-structures.

can be regarded as the inner product between the two mapped graphs, $\phi(G_i)$ and $\phi(G_j)$, in the kernel space $\mathcal{H}$, namely $k(G_i, G_j) = \langle \phi(G_i), \phi(G_j) \rangle_{\mathcal{H}}$.

By simply changing the decomposition, many different kernels can be obtained from the R-convolution kernel. When it comes to graphs, it is natural to decompose them into smaller substructures, such as paths, walks, trees and cycles, and build graph kernels based on similarities between those components, as it is easier to compare them. Figure 2.3 illustrates this procedure. The kernels differ mainly in the ways of composition and the similarity measures used to compare the substructures.

2.2.3 The pre-image problem

With the aforementioned definitions, we can now define the pre-image problem:

Definition 2.12 (the pre-image problem)**.** *Given a kernel $k : \mathcal{X}^2 \to \mathbb{R}$ with its corresponding mapping $\phi(\cdot)$, the pre-image problem of any $\psi \in \mathcal{H}$ consists in finding*

the element $x \in \mathcal{X}$ whose image, under the map $\phi(\cdot)$, is ψ, namely $\psi = \phi(x)$. Here, x is called the pre-image of ψ, namely $x = \phi^{-1}(\psi)$ where $\phi^{-1}(\cdot) : \mathcal{H} \to \mathcal{X}$ denotes the inverse mapping of $\phi(\cdot)$.

Although $\phi(\cdot)$ can be implicit and straightforward thanks to the kernel trick, the reverse map is difficult to compute in general. In most cases, finding the pre-image is an ill-posed problem, namely, at least one of the following conditions is not satisfied [Honeine and Richard, 2011]:

- There is a solution;
- The solution is unique;
- The solution continuously depends on the data (the stability condition).

Indeed, most $\psi \in \mathcal{H}$ do not have a valid pre-image, as the dimension of the kernel space is usually much higher than the input space. It may not be unique, even if it exists. Therefore, the pre-image problem generally consists in estimating an approximate solution, namely $\widehat{x}$ such that $\phi(\widehat{x}) \approx \psi$. Since $\psi = \sum_{i=1}^{N} \alpha_i \phi(x_i)$ according to the generalized representer theorem, then the pre-image problem consists in estimating $\widehat{x}$ such that $\phi(\widehat{x}) \approx \sum_{i=1}^{N} \alpha_i \phi(x_i)$. Many strategies to estimate pre-images have been proposed in the literature. They can be grouped into two categories, methods relying on minimizing the gap between $\phi(\widehat{x})$ and ψ with gradient-based techniques, and methods relying on dimension reduction techniques [Honeine and Richard, 2011].

The first class of pre-image methods rely on minimizing the distance between $\phi(\widehat{x})$ and $\sum_{i=1}^{N} \alpha_i \phi(x_i)$, generally by solving an optimization problems of the form:

$$\widehat{x} = \underset{x \in \mathcal{X}}{\operatorname{argmin}} \Big|\Big| \sum_{i=1}^{N} \alpha_i \phi(x_i) - \phi(x) \Big|\Big|_{\mathcal{H}}^2. \tag{2.5}$$

After expanding the objective function, the terms independent of x can be dropped. Using the kernel trick, the other two terms can be expressed in the form of an inner product, namely the kernel function. The problem is therefore reformed as

$$\widehat{x} = \underset{x \in \mathcal{X}}{\operatorname{argmin}} \Big(k(x,x) - 2 \sum_{i=1}^{N} \alpha_i k(x_i, x) \Big). \tag{2.6}$$

Gradient descent techniques rely on computing the gradient of the objective function given in (2.6) and nullifying it with an iterative update [Honeine and Richard, 2011]. However, the local minima may be reached as the

optimization problem is non-linear and non-convex. To tackle this issue, the proposed algorithms are often run multiple times with different starting values. The fixed-point iteration method takes advantage of the closed-form of the gradient of the objective function given in (2.6) for most kernels, by setting its expression to zero [Abrahamsen and Hansen, 2009]. Besides local minima, this technique also suffers from numerical instabilities (e.g., when the denominator of the iterative expression tends to zero). A regularized solution to prevent this phenomenon is proposed in [Abrahamsen and Hansen, 2009]. The bright side is that the consequential pre-image lies in the span of the given data, which reduces the search space.

The second class of methods relies on connecting the pre-image problem with dimension reduction problems. Indeed, the pre-image mapping can be viewed as embedding elements from a high-dimensional space $\mathscr{H}$ into a lower-dimensional space $\mathscr{X}$. In the same spirit of multi-dimensional scaling (MDS) [Cox and Cox, 2008], the principle of preserving pairwise distances in both spaces was used to solve the pre-image problem in [Kwok and Tsang, 2004]. This method aims at finding the pre-image that minimizes the difference between distances in the input space and their counterparts in the kernel space. To solve this problem, one method is to minimize the mean square error between these distances, namely by solving the following optimization problem:

$$\widehat{x} = \underset{x \in \mathscr{X}}{\operatorname{argmin}} \sum_{i=1}^{N} \left(||x - x_i||^2 - ||\psi - \phi(x_i)||^2_{\mathscr{H}} \right)^2, \tag{2.7}$$

where $||x - x_i||$ and $||\psi - \phi(x_i)||_{\mathscr{H}}$ are respectively the distances in the input and kernel spaces. When dealing with Euclidean input spaces, methods such as the fixed-point iteration are employed to solve this problem. Another way is to consider separately each distance, where the pre-image is eventually obtained by a least-square solution [Honeine and Richard, 2011]. Besides the distance-preserving method in MDS, the conformal map approach proposed in [Honeine and Richard, 2009] seeks to preserve inner product measures as well as the angular measures, by defining a coordinate system in the kernel space that is in isometry with the input space.

A thorough survey of the pre-image methods and their applications can be found in [Honeine and Richard, 2011]. While there are many diverse applications, including feature extraction, image denoising, and auto-localization in wireless sensor networks, most of the work on the pre-image problem has been addressing vector data, namely when input space is a Euclidean space.

The graph pre-image problem can be defined in a similar way. Defining a kernel on a space of graphs $\mathcal{G}$, the graph pre-image problem of some $\psi \in \mathcal{H}$ consists in estimating a graph $\widehat{G} \in \mathcal{G}$, such that $\phi(\widehat{G}) \approx \psi$. In order to consider a distance-preserving formalism in the same spirit of MDS and (2.7), one needs first to define a distance in the graph space $\mathcal{G}$ (with graph edit distance described in the next section), and then optimize on this space (see Chapter 5).

2.3 Graph edit distances

Definition 2.13 (Graph Edit Distance (GED))**.** *The Graph Edit Distance (GED) between two graphs* $G_1 = (V_1, E_1)$ *and* $G_2 = (V_2, E_2)$ *is defined as the cost of a minimal transformation [Riesen, 2015]:*

$$ged(G_1, G_2) = \min_{\pi \in \Pi(G_1, G_2)} C(\pi, G_1, G_2), \tag{2.8}$$

where (2.8)*,* $\pi(G_1, G_2)$ *is a transformation from* G_1 *to* G_2*, including a series of six elementary operations: removing a vertex or an edge, inserting a vertex or an edge, and substituting a label of a vertex or an edge by another label. This sequence of edit operations is known as an edit path from* G_1 *to* G_2*.*

As shown in [Bougleux et al., 2015a], the edit path π can be considered as a mapping function. Let ε be a dummy symbol denoting dummy vertices and edges as well as their labels. First, $\pi : V_1 \rightarrow V_2 \cup \varepsilon$ encodes the mapping of G_1's vertices to vertices of G_2. If a vertex v_i is deleted, we have $\pi(v_i) = \varepsilon$. Similarly, we denote as π^{-1} the mapping of V_2 to $V_1 \cup \varepsilon$. For the same edit path, we have thus $\pi(v_i) = v_j \Rightarrow \pi^{-1}(v_j) = v_i$. Given a mapping, the cost associated to vertex operations of an edit path represented by π is given by:

$$C_v(\pi, G_1, G_2) = \sum_{\substack{v \in V_2 \\ \pi^{-1}(v) \notin V_1}} c_{vfi}(\varepsilon, v) + \sum_{\substack{u \in V_1 \\ \pi(u) \notin V_2}} c_{vfr}(u, \varepsilon) + \sum_{\substack{u \in V_1 \\ \pi(u) \in V_2}} c_{vfs}(u, \pi(u)), \tag{2.9}$$

where $c_{vfr}, c_{vfi}, c_{vfs}$ are edit cost functions to perform respectively the removal, insertion, and substitution edit operations on vertices. The cost associated with

edge operations is defined as:

$$\mathrm{C}_e(\pi,\mathrm{G}_1,\mathrm{G}_2) = \sum_{\substack{e=(v_i,v_j)\in \mathrm{E}_2 \mid \\ \pi^{-1}(v_i)=\varepsilon \vee \\ \pi^{-1}(v_j)=\varepsilon \vee \\ (\pi^{-1}(v_i),\pi^{-1}(v_j))\notin \mathrm{E}_1}} c_{efi}(\varepsilon,e) + \sum_{\substack{e=(v_i,v_j)\in \mathrm{E}_1 \mid \\ \pi(v_i)=\varepsilon \vee \\ \pi(v_j)=\varepsilon \vee \\ (\pi(v_i),\pi(v_j))\notin \mathrm{E}_2}} c_{efr}(e,\varepsilon) + \sum_{\substack{e=(v_i,v_j)\in \mathrm{E}_1 \mid \\ \pi(v_i)\neq\varepsilon \wedge \\ \pi(v_j)\neq\varepsilon \wedge \\ (\pi(v_i),\pi(v_j))\in \mathrm{E}_2}} c_{efs}(e,\pi(e)), \quad (2.10)$$

where $c_{efr}, c_{efi}, c_{efs}$ are edit cost functions to perform respectively the removal, insertion, and substitution edit operations on edges. The final cost is then given by:

$$\mathrm{C}(\pi,\mathrm{G}_1,\mathrm{G}_2) = \mathrm{C}_v(\pi,\mathrm{G}_1,\mathrm{G}_2) + \mathrm{C}_e(\pi,\mathrm{G}_1,\mathrm{G}_2). \quad (2.11)$$

For graphs with symbolic labels, the delta function (2.1) is normally used for the comparison between labels, namely, the cost is equal to 1 if the two labels are the same and 0 otherwise. Then the edit cost functions can be defined by constants, namely:

$$\begin{cases} c_{vfi}(\varepsilon,v) = c_{vi}, \;\; c_{vfr}(v,\varepsilon) = c_{vr}, \;\; c_{vfs}(u,v) = c_{vs}, \\ c_{efi}(\varepsilon,e) = c_{ei}, \;\; c_{efr}(e,\varepsilon) = c_{er}, \;\; c_{efs}(e,f) = c_{es}. \end{cases} \quad (2.12)$$

The cost associated to vertex and edge operations is then respectively given by:

$$\mathrm{C}_v(\pi,\mathrm{G}_1,\mathrm{G}_2) = \sum_{\substack{v\in \mathrm{V}_2 \\ \pi^{-1}(v)=\varepsilon}} c_{vi} + \sum_{\substack{v\in \mathrm{V}_1 \\ \pi(v)=\varepsilon}} c_{vr} + \sum_{\substack{v\in \mathrm{V}_1 \\ \pi(v)\neq\varepsilon}} c_{vs}, \quad (2.13)$$

and

$$\mathrm{C}_e(\pi,\mathrm{G}_1,\mathrm{G}_2) = \sum_{\substack{e=(v_i,v_j)\in \mathrm{E}_2 \mid \\ \pi^{-1}(v_i)=\varepsilon \vee \\ \pi^{-1}(v_j)=\varepsilon \vee \\ (\pi^{-1}(v_i),\pi^{-1}(v_j))\notin \mathrm{E}_1}} c_{ei} + \sum_{\substack{e=(v_i,v_j)\in \mathrm{E}_1 \mid \\ \pi(v_i)=\varepsilon \vee \\ \pi(v_j)=\varepsilon \vee \\ (\pi(v_i),\pi(v_j))\notin \mathrm{E}_2}} c_{er} + \sum_{\substack{e=(v_i,v_j)\in \mathrm{E}_1 \mid \\ \pi(v_i)\neq\varepsilon \wedge \\ \pi(v_j)\neq\varepsilon \wedge \\ (\pi(v_i),\pi(v_j))\in \mathrm{E}_2}} c_{es}. \quad (2.14)$$

On the other hand, according to [Kaspar and Horst, 2010], the edit cost functions for graphs with non-symbolic labels can be defined as:

$$\begin{cases} c_{vfi}(\varepsilon,v) = c_{vi}, \;\; c_{efi}(\varepsilon,e) = c_{ei}, \\ c_{vfr}(v,\varepsilon) = c_{vr}, \;\; c_{efr}(e,\varepsilon) = c_{er}, \\ c_{vfs}(u,v) = c_{vs}\|\ell_v(u) - \ell_v(v)\|, \\ c_{efs}(e,f) = c_{es}\|\ell_e(e) - \ell_e(f)\|. \end{cases} \quad (2.15)$$

In (2.12) and (2.15), $c_{vr}, c_{vi}, c_{vs}, c_{er}, c_{ei}, c_{es}$ are the constant edit costs, namely coefficients applied for vertex removal, vertex insertion, vertex substitution, edge removal, edge insertion and edge substitution, respectively; $\|\cdot\|$ represents a distance measure between non-symbolic labels, such as the Euclidean norm. We denote by $\boldsymbol{c} = [c_{vr}, c_{vi}, c_{vs}, c_{er}, c_{ei}, c_{es}]^\top$ the edit costs vector.

2.4 Real-world graph datasets

Graph datasets have been springing up in recent years, while public collections of datasets containing various types of graphs being published. For instance, *GREYC's Chemistry database*[1] contains various chemical datasets of molecules, each of them being either a classification or regression problem. *TUDataset*[2] is an expanding database that collects benchmark datasets for the evaluation of graph kernels [Morris et al., 2020]. Graphs in it come from various domains, such as bioinformatics, chemoinformatics, computer vision, social networks, and synthetic graphs. *The Open Graph Benchmark (OGB)*[3] is a collection of realistic, large-scale, and diverse benchmark datasets for machine learning on graphs, with tools to automatically download and process the datasets alongside [Hu et al., 2020]. More databases can be found in the `datasets` module of the `PyTorch Geometric` library [Fey and Lenssen, 2019].

We exhibit 14 well-known benchmark datasets. These datasets come from different fields, and cover a wide range of graph properties. The diversity of these particularities allows to explore and examine comprehensively the behavior of methods and the library proposed in this thesis. Table 2.2 outlines the properties of these datasets.

Alkane is a dataset of 150 alkanes with only carbon atoms, which are represented by acyclic unlabeled graphs [Cherqaoui and Villemin, 1994]. The problem is to predict boiling points by regression.

Acyclic has 183 acyclic molecules with hetero atoms [Cherqaoui et al., 1994]. It is associated with a regression problem to predict the boiling points of the molecules.

MAO is a Monoamine Oxidase dataset of 68 cycle-included molecules labeled by two classes: 38 of them act as antidepressant drugs by inhibiting the monoamine

[1] Available at `https://brunl01.users.greyc.fr/CHEMISTRY`.
[2] Available at `www.graphlearning.io`.
[3] Available at `https://ogb.stanford.edu`.

Figure 2.4 – A sample of the *Letter* dataset.

oxidase and 30 do not [Brun, 2018]. Thus, it is associated with a classification problem.

PAH is a polycyclic aromatic hydrocarbon dataset composed of 94 cyclic unlabeled graphs [Brun, 2018]. Atoms are all carbons and chemical bonds are all aromatics. The associated target is to determine whether each molecule is cancerous or not.

MUTAG is composed of 188 MUTAGenic aromatic and heteroaromatic nitro compounds [Debnath et al., 1991]. The classification task associated is to correctly determine whether each compound has a mutagenic effect.

Monoterpens contains different classes of monoterpenoid molecules encoded by labeled graphs. Depending on the existence of the same precursor, molecules are grouped into 8 classes.

The ***Letter*** dataset involves graphs of distorted letter drawings of 15 capital letters of the Roman alphabet that consist of straight lines only (*A*, *E*, *F*, *H*, *I*, *K*, *L*, *M*, *N*, *T*, *V*, *W*, *X*, *Y*, *Z*) [Riesen and Bunke, 2008]. According to the strength of distortion, three sub-datasets consisting of 2250 graphs each are generated, namely ***Letter-high***, ***Letter-med***, and ***Letter-low***, corresponding respectively to distortions of high, medium, and low strengths. In each graph, lines are represented by undirected edges and ending points by vertices. Vertices are equipped with 2D non-symbolic labels "x" and "y", which represent positions of vertices in a 2D coordinate system. Figure 2.4 exhibits a sample of *Letter* dataset, in which the left image "A" is the original letter, while the other three are its distortions at different levels. The task is to classify each graph to the proper letter.

Enzymes has 600 enzymes from the Brenda enzyme database [Schomburg et al., 2004], which represents the tertiary structures of protein [Borgwardt et al., 2005]. The task is to predict the correct Enzyme Commission top-level class from all six classes.

AIDS constitutes 2000 molecule compounds represented by graphs, each associated with a Boolean value indicating whether it is active against HIV or not [Riesen and Bunke, 2008]. Vertices are labeled with chemical symbols and edges

with valency. A classification task on activity is assigned to it.

NCI1 and ***NCI109*** represent two balanced subsets of datasets of chemical compounds screened for activity against non-small cell lung cancer and ovarian cancer cell lines respectively [Wale et al., 2008]. Each subset is composed of 4110 and 4127 graphs respectively, with vertices labeled with types of atoms. The associated task concerns classifying a graph according to its activity against the cancers.

DD is composed of 1178 graphs representing protein structures [Dobson and Doig, 2003]. Each vertex is labeled by a symbol denoting an amino acid. If the distance between two vertices is less than 6 Angstroms, an edge is inserted. Graphs are grouped by whether they are enzymes or not, and a classification task is assigned.

Table 2.2 – Structures and properties of real-world graph datasets.

Datasets	Substructures			Numbers of Labels				Directed	N	$\bar{n}$	$\bar{m}$	d	Class Numbers	Tasks
	linear	non-linear	cyclic	symbolic		non-symbolic								
				vertices	edges	vertices	edges							
Alkane	✓	✓	✗	✗	✗	✗	✗	✗	150	8.87	7.87	1.75	-	R
Acyclic	✓	✓	✗	3	✗	✗	✗	✗	183	8.15	7.15	1.47	-	R
MAO	✓	✓	✓	3	4	✗	✗	✗	68	18.38	19.63	2.13	2	C
PAH	✓	✓	✓	✗	✗	✗	✗	✗	94	20.70	24.43	2.36	2	C
MUTAG	✓	✓	✓	7	11	✗	✗	✗	188	17.93	19.79	2.19	2	C
Monoterpens	✓	✓	✓	3	3	✗	✗	✗	286	11.00	11.07	2.01	8	C
Letter-high	✓	✓	✓	✗	✗	2	✗	✗	2250	4.67	4.50	1.89	15	C
Letter-med	✓	✓	✓	✗	✗	2	✗	✗	2250	4.67	3.21	1.35	15	C
Letter-low	✓	✓	✓	✗	✗	2	✗	✗	2250	4.67	3.13	1.32	15	C
Enzymes	✓	✓	✓	3	✗	18	✗	✗	600	32.63	62.14	3.86	6	C
AIDS	✓	✓	✓	38	3	4	✗	✗	2000	15.69	16.20	2.01	2	C
NCI1	✓	✓	✓	37	✗	✗	✗	✗	4110	29.87	32.30	2.16	2	C
NCI109	✓	✓	✓	38	✗	✗	✗	✗	4127	29.68	32.13	2.15	2	C
DD	✓	✓	✓	82	✗	✗	✗	✗	1178	284.32	715.66	4.98	2	C

"Substructures" are the sub-patterns that graphs contain;
"Numbers of labels" include numbers of symbolic and non-symbolic vertex and edge labels, with ✗ for no label;
"Directed" exhibits whether directed graphs are included;
N is the number of graphs;
$\bar{n}$ is the average number of graph vertices;
$\bar{m}$ is the average number of edges;
d is the average vertex degree;
"Tasks" are either regression ("R") or classification ("C").

Part II

Contributions

Chapter 3

Graph kernels based on sub-patterns

Contents

3.1 Overview

Graph kernels can be constructed using the global information, the local information, or both of them in graphs, by means of varied strategies. Among these strategies, there are kernels based on sub-structures, information propagation kernels, and deep graph kernels [Ghosh et al., 2018]. Of particular interest are kernels based on sub-patterns/structures. When comparing graphs and analyzing their properties, the similarity principle has been widely investigated [Johnson and Maggiora, 1990]. It states that molecules having more common substructures turn to have more similar properties. This principle can be generalized to other fields where data is modeled as graphs. It provides a theoretical support to construct graph kernels by studying graphs' substructures, which are also referred to as patterns. There are three major types of patterns, as illustrated by G_1, G_2 and G_3 in Figure 3.1. The most fundamental patterns are linear patterns, which are composed of sequences of vertices connected by edges. However, when a substructure contains vertices that have more than two neighbors, linear patterns are insufficient to completely describe the structure. This is where non-linear patterns become useful, with either non-linear (acyclic) patterns or cyclic patterns, which contain cycles.

Despite that non-linear patterns may encode more complex structural information than linear ones, the latter are of great interest for several reasons. First, non-linear patterns normally include or imply the linear ones. This is the case for example of the treelet pattern [Gaüzère et al., 2012]. A treelet is a subtree, rooted at a vertex v, that can be expanded by iteratively adding all neighbors of the leaf vertices as their child leaves as long as the size of the subtree is not bigger than 6. The treelet pattern is non-linear as a whole, while treelets whose maximal size is less than 4 are linear. A detailed description of the treelet kernel can be found in Section 3.5.1. Second, linear patterns require lower computational complexity than non-linear patterns in most cases. Moreover, it could be intractable to compute non-linear or cyclic-based kernels on large graphs. Therefore in this thesis, we fo-

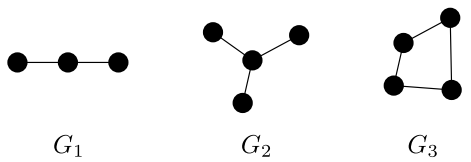

Figure 3.1 – Different types of graph patterns. G_1, G_2, G_3 are examples of linear patterns, non-linear (acyclic) patterns and cyclic patterns, respectively.

Table 3.1 – Characteristics of graph kernels based on linear patterns, and two on non-linear patterns

Kernels	Substructures			Labeling				Directed	Edge Weighted	Time Complexity (Gram Matrix)	Explicit Representation	Weighting
				symbolic		non-symbolic						
	linear	non-linear	cyclic	vertices	edges	vertices	edges					
Common walk	✓	✗	✗	✓	✓	✗	✗	✓	✗	$\mathcal{O}(\mathrm{N}^2 n^6)$	✗	a priori
Marginalized	✓	✗	✗	✓	✓	✗	✗	✓	✗	$\mathcal{O}(\mathrm{N}^2 r n^4)$	✗	✗
Sylvester equation	✓	✗	✗	✗	✗	✗	✗	✓	✓	$\mathcal{O}(\mathrm{N}^2 n^3)$	✗	a priori
Conjugate gradient	✓	✗	✗	✓	✓	✓	✓	✓	✓	$\mathcal{O}(\mathrm{N}^2 r n^4)$	✗	a priori
Fixed-point iterations	✓	✗	✗	✓	✓	✓	✓	✓	✓	$\mathcal{O}(\mathrm{N}^2 r n^4)$	✗	a priori
Spectral decomposition	✓	✗	✗	✗	✗	✗	✗	✓	✓	$\mathcal{O}(\mathrm{N}^2 n^2 + \mathrm{N} n^3)$	✗	a priori
Shortest path	✓	✗	✗	✓	✗	✓	✗	✓	✓	$\mathcal{O}(\mathrm{N}^2 n^4)$	✗	✗
Structural shortest path	✓	✗	✗	✓	✓	✓	✓	✓	✗	$\mathcal{O}(h\mathrm{N}^2 n^4 + \mathrm{N}^2 nm))$	✗	✗
Path kernel up to length h	✓	✗	✗	✓	✓	✗	✗	✓	✗	$\mathcal{O}(\mathrm{N}^2 h^2 n^2 d^{2h})$	✓	✓
Treelet	✓	✓	✗	✓	✓	✗	✗	✓	✗	$\mathcal{O}(\mathrm{N}^2 n d^5)$	✓	✓
Weisfeiler-Lehman (WL) subtree	✓	✓	✗	✓	✗	✗	✗	✓	✗	$\mathcal{O}(\mathrm{N} hm + \mathrm{N}^2 hn)$	✓	✗

The "Time complexity" column is a rough estimation for computing the Gram matrix. The "Explicit representation" column indicates whether the embedding of graphs in the representation space can be encoded by a vector explicitly; in other words, whether the patterns of graph kernels can be explicitly presented (see [Kriege et al., 2014] for more detailed analysis). The "Weighting" column indicates whether the substructures can be weighted in order to obtain a similarity measure adapted to the problem at hand, where "a priori" indicates that the weights are set while constructing kernels. For example, weight λ_h in (3.2) is set to γ^h when constructing the kernel by geometric series (see Section 3.2.1).

cus on studying and comparing graph kernels based on different linear patterns, as well as several kernels based on non-linear patterns for the sake of comparison.

A linear pattern is defined as a walk or a path. A walk is an alternating sequence of vertices and connecting edges; a path is a walk without repeated vertices. Major Kernels based on walks are the common walk kernel [Gärtner et al., 2003], the marginalized kernel [Kashima et al., 2003], and the generalized random walk kernel [Vishwanathan et al., 2010]. The shortest path kernel [Borgwardt and Kriegel, 2005], the structural shortest path kernel [Suard et al., 2007], and the path kernel up to length h [Ralaivola et al., 2005] are constructed based on paths, which are relieved from artifacts brought by walks due to tottering and halting (see Section 3.2.4). More recently, many developments have been carried out to enhance these graph kernels [Aziz et al., 2013, Xu et al., 2014, Sugiyama and Borgwardt, 2015].

In this chapter, we thoroughly study these graph kernels based on linear patterns, along with the comparison with two kernels based on non-linear patterns. We propose three strategies to reduce the computational complexity of graph kernels in time and memory usage and conduct comprehensive experiments and analyses for these kernels on various types of graph datasets. Our work aims to provide a better understanding of graph kernels and practice suggestions on how to choose and use them. We first study their mathematical representations and compare their computational complexities. Table 3.1 provides an insight into the characteristics of these kernels.

3.2 Graph kernels based on walks

Depending on how walks are generated, several graph kernels have been proposed.

3.2.1 Common walk kernel

The common walk kernel is based on the simple idea to compare all possible walks starting from all vertices in two graphs [Gärtner et al., 2003]. By assigning a transition probability to each edge while performing walks, the common walk kernel has the potential to deal with stochastic processes, thus is referred to as the random walk kernel in some literature [Vishwanathan et al., 2010, Borgwardt and Kriegel, 2005].

The straightforward way to compute this kernel is by brute force, i.e., searching

for all walks available and trying to match them one by one. To use this method, one needs to fix in advance the maximum length of walks to some value h, since the number of walks in a graph can be infinite. Two steps are required here: Step 1 is to find out all walks in each graph, where the Depth-first search scheme is usually applied [Cormen et al., 2009]. The time complexity of this step is $\mathcal{O}(nd^h)$ for a graph with n vertices and average vertex degree d. Step 2 is to compare walks one by one in the two graphs under study, which has a time complexity of $\mathcal{O}(n^2d^{2h})$. For a dataset of N graphs, the computation of the Gram matrix requires $\mathcal{O}(\mathrm{N}^2n^2d^{2h})$ operations. Therefore, the time complexity is exponential in the length of walks, which is impractical for large-scale graphs. To overcome this difficulty and better explore the infinite walk space, other computational methods have been proposed as given next.

In [Gärtner et al., 2003], the fully-labeled direct product graph is employed to reduce the computational complexity within kernels based on contiguous label sequences, which can deal with labels on vertices and/or edges. The direct product graph is defined as

Definition 3.1 (direct product graph)**.** *The direct product graph of two graphs* $\mathrm{G}_1 = (\mathrm{V}_1, \mathrm{E}_1)$ *and* $\mathrm{G}_2 = (\mathrm{V}_2, \mathrm{E}_2)$*, denoted* $\mathrm{G}_\times = \mathrm{G}_1 \times \mathrm{G}_2$*, is defined by*

$$\begin{cases} \mathrm{V}_\times(\mathrm{G}_1 \times \mathrm{G}_2) = \left\{(v_1, v_2) \in \mathrm{V}_1 \times \mathrm{V}_2 \;\middle|\; \ell_v(v_1) = \ell_v(v_2)\right\} \\ \mathrm{E}_\times(\mathrm{G}_1 \times \mathrm{G}_2) = \Big\{((u_1, u_2), (v_1, v_2)) \in \mathrm{V}^2(\mathrm{G}_1 \times \mathrm{G}_2) \;\Big|\; \\ \qquad\qquad (u_1, v_1) \in \mathrm{E}_1 \wedge (u_2, v_2) \in \mathrm{E}_2 \wedge \ell_e(u_1, v_1) = \ell_e(u_2, v_2)\Big\}, \end{cases} \tag{3.1}$$

where $\ell_v(\cdot)$ *and* $\ell_e(\cdot)$ *are the labeling functions defined in Section 2.1.*

In other words, vertices with the same labels from graphs G_1 and G_2 are compounded to new vertices of graph $\mathrm{G}_\times$, and an edge between two vertices in graph $\mathrm{G}_\times$ exists if and only if edges exist between their corresponding vertices in graphs G_1 and G_2, while these two edges have the same label. Figure 3.2 illustrates the direct product $\mathrm{G}_1 \times \mathrm{G}_2$ of two fully-labeled graphs G_1 and G_2, where $\{\ell_{v_i} \mid i = 1, 2, \ldots\}$ denotes the set of vertex labels and $\{\ell_{e_i} \mid i = 1, 2, \ldots\}$ the set of edge labels. This definition is a generalization of the directed product of unlabeled graphs, which considers that all vertices and edges have the same labels, and is shown by Figure 2 in [Vishwanathan et al., 2010].

A bijection exists between every walk in the direct product graph and one walk in each of its corresponding graphs, so that labels of all vertices and edges on these

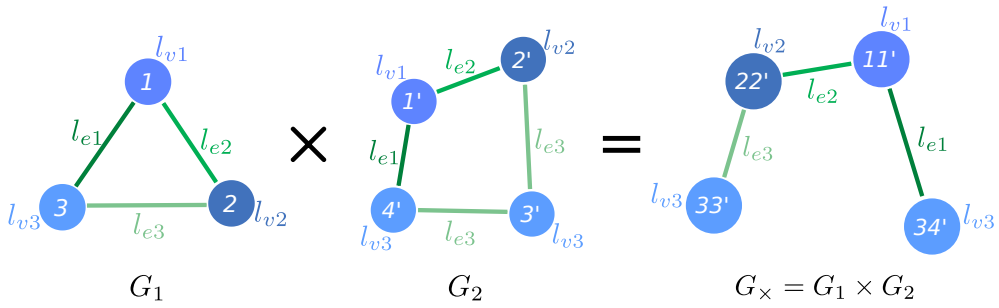

Figure 3.2 – Direct product of fully-labeled graphs

walks match by order. Consequently, it is equivalent to perform a walk on a direct product graph and on its two corresponding graphs simultaneously, which makes it possible to compute the kernel between two graphs by finding out all walks in their direct product graph. The direct product kernel, a.k.a. the common walk kernel, is then designed this way.

Definition 3.2 (common walk kernel)**.** *For a graph* G*, let* $\Phi(G) = (\Phi_{s_1}(G), \Phi_{s_2}(G), \dots)$ *be a map to label sequence feature space expanded by basis* $\Phi_S(G)$*, where* $S = (s_1, s_2, \dots)$ *is the set of all possible label sequences of walks, and* $\Phi_s(G)$ *the feature corresponding to label sequence* s*. For each possible sequence* s *of length* h*,* $\Phi_s(G) = \sqrt{\lambda_h}|W_s|$*, where* $|W_s|$ *is the number of walks that correspond to the label sequence* s *in* G*, and* $\lambda_h \in \mathbb{R}$ *is some fixed weight for length* h*. Then we have the direct product kernel* $k_\times(G_1, G_2) = \langle \Phi(G_1), \Phi(G_2) \rangle$*, with*

$$k_\times(G_1, G_2) = \sum_{i,j=1}^{|V_\times|} \left[\sum_{h=1}^{\infty} \lambda_h A_\times^h \right]_{ij}, \tag{3.2}$$

if the limit exists, where $A_\times$ *is the adjacency matrix of the direct product graph* $G_\times$*. As each component* $[A_\times]_{ij}$ *indicates whether an edge exists between vertex* v_i *and* v_j*,* $[A_\times^h]_{ij}$ *is the number of all possible walks of length* h *from vertex* v_i *to* v_j*.*

In practice, this limit cannot be computed directly. However, for $A_\times$ that satisfies certain properties and certain choices of λ_h, closed-forms can be constructed for computation. Two examples are given here [Gärtner et al., 2003]:

- The first one employs exponential series of square matrices $e^{\beta A_\times} = I + \beta A_\times / 1! + \beta^2 A_\times^2 / 2! + \beta^3 A_\times^3 / 3! + \dots$ Computing it normally requires the diagonalization of $A_\times$ as $T^{-1}DT$, where D is a diagonal matrix, and weight $\lambda_h = \beta^h / h!$ where β is a constant. In this way, $k_\times(G_1, G_2) = \sum_{i,j=1}^{|V_\times|} [T^{-1} e^{\beta D} T]_{ij}$, where $e^{\beta D}$ can be cal-

culated component-wise in linear time. Diagonalizing matrix $A_\times$ has roughly a cubic time complexity.

- The second example applies geometric series of matrices. Let the weights be $\lambda_h = \gamma^h$, where $\gamma < 1/\min\{\Delta^+(G), \Delta^-(G)\}$ and $\Delta^+(G), \Delta^-(G)$ are the maximal outdegree and indegree of graph G, respectively. Then the geometric series of a matrix is defined as $I + \gamma^1 A^1 + \gamma^2 A^2 + \ldots$ The limit of this series can be computed by inverting the matrix $I - \gamma A$, which is roughly of cubic time complexity.

The common walk kernel constructs an infinite sequence feature space consisting of all possible walk sequences in graphs whose lengths are theoretically up to infinity. When the kernel is represented by exponential or geometric series, closed-forms are available to compute it in $\mathcal{O}(n^6)$. However, these closed-forms require that the coefficient λ_h has forms chosen specially to converge, which not only is inflexible, but also causes the halting problem, which excessively restrains the effect of long walk sequences on the kernel (see Section 3.2.4). Moreover, it is still time-consuming in practice for large-scale graphs and is more likely to induce unnecessary artifacts into the kernel due to trivial walks, which are irrelevant to the task and therefore decrease the accuracy.

3.2.2 Marginalized kernel

The marginalized kernel relies on walks generated using marginal distributions on some hidden variables [Kashima et al., 2003], which is constructed as

$$k(G_1, G_2) = \sum_{w_1 \in W(G_1)} \sum_{w_2 \in W(G_2)} k_W(w_1, w_2) p_{G_1}(w_1) p_{G_2}(w_2), \tag{3.3}$$

where $W(G)$ is the set of all walks in graph G, k_W is a joint kernel between two walks, usually defined as a delta function of label sequences of the two measured walks, and $p_G(w)$ is the probability of traversing walk w in G. If $w = (v_1, v_2, \ldots, v_h)$, then

$$p_G(w) = p_0(v_1) \prod_{i=2}^{h} p_t(v_i | v_{i-1}) p_q(v_h), \tag{3.4}$$

where the initial probability distribution $p_0(v)$ indicates the probability that walk w starts from vertex v, the transition probability on vertex v_{i-1}, denoted $p_t(v_i|v_{i-1})$, describes the probability of choosing v_i as the next vertex of v_{i-1}, and the termination probability $p_q(v_h)$ gives the probability that walk w stops on vertex v_h. The

latter two probability satisfy the relation $\sum_{j=1}^{n} p_t(v_j|v_i) + p_q(v_i) = 1$. Without any prior knowledge, p_0 is set to be uniform over all vertices of graph G, $p_t(v_i|v_{i-1})$ is uniform over all neighbors of vertex v_{i-1}, and p_q is a constant.

In [Kashima et al., 2003], an efficient method to compute this kernel is proposed as

$$k(\mathrm{G},\mathrm{G}') = \sum_{v_1, v_1'} s(v_1, v_1') \lim_{h\to\infty} \mathrm{R}_h(v_1, v_1'),$$

where $s(v_1, v_1') = p_0(v_1)p_0'(v_1')k_v(v_1, v_1')$ and $\mathrm{R}_h(v_1, v_1')$ is updated recursively in r iterations with

$$\mathrm{R}_h(v_1, v_1') = r_1(v_1, v_1') + \sum_{v_i \in \mathrm{V}} \sum_{v_j' \in \mathrm{V}'} t(v_i, v_j', v_1, v_1')\mathrm{R}_{h-1}(v_i, v_j'),$$

where

$$\begin{cases} r_1(v_1, v_1') = q(v_1, v_1') = \mathrm{R}_1(v_1, v_1'), \\ q(v_h, v_h') = p_q(v_h)p_q(v_h'), \\ t(v_i, v_i', v_{i-1}, v_{i-1}') = p_t(v_i|v_{i-1})p_t'(v_i'|v_{i-1}')k_v(v_i, v_i')k_e(e_{v_{i-1},v_i}, e'_{v_{i-1}',v_i'}). \end{cases}$$

By applying this method, the time complexity of the marginalized kernel is the same as solving a linear system with n^2 equations and n^2 unknown variables, which boils down to $\mathcal{O}(r\,n^4)$.

3.2.3 Generalized random walk kernel

The generalized random walk kernel, as a unified framework for random walk kernels, was proposed in [Vishwanathan et al., 2010]. Based on the idea of performing random walks on a pair of graphs and then counting the number of matching walks, both the common walk kernel and the marginalized kernel are special cases of this kernel. Besides, it is proven in the same paper that certain rational kernels [Cortes et al., 2004] also boil down to this kernel when specialized to graphs.

Similar to the marginalized kernel, the generalized random walk kernel introduces randomness with the construction of graphs' subpatterns, namely random walks. First, an initial probability distribution over vertices is given, denoted p_0, which determines the probability that walks start on each vertex, same as initial probability distribution p_0 of the marginalized kernel. Then, a random walk generates a sequence of vertices $v_{i_1}, v_{i_2}, v_{i_3}, \ldots$ according to a conditional probability

$p(i_{k+1} \mid i_k) = \mathrm{A}_{i_{k+1},i_k}$, where A is the normalized adjacency matrix of the graph. This probability, which plays a similar role as the transition probability p_t of the marginalized kernel, chooses $v_{i_{k+1}}$ as the next vertex of v_{i_k} being proportional to the weight of the edge $(v_{i_k}, v_{i_{k+1}})$, namely $p_t(i_{k+1} \mid i_k) = w_{i_{k+1}} / \sum_{j \in \mathrm{N}(i_k)} w_j$, where $\mathrm{N}(i_k)$ is the set of neighbors of the vertex i_k. The edge weight here is a special label that represents the transition probability from one vertex to another rather than a property of the edge itself. Finally, similar to termination probability p_q of the marginalized kernel, a stopping probability distribution $q = (q_{i_1}, q_{i_2}, \ldots)$, $n_{i_k} \in \mathrm{V}$ is associated with a graph $\mathrm{G} = (\mathrm{V}, \mathrm{E})$ over all its vertices, modeling the phenomenon where a random walk stops at vertex n_{i_k}. Both the initial probability and the stopping probability are practically set as the uniform distribution without prior knowledge.

Similar to the common walk kernel, defining the generalized random walk kernel takes advantage of the direct product. However, when performing the transformation, graphs are considered unlabeled, which is a special case of Definition 3.1. It is worth noting that the direct product graph of unlabeled graphs often has much more edges than the labeled one, as illustrated in Figure 3.3 when taking the graphs in Figure 3.2 as unlabeled. The generalized random walk kernel between two graphs G_1 and G_2 is defined in [Vishwanathan et al., 2010] as

$$k(\mathrm{G}_1, \mathrm{G}_2) = \sum_{h=0}^{\infty} f(h) q_\times^\top \mathrm{W}_\times^h p_\times. \tag{3.5}$$

In this expression, $f(h)$ is the weight chosen a priori for random walks of length h, $p_\times = p_{0_1} \otimes p_{0_2}$ and $q_\times = q_1 \otimes q_2$ are the initial probability distribution, and the stopping probability distribution on $\mathrm{G}_\times$, respectively, where the operator $\otimes$ denotes the Kronecker product and $\mathrm{W}_\times \in \mathbb{R}^{n \times n}$ is the weight matrix.

Assuming the initial and the stopping probability distributions to be uniform and setting $\mathrm{W}_\times$ as the unnormalized adjacency matrix of $\mathrm{G}_\times$, (3.5) can be transformed to the common walk kernel. Meanwhile, applying $f(h) = 1$ and $\mathrm{W}_\times$ as a specific form, the marginalized kernel can be recovered from (3.5).

The complexity of the direct computation is $\mathcal{O}(\mathrm{N}^2 n^6)$. Four methods are presented to accelerate the computation [Vishwanathan et al., 2010]:

The *Sylvester equation method* is based on the generalized Sylvester equation $\mathrm{M} = \sum_{i=1}^{d} \mathrm{S}_i \mathrm{M} \mathrm{T}_i + \mathrm{M}_0$. For graphs with symbolic edge labels, when $f(h) = \lambda_h$, the kernel in (3.5) can be computed by $q_\times^\top vec(\mathrm{M})$, with $vec(\cdot)$ the column-stacking operator and M the solution of the generalized Sylvester equation $\mathrm{M} =$

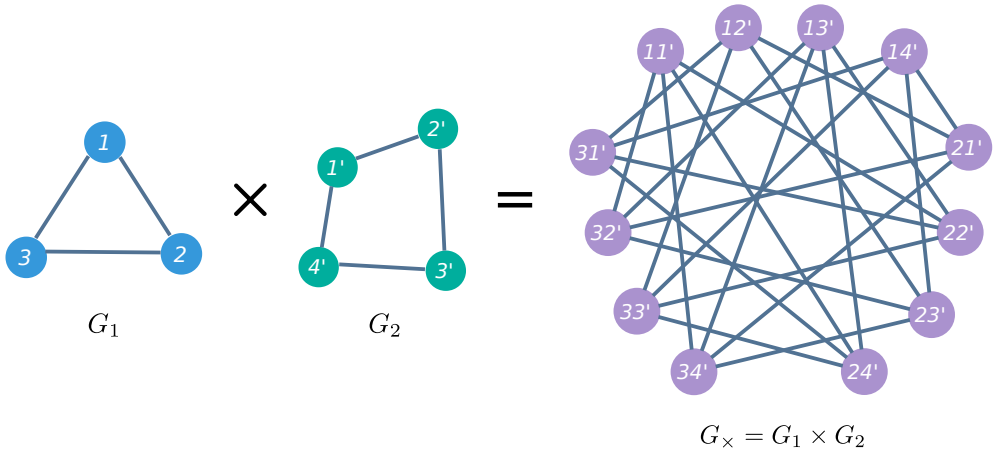

Figure 3.3 – Direct product of unlabeled graphs (compared to the labeled ones in Figure 3.2)

$\sum_{i=1}^{d} \lambda\, {}^{i}\mathrm{A}_2 \mathrm{M}\, {}^{i}\mathrm{A}_1^\top + \mathrm{M}_0$, where d is the number of different edge labels, $vec(\mathrm{M}_0) = p_\times$ and ${}^{i}\mathrm{A}$ is the normalized adjacency matrix of graph G filtered by the i-th edge label ${}^{i}\ell_e$ of G, namely, ${}^{i}\mathrm{A}_{jk} = \mathrm{A}_{jk}$ if $\ell_e(v_j, v_k) = {}^{i}\ell_e$, and zero otherwise. When $d = 1$, this equation can be computed in cubic time, while its time complexity remains unknown when $d > 1$.

This method does not directly compute weight matrices of direct product graphs. Benefiting from the Kronecker product, only (normalized) adjacency matrices of original graphs are required, which have the size of n^2 and can be precomputed for each graph. Besides, computing $q_\times^\top vec(\mathrm{M})$ requires $\mathcal{O}(\mathrm{N}^2 n^2)$ time. Thus, for N unlabeled graphs, the complexity of computing the corresponding Gram matrix is $\mathcal{O}(\mathrm{N}^2 n^3)$, which is reduced compared to the direct computation.

However, it has a strong drawback: libraries currently available to solve the generalized Sylvester equation, such as `dlyap` solver in MATLAB and `control.dlyap` function in *Python Control Systems Library* [Murray et al., 2018], can only take d as 1, which means the solver is limited to edge-unlabeled graphs. Some contributions exist to solve the general problem [Bouhamidi and Jbilou, 2008], but their implementation are not available yet. Integrating such work into our library is included in mid-term schedule.

The second method is the *conjugate gradient method*. It solves the linear system $(\mathrm{I} - \lambda \mathrm{W}_\times)x = p_\times$ for x, using a conjugate gradient solver, and then computes $q_\times^\top x$. This procedure can be done in $\mathcal{O}(r n^4)$ for r iterations.

The third method, *fixed-point iterations*, rewrites $(\mathrm{I} - \lambda \mathrm{W}_\times)x = p_\times$ as $x = p_\times + \lambda \mathrm{W}_\times x$. Thus, it computes x by finding a fixed point of the equation, namely by iter-

ating $x_{t+1} = p_\times + \lambda W_\times x_t$. The worst-case time complexity is $\mathcal{O}(r n^4)$ for r iterations.

The *spectral decomposition* method applies the $W_\times = P_\times D_\times P_\times^{-1}$ decomposition, where the columns of $P_\times$ are its eigenvectors, and $D_\times$ is a diagonal matrix of corresponding eigenvalues. This method can be performed in $\mathcal{O}(N^2 n^2 + N n^3)$ for N graphs.

The generalized random walk kernel provides a quite flexible framework for walk-based kernels (see (3.5)). However, the problems lie in the methods to compute the kernel as introduced above. Despite the improvement in time complexity, the shortcomings of these methods are obvious. First, some methods can only be applied for special types of graphs. By definition, the Sylvester equation method can only be applied for graphs unlabeled or with symbolic edge labels. The symbolic vertex labels of two vertices on an edge can be added to the edge label of that edge. However, due to the lack of solvers, no label can be dealt with in practice. The spectral decomposition method, on the other hand, can only tackle unlabeled graphs. Secondly, each method is designated for a special case of the generalized random walk kernel. The Sylvester equation method, the conjugate gradient method, and the fixed-point iterations are specified for the geometric kernel only, namely, $f(h)$ set to λ^h in (3.5). The spectral decomposition method works on any $f(h)$ that makes (3.5) converge, but is only efficient for unlabeled graphs.

For conciseness in this thesis, the generalized random walk kernel computed by the Sylvester equation, the conjugate gradient, the fixed-point iterations, and the spectral decomposition are denoted as **the Sylvester equation kernel**, **the conjugate gradient kernel**, **the fixed-point kernel** and **the spectral decomposition kernel**, respectively. In our implementation, uniform distributions are applied by default for both starting and stopping probabilities (*i.e.*, p_0 and q), as recommended in [Vishwanathan et al., 2010]. Users are able to introduce prior knowledge with edge weights.

3.2.4 Problems raised by walks

There are two problems that may lead to worsening the performance of kernels based on walks: tottering and halting. We discuss these problems in this section.

Tottering. When constructing a walk in a graph, two connected vertices on this walk may appear multiple times as the transition scheme allows transiting back. This phenomenon, called tottering, brings tottering artifacts into the walk. As Fig-

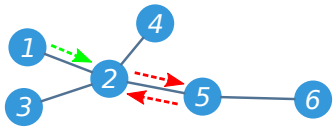

Figure 3.4 – A tottering example: walk (v_1, v_2, v_5, v_2) has tottering between vertices v_2 and v_5, *i.e.*, (v_2, v_5, v_2).

ure 3.4 shows, a tottering brings unnecessary structure to the pattern and may worsen the performance of graph kernels.

To avoid this problem, [Mahé et al., 2004] propose a technique for the marginalized kernel. It first transforms each graph $G = (V, E)$ to $G' = (V', E')$, with

$$\begin{cases} V' = V \cup E \\ E' = \left\{(v, (v, t)) \;\middle|\; v \in V, (v, t) \in E\right\} \cup \left\{((u, v), (v, t)) \;\middle|\; (u, v), (v, t) \in E, u \neq t\right\}, \end{cases} \tag{3.6}$$

and labels its vertices and edges as follows: For a vertex $v' \in V'$, if $v' \in V$, the label $\ell'_v(v') = \ell_v(v')$; if $v' = (u, v) \in E$, then the label $\ell'_v(v') = \ell_v(v)$. For an edge $e' = (v'_1, v'_2) \in E'$, where $v'_1 \in V \cup E$ and $v'_2 \in E$, the label $\ell'_e(e') = \ell_e(v'_2)$. And then, it computes the marginalized kernel between transformed graphs. This extension is able to remove tottering from walks for the marginalized kernel; hence, it enhances the performance of the kernel. However, this improvement is only minor according to the experiments carried out in [Mahé et al., 2004]. Meanwhile, it may significantly enlarge the size of graphs, bringing computational complexity problems. For a graph with n vertices, m edges and average vertex degree d, the transformed graph may at most have $n+m$ vertices and $nd+m^2$ edges; hence, the worst case time complexity of the kernel is $\mathcal{O}((n+m)^2)$, which is not practical for graphs with a high average vertex degree. For all these reasons, experiments conducted in Section 3.7 evaluate the conventional marginalized kernel with tottering.

Halting. Besides tottering, a problem called halting may occur for common walk kernels [Sugiyama and Borgwardt, 2015], where walk patterns with longer lengths contribute less to the kernel values. It is as if the common walk halts after several steps of computation. For example, as shown in Section 3.2.1, the geometric common walk kernel applies geometric series as weights for walks with different lengths, namely $\lambda_h = \gamma^h$, for $\gamma < 1$. When γ is small and h is big, λ_h becomes significantly small; when γ is small enough, walks of length 1 dominate the other walks in the final results. Thus, the kernel degenerates to the comparison of single vertices and edges, and most of the structure information is lost.

To overcome these issues, several graph kernels based on paths have been proposed, as described in the following section.

3.3 Graph kernels based on paths

3.3.1 Shortest path kernel

The shortest path kernel is built on the comparison of shortest paths between any pair of vertices in two graphs [Borgwardt and Kriegel, 2005]. The first step to compute this kernel is to transform the original graphs into shortest-paths graphs by Floyd-Warshall's algorithm [Floyd, 1962]. A shortest-paths graph contains the same set of vertices as the original graph, while between each pair of vertices there is an edge that is labeled by the shortest distance between these two vertices. Then the shortest path kernel is defined on the Floyd-transformed graphs as follows:

Definition 3.3 (shortest path kernel)**.** *Let* $G_1^\star = (V_1, E_1)$ *and* $G_2^\star = (V_2, E_2)$ *be the Floyd-transformed graphs of two graphs* G_1 *and* G_2*, respectively. The shortest path graph kernel between graphs* G_1 *and* G_2 *is defined as*

$$k_{sp}(G_1, G_2) = \sum_{e_1 \in E_1} \sum_{e_2 \in E_2} k_w(e_1, e_2), \tag{3.7}$$

where k_w *is a positive semi-definite kernel on length 1 walks.*

The basic definition of $k_w(e_1, e_2)$ is the product of kernels on vertices and edges encountered along the walk. The kernel for symbolic vertex labels is usually the delta function of labels of two compared vertices, while the kernel for non-symbolic vertex attributes is not given for general cases. In this thesis, we consider the basic definition where the labels of the compared edges are defined by the weighted lengths of their corresponding shortest paths. Nevertheless, more information can be added for more thorough studies [Borgwardt and Kriegel, 2005].

The Floyd-Warshall's algorithm, required in the shortest path kernel to perform the Floyd-transformation, can be done in $\mathcal{O}(n^3)$. For a connected graph G with n vertices, its shortest-paths graph $G^\star$ contains n^2 edges. Assuming that vertex kernels and edge kernels are computed in $\mathcal{O}(1)$, then the pairwise comparison of all edges in two shortest-paths graphs has a time complexity of $\mathcal{O}(n^4)$, which is also the complexity to compute the shortest path kernel.

Compared to kernels based on walks, the shortest path kernel has some advantages: It avoids tottering while remaining simple both conceptually and practically. However, this comes with a cost. Its major shortcomings are:

- It simplifies the graph structure by Floyd transformation and only considers information concerning shortest distances. Only attributes of start and end vertices of shortest paths are considered, while intermediate vertices and edges are ignored.
- It cannot deal with graphs whose edges bear continuous attributes other than distances. Symbolic edge labels are omitted as well. The loss of structure information may crucially decrease the performance accuracy [Borgwardt and Kriegel, 2005].
- Although non-symbolic vertex attributes are implied, the kernel for them is not given explicitly for general cases, and it is not clear how to bind it with the kernel for symbolic vertex labels. This issue has not been studied properly in literature, nor solved by any Python or C++ implementation in general case.

To tackle the last issue, our implementation provides a flexible scheme, where the vertex kernel can be customized by users. In experiments, we introduce a kernel for vertices as a product of two kernels: the delta function for symbolic vertex labels and the Gaussian kernel for non-symbolic vertex attributes. The product is applied because the delta function is binary, thus the kernel between two vertices is equal to zero if their symbolic labels are different.

3.3.2 Structural shortest path kernel

The structural shortest path kernel is an extension of the shortest path kernel, as well as a special case of the kernel on bags of paths [Suard et al., 2007]. This kernel takes into consideration vertices and edges on shortest paths, instead of the shortest distance between two vertices. As a result, edge weights cannot be taken into account.

To construct this kernel, all shortest paths between all vertices in each graph are obtained, where Dijkstra's algorithm is used [Dijkstra, 1959]. Then, the kernel function on any two shortest paths p and p' of two graphs is defined as

$$k_p(p,p') = k_v\left(\ell_v(v_1),\ell_v(v'_1)\right)\prod_{i=2}^{n} k_e\left(\ell_e(v_{i-1},v_i),\ell_e(v'_{i-1},v'_i)\right)k_v\left(\ell_v(v_i),\ell_v(v'_i)\right), \quad (3.8)$$

where v_i and v'_i, for $i = 1, 2, \ldots, n$, are vertices on paths p and p', $\ell_v(\cdot)$ and $\ell_e(\cdot)$ are label functions of vertices and edges, and functions k_v and k_e are kernels on labels of vertices and edges, respectively. In general, these two kernel functions are simply defined as the delta function for symbolic labels and the Gaussian kernel for non-symbolic labels, which will be multiplied if both symbolic and non-symbolic labels exist.

The structural shortest path kernel can then be derived from k_p given in (3.8). We use here the simple and straightforward mean average kernel:

$$k_{ssp}(\mathrm{G}_1, \mathrm{G}_2) = \frac{1}{n_1}\frac{1}{n_2}\sum_{p_i \in \mathrm{P}_1}\sum_{p_j \in \mathrm{P}_2} k_p(p_i, p_j), \tag{3.9}$$

where P_1 and P_2 are respectively the shortest path sets of graphs G_1 and G_2. Other approaches can also be applied, such as the max matching kernel and the path level-set based kernel [Suard et al., 2007].

Given graphs with n vertices and m edges, the time complexity of repeated Dijkstra's algorithm using Fibonacci heaps is $\mathcal{O}(n^2 \log n + nm)$ [Bajema and Merlin, 1987]. The complexity to match all paths in two graphs is $\mathcal{O}(hn^4)$, where h is the average length of the shortest paths. Hence, the complexity of the kernel computation is $\mathcal{O}(hn^4 + nm)$.

Compared to the shortest path kernel, the structural shortest path kernel involves more structural information. However, since both kernels adopt only the shortest paths, structures of other paths are still hidden. The path kernel allows to overcome this issue.

3.3.3 Path kernel up to length *h*

The path kernel compares all possible paths rather than the shortest ones [Ralaivola et al., 2005]. The simple path kernel between graphs G_1 and G_2 is defined as

$$k_{ph}(\mathrm{G}_1, \mathrm{G}_2) = \sum_{p \in \mathrm{P}(\mathrm{G}_1) \cup \mathrm{P}(\mathrm{G}_2)} \phi_p(\mathrm{G}_1)\phi_p(\mathrm{G}_2), \tag{3.10}$$

where $\mathrm{P}(\mathrm{G})$ is the set of all paths in graph G, and $\phi_p(\mathrm{G})$ denotes the feature map of path p for graph G. Two definitions of $\phi_p(\mathrm{G})$ are provided: the binary feature map, where $\phi_p(\mathrm{G}) = \mathbf{1}_{\mathrm{P}(\mathrm{G})}(p)$, and the counting feature map, defined as $\phi_p(\mathrm{G}) = |\{p \mid p \in \mathrm{P}(\mathrm{G})\}|$.

Based on the definitions of $\phi_p(\mathrm{G})$, different types of path kernels can be con-

structed. The Tanimoto kernel, based on the binary feature map, is defined as

$$k_{ph}^{t}(\mathrm{G}_1,\mathrm{G}_2) = \frac{k_{ph}(\mathrm{G}_1,\mathrm{G}_2)}{k_{ph}(\mathrm{G}_1,\mathrm{G}_1) + k_{ph}(\mathrm{G}_2,\mathrm{G}_2) - k_{ph}(\mathrm{G}_1,\mathrm{G}_2)}, \tag{3.11}$$

where $k_{ph}(\mathrm{G}_1,\mathrm{G}_2)$ is the kernel defined as (3.10) corresponding to the binary feature map. When $\phi_p(\mathrm{G})$ takes the form of the counting feature map, then the MinMax kernel can be constructed as

$$k_{ph}^{m}(\mathrm{G}_1,\mathrm{G}_2) = \frac{\sum\limits_{p\in \mathrm{P}(\mathrm{G}_1)\cup \mathrm{P}(\mathrm{G}_2)} \min(\phi_p(\mathrm{G}_1),\phi_p(\mathrm{G}_2))}{\sum\limits_{p\in \mathrm{P}(\mathrm{G}_1)\cup \mathrm{P}(\mathrm{G}_2)} \max(\phi_p(\mathrm{G}_1),\phi_p(\mathrm{G}_2))}. \tag{3.12}$$

These two kernels are related to the Tanimoto similarity measure in the chemistry literature, and provide normalization for the path kernel. While the MinMax kernel considers the frequency of each path rather than just its appearance, it measures more precisely the similarity between graphs of different sizes.

Similar to walks in the common walk kernel, the number of paths in a graph can be infinite. However, unlike the common walk kernel, no closed-form solution has been raised for this phenomenon in the path kernel. The Depth-first search scheme is then applied to find all paths, which limits the maximum length of paths to the depth h. Our implementation applies a trie data structure to store paths in graphs, which saves tremendous memory compared to storing paths directly (e.g., in a list) [Fredkin, 1960]. Thus, the path kernel between two graphs is computed in $\mathcal{O}(h^2n^2d^{2h})$.

The path kernel up to length h encodes information of all paths no longer than h in a graph, which is more expressive than other kernels based on paths. Yet, the limitation of paths' maximum length may be a significant drawback, especially for the running time and memory usage in large-scale graphs.

3.4 Graph kernels related to walks and paths

These two classes of kernels based on linear patterns are the cornerstone of graph kernels. While many other graph kernels have been proposed under different frameworks, they are tightly connected to walks and paths.

The R-*convolution kernels* [Haussler, 1999] are able to compare two objects by the similarities between their substructures, thus provide the base framework for

all kernels mentioned above (See Section 2.2.2). The common walk kernel, as one special case, can be traced back to *the diffusion kernel* [Kondor and Lafferty, 2002], established on the matrix exponentiation. The *exponential kernel* and *the geometric kernel*, proposed in [Gärtner, 2003a], construct kernels with exponential and geometric series of matrices, respectively, where the former one is founded on the diffusion kernel. These two series are directly applied in the computation of the common walk kernel, as described in Section 3.2.1. More recently, due to the phenomenon of halting, the authors of [Sugiyama and Borgwardt, 2015] argued that the common walk kernel is surpassed by *the k-step random walk kernel* in terms of accuracy and possibly computational complexity, where the latter kernel limits the maximum length of walks to k rather than infinity as in the former one (See Section 3.2.4). Meanwhile, when the common walk kernel based on geometric series halts, it boils down to *linear kernels on label histograms* [Sugiyama and Borgwardt, 2015], which compare numbers of vertices that have the same labels in two graphs, as recently implemented in [Sugiyama et al., 2017].

Improvements for other walk-based graph kernels have been introduced as well. [Mahé et al., 2004] proposed *an extension of the marginalized graph kernel* by the graph transformation (3.6), aiming to ameliorate the accuracy by avoiding the tottering problem (See Section 3.2.4). For the generalized random walk kernel, a set of *fast random walk kernels* (ARK) is proposed in [Kang et al., 2012], which speeds up the kernel computation by considering a lower rank approximation of the weight matrix $W_\times$ as in (3.5); however, it suffers a visible loss of accuracy at the same time. Moreover, the authors of [Vishwanathan et al., 2010] proved that the generalized random walk kernel to weighted automata can be viewed as a special case of *the rational kernel*, which computes similarities between weighted automata [Cortes et al., 2004]. In the meantime, *the Fisher kernel* [Jaakkola et al., 1999] from generative models can be viewed as a special case of the marginalized kernel [Ghosh et al., 2018]. Other novel walk-based graph kernels involve *quantum walk kernels* [Bai et al., 2015, Rossi et al., 2013, Bai et al., 2017], which combine the quantum Jensen-Shannon divergence [Lamberti et al., 2008, Majtey et al., 2005] with quantum walks [Farhi and Gutmann, 1998] and *RetGK*, a graph kernel based on return probabilities of random walks [Zhang et al., 2018].

Path-based kernels share similar progress. *The graph hopper kernel* was designed in [Feragen et al., 2013], exploring the sparsity and small diameters of the real-world graphs. It compares shortest paths by vertex kernels encountered

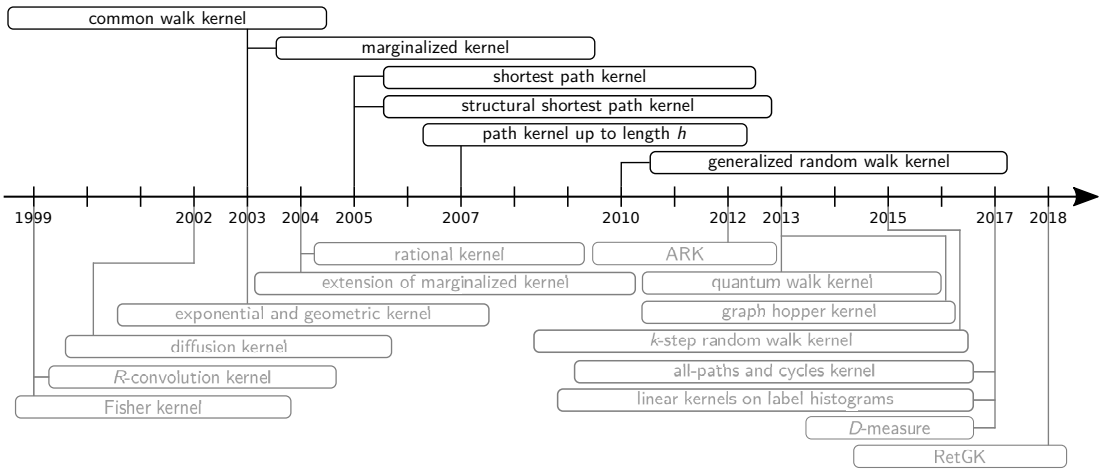

Figure 3.5 – Timeline of graph kernels based on linear patterns (above the timeline) and kernels related to them (below the timeline).

while hopping along those paths, decomposing the graph kernel as a sum of vertex kernels with weights, which encode the graph structure. The authors of [Giscard and Wilson, 2017] then devised *the all-paths and cycles graph kernel*, which by applying a path number counting algorithm based on the work in [Giscard and Rochet, 2018], is able to compute the path kernel up to a moderate length more efficiently and more enriched by encompassing simple cycles. On the other hand, the authors of [Schieber et al., 2017] proposed the D-measure to extract global information of graphs by quantifying differences among the probability distributions of distances defined by the shortest path lengths between vertices.

Figure 3.5 exhibits these kernels in a timeline. For more information about the development of graph kernels, we refer interested readers to [Borgwardt et al., 2020, Kriege et al., 2020, Ghosh et al., 2018, Kriege et al., 2017, Gaüzère et al., 2015a, Gärtner, 2003b].

3.5 Graph kernels based on non-linear patterns

In this section, we introduce two benchmark graph kernels based on non-linear patterns for comparison, namely the treelet kernel and the Weisfeiler-Lehman subtree kernel.

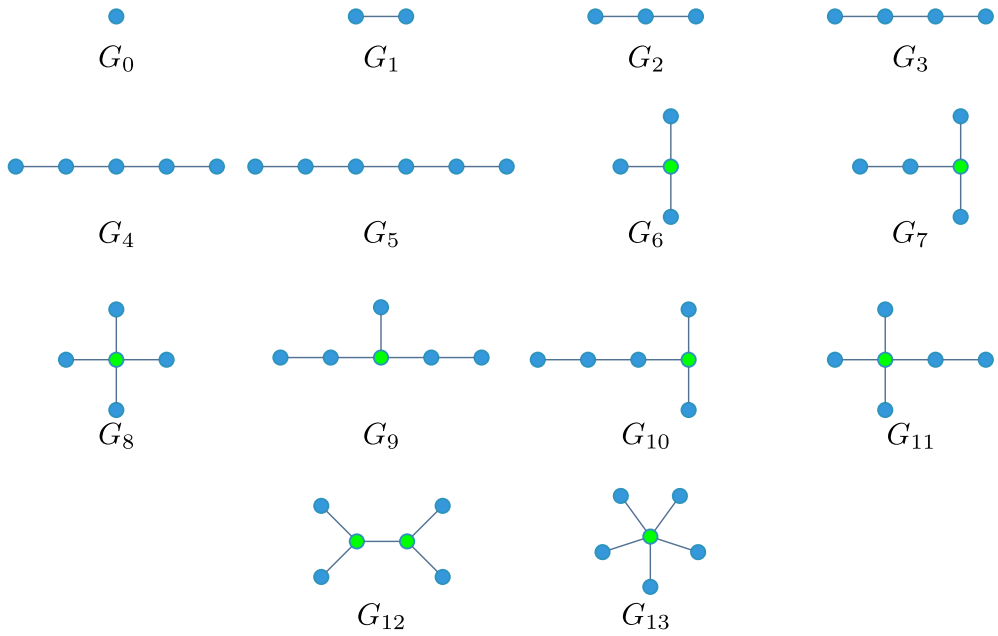

Figure 3.6 – Tree patterns of the treelet kernel.

3.5.1 Treelet kernel

The treelet kernel is built on the distributions of the sub-pattern *treelet* in graphs [Gaüzère et al., 2015b] (see also [Bougleux et al., 2012, Feragen et al., 2013, Borgwardt et al., 2020]). A treelet is a subtree of size up to 6, in which vertices and edges can be labeled. The size number 6 is chosen as a trade-off between the expressiveness and complexity of the kernel. When labels are not considered, there are 14 non-isomorphic treelets, a.k.a., tree patterns in total, as shown in Figure 3.6 [Gaüzère et al., 2012].

No matter whether a graph is labeled or not, the 14 tree patterns have to be enumerated first. Using a recursive depth-first search for all paths in the graph with lengths up to 6, the distribution of the 6 linear patterns G_0, G_1, G_2, G_3, G_4, G_5 as shown in Figure 3.6 can be found [Shervashidze et al., 2009]. The foundation to detect the remaining patterns is finding the vertices of degrees 3, 4, and 5, which are shown as green nodes with borders in Figure 3.6. Patterns G_6, G_8, G_{13} can be extracted directly from these vertices; Patterns G_7, G_9, G_{10}, G_{12} are enumerated from the neighborhood of pattern G_6; Pattern G_{11} is constructed from the neighborhood of pattern G_8.

When labels are considered, each treelet is assigned with a code so that the isomorphism is ensured between two treelets with the same code. A code is composed

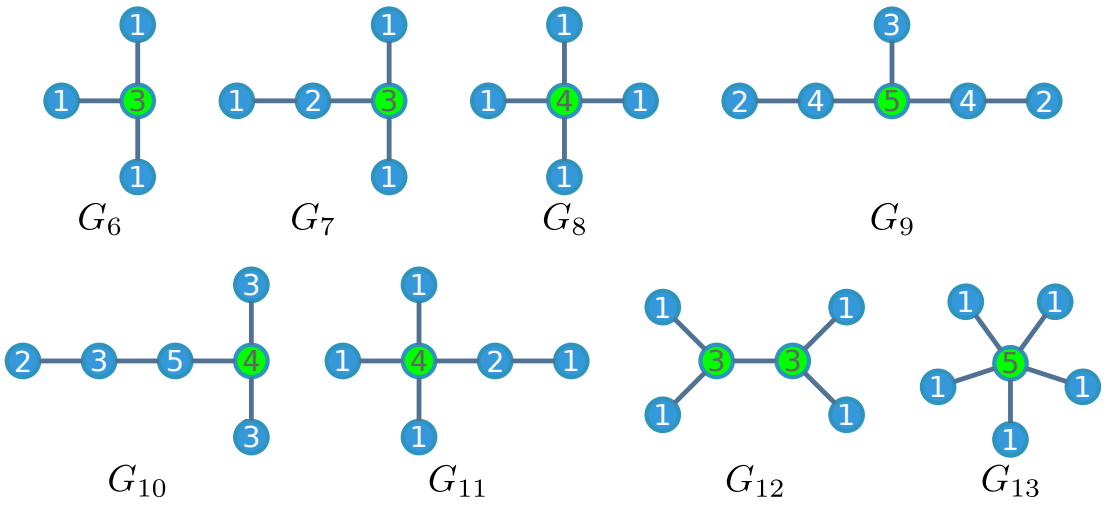

Figure 3.7 – Extended labels of non-linear patterns.

of two parts: a) the index of the corresponding tree pattern indicating the underlying structure and b) a sequence of labels of vertices and edges that forms a canonical key. The key for a linear pattern is the label sequence (Definition 2.10) of that pattern, i.e., the sequence of alternated labels of vertices and edges on that pattern that has the lowest lexicographic order. For a non-linear pattern, the key is based on the extended label of each vertex [Morgan, 1965], which is computed by an iterative process. The extended label of each vertex is initialized by its degree, extended by the sum of extended labels of its neighbors iteratively, and stops updating once the number of different labels of the pattern no longer increases. For isomorphic graphs, the set of extended labels is the same; for each tree pattern, the set of extended labels is unique. The extended labels of each non-linear pattern in Figure 3.6 are exhibited in Figure 3.7.

A set of extended labels may correspond to several permutations of treelets with labels. To distinguish among them, a rooted tree is constructed for each treelet, which encodes the partial order relationship between adjacent vertices. The vertex with the maximum extended label in each treelet is chosen as the root of the tree. Notice there are two rooted trees associated with treelets endowing the pattern G_{12}.

The canonical key of a treelet then is constructed by traversing its rooted tree. To define a unique key, the children of each internal nodes in the rooted tree need to be sorted by the following recursive process: At the beginning, define the key of each leaf $key(v)$ as null; After that, sort the set of children $v_1, v_2, \ldots, v_n$ of each internal node v first according to their extended labels and then the concatenation of the

vertex label $\ell_v(v_i)$, the edge label $\ell_e(v, v_i)$, and the canonical key $key(v_i)$, where $1 \leqslant i \leqslant n$. The key of vertex v is then defined as

$$key(v) = \left(\bigodot_{i=1}^{n} \ell_v(v_i).\ell_e(v, v_i) \right). \bigodot_{i=1}^{n} key(v_i), \tag{3.13}$$

where "$\odot$" and "." are the concatenation operator. Finally, the key of a tree rooted on vertex v_r is defined as $\ell_v(v_r).key(v_r)$.

As treelets G_6 to G_{11} correspond to a single-rooted tree, their canonical codes compose of the concatenation of their tree pattern indices and the key of their rooted trees. Meanwhile, treelet G_{12} corresponds to two rooted trees, so its code concatenates its tree pattern index with the key of the rooted trees that has the lower lexicographic order. The labeled treelets are then enumerated based on the canonical code. The treelet kernel is defined as

$$k_{treelet}(G_1, G_2) = \sum_{t \in \mathcal{T}(G_1) \cap \mathcal{T}(G_2)} k(f_t(G_1), f_t(G_2)), \tag{3.14}$$

where $\mathcal{T}(G)$ denotes the set of treelets in G, $\mathcal{T}(G_1) \cap \mathcal{T}(G_2)$ denotes the set of common treelets in G_1 and G_2, $k(\cdot, \cdot)$ is a kernel between real numbers (the RBF and the polynomial kernels are applied in our experiments in Section 3.7), and the function $f : \mathcal{G} \to \mathbb{R}^n$ associates a graph G to a vector encoding the distribution of its treelets:

$$f(G) = [f_t(G)]_{t \in \mathcal{T}(G)}, \text{ with } f_t(G) = |t|. \tag{3.15}$$

The treelet kernel takes advantage of informativeness by exploring both linear and non-linear (tree) patterns and thus acquires the possibility to obtain better task performances. Meanwhile, the finite set of treelets limits the complexity of the kernel computation, which is in $\mathcal{O}(N^2 n d^5)$ for the Gram matrix of a dataset consisting of N graphs with n average vertices and d average vertex degree, where d is small in many applications such as in chemo-/bio-informatics or social networks field. Moreover, as the treelets are enumerated explicitly, an all-pairs comparison is avoided and a treelet selection step can be integrated. The drawback of this kernel lies in the fact that the treelet set is fixed a priori, which may not suit well any datasets or tasks.

Improvements and extensions for this kernel have been proposed. In [Gaüzère et al., 2012], selection algorithms to weight the treelets, with the good in-

tention to exclude treelets irrelevant to a given property. This method achieves the best results on regression tasks, but may be outperformed by other kernels on classification tasks. In [Gaüzére et al., 2012], a treelet kernel based on cyclic information is proposed, which allows encoding topological relationships between relevant cycles. When it comes to the chemoinformatics field, domain knowledge such as stereoisomerism and chiral information can be incorporated into the kernel [Gaüzère et al., 2015b, Grenier et al., 2013].

3.5.2 Weisfeiler-Lehman subtree kernel

Another well-known graph kernel that can endow non-linear structural information is the Weisfeiler-Lehman (WL) kernel [Shervashidze et al., 2011, Shervashidze and Borgwardt, 2009] (see also [Morris et al., 2017]), which is based on the 1-dimensional variant of the Weisfeiler-Lehman test of isomorphism [Weisfeiler and Lehman, 1968] (also known as "naive vertex refinement" and "color refinement algorithm" [Grohe et al., 2017]). This test is an iterative procedure. In each iteration i, the label $\ell^{(i-1)}(v)$ of a vertex v is updated by a new one $\ell^{(i)}(v)$, which is a compressed notation that represents the argumentation of $\ell^{(i-1)}(v)$ by the sorted multiset of v, namely the set of vertex labels of v's neighbors. This relabeling process for a graph $G = (V, E, \ell^{(i)})$ can be denoted by a function $r((V, E, \ell^{(i)})) = (V, E, \ell^{(i+1)})$. The iterations stop when the sets of vertex labels of the two compared graphs G_1 and G_2 differ, or a given maximum number of the iterations i_{max} is reached.

Definition 3.4 (Weisfeiler-Lehman sequence)**.** *For a graph* $G = (V, E, \ell) = (V, E, \ell^{(0)})$*, its Weisfeiler-Lehman graph at height* i *is defined as* $G^{(i)} = (V, E, \ell^{(i)})$*, and its Weisfeiler-Lehman sequence up to height* h *is defined as the sequence of Weisfeiler-Lehman graphs*

$$\left\{G^{(0)}, G^{(1)}, \ldots, G^{(h)}\right\} = \left\{(V, E, \ell^{(0)}), (V, E, \ell^{(1)}) \ldots, (V, E, \ell^{(h)})\right\},$$

where $G^{(0)} = G$.

Definition 3.5 (Weisfeiler-Lehman kernel)**.** *The Weisfeiler-Lehman kernel with* h *iterations between graphs* G_1 *and* G_2 *is defined as*

$$k_{wl}^{(h)}(G_1, G_2) = \sum_{i=0}^{h} k(G_1^{(i)}, G_2^{(i)}), \tag{3.16}$$

where $\left\{G_1^{(0)}, G_1^{(1)}, \ldots, G_1^{(h)}\right\}$ *and* $\left\{G_2^{(0)}, G_2^{(1)}, \ldots, G_2^{(h)}\right\}$ *are respectively the Weisfeiler-Lehman sequence of* G_1 *and* G_2*, and* $k(\cdot,\cdot)$ *can be any graph kernel and serves as the base kernel. This kernel will be abbreviated to the WL kernel in the remainder of the thesis for conciseness.*

Three base kernels are introduced in [Shervashidze et al., 2011], namely the subtree kernel [Shervashidze and Borgwardt, 2009], the edge kernel, and the shortest path kernel [Borgwardt and Kriegel, 2005]. Of particular interest is the Weisfeiler-Lehman (WL) subtree kernel, which is a natural instance of the WL kernel.

Definition 3.6 (Weisfeiler-Lehman subtree kernel)**.** *Given two graphs* G_1 *and* G_2*, let* $L^{(i)} = \left\{l \mid l \in L_1^{(i)} \bigcup L_2^{(i)}\right\}$ *be the union set of vertex label sets* $L_1^{(i)}$ *and* $L_2^{(i)}$ *of graphs* $G_1^{(i)}$ *and* $G_2^{(i)}$*, where* $L_1^{(i)} = \left\{l_1^{(i)} \mid l_1^{(i)} \in \ell_1^{(i)}\right\}$ *and* $L_2^{(i)} = \left\{l_2^{(i)} \mid l_2^{(i)} \in \ell_2^{(i)}\right\}$*,* $G_1^{(i)} = (V_1, E_1, \ell_1^{(i)})$ *and* $G_2^{(i)} = (V_2, E_2, \ell_2^{(i)})$ *are respectively the Weisfeiler-Lehman graphs at height* i *of* G_1 *and* G_2*,* $0 \leqslant i \leqslant h$*. Order every* $L^{(i)}$*, and denote the number of occurrences of a label* $l^{(i,j)} \in L^{(i)}$ *in the graph* G *as* $c^{(i)}(G, l^{(i,j)})$*, where* $1 \leqslant j \leqslant |L^{(i)}|$*. The Weisfeiler-Lehman subtree kernel between* G_1 *and* G_2 *with* h *iterations is then defined as*

$$k_{wl-subtree}^{(h)}(G_1, G_2) = < \phi_{wl-subtree}^{(h)}(G_1), \phi_{wl-subtree}^{(h)}(G_2) >, \tag{3.17}$$

with

$$\phi_{wl-subtree}^{(h)}(G) = [c^{(i)}(G, l^{(i,j)})], \text{ for } 1 \leqslant j \leqslant |L^{(i)}| \text{ and } 0 \leqslant i \leqslant h$$

being a vector, where each element in this vector is given by a certain j *and* i*. This kernel will be abbreviated to the WL subtree kernel in the remainder of the thesis for conciseness.*

Figure 3.8 illustrates the procedure to compute the WL subtree kernel with $h = 1$, where labels "A" to "D" and "0" to "4" inside the nodes are vertex labels. The WL subtree kernel shares the same time complexity with the WL kernel of $\mathcal{O}(hm)$, where m is the average number of edges in a graph. To compute the Gram matrix for N graphs, one can repeat the same procedure between each pair of graphs. The time required for this native method is in $\mathcal{O}(N^2 hm)$. In contrast, the same procedures can be processed over N graphs simultaneously. Algorithm 3.1 details this procedure. From Step 6 to Step 8, the label compression, the relabeling, and the feature vector construction are performed over all labels in all graphs at the same time. The time complexity of this algorithm is thereby in $\mathcal{O}(Nhm + N^2 hn)$ for N graphs, with n being

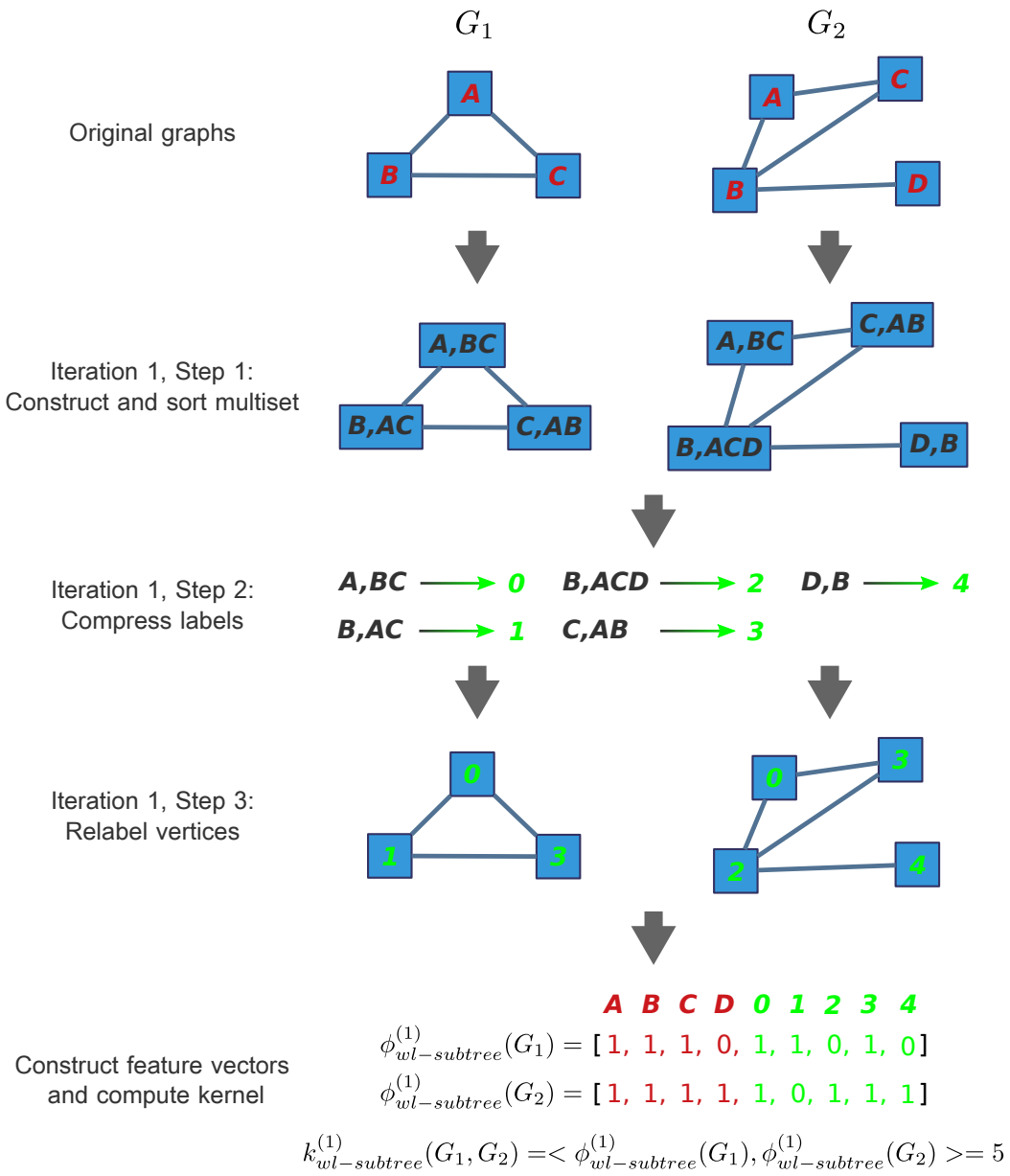

Figure 3.8 – The procedure to compute the WL subtree kernel with $h = 1$.

the average number of vertices in a graph. In general, m is bigger than n, which enlarges the speed advantage of the simultaneous computation over the naive one.

A drawback of Algorithm 3.1 is that the parallel schemes over pairs of graphs are difficult to be deployed due to its special procedure (see Section 3.6.2 for detailed description). Assuming the use of a parallel scheme, the complexity of the naive pairwise computation is reduced to $\mathcal{O}(\frac{\mathrm{N}^2 hm}{p})$. The ratio between the complexity of the two algorithms is in $\mathcal{O}(\frac{p}{\mathrm{N}} + \frac{pn}{m})$. As long as $p \geqslant \frac{m}{n}$, the ratio is bigger than 1 and

Algorithm 3.1 becomes slower than the naive computation. Therefore, it is not recommended for graphs with low density. In Section 3.7, the naive computation is used with a parallelization schema. Nevertheless, the time complexity of the WL (subtree) kernel is linear to the numbers of vertices and/or edges in a graph; hence, the WL kernel opens the door to design graph kernels on large-scale graphs.

Variants and improvements of the WL kernel involve multiple aspects. To strike the balance between the global and local information in graphs, *glocalized Weisfeiler-Lehman graph kernels* are proposed in [Morris et al., 2017]. A local variant of the k-dimensional Weisfeiler-Lehman algorithm is used to capture graph properties, rather than the 1-dimensional one used in the WL kernel. A stochastic approximation approach is proposed to reduce the time complexity to compute this kernel. In [Yanardag and Vishwanathan, 2015b], the authors proposed *a structural smoothing framework* for graphs inspired by smoothing methods used in natural language processing. It considers the partial similarity between compressed multiset labels in the WL kernel, rather than performs an exact matching by assigning binary values. *The Weisfeiler–Lehman optimal assignment kernel* seeks to achieve high predictive performance by adopting the optimal assignment kernel [Kriege et al., 2016], where a base kernel is defined on the compressed subtree labels. A deep variant of this kernel is proposed in [Kriege, 2019]. A method that can be used to improve the WL kernel is proposed in [Kersting et al., 2014], which is implemented by matrix-matrix/vector multiplications, thus readily parallelizable and scalable in terms of time and space. In [Yao and Holder, 2014], a framework combines the WL kernel with an incremental support vector machine to tackle classification tasks on large-scale dynamic graphs. In [Rieck et al., 2019], the WL subtree kernel is augmented to capture low-dimensional topological information in graphs, such as connected components and cycles. *Wasserstein Weisfeiler-Lehman graph kernels* [Togninalli et al., 2019] introduce the Wasserstein distance between distributions of vertex feature vectors in two graphs, allowing capturing the distribution information of individual sub-patterns.

Various graph kernels and works are related to or developed on the WL kernel. During its construction, the compressed label $l^{(i)}(v)$ encodes the neighborhood information of the vertex v, which corresponds to subtree patterns of height i rooted at v, the base patterns to construct several kernels [Ramon and Gärtner, 2003, Bach, 2008]. *The neighborhood hash kernel* [Hido and Kashima, 2009] developed in parallel shares the same framework of the iterative refinement of vertex labels,

where the labels of a vertex are combined with labels of its neighbors and then refined using a hash-like map. Hence the time complexity of this kernel is also linear to the number of edges, which endows it with the ability to deal with large-scale graphs. Rather than the perfect hashing used by the WL kernel, the neighborhood hash kernel uses hash functions based on simpler logical operations on bit-representations of vertex labels. This feature makes it slightly faster to compute the latter kernel, whereas the price to pay is the possibility of accidental hashing collisions, namely when the same values are assigned to different labels. *The nested subtree hash kernel* introduced in [Li et al., 2012] aims to enable classification on large-scale graphs over streams, where the sub-patterns increase when graphs are fed in. It is proved that this kernel is an unbiased and highly concentrated estimator of the WL kernel. Algorithms based on the WL kernel have been also developed for applications in wide domains, such as program similarity computation [Li et al., 2016] and the fast computation of the kernel on the resource description framework data that represents semantic webs [de Vries, 2013]. Relationships between the WL kernel and graph neural networks are explored in [Morris et al., 2019].

Many general graph kernel frameworks implement specific cases of the WL kernel. Fitting it into *the hash graph kernel* framework [Morris et al., 2016] enables the WL kernel to deal with continuous labels while maintaining the scalability, where randomized hash functions are used to iteratively turn continuous labels into discrete ones. Another attempt to tackle continuous labels with the WL subtree kernel is to use *the graph invariant kernel* framework [Orsini et al., 2015, Borgwardt et al., 2020]. In [Yanardag and Vishwanathan, 2015a], *the deep graph kernel* framework is applied on the WL subtree kernel, where a weight is given on each sub-pattern. In [Nikolentzos et al., 2018], *the core-based kernel framework* is applied on the WL kernel, which considers sub-structures at multiple different scales. *Message passing graph kernels* are constructed by considering a vertex kernel [Nikolentzos and Vazirgiannis, 2018]. A similar iterative procedure is applied, where the kernel between vertices in the new iteration is updated by adding the kernel between their neighbors using the weighted sum. In this way, graphs with continuous attributes can be taken into consideration. The WL subtree kernel is included in this framework.

Algorithm 3.1 Simultaneous computation of the WL subtree kernel

Input: Graph dataset $\mathbb{G}_N = \{G_1, \ldots, G_N\}$, an alphabet set Σ,
a relabeling function f, the thresholds of stopping criteria h.
Output: The gram matrix K of $\mathbb{G}_N$.

Let $i = 1$ and $\mathbb{G}_N^{(0)} = \{G_j^{(0)} = G_j \mid 0 \leqslant j \leqslant N\}$.
Compute $K^{(0)}$, with $K^{(0)}_{j_1,j_2} = k^{(0)}_{wl-subtree}(G_{j_1}, G_{j_1})$ using (3.17).
while $i \leqslant h$ **do**
(Construct multisets)
for $G^{(i-1)} = (V, E, \ell^{(i-1)}) \in \mathbb{G}_N^{(i-1)}$ **do**
for $v \in V$ **do**
Assign a multiset label $M^{(i)}(v) = \{\ell^{(i-1)}(u) \mid (v,u) \in E\}$ to v.
end for
end for
Denote the set containing all $M^{(i)}(v)$ as $\mathbb{M}^{(i)}$.
(Sort each multiset)
for $M^{(i)}(v) \in \mathbb{M}^{(i)}$ **do**
Sort $M^{(i)}(v)$ in ascending order (e.g., in lexicographical order).
$s^{(i)}(v) = \ell^{(i-1)}(v).\bigodot_{(v,u)\in E} \ell^{(i-1)}(u)$.
end for
Denote the set containing all $s^{(i)}(v)$ as $\mathbb{S}^{(i)}$.
(Compress labels)
for $s^{(i)}(v) \in \mathbb{S}^{(i)}$ **do**
Map $s^{(i)}(v)$ to a compressed label using $f : \mathbb{S}^{(i)} \to \Sigma$,
where $\forall$ vertices v_1, v_2, $f(s^{(i)}(v_1)) = f(s^{(i)}(v_2)) \Leftrightarrow s^{(i)}(v_1) = s^{(i)}(v_2)$.
end for
(Relabel)
for $G^{(i-1)} = (V, E, \ell^{(i-1)}) \in \mathbb{G}_N^{(i-1)}$ **do**
for $v \in V$ **do**
$\ell^{(i)}(v) = f(s^{(i)}(v))$.
end for
end for
Denote the set containing all $\ell^{(i)}(v)$ as $L^{(i)}$.
(Compute Gram matrix)
for $G^{(i)} \in \mathbb{G}_N^{(i)}$ **do**
$\phi^{(i)}_{wl-subtree}(G^{(i)}) = [c^{(i)}(G^{(i)}, l^{(i,1)}), \ldots, c^{(i)}(G^{(i)}, l^{(i,|L^{(i)}|)})]$,
where $c^{(i)}(G^{(i)}, l^{(i,k)})$ is the number of occurrence of $l^{(i,k)} \in L^{(i)}$ in $G^{(i)}$.
end for
(Compute kernel)
for $j_1, j_2 = 1$ to N **do**
$K^{(i)}_{j_1,j_2} = K^{(i-1)}_{j_1,j_2} + < \phi^{(i)}_{wl-subtree}(G^{(i)}_{j_1}), \phi^{(i)}_{wl-subtree}(G^{(i)}_{j_2}) >$.
end for
$i = i + 1$.
end while
$K = K^{(h)}$.

3.6 Acceleration strategies: FCSP, parallelization, and trie structure

The computational complexity limits the practicability and scalability of graph kernels. In this section, we present 3 strategies to reduce the computing time and memory usage to compute graph kernels: The *Fast Computation of Shortest Path Kernel* method, parallelization, and the trie structure. Datasets and environment settings applied in this section are respectively described in detail in Section 2.4 and Section 3.7.

3.6.1 The *Fast Computation of Shortest Path Kernel* method

To compute the shortest path kernel between 2 graphs G_1 and G_2, shortest paths between all pairs of vertices in both graphs are compared. Each time we compare 2 shortest paths, both their corresponding pairs of vertices are compared once, causing significant redundancy during the vertex comparison procedure. To illustrate such redundancy, consider the example given in Figure 3.9. When comparing shortest paths (v_1, v_2) and (v_5, v_4, v_7), the vertex kernels $k_v(v_1, v_5)$ and $k_v(v_2, v_7)$ need to be computed; while comparing paths (v_1, v_2, v_3) and (v_5, v_4, v_7, v_6), the vertex kernels $k_v(v_1, v_5)$ and $k_v(v_3, v_6)$ are evaluated. Kernel between vertices v_1 and v_5 is unnecessarily computed twice. In general, if G_1 has n_1 vertices and G_2 has n_2 vertices, then there are at most n_1^2 shortest paths in G_1 and n_2^2 shortest paths in G_2, thus $n_1^2 n_2^2$ comparisons between shortest paths and $2n_1^2 n_2^2$ comparisons between vertices are required to compute the kernel. Each pair of vertices is compared $2n_1 n_2$ times on average.

The *Fast Computation of Shortest Path Kernel* (FCSP) method reduces this redundancy. Initially introduced in [Xu et al., 2014] for vertex comparing in the shortest path kernel, we extend it to the structural shortest path kernel, and integrate

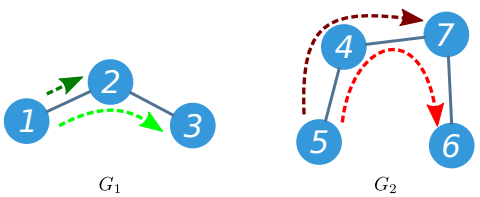

Figure 3.9 – An example of the redundant comparisons between vertices and edges.

Algorithm 3.2 Fast computation of the shortest path kernel (FCSP)

Input: Shortest-path graphs $G_1^\star = (V_1, E_1)$ and $G_2^\star = (V_2, E_2)$.
A vertex kernel k_v, an edge kernel k_e.
Output: The shortest path kernel k between $G_1^\star$ and $G_2^\star$.

Compare and store vertex kernels:
for $v_i \in V_1$ **do**
  for $v_j \in V_2$ **do**
    $VK[(v_i, v_j)] = k_v(v_i, v_j)$.
  end for
end for

Compute kernel:
Let $k = 0$.
for $e_i = (v_{i_1}, v_{i_2}) \in E_1$ **do**
  for $e_j = (v_{j_1}, v_{j_2}) \in E_2$ **do**
    $k = k + VK[(v_{i_1}, v_{j_1})] * VK[(v_{i_2}, v_{j_2})] * k_e(e_i, e_j)$.
  end for
end for

it for edge comparison. In the following, we describe the FCSP method and these proposed extensions, as well as some experimental analysis.

The FCSP is presented in Algorithm 3.2. Instead of comparing vertices during the procedure of comparing shortest paths, for 2 shortest-paths graphs $G_1^\star$ and $G_2^\star$, FCSP first compares each vertex in $G_1^\star$ with each vertex in $G_2^\star$, and then stores the comparison results in VK. For the convenience of extension, VK is a dictionary of length $n_1 n_2$, rather than a $n_1 \times n_2$ matrix which is used in [Xu et al., 2014]. Each item in VK is a key-value pair. The key is a tuple (v_1, v_2), where v_1 and v_2 are respectively vertices in $G_1^\star$ and $G_2^\star$, and the value is the vertex kernel between v_1 and v_2. After this, when comparing shortest paths, comparison results of corresponding vertices are retrieved from VK. Notice the edge kernel k_e in Algorithm 3.2 deals with weighted lengths of the shortest paths. This method reduces vertex comparisons to at most $n_1 n_2$ times, with an additional memory usage of size $\mathcal{O}(n_1 n_2)$. In practice, FCSP can reduce the time complexity up to several orders of magnitude. See Figure 3.10 for details.

In our implementation, we apply this vertex comparing method to the shortest path kernel, as recommended by [Xu et al., 2014]. Moreover, we also extend this strategy and apply it to the structural shortest path kernel. This allows reducing

Algorithm 3.3 FCSP considering edges

Input: Graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$.
A vertex kernel k_v, an edge kernel k_e.
Output: The structural shortest path kernel k between G_1 and G_2.

Compare and store vertex kernels:
for $v_i \in V_1$ **do**
  for $v_j \in V_2$ **do**
    $VK[(v_i, v_j)] = k_v(v_i, v_j)$.
  end for
end for

Compare and store edge kernels:
for $e_i \in E_1$ **do**
  for $e_j \in E_2$ **do**
    $EK[(e_i, e_j)] = k_e(e_i, e_j)$.
  end for
end for

Compute kernel:
Extract all shortest paths $P_1 = \{p_{1,1}, \ldots, p_{1,h_1}\}$ and $P_2 = \{p_{2,1}, \ldots, p_{2,h_2}\}$ between all pairs of vertices in G_1 and G_2, respectively.
Let $k = 0$.
for $p_i = (v_{i,0}, e_{i,1}, v_{i,1} \ldots, e_{i,n_i}, v_{i,n_i}) \in P_1$ **do**
  for $p_j = (v_{j,0}, e_{j,1}, v_{j,1} \ldots, e_{j,n_j}, v_{j,n_j}) \in P_2$ **do**
    if $n_i == n_j$ **then**
      $k_p = VK[(v_{i,0}, v_{j,0})]$.
      for $m \in \{1, \ldots, n_i\}$ **do**
        $k_p = k_p * EK[(e_{i,m}, e_{j,m})] * VK[(v_{i,m}, v_{j,m})]$.
      end for
      $k = k + k_p$.
    end if
  end for
end for

more redundancy since this kernel requires comparisons between all vertices on each pair of shortest paths. If the average length of the shortest paths in G_1 and G_2 is h, the new method is at most $n_1 n_2 h$ times faster than the direct comparison.

We further extend this strategy to edge comparison when computing the structural shortest kernel. Algorithm 3.3 details this extension. First, same as in Algorithm 3.2, a sub-kernel between each vertex of graph G_1 and each vertex of

graph G_2 are computed and then stored in the dictionary VK. After that, each edge of G_1 is compared with each edge of G_2 using an edge kernel, and the result stored in a dictionary EK of the same type containing key-value pairs. Each key stores an edge pair (e_1, e_2) and value the corresponding comparison result. Finally, when computing the graph kernels, sub-kernels between vertices and edges are respectively acquired from VK and EK with the time complexity of $\mathcal{O}(1)$. When undirected graphs are considered, EK stores all possible permutations of (e_1, e_2), as an edge may start from its both sides. That is to say, let $e_1 = (v_{11}, v_{12})$ and $e_2 = (v_{21}, v_{22})$, then $((v_{11}, v_{12}), (v_{21}, v_{22}))$, $((v_{12}, v_{11}), (v_{21}, v_{22}))$, $((v_{11}, v_{12}), (v_{22}, v_{21}))$, and $((v_{12}, v_{11}), (v_{22}, v_{21}))$ are all entries of EK. Notice that G_1 and G_2 are the original graphs between which the graph kernel is evaluated, rather than the shortest-paths graphs as in Algorithm 3.2. If G_1 has m_1 edges and G_2 has m_2 edges, then it requires $m_1 m_2$ times of edge label comparisons, compared to $n_1^2 n_2^2 h$ times by the original method. The former is often much smaller, as long as the vertex degrees of graphs are not too high, which is generally the case in many fields, such as bioinformatics and social media. Table 2.2 in Page 34 lists the average vertex degrees of datasets that we use in experiments, which vary from 1.32 to 4.98.

Figure 3.10 compares runtimes to compute Gram matrices with and without the FCSP method. To avoid the influence of parallelization on runtime (see Section 3.6.2 for more detail), all computations are serially run on a single CPU. Graphs without edges are omitted for the shortest path kernel. On the small dataset *Alkane*, the FCSP achieves similar performance with the naive computation; while in every other circumstance, the FCSP dominates the performance, with a speedup up to more than 28 times for the shortest path kernel (on the *PAH* dataset) and 37 times for the structural shortest path kernel (on the *PAH* dataset).

It may be interesting to generalize this strategy to any graph kernels based on bags of patterns, as long as it requires explicit comparisons between vertices and/or edges.

3.6.2 Parallelization

Parallelization may significantly reduce the computational time. The basic concept of parallelization is to split a set of computation tasks into several pieces, and then carry them out separately on multiple computation units such as CPUs or GPUs. We implement parallelization with Python's `multiprocessing.Pool` module in two

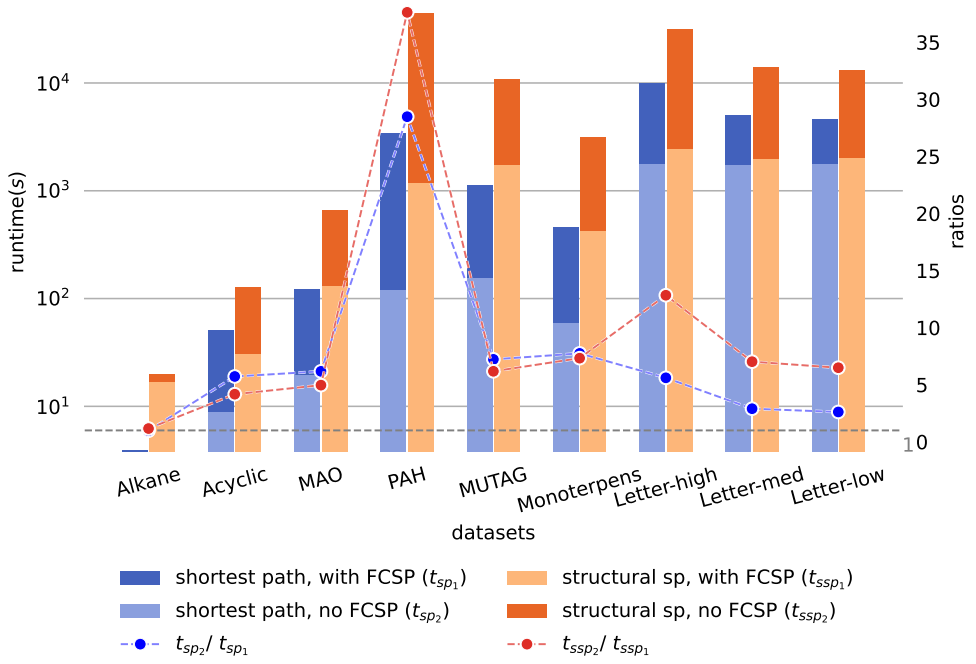

Figure 3.10 – Left y-axis: Runtimes (in seconds) to compute Gram matrices with the FCSP method (bottom of each pillar) and without it (top of each pillar) of the shortest path kernel (blue pillar) and the structural shortest path kernel (orange pillar) with parallelization. Right y-axis: The blue dots are the ratios between the runtimes to compute Gram matrices of the shortest path kernel with the naive and the FCSP method, while the red dots are the ones of the structural shortest path kernel.

aspects: In cross-validation, parallelization is carried out over the set of trials; when computing the Gram matrices of graph kernels, parallelization is performed over pairs of graphs. The special cases of the latter are listed as follows:

- *Marginalized kernel*: when the technique to remove tottering is applied (see Section 3.2.4), parallelization is performed first over graphs to construct untottering graphs and then over pairs of these untottering graphs to compute the Gram matrix.
- *Shortest path kernel*: parallelization is performed first over graphs to compute their corresponding shortest-paths graphs and then over pairs of these shortest-paths graphs to compute Gram matrix (see Section 3.3.1).
- *Structural shortest path kernel*: parallelization is performed first over graphs to extract all shortest paths in each graph and then over pairs of sets of shortest paths corresponding to pairs of graphs to compute the Gram matrix (see Section 3.3.2).

- *Path kernel up to length h*: parallelization is performed first over graphs to extract the set of paths with length up to h in each graph and then over pairs of sets of paths corresponding to pairs of graphs to compute the Gram matrix (see Section 3.3.3). The set of paths can be extracted in the form of a trie (see Section 3.6.3).
- *Treelet kernel*: First, parallelization is performed over graphs. For each graph, a set is constructed to record the number of the occurrence of each treelet in the form of its canonical key. Then, parallelize over pairs of the constructed sets that correspond to pairs of graphs to evaluate the Gram matrix (see Section 3.5.1).
- *WL subtree kernel*: parallelization is performed over pairs of graphs. Notice in this case it is unable to use the simultaneous operations on labels presented in Algorithm 3.1. Instead, operations on labels are carried out for each graph every time a kernel between a pair of graphs is evaluated (see Section 3.5.2).

A widely used measure of the quality of a parallelization on a task is *speedup* [Amdahl, 1967], which is the ratio between the times used before and after the parallelization. Amdahl's law is a rule of thumb to evaluate the speedup [Amdahl, 1967], which is defined as

$$\mathrm{S}_{latency}(n) = \frac{1}{(1-p) + \frac{p}{n}}, \tag{3.18}$$

where $\mathrm{S}_{latency}$ is the theoretical speedup, n is the number of threads, i.e., computing cores over which the task is parallelized, and p is the proportion of the runtime to execute the parallelized part in the task before parallelization. When $n \to \infty$, $\mathrm{S}_{latency}$ reaches its upper limit $\frac{1}{1-p}$.

Amdahl's law shows that the proportion of the parallelized part and the number of computing cores will restrict the speedup of a parallelization. However, this definition is based on the assumption that the time t_p to execute the parallelized part is reversely proportional to the number of computing cores n, which is untrue in practice. The reason is that extra work needs to be involved to carry out the parallelization, such as splitting the data, sending data to each computing core, and collecting results from these cores. To consider this situation, we modify Amdahl's law as

$$\mathrm{S}'_{latency}(n) = \frac{1}{(1-p) + \frac{p}{n} + \frac{1}{r}}, \tag{3.19}$$

where r is the ratio between the time to execute the parallelized part before parallelization and the extra time required to carry out the parallelization. Factors that

may impact r include the number of computing cores n, the transmission bandwidth between computing cores, the method to split the data, and the computational complexity to tackle one piece of data. In the remainder of this section, we examine two crucial factors: the number of computing cores n and `chunksize`.

Number of computing cores

As shown in (3.19), the number of computing cores indicates the potential power of parallelization. Figure 3.11 reveals the influence of parallelization and CPU core numbers (7 versus 28), on runtime to compute Gram matrices and to perform model selections for the shortest path kernel on 8 datasets. Moreover, we present the *speedup* between runtimes to compute the Gram matrices on 28 and 7 cores. The *speedup* for large-scale datasets are around 4, which turns out to be the inverse ratio of the number of CPU cores (28/7). It is worth noting that the *Letter-med* dataset has the largest number of graphs (but with relatively "small" graphs), and *Enzymes* has the "average-largest" graphs (the second one being *PAH*).

`chunksize`

Ideally, parallelization is more efficient when more computing cores are applied. However, as (3.19) reveals, this efficiency may be suppressed by the parallel procedure required to distribute data to computing cores and collect returned results. Parallelizing relatively small graphs to a large number of computing cores may be more time consuming than non-parallel computation. For instance, Figure 3.11 shows that it takes almost the same time to compute the Gram matrix of the small dataset *Alkane* on 28 and 7 CPU cores, indicating that the time efficiency raised by applying more CPU cores is nearly neutralized by the cost to allocate these cores. To tackle this problem, it is essential to choose an appropriate `chunksize`, which describes how many data are grouped together as one piece to be distributed to a single computing core.

In Figure 3.12, runtimes to compute the Gram matrices of the shortest path kernel with different `chunksize` values are compared on 28 CPU cores. When `chunksizes` are too small, the runtimes become slightly high, as the parallel procedure costs too much time; as `chunksizes` become bigger, the runtimes turn smaller, and then reach the minima; after that, the runtimes may become much bigger as `chunksizes` continue growing, due to the waste of computational resources. The

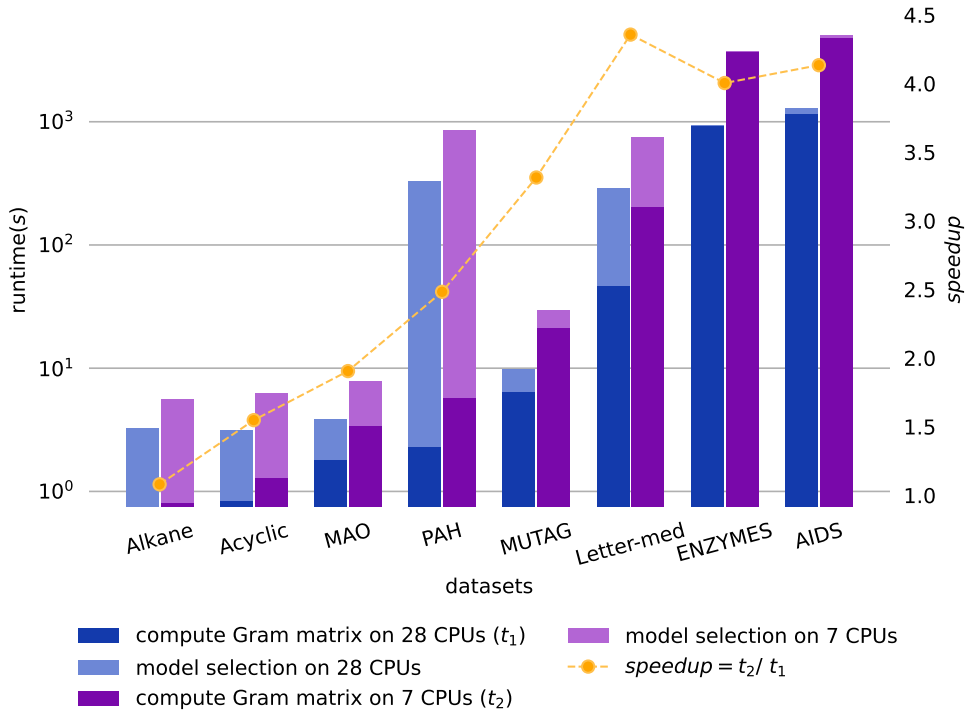

Figure 3.11 – Left y-axis: Runtimes (in seconds) to compute Gram matrices (bottom of each pillar) and perform model selections (top of each pillar) of the shortest path kernel on each dataset on 28 CPU cores (blue pillar) and 7 CPU cores (purple pillar) with parallelization. Right y-axis: The orange dots are the ratios between the runtimes to compute Gram matrices of each dataset on 28 and 7 cores.

minimum runtime of each dataset (shown with the vertical dot lines) varies due to the time and memory consumed to compute Gram matrices. Computations with wise `chunksize` choices could be more than 20 times faster than the worst choices. In our experiments, for convenience of implementations and comparisons, the `chunksize` to compute an N × N Gram matrix on $n_\heartsuit$ CPU cores is set to 100 if $N^2 > 100 n_\heartsuit$; and $N^2 / n_\heartsuit$ otherwise. The value 100 is chosen since the corresponding runtimes are close enough to their minima on all the datasets.

The ratio between runtimes of the worst and the best `chunksize` settings for each graph kernel on each dataset is shown in Figure 3.13. On all available settings, the proper choices of the `chunksizes` speed up the computation. Some are more than 30 times faster than the worst `chunksize` settings (i.e., the path kernel up to length h and the treelet kernel on *Letter-med*).

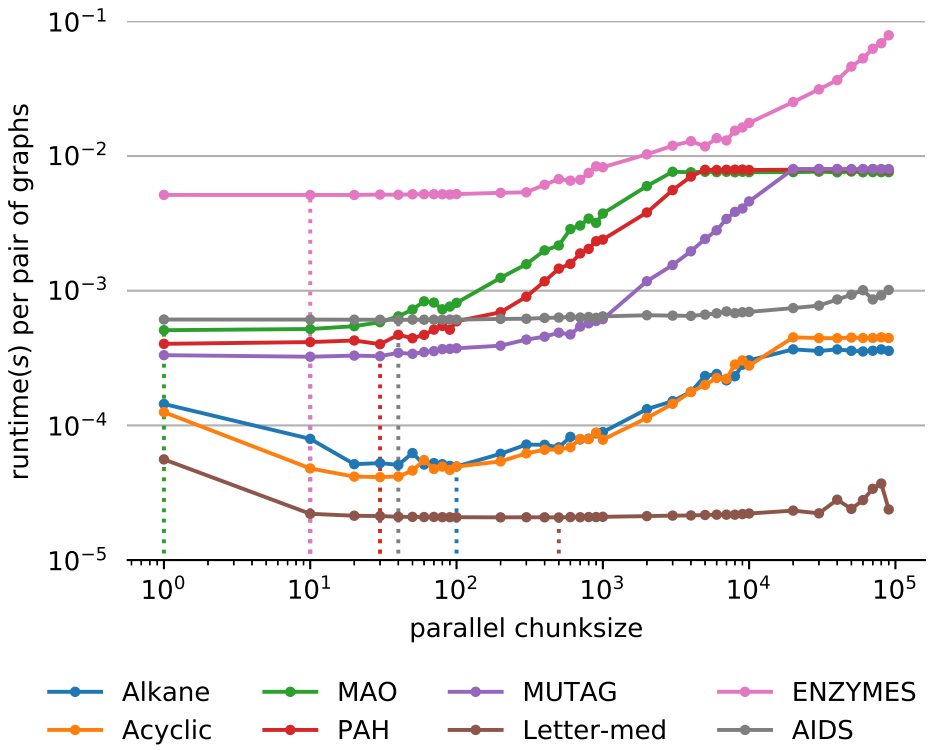

Figure 3.12 – Runtimes to compute the Gram matrices of the shortest path kernel on each dataset on 28 CPU cores with different `chunksize` values.

3.6.3 The trie Structure

In some graph kernels that require comparison of paths (i.e., the path kernel up to length h), the paths are pre-computed for the sake of time complexity. However, for large datasets, when the maximum limits of the lengths of paths are high, storing these paths becomes memory-consuming. In [Ralaivola et al., 2005], the authors proposed a suffix tree data structure for fast computation of path kernels. Inspired by that, we employ the trie data structure [Fredkin, 1960] to store paths in order to tackle the memory problem, as described next.

A trie, also called a prefix tree, is a data structure to store dictionaries, where the keys are normally strings. Each node in a trie represents a string or a prefix of a string except for the root node, which is associated with an empty string. The string associated with the child of a node shares the same prefix with the string associated with that node. Each node may also be associated with a value that represents the number of occurrences of the associated string. Figure 3.14 illustrates a trie to store a set of strings. Each orange node corresponds to a string on the left and the

	common walk	marginalized	Sylvester equation	conjugate gradient	fixed-point iterations	spectral decomposition	shortest path	structural SP	path up to length h	treelet	WL subtree
Alkane	1.09	24.94	8.1	23.14	24.22	1.71	20.39	23.29	2.75	2.51	1.02
Acyclic	1.24	25.19	8.9	19.11	18.27	1.51	20.36	22.56	4.89	2.72	1.02
MAO	1.68	24.59	4.32	1.14	1.06	7.25	20.96	24.91	6.46	2.0	1.03
PAH	1.32	26.05	5.17	1.15	1.16	10.19	24.98	26.98	2.06	3.26	1.04
Mutag	1.16	27.23	12.09	1.39	1.57	5.69	25.43	27.47	13.5	6.17	1.01
Letter-med	1.43	1.54	14.83	1.46	1.65	1.61	2.63	2.37	31.54	30.35	1.01
Enzymes	1.43	14.66	1.85	2.03	1.73	inf	14.52	inf	16.23	17.22	1.01
AIDS	1.08	1.59	2.0	1.07	1.08	inf	1.8	2.15	1.8	21.04	1.01
NCI1	inf	1.17	1.94	inf	inf	inf	1.27	inf	4.14	23.94	1.07
NCI109	inf	1.27	1.93	inf	inf	inf	1.3	inf	6.04	32.0	1.07
D&D	inf	inf	inf	inf	inf	inf	inf	inf	inf	inf	1.01

Figure 3.13 – The ratio between runtimes of the worst and the best `chunksize` settings for each graph kernel on each dataset. Darker color indicates a better result. Gray cells with the "inf" marker indicate that the computation of the graph kernel on the dataset is omitted due to much higher consumption of computational resources than other kernels.

number inside the node counts the occurrence of that string. The latter is not necessarily needed to build a trie. Notice a string is implied in the structure of the trie itself, namely the position of the corresponding node, rather than stored directly in that node. Trie tends to occupy less space when common prefixes take larger proportions. In the example given in Figure 3.14, the original strings require 33 units of space, while the corresponding trie takes only 13. We consider the following basic operations of a trie:

- *Insertion*: To insert a string s to a trie of length h, successively look up each character c in s starting in the children of the root node n_r as the current parent. If child n_i is associated with c, then set n_i as the current parent and move to the next character; otherwise insert s as a new sub-tree of n_r. The worst time complexity of insertion is $\mathcal{O}(h)$.
- *Construction*: Constructing a trie involves repeating the insertion process. For n strings with the average length h, the time complexity is $\mathcal{O}(hn)$.
- *Lookup*: Look up a string in a trie uses the same procedure as inserting it, ex-

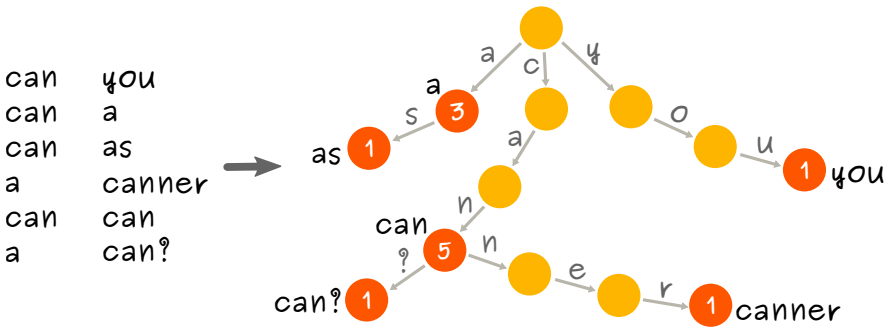

Figure 3.14 – An illustration of a trie structure to store a set of strings.

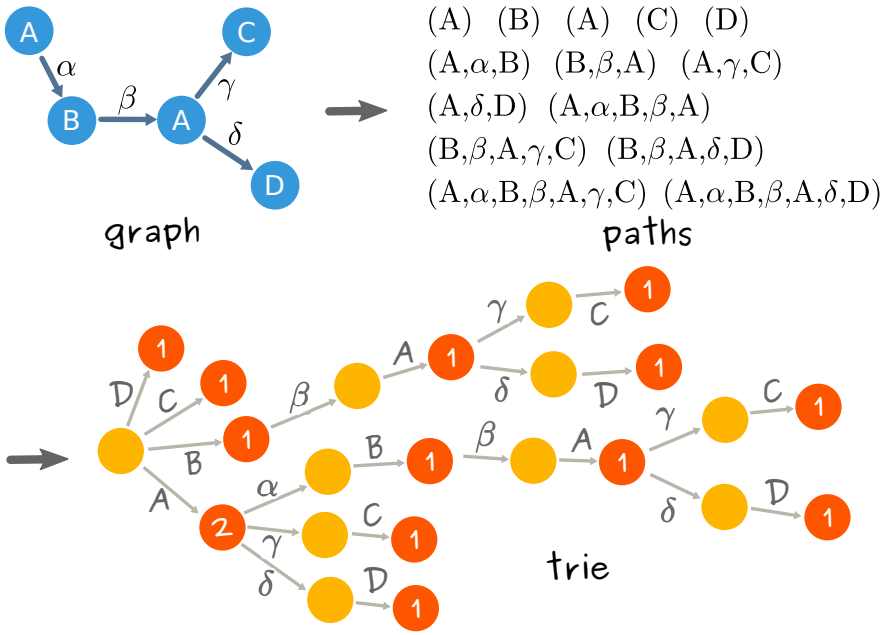

Figure 3.15 – Construction of a trie from paths in a graph.

cept that no node is inserted into the trie. Thereby the worst time complexity is also $\mathcal{O}(h)$ for a length-h string.

In a labeled graph, we use trie to combine the common prefixes of paths together to reduce space complexity. Vertex and edge labels with arbitrary dimensions can be handled, as long as they are symbolic. Figure 3.16 shows an example of how all paths are extracted from a graph, with labels of vertices (the modern English letters) and edges (the Greek letters) constituting label sequences, and construct a trie from them. Each node in the trie (except for the root) represents a label sequence and each orange one represents a path. The trie reduces the required space from 46 units to 23 units. Take the path kernel up to length h for example. Let n be the average vertex number of each graph, d be the average vertex degree, and l the different

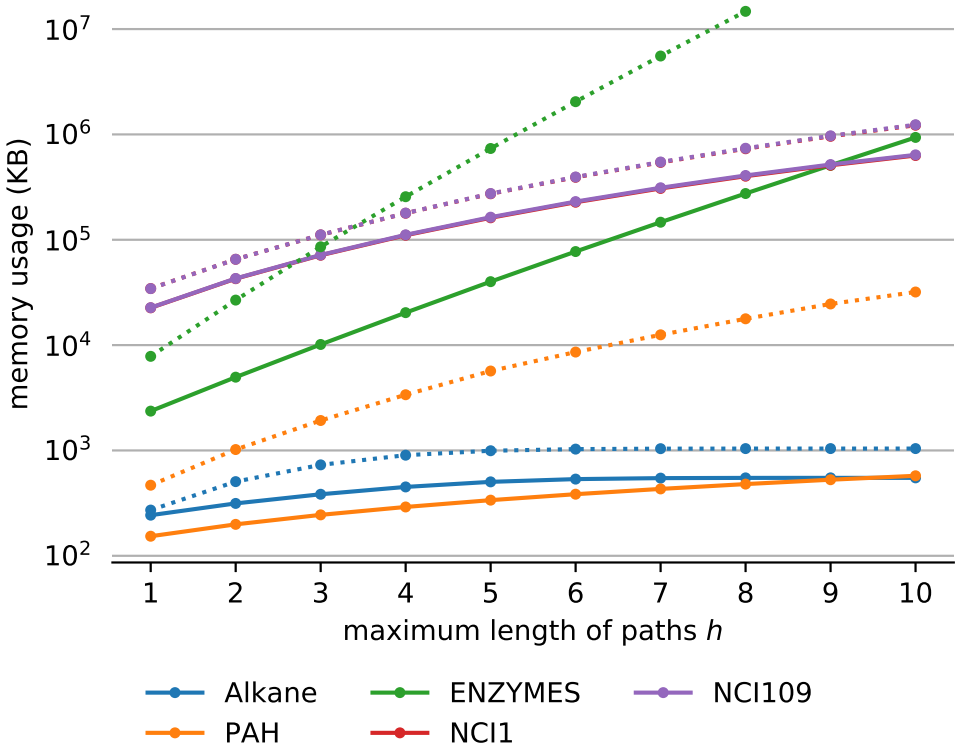

Figure 3.16 – Memory usages to store paths in all graphs in each dataset under different maximum length of paths h. Dot lines represent explicit storage of paths, and solid lines represent storage using the trie structure. Note that lines of datasets *NCI1* and *NCI109* overlap.

vertex labels in total (for conciseness, only symbolic vertex labels are considered here). When paths are stored explicitly in memory (e.g. using the Python `list` object), the space complexity to store paths in one graph is in $\mathcal{O}(n(1+d+\cdots+d^h))$; The trie reduces it to $\mathcal{O}(l(1+l+\cdots+l^h))$, making it very efficient for large datasets and graphs, or when h is high. See Figure 3.16 for a comparison.

The evaluation of the path kernel up to length h between two graphs G_1 and G_2 can be carried out while traversing the corresponding tries. Algorithm 3.4 formalizes this procedure for the Tanimoto kernel (see Section 3.3.3) in four steps. Step 1, construct the tries for T_1 (resp. T_2) for the paths up to length h in G_1 (resp. G_2) and initialize the numerator sum_n and the denominator sum_d of (3.11), namely the intersection and the union of path sets in G_1 and G_2, respectively. Step 2, find all paths in T_1 with a Deep-first search. For each path, increase sum_d by 1 and check whether it is in T_2. If so, then increase sum_n by 1 (the first function in Algo-

Algorithm 3.4 The evaluation of the path kernel up to length h with the Tanimoto kernel using trie

Input: Graphs G_1 and G_2.
Output: The path kernel k up to length h between G_1 and G_2.

1: Construct the tries T_1 (resp. T_2) for the paths up to length h in G_1 (resp. G_2), as illustrated by Figure 3.15.
2: Let $sum_n = 0$ and $sum_d = 0$.
3: Let n_{r_1} (resp. n_{r_2}) be the root node of T_1 (resp. T_2).
4: Update sum_n and sum_d by function `TRAVERSE_TRIE1`(n_{r_1}, T_2, sum_n, sum_d) in Algorithm 3.5.
5: Update sum_n and sum_d by function `TRAVERSE_TRIE2`(n_{r_2}, T_1, sum_n, sum_d) in Algorithm 3.5.
6: $k = sum_n / sum_d$.

rithm 3.5). Step 3, find the paths in T_2 that are not in T_1 by the similar procedure of Step 2 (the second function in Algorithm 3.5). Step 4, compute the kernel as the ratio of sum_n and sum_d. For the MinMax kernel (3.12), a similar four-step process can be used by modifying Step 2 and Step 3. Each time a path is found, the numbers of its occurrence in two graphs are also retrieved. sum_n and sum_d are increased by the comparing results of these numbers, rather than by 1 (Lines 5, 6, and 22 in Algorithm 3.5).

Although both saving paths to the trie structure and retrieving paths from it require extra computing time, less memory usage may avoid swapping between memory and hard disk, which saves more time in practice. As a result, the users should use the trie structure according to the limits of their computing resources. In our implementation, it is implemented for the path kernel up to length h.

Algorithm 3.5 Auxiliary functions for Algorithm 3.4

```
1: function TRAVERSE_TRIE1(n_r, T_2, sum_n, sum_d, s = ())
2:     for each child n_c of n_r do
3:         Add the label ℓ corresponding to n_c to the end of s.
4:         if s represents a path then
5:             sum_d += 1.
6:             Look up s in T_2, if found, then sum_n += 1.
7:         end if
8:         if n_c has children then
9:             Run function TRAVERSE_TRIE1(n_c, T_2, sum_n, sum_d, s).
10:        else
11:            Remove the last label from s.
12:        end if
13:    end for
14:    if s is not empty then
15:        Remove the last label from s.
16:    end if
17: end function

18: function TRAVERSE_TRIE2(n_r, T_1, sum_n, sum_d, s = ())
19:    for each child n_c of n_r do
20:        Add the label ℓ corresponding to n_c at the end of the label sequence s.
21:        if s represents a path then
22:            Look up s in T_1, if not found, then sum_d += 1.
23:        end if
24:        if n_c has children then
25:            Run function TRAVERSE_TRIE2(n_c, T_1, sum_n, sum_d, s).
26:        else
27:            Remove the last label from s.
28:        end if
29:    end for
30:    if s is not empty then
31:        Remove the last label from s.
32:    end if
33: end function
```

3.7 Experiments and analyses

In this section, we perform each graph kernel on synthesized graph datasets as well as real-world benchmark graph datasets listed in Table 2.2. We analyze the accuracy and computational complexity according to the type of graphs, thus offering some advice to choose graph kernels based on the type of the dataset at hand, and discussing which ones work on particular graphs.

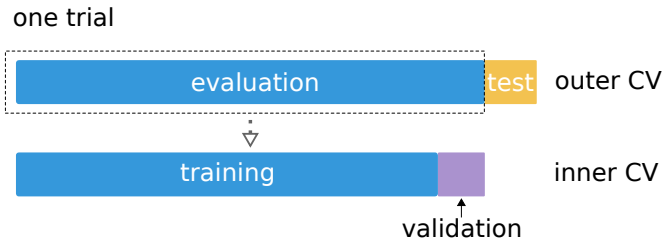

Figure 3.17 – Illustration of the two-layer nested cross validation.

The methods used for prediction are kernel ridge regression [Murphy, 2012] and support vector machines [Boser et al., 1992] for classification. A two-layer nested cross validation (CV) method is applied to select and evaluate models as follows. As Figure 3.17 shows, in the outer CV, the whole dataset is first randomly split into 10 folds, nine of which serve for model evaluation and one for an unbiased estimate of the performance. Then, in the inner CV, the evaluation set is split into 10 folds, nine of which are used for training, and the remaining split is used for validating the tuning of the hyper-parameters. This procedure is repeated 30 times, a.k.a., 30 trials, and the final results correspond to the average over these trials. Strategies proposed in Section 3.6 are applied through the experiments unless stated otherwise. Table 3.2 shows the environment settings for the experiments.

Table 3.2 – Environment settings for experiments.

Environments	Settings
CPU	Intel(R) Xeon(R) E5-2680 v4 @ 2.40GHz
# CPU cores	28
Memory (in total)	252 GB
Operating system	CentOS Linux release 7.3.1611, 64 bit
Python version	3.6.9

3.7.1 Performance on synthesized graphs

In this section, the performance and properties of each graph kernel are studied using synthesized graphs, with runtimes estimated on the Gram matrix computations.

First, we study the scalability of the kernels by increasing the number of graphs in the dataset (from 100 to 1000 graphs), where the generated unlabeled graphs consist of 20 vertices and 40 edges randomly assigned to pairs of vertices. Figure 3.18(a) shows the runtimes for all kernels. Since all the lines are linear with the y-axis being in square root scale, this indicates that the runtimes are quadratic in the number of graphs.

The scalability is also analyzed by increasing the number of vertices (from 10 to 100), where 10 datasets of 100 unlabeled graphs each were generated. The average degree of all generated graphs is equal to 2 and the edges are randomly assigned to pairs of vertices. Figure 3.18(b) shows the runtimes increasing in the number of vertices for all kernels. The path kernel up to length h and the WL subtree kernel have the best scalability. One reason that they are fast is that only a small part of sub-patterns in these two kernels need to be taken into consideration for a good performance.

To study the scalability of the kernels w.r.t. the average vertex degree, we generated unlabeled graphs consisting of 20 vertices with increasing degrees (from 1 to 10). The edges are randomly assigned to pairs of vertices. For each of the 10 degree values, we generated 100 graphs. Figure 3.18(c) shows that most kernels have good scalability to the degrees, the worst being the treelet kernel, as the number of treelets in each graph increases rapidly with the degree.

To study the scalability of the kernels w.r.t. the alphabet sizes of symbolic vertex labels, we generated unlabeled graphs of 20 vertices and 40 edges randomly assigned to pairs of vertices. The vertices are symbolically labeled with increasing alphabet sizes, and the edges are unlabeled. For each alphabet size (from 0 to 20), we generated 100 graphs. As Figure 3.18(d) shows, the runtimes of the path kernel up to length h increases significantly with the alphabet size. This is because the trie structures to store paths become bigger, and it requires more time to construct and compare them. In contrast, the runtimes of the common walk kernel and the structural shortest path kernel become smaller when the alphabet size becomes bigger. The former is related to smaller direct product graphs (see [Gärtner et al., 2003]), and the latter is caused by the reduced comparison between vertex labels through the shortest paths.

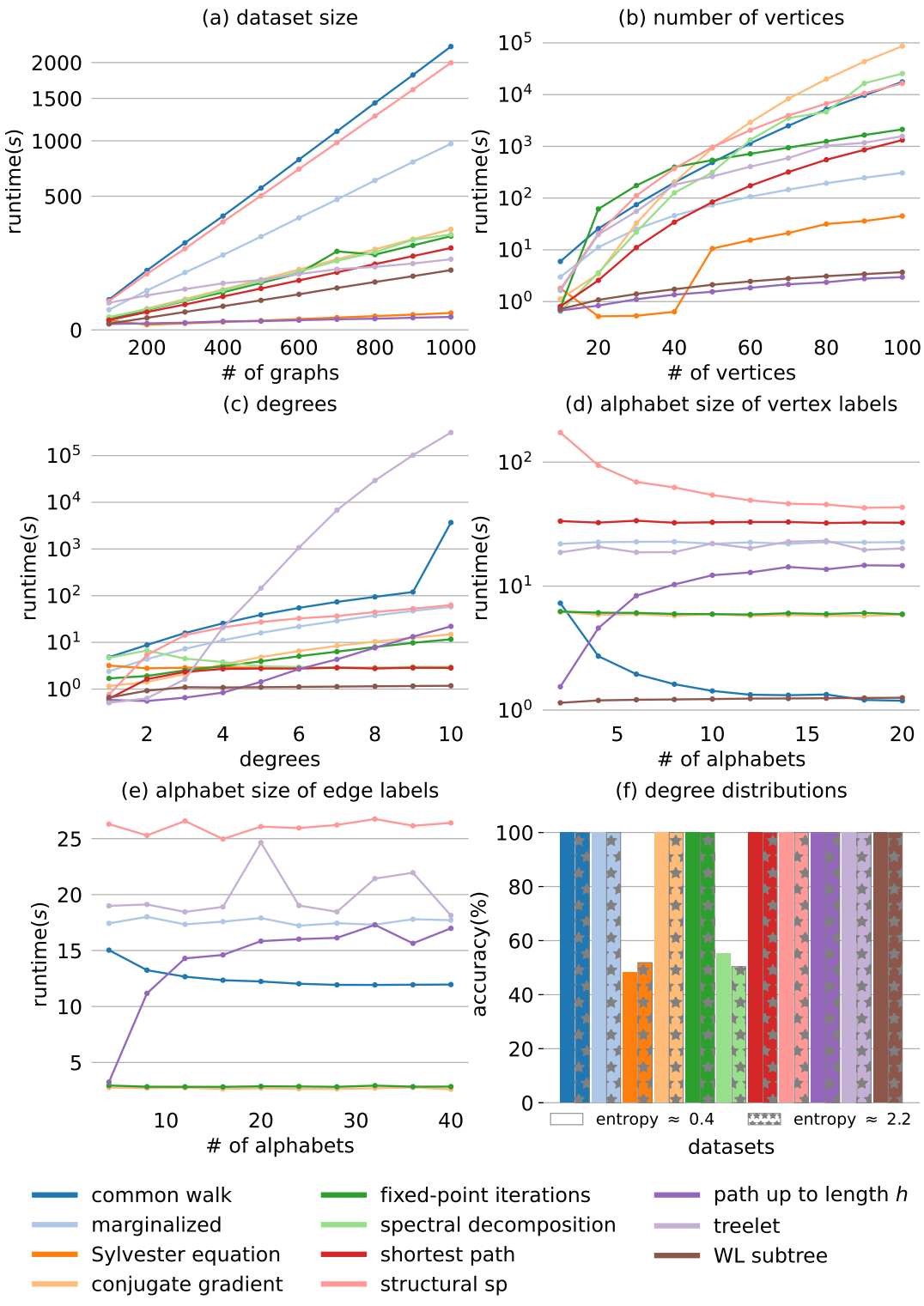

Figure 3.18 – Performance of all graph kernels on synthesized graphs.

The scalability of the kernels w.r.t. the alphabet sizes of edge labels is studied in the same way, except that the edges are symbolically labeled with increasing alphabet sizes, and the vertices are unlabeled. For each alphabet size $(0,4,8,12,\ldots,40)$, we generated 100 graphs. According to Figure 3.18(e), the runtimes of the path kernel up to length h increase with the alphabet sizes, caused by the aforementioned reason concerning the trie structures. In general, the influence of the alphabet size on the runtime is small for all kernels.

Finally, we studied the classification performances of the kernels on graphs with different amounts of entropy on degree distributions. For this reason, we generated two sets of 200 graphs with 40 vertices. Graphs of the first set have low entropy on degree distributions, while graphs of the second set have high entropy (0.4 versus 2.2 on average). Each set has two classes, one consisting of half of the graphs with one label on vertices and the other class another label. A classification task is performed on each set using the SVM classifier, and the accuracy is evaluated. Figure 3.18(f) shows that all graph kernels achieve equivalent accuracy on both sets of different degree distributions. Except for the Sylvester equation kernel or the spectral decomposition kernel that cannot deal with labels, all graph kernels achieve high accuracy. It indicates that these graph kernels are suitable for various degree distributions.

As a conclusion, the dataset size and the number of vertices have the most significant effect on the computation runtimes of the aforementioned graph kernels. Some graph kernels may be limited for large datasets due to their scalability. For the treelet kernel, the influence of vertex degree is also significant. These effects should be examined carefully before using the kernels. We provide in the following a deeper

Table 3.3 – The ranges of hyper-parameters for each kernel

Kernels	Hyper-parameter ranges
Common walk	method: geo, γ: [0.01, 0.02, …, 0.15] method: exp, γ: [0,1, 2, …, 15]
Marginalized	iter: $[1,5,\ldots,20]$, p_q: [0.1, 0.2, …, 1.0]
Sylvester equation	λ: [1-10, 1e-9, …, 1e-1]
Conjugate gradient	λ: [1-10, 1e-9, …, 1e-1]
Fixed-point iterations	λ: [1-10, 1e-9, …, 1e-1]
Spectral decomposition	λ: [1-10, 1e-9, …, 1e-1]
Path kernel up to length h	h: [1, 2, …, 10], $kfunc$: [MinMax, Tanimoto]
Treelet	kernel: gaussian, γ: [1e-10, 1e-9, …, 1] kernel: polynomial, d: [1, 2, …, 10], c: [1, 10, …, 1e10], γ: [1e-10, 1e-9, …, 1]
WL subtree	height: [0, 1, …, 10]

analysis by analyzing the performance on well-known real-world datasets.

3.7.2 Performance on the real-world datasets

In this section, experiments and analyses are performed on real-world datasets listed in Table 2.2 on Page 34. Table 3.3 summarizes the ranges of hyper-parameters over which each graph kernel is optimized with a grid search CV. Tables 3.4 and 3.5 gather the performances of all these kernels on all datasets for regression and classification tasks, respectively.

Table 3.4 – Results of all graph kernels on datasets for regression tasks

Datasets	Kernels	Train Perf	Valid Perf	Test Perf	Parameters	t_{gm}	t_{all}	
	Common walk	6.76±0.72	10.79±2.08	15.52±15.10	method: geo, γ: 0.06, α: 1e-10	2.24"/3.04"±0.83"	184.17"	1
	Marginalized	41.82±2.41	42.38±2.16	43.75±18.88	iter: 16, p_q: 0.1, α: 1e-10	4.68"/3.25"±1.48"	330.01"	2
	Sylvester equation	6.89±0.35	12.60±1.28	8.97±8.84	λ: 0.01, α: 3.16e-9	**0.37"**/0.38"±0.02"	20.11"	3
	Conjugate gradient	7.17±0.48	12.37±1.56	11.13±11.10	λ: 0.1, α: 1e-8	0.76"/0.66"±0.04"	26.19"	4
	Fixed-point iterations	14.66±0.38	17.35±0.91	12.78±2.33	λ: 1e-3, α: 1e-8	0.64"/0.60"±0.06"	20.28"	5
Alkane	Spectral decomposition	10.62±0.36	13.33±1.13	12.95±6.74	λ: 0.1, α: 1e-10	0.59"/0.65"±0.08"	43.84"	6
	Shortest path	7.87±0.16	8.76±0.22	7.81±1.51	α: 1e-8	0.75"	**3.23"**	7
	Structural SP	7.89±0.17	11.04±0.30	8.65±1.55	α: 0.1	1.05"	**3.20"**	8
	Path up to length h	0.52±0.03	6.96±1.04	9.00±12.87	h: 9, k_func: MinMax, α: 3.16e-3	0.48"/0.51"±0.04"	52.20"	9
	Treelet	1.10±0.04	2.57±0.21	**2.53±1.32**	kernel: gaussian, γ: 1e-6, α: 1e-10	0.48"/0.50"±0.04"	212.68"	10
	WL subtree	5.22±0.11	21.99±4.36	26.42±41.59	height: 2, α: 3.16e-4	**0.38"**/1.45"±0.89"	53.65"	11
	Common walk	7.60±0.22	12.77±1.00	12.93±3.91	method: geo, γ: 0.04, α: 1e-8	1.84"/2.27"±0.47"	177.87"	12
	Marginalized	11.17±0.42	17.77±1.50	18.77±3.75	iter: 19, p_q: 0.3, α: 1e-5	6.66"/4.16"±1.93"	400.54"	13
	Sylvester equation	30.75±0.50	31.83±0.49	32.50±4.30	λ: 0.01, α: 3.16e-10	**0.41"**/0.66"±0.83"	24.71"	14
	Conjugate gradient	9.07±0.31	12.81±0.81	13.15±3.64	λ: 0.01, α: 3.16e-9	0.95"/0.92"±0.12"	31.89"	15
	Fixed-point iterations	11.30±0.72	13.06±0.97	14.20±5.93	λ: 1e-3, α: 3.16e-9	0.87"/0.77"±0.11"	23.28"	16
Acyclic	Spectral decomposition	30.97±0.48	31.90±0.60	33.05±4.34	λ: 0.1, α: 1e-9	0.96"/0.79"±0.11"	50.60"	17
	Shortest path	6.28±0.21	9.77±0.68	9.03±2.36	α: 1e-3	0.84"	**3.15"**	18
	Structural SP	3.78±0.13	12.62±1.12	13.10±4.78	α: 1e-3	1.73"	**4.43"**	19
	Path up to length h	1.89±0.14	6.83±0.43	**6.66±1.63**	h: 2, k_func: MinMax, α: 3.16e-3	0.50"/0.50"±0.04"	55.38"	20
	Treelet	3.38±0.16	6.16±0.39	**5.99±1.45**	kernel: poly, d: 1, c: 1e+3, α: 1e-3	0.51"/0.49"±0.02"	274.67"	21
	WL subtree	13.19±0.63	16.88±1.02	19.80±6.12	height: 1, α: 3.16e-10	**0.37"**/2.18"±1.32"	78.12"	22

"Parameters" indicates the hyper-parameters values selected by CV, with grid search values of α and C being [1e-10, 1e-9, ..., 1e10]. Ranges of all the parameters can be found in the demos of our Python library (introduced in Chapter 6).
"t_{gm}" is the time to compute Gram matrix/matrices in seconds. Note for kernels that need to tune hyper-parameters to compute Gram matrices, multiple Gram matrices are computed, and average time consumption and its confidence are obtained over the hyper-parameter grids, which are shown after the label "/". The time shown before "/" is the one spent on building the Gram matrix corresponding to the best test performance. Once hyper-parameters are fixed, learning is only performed on a single Gram matrix.
"t_{all}" exhibits the total time consumed to compute Gram matrix/matrices as well as to perform model selection for each kernel. For regression tasks (*Acyclic* and *Alkane* in this Table), the performances are given in terms of errors of boiling points.
The last column is the row number.

Table 3.5 – Results of all graph kernels for classification tasks (accuracy in percentage)

Datasets	Kernels	Train Perf	Valid Perf	Test Perf	Parameters	t_{gm}	t_{all}	
	Common walk	98.26±0.34	90.62±2.28	**93.00±8.16**	method: exp, β: 6, C: 3.16e+2	10.48"/6.47"±4.07"	1185.16"	23
	Marginalized	97.29±1.12	88.37±3.20	85.62±12.25	iter: 7, p_q: 0.5, C: 1e+7	4.24"/4.85"±2.26"	5609.37"	24
	Sylvester equation	90.72±1.40	87.09±2.67	84.52±13.23	λ: 0.1, C: 1e+7	**0.37"**/0.34"±0.03"	21.07"	25
	Conjugate gradient	98.15±0.47	86.41±3.71	88.57±10.93	λ: 0.1, C: 3.16e+6	0.86"/0.77"±0.04"	73.95"	26
	Fixed-point iterations	82.44±1.30	78.27±3.00	73.71±11.86	λ: 1e-3, C: 1e+10	1.05"/0.94"±0.15"	21.99"	27
MAO	Spectral decomposition	79.50±1.78	79.33±1.92	77.67±15.93	λ: 1e-7, C: 3.16e+9	**0.34"**/1.38"±1.06"	55.28"	28
	Shortest path	97.43±0.76	88.51±2.10	87.81±7.38	C: 3.16e+3	1.79"	**3.82"**	29
	Structural SP	96.70±0.76	90.79±2.44	91.62±9.16	C: 1e+3	7.63"	9.60"	30
	Path up to length h	98.20±1.00	91.11±2.59	85.43±12.60	h: 9, k_func: MinMax, C: 10	1.03"/0.72"±0.22"	52.21"	31
	Treelet	97.71±0.62	90.92±2.49	91.19±9.74	kernel: poly, d: 4, c: 1e+7, C: 1e+2	0.48"/0.52"±0.05"	1091.97"	32
	WL subtree	95.90±0.84	90.70±2.00	**93.05±8.66**	height: 6, C: 10	0.43"/0.56"±0.36"	29.82"	33
	Common walk	76.26±1.31	72.44±2.24	71.80±11.81	method: geo, γ: 0.11, C: 3.16e+4	11.59"/36.39"±23.57"	1574.13"	34
	Marginalized	63.37±2.20	63.52±2.18	57.67±18.51	iter: 4, p_q: 0.4, C: 1e-5	7.88"/11.27"±5.42"	827.50"	35
	Sylvester equation	74.47±1.30	71.88±2.51	71.50±12.36	λ: 0.1, C: 1e+4	**0.37"**/0.38"±0.05"	43.13"	36
	Conjugate gradient	75.62±2.08	71.69±2.49	73.93±13.89	λ: 0.1, C: 3.16e+4	1.57"/1.37"±0.12"	68.17"	37
	Fixed-point iterations	63.29±1.80	63.39±1.93	58.33±15.10	λ: 1e-4, C: 1e-8	2.46"/1.79"±0.44"	**30.33"**	38
PAH	Spectral decomposition	73.54±1.61	71.09±3.29	70.73±12.70	λ: 0.1, C: 3.16e+5	0.45"/2.33"±1.91"	78.05"	39
	Shortest path	79.53±1.26	76.66±2.55	69.40±11.57	C: 3.16e+2	2.30"	329.48"	40
	Structural SP	77.39±1.85	74.22±2.50	**74.50±13.39**	C: 3.16e+2	20.91"	776.99"	41
	Path up to length h	76.33±1.61	72.51±2.34	**75.27±13.72**	h: 1, k_func: MinMax, C: 10	0.53"/0.53"±0.04"	49.26"	42
	Treelet	82.89±1.64	70.66±3.23	66.30±12.68	kernel: gaussian, C: 1e+3	0.58"/0.58"±0.04"	8419.49"	43
	WL subtree	100.00±0.00	77.86±2.62	**75.93±10.83**	height: 14, C: 1e+2	1.86"/0.94"±0.66"	**37.39"**	44
	Common walk	91.88±0.98	88.09±1.31	85.96±7.92	method: geo, γ: 0.02, C: 1e+4	9.86"/19.02"±8.71"	2945.88"	45
	Marginalized	86.07±0.91	78.84±1.52	76.11±7.90	iter: 7, p_q: 0.8, C: 1e+6	19.72"/23.04"±11.57"	72207.27"	46
	Sylvester equation	84.89±1.24	83.58±1.90	82.77±7.23	λ: 0.1, C: 3.16e+3	0.51"/0.50"±0.03"	56.55"	47
	Conjugate gradient	92.19±0.76	87.14±1.60	86.18±5.83	λ: 1e-3 C: 3.16e+6	2.84"/2.73"±0.09"	74.39"	48
	Fixed-point iterations	92.31±0.73	87.34±1.51	86.58±6.66	λ: 1e-3, C: 1e+6	4.25"/3.35"±0.62"	45.36"	49
MUTAG	Spectral decomposition	83.71±0.90	83.41±1.14	84.05±7.85	λ: 1e-7, C: 3.16e+8	0.92"/5.94"±5.14"	159.06"	50
	Shortest path	98.23±0.40	84.39±2.35	81.84±6.63	C: 1e+3	4.89"	**7.58"**	51
	Structural SP	100.00±0.00	84.66±1.57	86.26±5.14	C: 3.16e+9	68.85"	71.18"	52
	Path up to length h	96.06±0.55	89.89±1.29	88.47±5.84	h: 2, k_func: MinMax, C: 1e+8	0.52"/0.86"±0.35"	51.17"	53
	Treelet	98.88±0.25	90.33±1.45	**90.79±4.62**	kernel: poly, d: 3, c: 1e+8, C: 3.16e+1	0.55"/0.57"±0.04"	152.88"	54
	WL subtree	92.72±0.72	87.24±1.36	87.18±5.69	height: 1, C: 3.16e+4	**0.33"**/1.56"±1.07"	41.27"	55

Letter-med	Common walk	39.40±0.34	36.53±0.72	36.16±2.94	method: geo, γ: 0.11, C: 3.16e+6	95.77"/102.20"±6.58"	377412.73"	56
	Marginalized	7.70±0.10	5.59±0.60	5.20±0.82	iter: 4, p_q: 0.8, C: 1e+10	75.03"/120.94"±60.63"	216051.89"	57
	Sylvester equation	39.14±0.31	36.26±0.65	37.27±1.93	λ: 0.1, C: 1e+6	13.76"/13.63"±0.50"	29832.59"	58
	Conjugate gradient	98.32±0.11	92.73±0.32	93.12±1.28	λ: 0.1, C: 1e+2	100.80"/92.35"±4.09"	3281.90"	59
	Fixed-point iterations	97.02±0.14	91.45±0.37	91.30±1.56	λ: 1e-4, C: 1e+5	78.45"/70.97"±7.07"	2481.78"	60
	Spectral decomposition	38.44±0.41	36.10±1.10	36.38±2.61	λ: 0.1, C: 3.16e+6	56.87"/60.19"±3.24"	27308.51"	61
	Shortest path	98.96±0.07	93.87±0.29	93.72±1.12	C: 10	36.98"	**255.48"**	62
	Structural SP	99.10±0.08	94.84±0.23	**94.88±1.24**	C: 10	41.92"	**257.96"**	63
	Path up to length h	49.62±0.29	45.73±0.71	43.83±2.31	h: 9, k_func: MinMax, C: 1e+7	**11.98"**/12.08"±0.20"	4707.17"	64
	WL subtree	40.82±0.34	36.90±0.91	36.13±2.70	height: 9, C: 1e+4	178.13"/104.55"±60.96"	4047.52"	65
Enzymes	Common walk	71.86±0.94	42.01±1.44	42.81±4.66	method: geo, γ: 0.03, C: 1e+5	907.43"/7960.34"±3246.14"	510920.78"	66
	Marginalized	68.52±0.78	45.72±1.51	45.92±4.79	iter: 19, p_q: 0.1, C: 1e+4	2426.77"/1513.51"±743.16"	96652.44"	67
	Sylvester equation	27.53±0.61	22.83±1.19	23.24±4.42	λ: 0.01, C: 3.16e+6	**5.19"**/5.20"±0.05"	1019.81"	68
	Conjugate gradient	100.00±0.00	61.97±1.33	60.89±5.62	λ: 1e-5, C: 1e+6	416.47"/418.57"±4.48"	4309.13"	69
	Fixed-point iterations	100.00±0.00	61.35±0.98	63.11±3.83	λ: 1e-4, C: 1e+5	741.70"/610.72"±102.94"	4978.36"	70
	Spectral decomposition	27.09±0.72	23.15±1.59	23.68±3.87	λ: 0.1, C: 1e+8	4939.35"/2493.84"±2486.74"	57494.36"	71
	Shortest path	100.00±0.00	68.86±1.91	**70.09±4.20**	C: 1e+6	704.54"	717.35"	72
	Path up to length h	100.00±0.00	57.53±1.53	57.49±5.19	h: 10, k_func: MinMax, C: 3.16e+2	911.77"/142.91"±279.98"	3123.30"	73
	Treelet	99.02±0.14	51.17±1.53	52.23±3.94	kernel: poly, d: 2, c: 1e+10, C: 3.16e+2	120.15"/121.08"±0.71"	16576.86"	74
	WL subtree	100.00±0.00	51.81±1.28	50.76±5.98	height: 4, C: 3.16e+2	19.88"/25.70"±17.76"	**433.76"**	75
AIDS	Common walk	95.68±0.21	94.62±0.28	94.71±1.64	method: geo, γ: 0.1, C: 1e+2	674.46"/673.64"±1.07"	11263.51"	76
	Sylvester equation	92.38±0.32	91.96±0.36	92.42±1.66	λ: 0.1, C: 3.16e+6	26.04"/23.69"±0.83"	2009.46"	77
	Conjugate gradient	99.30±0.06	98.78±1.05	98.93±0.58	λ: 0.01, C: 3.16e+2	836.96"/810.79"±12.78"	9196.29"	78
	Fixed-point iterations	99.29±0.06	98.73±0.12	98.57±0.61	λ: 1e-3, C: 3.16e+3	1226.56"/1073.06"±149.68"	9498.77"	79
	Spectral decomposition	87.98±0.31	87.79±0.39	87.21±2.62	λ: 0.1, C: 3.16e+6	154.16"/5556.74"±5542.56"	113636.62"	80
	Shortest path	99.91±0.02	99.13±0.11	99.26±0.55	C: 10	892.26"	**994.27"**	81
	Structural SP	99.80±0.03	98.90±0.10	98.84±0.63	C: 3.16	8021.98"	8125.28"	82
	Path up to length h	99.70±0.04	99.64±0.07	**99.65±0.40**	h: 1, k_func: MinMax, C: 3.16	**5.09"**/38.60"±28.42"	2826.33"	83
	Treelet	99.92±0.03	99.54±0.08	99.54±0.36	kernel: poly, d: 1 c: 1e+3, C: 3.16e+2	8.27"/7.46"±0.49"	5692.29"	84
	WL subtree	99.97±0.02	98.74±0.09	98.63±0.67	height: 10, C: 10	325.07"/164.66"±105.33"	2657.85"	85
NCI1	Sylvester equation	60.43±0.28	60.30±0.32	59.76±2.42	λ: 0.01, C: 3.16e+5	198.94"/199.35"±0.60"	8093.69"	86
	Conjugate gradient	72.97±0.26	70.96±0.40	71.34±1.85	λ: 0.01, C: 1e+6	10669.01"/9050.21"±907.11"	150599.44"	87
	Structural SP	92.75±0.13	80.13±3.86	79.88±1.71	C: 3.16e+2	132848.44"	135483.96"	88
	Path up to length h	97.86±0.09	84.22±0.37	**84.84±1.79**	h: 10, k_func: MinMax, C: 3.16	305.64"/108.88"±90.97"	16933.64"	89
	Treelet	64.96±0.38	64.76±0.39	64.84±2.16	kernel: gaussian, γ: 1e-3, C: 3.16e-2	**30.14"**/29.95"±0.27"	**7062.45"**	90
	WL subtree	99.59±0.04	85.12±0.38	**84.63±1.58**	height: 8, C: 10	1705.20"/1039.54"±722.29"	17484.36"	91

	Sylvester equation	60.30±0.39	60.13±0.45	60.62±1.95	λ: 0.01, C: 1e+5	201.48"/201.52"±0.11"	8179.15"	92
	Conjugate gradient	68.19±0.39	67.03±0.56	67.60±1.93	λ: 1e-3, C: 3.16e+6	35100.94"/23710.66"±8141.48"	195876.96"	93
	Fixed-point iterations	68.19±0.26	67.16±0.36	67.25±2.05	λ: 1e-3, C: 1e+9	35198.13"/23880.69"±8303.16"	197193.98"	94
NCI109	Structural SP	89.16±0.21	78.89±0.40	79.04±1.80	C: 10	134539.59"	141844.22"	95
	Path up to length h	97.97±0.08	83.77±0.26	**83.94±1.40**	h: 10, k_func: MinMax, C: 3.16	311.85"/111.22"±92.84"	17261.64"	96
	Treelet	64.37±0.23	64.31±0.23	63.46±2.06	kernel: gaussian, γ: 1e-3, C: 1e-2	**30.19"**/29.86"±0.25"	**7064.00"**	97
	WL subtree	99.41±0.05	85.14±0.30	**85.47±1.58**	height: 7, C: 10	1441.58/1018.67"±709.73"	17308.16"	98
DD	Path up to length h	100.00±0.00	80.92±0.58	**81.40±3.68**	h: 2, k_func: MinMax, C: 1e+2	**192.11"**/472.38"±638.44"	**1560.70"**	99
	WL subtree	100.00±0.00	79.36±0.52	77.30±3.76	height: 6, C: 1e+3	1062.89"/886.16"±83.81"	10143.36"	100

Same legends as Table 3.4. For the large-scale datasets, graph kernels are neglected if their time or memory consumption is much higher than other kernels.

Table 3.6 – Accuracy achieved by graph kernels, in terms of regression error (the upper table) and classification rate (the lower table) estimated on the test set. The red color indicates the worst results and dark green the best ones. Gray cells with the "inf" marker indicate that the computation of the graph kernel on the dataset is omitted due to much higher consumption of computational resources than other kernels.

	common walk	marginalized	Sylvester equation	conjugate gradient	fixed-point iterations	spectral decomposition	shortest path	structural SP	path up to length h	treelet	WL subtree
Alkane	15.52	43.75	8.97	11.13	12.78	12.95	7.81	8.65	9.0	2.53	26.42
Acyclic	12.93	18.77	32.5	13.15	14.2	33.05	9.03	13.1	6.66	5.99	19.8

MAO	93.0	85.62	84.52	88.57	73.71	77.67	87.81	91.62	85.43	91.19	93.05
PAH	71.8	57.67	71.5	73.93	58.33	70.73	69.4	74.5	75.27	66.3	75.93
Mutag	85.96	76.11	82.77	86.18	86.58	84.05	81.84	86.26	88.47	90.79	87.18
Letter-med	36.16	5.2	37.27	93.12	91.3	36.38	93.72	94.88	43.83	inf	36.13
Enzymes	42.81	45.92	23.24	60.89	63.11	23.68	70.09	inf	57.49	52.23	50.76
AIDS	94.71	inf	92.42	98.93	98.57	87.21	99.26	98.84	99.65	99.54	98.63
NCI1	inf	inf	59.76	71.34	inf	inf	inf	79.88	84.84	64.84	84.63
NCI109	inf	inf	60.62	67.6	67.25	inf	inf	79.04	83.94	63.46	85.47
D&D	inf	inf	inf	inf	inf	inf	inf	inf	81.4	inf	77.3

The overall performance

Table 3.6 provides an outline of the accuracies achieved by the aforementioned graph kernels as given in terms of test performance in Tables 3.4 and 3.5. Each row corresponds to a dataset and each column to a graph kernel. All kernels achieve better results compared with random assignment. It can be seen that, generally speaking, graph kernels based on paths have better accuracy than those based on walks for both regression and classification tasks, proving that the application of walk patterns are constrained by their common shortcomings, such as tottering and halting. Moreover, due to their mathematical structures, the common walk kernel and the marginalized kernel are two of the slowest to compute. For relatively large datasets, such as *Enzymes*, the average time to compute Gram matrices for these two kernels are more than 60 times slower than the fastest kernels. For even larger datasets,

such as *NCI1*, *NCI109*, and *DD*, these two kernels are ignored in our experiments due to their expensive time complexity. As a result, these two kernels are not recommended to be applied in real tasks, but can still be considered as baselines to test the performance of newly constructed kernels.

Among other kernels based on walks, the Sylvester equation kernel and the spectral decomposition kernel cannot tackle any labeling information. On unlabeled graphs, such as *Alkane* and *PAH*, accuracies that they provide are noteworthy; however, under other circumstances, the conjugate gradient kernel and the fixed-point kernel may offer better accuracies. The latter two kernels are able to tackle symbolic and non-symbolic labels on both vertices and edges; therefore, they are among the best kernels based on walks with respect to accuracy on all considered datasets. Their accuracies are sometimes competitive with those of kernels based on paths, such as on *MUTAG* and *Letter-med* datasets.

Among kernels based on paths, the shortest path kernel and the structural shortest path kernel provide the ability to tackle symbolic and non-symbolic labels. The latter takes better structure information into consideration than the former one; thus, it yields higher accuracy on most datasets while requiring much more computational resources. The path kernel up to length h is capable of tackling symbolic labels only, where it offers the best accuracy in most cases. More importantly, due to its relatively low computational complexity, it is possible to apply this kernel to large datasets, such as *DD* whose average number of vertices is 284.32.

Additionally, the performance of two well-known graph kernels based on non-linear patterns, namely the treelet kernel and the WL subtree kernel are exhibited. Several graph kernels based on linear patterns, especially paths, provide competitive or even higher accuracies than these two kernels. On the *MAO* dataset, the common walk kernel achieves 93% accuracy (Table 3.5, Line 1), which is comparable to the accuracy of the WL subtree kernel (93.05%, Table 3.5, Line 11) and is higher than that of the treelet kernel (91.19%, Table 3.5, Line 10). The shortest path kernel achieves the highest accuracy on dataset *Enzymes* (70.09%, Table 3.5, Line 49), which is about 20% higher than the treelet kernel and the WL subtree kernel (Table 3.5, Lines 51 and 52). The structural shortest path kernel has the equivalent accuracy as the WL subtree kernel on *PAH* (Table 3.5, Lines 19 and 22). The path kernel up to length h achieves equivalent or higher accuracy with runtime comparable to or lower than kernels based on non-linear patterns, on the datasets *Acyclic*, *PAH*, *MUTAG*, *Enzymes*, as well as larger datasets such as *AIDS*, *NCI1*, *NCI109* and *DD*. On

Table 3.7 – Time used to compute Gram matrices of graph kernels (in $\log_{10}$ of seconds). Same color legends as Table 3.6 are used.

	common walk	marginalized	Sylvester equation	conjugate gradient	fixed-point iterations	spectral decomposition	shortest path	structural SP	path up to length h	treelet	WL subtree
Alkane	0.48	0.51	-0.42	-0.18	-0.22	-0.19	-0.12	0.02	-0.29	-0.3	0.16
Acyclic	0.36	0.62	-0.18	-0.04	-0.11	-0.1	-0.08	0.24	-0.3	-0.31	0.34
MAO	0.81	0.69	-0.47	-0.11	-0.03	0.14	0.25	0.88	-0.14	-0.28	-0.25
PAH	1.56	1.05	-0.42	0.14	0.25	0.37	0.36	1.32	-0.28	-0.24	-0.03
Mutag	1.28	1.36	-0.3	0.44	0.53	0.77	0.69	1.84	-0.07	-0.24	0.19
Letter-med	2.01	2.08	1.13	1.97	1.85	1.78	1.57	1.62	1.08	inf	2.02
Enzymes	3.9	3.18	0.72	2.62	2.79	3.4	2.85	inf	2.16	2.08	1.41
AIDS	2.83	inf	1.37	2.91	3.03	3.74	2.95	3.9	1.59	0.87	2.22
NCI1	inf	inf	2.3	3.96	inf	inf	inf	5.12	2.04	1.48	3.02
NCI109	inf	inf	2.3	4.37	4.38	inf	inf	5.13	2.05	1.48	1.48
D&D	inf	inf	inf	inf	inf	inf	inf	inf	2.67	inf	2.95

the *AIDS* dataset, all exhibited kernels have comparable accuracies (Table 3.5, Lines 53 to 57).

The treelet kernel and the WL subtree kernel are not able to tackle non-symbolic labels. More recent work is able to tackle this problem (a collection of these improvements are introduced in Section 3.5.2), which may affect the performance of these kernels on datasets such as *Letter-med*, *Enzymes*, and *AIDS*. However, on other datasets that do not contain non-symbolic labels, the performance remains the same, and the aforementioned analyses still stand.

Besides accuracy, we furthermore examined the computational complexity of each kernel. Table 3.7 displays the time consumed to compute the Gram matrix of each kernel on each dataset. The results are consistent with the time complexities for graph kernels. In most cases, the computation is efficient and it takes seconds or minutes to compute the whole Gram matrix. On the largest dataset (i.e., *DD*, which contains 1178 graphs with 284 vertices and 715 edges per graph on average), two graph kernels can still be computed in tolerable time. For example, the path kernel up to length h can be computed within 8 minutes on *DD*, which benefits from not

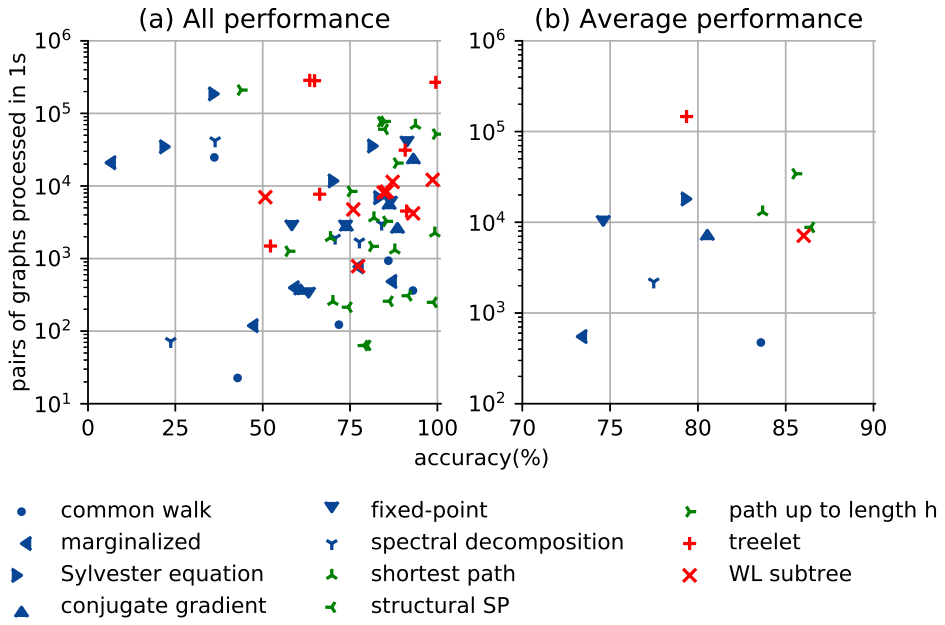

Figure 3.19 – Comparison of computational complexity versus accuracy of all graph kernels on all datasets (a), as well as the average performance of each kernel over all datasets (b). Markers correspond to different kernels; Colors blue, green, and red depict graph kernels based on walks, paths, and non-linear patterns, respectively.

only its relatively lower time complexity, but also the trie structure applied to it (see Section 3.6.3). Notice that it takes too much time to compute some graph kernels on some large datasets. For instance, the time complexity for the common walk kernel is in $\mathcal{O}(n^6)$ per pair of graphs. To this end, it is irrational to apply this kernel on graphs with large amounts of vertices.

The joint performance of the computational complexity and accuracy of each graph kernel on each classification dataset is shown in Figure 3.19. Performances that cannot be acquired in reasonable time are omitted. In Figure 3.19(b), the performance of each kernel is averaged on at least 5 datasets. Note that for datasets *Letter-med* and *Enzymes*, kernels that cannot tackle non-symbolic labels are omitted. We provide general conclusions on these graph kernels. From a global viewpoint, all kernels provide good accuracy on all datasets. We can see that the marginalized kernel has the worst accuracy, with some regular computational time. The structural shortest path kernel provides the best accuracy in general, the price to pay being its computational complexity. The Sylvester equation kernel, the spectral decomposition kernel, and the path kernel up to length h have a good compro-

mise between time complexity and accuracy. Kernels based on non-linear patterns are among the best trade-offs; meanwhile, the Sylvester kernel, the conjugate gradient kernel, the shortest path kernel, and the path kernel up to length h achieve competitive or better trade-offs. Along with the preceding analyses, these facts prove that these kernels can be reliable for classification and regression problems on graphs, as well as qualified benchmark kernels for future graph kernels design. Figure 3.19 provides a helpful guidance to find the best trade-offs for each kernel. Based on this analysis, before choosing a graph kernel, one can have a rough expectation of its performances.

We then further analyze these kernels based on different types and characteristics of datasets.

Labeled and unlabeled graphs

To study the influence of labeling on the performance of graph kernels, we examine 3 datasets that have similar properties (*e.g.* $\bar{n}$, $\bar{m}$ and d in Table 2.2), except for labeling: *PAH* is unlabeled, *MAO* has 3 symbolic vertex labels and 4 symbolic edge labels, and *MUTAG* has 7 symbolic vertex labels and 11 symbolic edge labels.

Figure 3.20 exhibits the accuracy of each kernel and the average time to compute each kernel between a pair of graphs. We can see that for almost all kernels, the classification accuracies on dataset *PAH* are significantly lower and the confidence intervals around them are wider than the other two datasets, as *PAH* contains no labeling information. On each dataset, accuracies of kernels based on walks, paths, and non-linear patterns are competitive. Meanwhile, the second figure exhibits the influence of graph structures on time complexity. Take the common walk kernel for instance, whose time complexity is in $\mathcal{O}(n^6)$, the runtime is the shortest on *MUTAG* and the longest on *PAH* due to the different average number of graph vertices of each graph. The runtime for each dataset is also consistent with the time complexity for each kernel in Table 3.1. The Sylvester equation kernel and the path kernel up to length h have competitive speed with kernels based on non-linear patterns with equivalent accuracies.

Graphs with symbolic and non-symbolic labels

Non-symbolic labels are able to introduce continuous attributes to graphs. Among all graph kernels, the shortest path kernel is able to tackle symbolic and non-

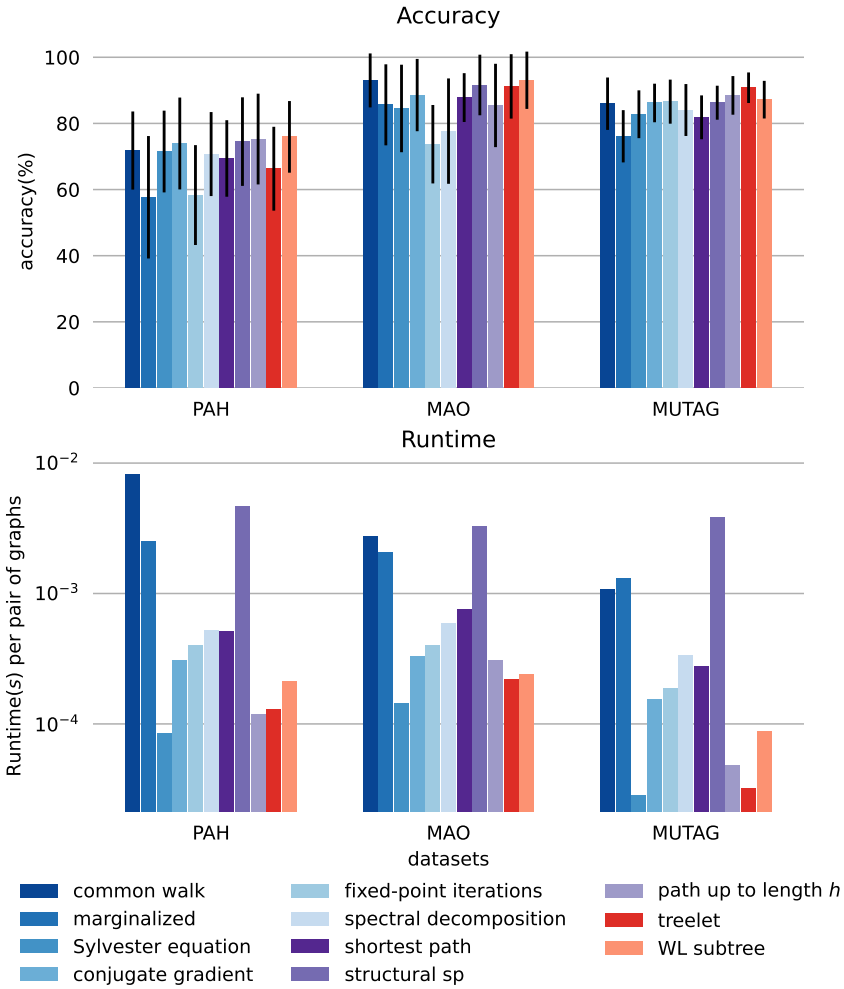

Figure 3.20 – Comparison of accuracy and runtime of all kernels on unlabeled (*PAH*) and labeled datasets (*MAO, MUTAG*). Accuracy is the mean value on 30 trials (pillars), with confidence intervals around it (error bars).

symbolic vertex labels, whereas the conjugate gradient kernel, the fixed-point kernel, and the structural shortest path kernel can deal with both non-symbolic labels of vertices and edges. In available datasets, *Letter-med* and *Enzymes* contain non-symbolic vertex labels. We compute accuracy and time complexity of each aforementioned kernel on these 2 datasets, then we remove the non-symbolic labels from the datasets and compute the performance again.

Figure 3.21 shows that, with non-symbolic labels, the classification accuracy of all kernels exceeds 90% on dataset *Letter-med*, and more than 60% on dataset *Enzymes*; these accuracies drop to about 35% when non-symbolic labels are removed, which are still better than random assignments because of the large numbers of

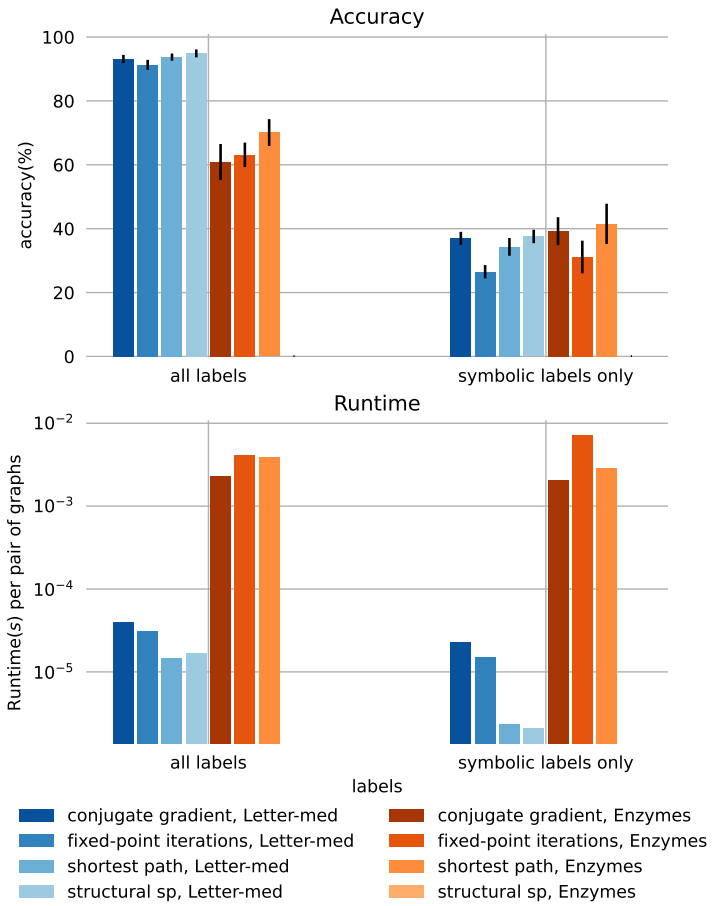

Figure 3.21 – Comparison of accuracy and runtime of graph kernels on datasets with and without non-symbolic labels. The last pillar was removed due to its high computational time.

competing classes (15 and 6, respectively). It reveals how these graph kernels can take advantage of non-symbolic labels, which carry out essential information of dataset structures. This consequence is corroborated by the results revealed in Table 3.5 and Table 3.6 where graph kernels that cannot tackle non-symbolic labels work poorly on *Letter-med* and *Enzymes*, such as the common walk kernel and the marginalized kernel.

As a result, non-symbolic labels should always be well examined before designing graph kernels. When only linear patterns are included, the shortest path kernel, the conjugate gradient kernel, the fixed-point kernel, and the structural shortest path kernel would be the first to consider.

We then split datasets of classification tasks in Table 2.2 into 2 groups: graphs containing non-symbolic labels (including *Letter-med*, *Enzymes*, *AIDS*) and those

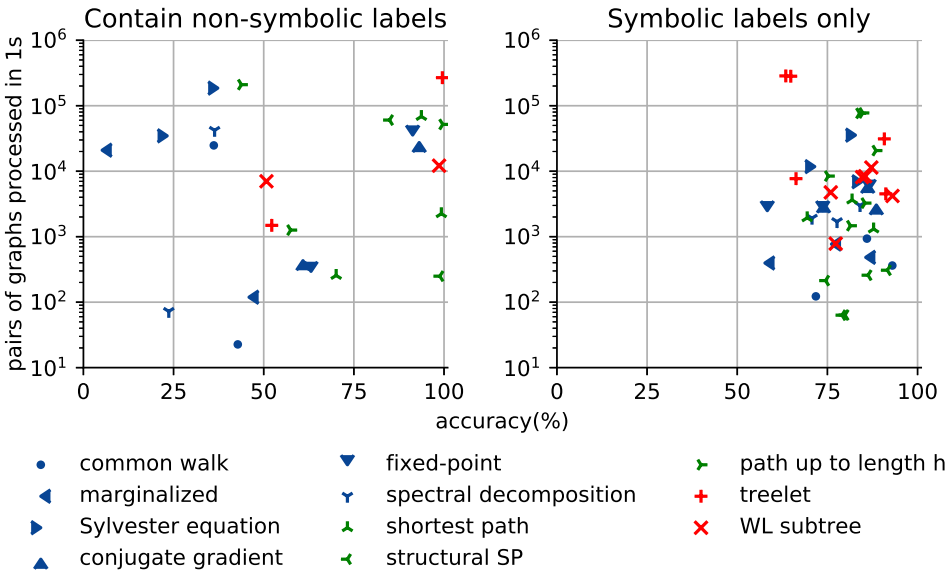

Figure 3.22 – Comparison of computational complexity versus accuracies of all graph kernels on graphs with and without non-symbolic labels. Markers correspond to different kernels; Colors blue, green, and red depict graph kernels based on walks, paths, and non-linear patterns, respectively.

with only symbolic labels (including all the rests). Figure 3.22 exhibits the time complexity and classification accuracies of all kernels on each type of datasets. On datasets that contain non-symbolic labels, the conjugate gradient kernel, the fixed-point kernel, the shortest path kernel, and the structural shortest path kernel yield higher accuracy, except on *AIDS*, where all kernels achieve high accuracy. On datasets without non-symbolic labels, the performance of each kernel varies with respect to datasets. No kernel dominates all others on all datasets. Other properties of datasets should be taken into consideration under this circumstance.

Graphs with different average vertex numbers

The average vertex number of a graph dataset largely influences the time complexity of computing graph kernels. To examine it, we choose 3 datasets with a relatively wide range of vertex numbers, namely *PAH*, *MUTAG* and *Enzymes*, corresponding to unlabeled, symbolic labeled, and non-symbolic labeled graphs, respectively. For each dataset, we order the graphs according to the vertex number, and then split them into 5 subsets with different average vertex numbers.

Figures 3.23(a)(b)(c) show the evolution in the runtime to compute Gram matrices with the growth of average vertex numbers. The runtimes of the common walk

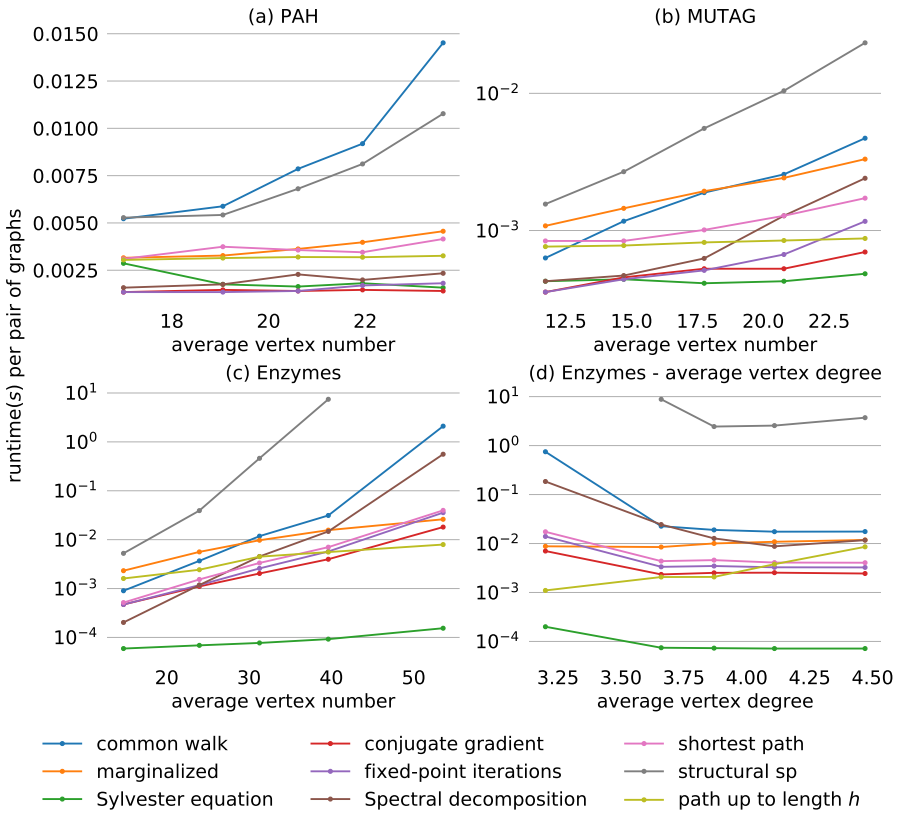

Figure 3.23 – Comparison of the runtime of each kernel on datasets with different average vertex numbers and average vertex degrees.

kernel and the structural shortest path kernel grow the fastest, the runtimes of the Sylvester equation kernel and the path kernel up to length h remain relatively stable, while the increase rates of runtimes for other kernels are in the middle. This result is consistent with the time complexity of computing the Gram matrix of each kernel, where average vertex numbers to different powers are involved (see Table 3.1). However, the time complexity is affected by other factors, such as average vertex degrees, which causes fluctuations and decreases to the runtime as the average vertex numbers grow. This phenomenon is more observable for small datasets with a narrower range of vertex numbers, such as *PAH* shown in Figure 3.23(a).

According to the average graph vertex numbers $\bar{n}$, datasets that concern classification tasks in Table 2.2 can be classified into small graphs (including *Letter-med*), big graphs (including *DD*), and medium graphs (including all the rests). Figures 3.24(a)(b)(c) exhibit the computational time and the classification accuracy of each kernel on these datasets. For small datasets (*Letter-med*), the kernels based on

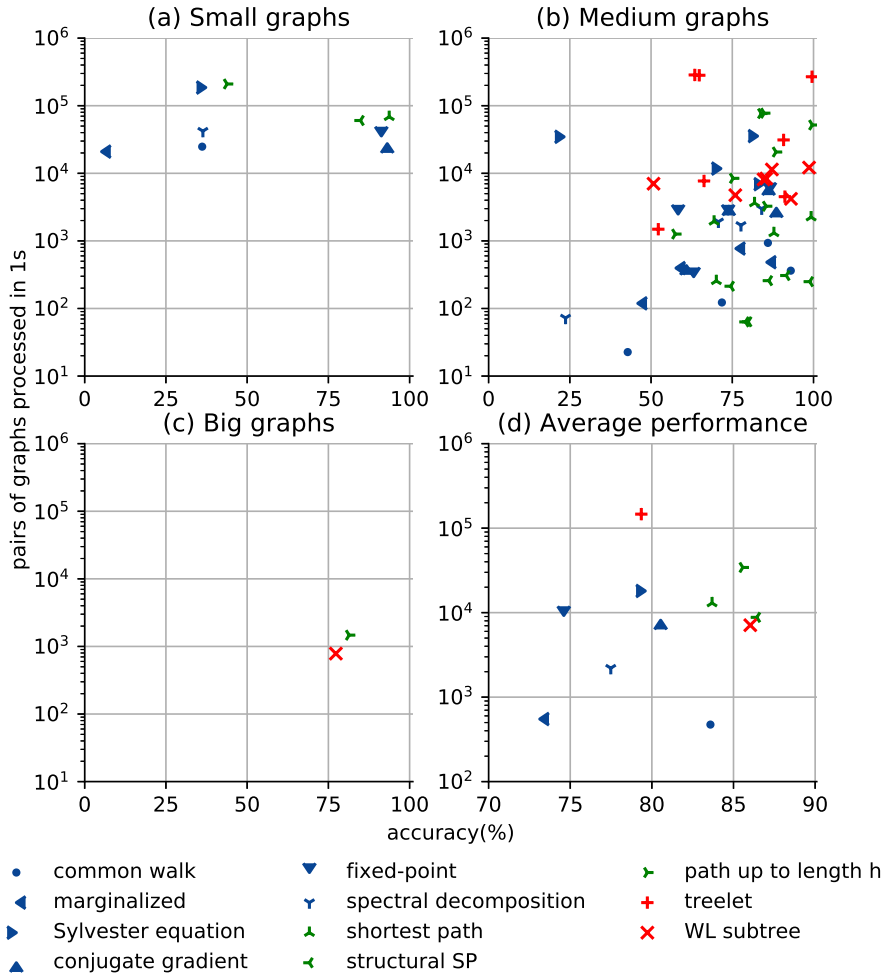

Figure 3.24 – Comparison of computational complexity versus accuracies of all graph kernels on small (a), medium (b), and big (c) datasets. Markers correspond to different kernels; Colors blue, green, and red depict graph kernels based on walks, paths, and non-linear patterns, respectively.

shortest paths, the conjugate gradient kernel, and the fixed-point kernel achieve the best compromise between computational complexity and accuracy. Kernels based on non-linear patterns are omitted as they are not suitable for the *Letter-med* dataset (Figure 3.24(a)). As sizes of graphs grow, kernels based on walks, paths, and non-linear patterns may all strike good trade-offs between computational complexity and accuracy for certain datasets (Figure 3.24(b)). Accuracies of the latter two groups of kernels are higher in general. On the big dataset *DD*, the path kernel up to length h performs better than the WL subtree kernel (Figure 3.24(c)).

Graphs with different average vertex degrees

As vertex number, vertex degree plays an important role in time complexity of computing graph kernels. Large vertex degrees indicate "dense" graphs where more edges and connections exist, leading to a much larger number of linear patterns inside graphs, such as walks and paths, and more time to explore them. Applying the same method as in Section 3.7.2, we choose a dataset with a relatively wide range of vertex degrees, namely *Enzymes*, order it based on the vertex degree, and split it into 5 subsets. Figure 3.23(d) reveals the relationship between the runtime to compute the graph kernel and the average vertex degree of each subset.

One can find from Table 3.1 that the time complexity of only two kernels based on linear patterns is directly affected by the average vertex degrees m: the structural shortest path kernel and the path kernel up to length h. As a result, Figure 3.23(d) shows that the runtime of the path kernel up to length h increases as the average vertex degree grows. Runtimes of most of the other kernels, however, are high for the first subset (i.e., with the smallest average vertex degree of $m = 3.2$), and stays stable afterwards. Other than the average vertex degree of each subset, this runtime is mainly influenced by the average vertex number, which is much bigger for the first subset than the others.

These experiments once again reveal the fact that, in practice, graph kernels based on linear patterns can achieve high performance on graphs containing linear and non-linear sub-structures compared to graph kernels based on non-linear patterns, even though these kernels were not specifically designed for the latter structures. In conclusion, these linear pattern kernels are worth investigating for any dataset. Structures and properties of datasets should be carefully inspected for choosing the proper graph kernels.

3.8 Conclusion

In this chapter, an extensive analysis of graph kernels based on linear patterns was performed. Although graph kernels based on linear patterns have been designed for linear structures, they were applied with success on datasets containing non-linear structures. We examined the influence of several factors, such as labeling, average vertex numbers, and average vertex degrees on the performance of graph kernels. We found the following observations. All kernels work on graphs with symbolic ver-

tex labels, while some kernels, such as the marginalized kernel, tend to fail to capture structure information of graphs when labels are absent. On the other hand, the existence of symbolic labels matters as well. The conjugate gradient kernel, the fixed-point kernel, the shortest path kernel, and the structural shortest path kernel, possess the ability to tackle non-symbolic labels, while the other kernels hold no such ability and thus are strongly discouraged when considering graphs without symbolic labels. A special notice is that the shortest path kernel can only tackle vertex labels.

The computational complexity – a major issue in designing and working with graph kernels – was extensively addressed in this chapter. We proposed three strategies to accelerate the computation of graph kernels. The average vertex numbers and average vertex degrees restrict kernels' scalability. The time complexity of all kernels is polynomial to the average vertex numbers, with the common walk kernel being the worst one, and thus it should be avoided for large-scale datasets. Average vertex degrees had a trivial influence on the time complexity, which remained low on all datasets.

Finally, we can conclude that, in general, many graph kernels based on linear patterns achieved competitive accuracy compared to the ones based on non-linear patterns with tolerable time complexity, where those based on paths were generally better than the ones based on walks. Work conducted in this chapter builds the foundation for the graph pre-image problem.

Chapter 4

Stability and metric learning of graph edit distances

Contents

4.1 Overview

Graphs provide a flexible representation framework to encode relationships between elements. However, graph spaces cannot be endowed with the mathematical tools and properties associated with Euclidean spaces. This issue prevents the use of classical machine learning methods mainly designed to operate on vector representations. To learn models on graphs, several approaches have been designed to leverage this flaw and among these, we can cite graph embedding strategy [Goyal and Ferrara, 2018, Cai et al., 2018, Gibert et al., 2012], graph kernels [Borgwardt et al., 2020, Kriege et al., 2020, Ghosh et al., 2018, Kriege et al., 2017, Gaüzère et al., 2015a, Gärtner, 2003b] and more recently graph neural networks [Wu et al., 2020, Zhang et al., 2020, Zhou et al., 2018a, Bronstein et al., 2017, Balcilar et al., 2021]. Despite their state-of-the-art performances, they seldom operate directly in a graph space, hence reducing the interpretability of the underlying operations.

To overcome these issues, one needs to preserve the properties of a graph space. For this purpose, one needs to define a dissimilarity measure or a metric in the graph space, in order to constitute the minimal requirement to implement simple machine learning algorithms like the k-nearest neighbors (see Section 1.2.2 for details). One of the most used dissimilarity measures between graphs is the graph edit distance (GED) [Bunke and Allermann, 1983, Sanfeliu and Fu, 1983]. As described in Section 2.3, the GED of two graphs G_1 and G_2 is the minimal amount of distortion required to transform G_1 into G_2. This distortion is encoded by a set of edit operations whose sequence constitutes an edit path. These edit operations include vertex and edge substitutions (v_s and e_s), removals (v_r and e_r), and insertions (v_i and e_i), as shown in Figure 4.1. Depending on the context, each edit operation e included in an edit path γ is associated with a non-negative cost $c(e)$. The sum of all edit operation costs included within the edit path defines the cost associated with this edit path. The minimal cost among all edit paths defines the GED between G_1 and G_2, namely

$$\texttt{ged}(G_1, G_2) = \min_{\gamma \in \Gamma(G_1, G_2)} C(\gamma). \tag{4.1}$$

Therefore, the GED is null when comparing two isomorphic graphs. This situation relates to an exact graph matching.

Evaluating GED is computationally costly. Even for uniform edit costs, this computation problem is *NP*-hard [Zeng et al., 2009]. Thus, it cannot

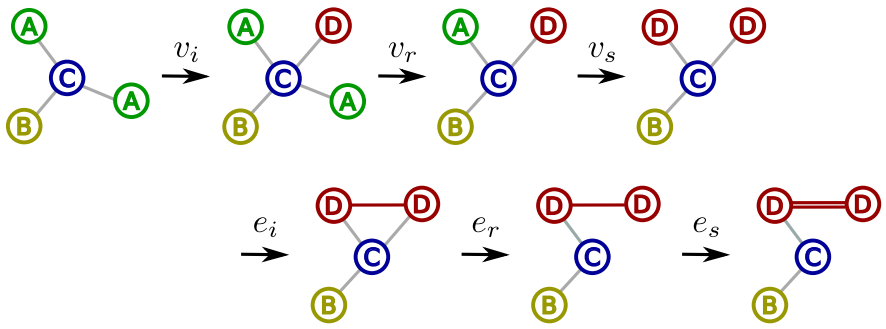

Figure 4.1 – An illustration of graph edit operations.

be done in practice for graphs having more than 12 vertices in the general case [Neuhaus et al., 2006]. To avoid this computational burden, strategies to approximate GED in a limited computational time have been proposed [Abu-Aisheh et al., 2017, Blumenthal et al., 2020] with acceptable classification or regression performances. Of particular interest are the two famous methods, `bipartite` [Riesen, 2015] and `IPFP` [Bougleux et al., 2016], where upper and lower bounds are estimated as an approximation of GED. The computation of the bounds relies highly on the design of the algorithm, as well as the randomness during the procedure, which leads to a reduction of stability. In this thesis, we define the instability of a GED heuristic in terms of the variability of the GED approximations over repeated trials; see Section 4.3 for details. Methods that can potentially alleviate this problem are proposed. For instance, by carrying out several local searches in parallel, the multi-start counterparts of `bipartite` and `IPFP`, named m`bipartite` and m`IPFP` respectively, may acquire better approximation with higher stability [Daller et al., 2018]. Description and analyses of these approximations and the root of randomness are presented in Section 4.3.

An essential ingredient of GED is the underlying edit cost function $c(e)$, which quantifies the distortion carried by any edit operation e. The values of the edit costs for each edit operation have a major impact on the computation of GED and its performance, including its stability. Thus, the cost edit function may be different depending on the data encoded by the graph and the one as the target of the task. Generally, they are fixed a priori by an expert of the domain, and are provided with the datasets.

However, these predefined costs are not optimal for all tasks, in the same spirit as the no free lunch theorem for machine learning and statistical inference [Wolpert and Macready, 1997]. Moreover, these costs may greatly influence both the

task performance and the computational time required to compute the graph edit distance. In [Bunke, 1999], the authors show that a particular set of edit costs may reduce the problem of computing graph edit distance to well-known problems in graphs like (sub)graph isomorphism or finding the maximum common subgraph of a pair of graphs. In [Abu-Aisheh et al., 2017], the authors evaluate the influence of different cost settings for different methods. These points show again the importance of the underlying cost function when computing a graph edit distance.

It is interesting to challenge the costs given by experts. By adapting costs according to some given targets, the GED between graphs may be changed. The optimization of this metric furthermore relates to a graph space tuning based on the information of the target space. For a prediction task, this target space may be founded on a metric measure between target values; while in terms of pre-image problem, the target space is constructed by graph kernels. Thus, a metric learning framework can be applied to connecting the GED space and either of the latter two spaces.

In this chapter, we propose a simple strategy to optimize edit costs according to a particular prediction task, and thus avoid the use of predefined costs. The idea is to align the metric in the graph space (namely, the GED) to the prediction space. While the concept of aligning the metric to the target space has been largely used in machine learning (e.g. with the so-called kernel-target alignment [Cristianini et al., 2002]), this is the first time that such a line of attack is investigated to estimate the optimal edit costs. With a distance-preserving principle, we provide a simple linear optimization procedure to optimize a set of constant edit costs. The edit costs resulting from the optimization procedure can then be analyzed to understand how the graph space is structured. The relevance of the proposed method is demonstrated on two regression tasks, showing that the optimized costs lead to a prediction error lower than the random and expert costs.

The remaining part of the chapter is organized as follows. Section 4.2 introduces widely used GED heuristic paradigms. Section 4.3 examines the stability of these heuristics and ensures the necessary parameter choices to compute them. Then, Section 4.4 presents a metric learning approach to graph edit costs for regression, providing the state of the art, the problem formulation, the proposed optimization method, and the results from conducted experiments. Finally, we conclude and open perspectives on this work, as well as its prospective usage in the graph pre-image problem. For the sake of conciseness, we restrict to undirected graphs in this chapter unless otherwise specified.

4.2 Graph edit distances heuristics

Over the years, many heuristics have been proposed to approximate the GED. The authors of [Blumenthal et al., 2020] categorize these heuristics according to their underlying paradigms. The first paradigm is based on transforming the computation to a linear sum assignment problem with error correction (`LSAPE-GED`) derived from [Bunke, 1999], where the famous `bipartite` method lies. The second paradigm is based on local search (`LS-GED`), which includes the instantiation `IPFP`. The third one is based on linear programming (`LP-GED`) and other heuristics are categorized as miscellaneous. As milestones and baselines to many other methods, both `bipartite` and `IPFP` achieve high performance [Abu-Aisheh et al., 2017]. Thereby, in the following sections, we focus on these two heuristics and the related paradigms, provide a novel formalism to introduce our method, and conduct analyses and experiments to evaluate its relevance.

4.2.1 The `LSAPE-GED` paradigm

GED can be approximated by solving a linear sum assignment problem with edition or error correction (LSAPE). For any two sets V_1 and V_2, consider a transformation from V_1 to V_2, with elementary operations on each element $i \in V_1$: substitution ($i \to j$), insertion ($\varepsilon \to j$), and removal ($i \to \varepsilon$), where $j \in V_2$ and ε represents a dummy element. An assignment with edition, also known as the ε-assignment [Bougleux and Brun, 2016], is a bijection between set $V_1^\varepsilon = V_1 \cup \{\varepsilon\}$ and set $V_2^\varepsilon = V_2 \cup \{\varepsilon\}$ relaxed on ε, namely $\pi : V_1^\varepsilon \to V_2^\varepsilon$ where $|\pi(i)| = 1$ for any $i \in V_1$, $|\pi^{-1}(j)| = 1$ for any $j \in V_2$, and $\pi(\varepsilon) = \varepsilon$. We denote the set of all possible ε-assignments from V_1^ε to V_2^ε as $\Pi(V_1, V_2)$.

Each elementary operation in an ε-assignment π can be associated with a non-negative cost c. Consequently, a cost C is associated with π, namely

$$C(\pi) = \sum_{\substack{i \in V_1 \\ \pi(i) = j}} c(i, j) + \sum_{\substack{j \in V_2 \\ \pi^{-1}(j) = \varepsilon}} c(\varepsilon, j) + \sum_{\substack{i \in V_1 \\ \pi(i) = \varepsilon}} c(i, \varepsilon), \tag{4.2}$$

where each term on the right side successively represents substitutions, insertions, and removals. The costs of all operations induced by π can be represented by a matrix $\mathbf{C} \in \mathbb{R}^{(|V_1|+1)\times(|V_2|+1)}$. LSAPE aims at minimizing this cost over all $\pi \in \Pi$, namely

finding

$$C^\star(\pi^\star) = \min_{\pi \in \Pi(V_1, V_2)} C(\pi). \tag{4.3}$$

We denote the set of all optimal solutions as $\Pi^\star(V_1, V_2)$. Variants of the Hungarian algorithm have been used to acquire an optimal solution of (4.3) [Kuhn, 1955, Munkres, 1957], with a time complexity of $\mathcal{O}(\min\{|V_1|, |V_2|\}^2 \max\{|V_1|, |V_2|\})$ and space complexity of $\mathcal{O}(|V_1||V_2|)$ [Bougleux et al., 2020].

The GED between two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ can be approximated by solving an LSAPE between vertex sets V_1 and V_2. Each row and column in the cost matrix $\mathbf{C}$ respectively correspond to a vertex in V_1 and V_2, each entry $\pi^\star(G_1, G_2) = \pi^\star(V_1, V_2) \in \Pi^\star(G_1, G_2)$ represents a optimal feasible transformation from G_1 to G_2 as in (4.16), and the optimal cost $C^\star$ in (4.3) corresponds to the approximation of GED. This paradigm, named `LSAPE-GED`, is presented in Algorithm 4.1 [Blumenthal et al., 2020].

Algorithm 4.1 Approximation of GED using `LSAPE-GED`

Input: Graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$.
Vertex edit cost c_v, edge edit cost c_e.
Output: An approximation of GED between G_1 and G_2.

1: Construct the cost matrix $\mathbf{C} \in \mathbb{R}^{(|V_1|+1)\times(|V_2|+1)}$ and the corresponding LSAPE using G_1, G_2, c_v, and c_e.
2: Compute a solution $\pi \in \Pi(|V_1|, |V_2|)$ to the LSAPE problem using a solver (e.g., using a variant of Hungarian algorithm).
3: Set the upper bound as $C(\pi)$, namely the required approximation.

The `bipartite` heuristic

The `bipartite` is a representative heuristic under the `LSAPE-GED` paradigm. The name comes from its relevance to the weighted bipartite graph matching problem [Bougleux and Brun, 2016, Bougleux et al., 2017]. It constructs the cost matrix $\mathbf{C}$ by adding the cost of vertices and the cost of the edges adjacent to them. For each vertex pair $(v_i, u_j) \in V_1 \times V_2$, construct an auxiliary edge cost matrix $\mathbf{C}_e^{i,j} \in \mathbb{R}^{(\deg(v_i)+1)\times(\deg(u_j)+1)}$ and the corresponding LSAPE, where $\deg(v)$ is the degree of vertex v. The k-th ($1 \leqslant k \leqslant \deg(v_i)$) row and l-th column ($1 \leqslant l \leqslant \deg(u_j)$) in $\mathbf{C}_e$ correspond respectively to the k-th edge e_k^i in the set of edges adjacent to v_i and to the

l-th edge e_l^j in the set of edges adjacent to u_j. The entries in $\mathbf{C}_e^{i,j}$ are then given by

$$\begin{cases} c_{k,l}^{i,j} = c_{efs}(e_k^i, e_l^j) \\ c_{k,\deg(u_j)+1}^{i,j} = c_{efr}(e_k^i, \varepsilon) \\ c_{\deg(v_i)+1,l}^{i,j} = c_{efi}(\varepsilon, e_l^j), \end{cases} \tag{4.4}$$

for $k = 1,2,\ldots,\deg(v_i)$ and $l = 1,2,\ldots,\deg(u_j)$, where c_{efs}, c_{efr}, c_{efi} are respectively the cost functions of edge substitution, removal, and insertion, as defined in (2.10). After that, an optimal LSAPE solution $\pi^{i,j}$ for $\mathbf{C}_e^{i,j}$ and the corresponding cost $\mathrm{C}_e^{i,j}(\pi^{i,j})$ are computed. Then, the entries in $\mathbf{C}$ are constructed as

$$\begin{cases} c_{i,j} = c_{vfs}(v_i, v_j) + \mathrm{C}_e^{i,j}(\pi^{i,j}) \\ c_{i,|\mathrm{V}_2|+1} = c_{vfr}(v_i, \varepsilon) + \sum_{k=1}^{\deg(v_i)} c_{efr}(e_k^i, \varepsilon) \\ c_{|\mathrm{V}_1|+1,j} = c_{vfi}(\varepsilon, u_j) + \sum_{l=1}^{\deg(u_j)} c_{efi}(\varepsilon, e_l^j), \end{cases} \tag{4.5}$$

where c_{vfs}, c_{vfr}, c_{vfi} are respectively the cost functions of vertex substitution, removal, and insertion, as defined in (2.9).

Besides `bipartite`, other heuristics inheriting the `LSAPE-GED` were proposed as well. For instance, when constructing the `LSAPE` problem, the algorithm `STAR` considers the neighbors of each pair of vertices [Zeng et al., 2009]; the algorithm `SUBGRAPH` considers more global information, namely graphlets [Carletti et al., 2015]; while the algorithm `WALKS` associates each vertex in the input graphs to the set of walks of a given size starting at this vertex [Gaüzère et al., 2014]. Moreover, based on the upper bound computed by heuristics under the `LSAPE-GED` paradigm, possible refinement can be carried out, such as the procedure of the `LS-GED` paradigm.

4.2.2 The `LS-GED` paradigm

The local search (`LS-GED`) paradigm is composed of two steps. First, the transformation π and the cost $\mathrm{C}(\pi)$ are initialized randomly or by a heuristic, such as one under the `LSAPE-GED` paradigm. Then, starting at these initial results, a refinement procedure is carried out by a local search method to search for improved transformation with a lower cost. This paradigm is summarized in Algorithm 4.2 [Blumenthal et al., 2020]. With different strategies applied in the second step, vari-

Algorithm 4.2 Approximation of GED using `LS-GED`

Input: Graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$.
Vertex edit cost c_v, edge edit cost c_e.
Output: An approximation of GED between G_1 and G_2.

1: Compute the initial GED with an upper bound $C_0(\pi_0)$ and the corresponding transformation $\pi_0 \in \Pi(G_1, G_2)$ randomly or by a heuristic (e.g. `bipartite`).
2: Construct a new transformation $\pi \in \Pi(G_1, G_2)$ by a local search strategy starting at π_0 using G_1, G_2, c_v, and c_e so that $C(\pi) < C_0(\pi_0)$.
3: Set the upper bound as $C(\pi)$, namely the required approximation.

ous heuristics have been designed. `IPFP` is a well-known representative one.

The `IPFP` heuristic

GED can be modeled as a quadratic problem. The `LSAPE-GED` paradigm simplifies this problem by only considering the linear part of GED, namely the costs of vertex transformations, costs of edge transformations can only be implied as patches, as done by the `bipartite` method. In contrast, `IPFP` heuristics under the `LS-GED` paradigm provides a method to extend `LSAPE-GED`, by including the edge transformations as a quadratic part of GED. We present this idea by following a narration similar to [Bougleux et al., 2016].

An ε-assignment $\pi : V_1^\varepsilon \to V_2^\varepsilon$ transforms vertices in a graph $G_1 = (V_1, E_1)$ to vertices in a graph $G_2 = (V_2, E_2)$. Simultaneously, the transformation between E_1 and E_2 is conducted as follows:

- *Substitution:* an edge $(v_i, v_j) \in E_1$ is substituted by an edge $(\pi(v_i), \pi(v_j) \in E_2$ if and only if $\pi(v_i) \in V_2 \wedge \pi(v_j) \in V_2 \wedge (\pi(v_i), \pi(v_j)) \in E_2$.
- *Removal:* an edge $(v_i, v_j) \in E_1$ is removed (i.e., mapped to ε) if and only if $\pi(v_i) \notin V_2 \vee \pi(v_j) \notin V_2 \vee (\pi(v_i), \pi(v_j)) \notin E_2$.
- *Insertion*: an edge $(u_k, u_l) \in E_2$ is inserted to E_1 if and only if the substitutions did not affect it, namely $\pi^{-1}(u_k) \notin V_1 \vee \pi^{-1}(u_l) \notin V_1 \vee (\pi^{-1}(u_k), \pi^{-1}(u_l)) \notin E_1$.

For any $v_i, v_j \in V_1^\varepsilon$ and $u_k, u_l \in V_2^\varepsilon$, the cost to transform (v_i, v_j) to (u_k, u_l) can be presented as

$$\begin{aligned} q((v_i, v_j), (u_k, u_l)) \; = \; & c_{efs}((v_i, v_j), (u_k, u_l))\delta_{(v_i,v_j)}(E_1)\delta_{(u_k,u_l)}(E_2) \\ & + c_{efr}((v_i, v_j), \varepsilon)\delta_{(v_i,v_j)}(E_1)(1 - \delta_{(u_k,u_l)}(E_2)) \\ & + c_{efi}(\varepsilon, (u_k, u_l))(1 - \delta_{(v_i,v_j)}(E_1))\delta_{(u_k,u_l)}(E_2), \end{aligned} \tag{4.6}$$

where the delta function $\delta_x(S) = 1$ if $x \in S$ and 0 otherwise. Then the cost $Q(\pi)$ of transform π can be written as

$$Q(\pi) = g \sum_{v_i \in V_1^\varepsilon} \sum_{u_k \in \pi(v_i)} \sum_{v_j \in V_1^\varepsilon} \sum_{u_l \in \pi(v_j)} q((v_i, v_j), (u_k, u_l)), \tag{4.7}$$

where the coefficient is set to $g = 0.5$ if the graphs are undirected, as $d((v_i, v_j), (u_k, u_l)) = d((u_k, u_l), (v_i, v_j))$, and $g = 1$ otherwise. We define a binary matrix $\mathbf{X} \in \{0,1\}^{(|V_1|+1)\times(|V_2|+1)}$ equivalent to an ε-assignment π whose entries are

$$\begin{cases} x_{i,j} = \delta_{\pi(v_i),v_j}, \ \forall (v_i, v_j) \in V_1 \times V_2 \\ x_{i,|V_2|+1} = \delta_{\pi(v_i),\varepsilon}, \ \forall v_i \in V_1 \\ x_{|V_1|+1,j} = \delta_{\pi^{-1}(v_j),\varepsilon}, \ \forall v_j \in V_2 \\ x_{|V_1|+1,|V_2|+1} = 1, \end{cases} \tag{4.8}$$

where the Kronecker delta function $\delta_{x,y}$ equals to 1 if $x = y$ and 0 otherwise (see (2.1)). The corresponding binary vector $\mathbf{x} = \mathtt{vec}(\mathbf{X}) \in \{0,1\}^{(|V_1|+1)(|V_2|+1)}$ vectorizes $\mathbf{X}$ by concatenating its rows. Using $\mathbf{x}$, (4.7) can be compacted as

$$Q(\pi) = g \sum_{i=1}^{|V_1|+1} \sum_{k=1}^{|V_2|+1} \sum_{j=1}^{|V_1|+1} \sum_{l=1}^{|V_2|+1} q((v_i, v_j), (u_k, u_l)) = g\mathbf{x}^\top \mathbf{Q}\mathbf{x}, \tag{4.9}$$

where $\mathbf{Q} \in \mathbb{R}^{((|V_1|+1)(|V_2|+1))\times((|V_1|+1)(|V_2|+1))}$ is a matrix containing all costs between edges, whose (ik, jl)-th entry is $q((v_i, v_j), (u_k, u_l))$ in (4.6). As a result, the cost of the transformation π can be formalized as

$$C(\mathbf{x}) = g\mathbf{x}^\top \mathbf{Q}\mathbf{x} + \mathbf{c}^\top \mathbf{x}, \tag{4.10}$$

where $\mathbf{c} = \mathtt{vec}(\mathbf{C})$ is the edit cost vector. A tighter form of (4.10) is given in some literature [Bougleux et al., 2016], and can be summarized as follows. Let $\Delta = \mathbf{Q}$ for undirected graphs and otherwise $\Delta = \mathbf{Q} + \mathbf{Q}^\top$, meanwhile $\widehat{\Delta} = \frac{1}{2}\Delta + \mathtt{diag}(\mathbf{c})$. By rewriting (4.10), the approximation of GED can be then formalized as a quadratic assignment problem with edition (QAPE):

$$\mathtt{ged}(G_1, G_2) = \min_{\mathbf{X} \in \Pi(|V_1|,|V_2|)} \mathbf{x}^\top \widehat{\Delta}\mathbf{x}, \tag{4.11}$$

where $\Pi(|V_1|, |V_2|)$ represents the set of all possible maps between G_1 and G_2.

Algorithm 4.3 Approximation of GED using IPFP

Input: Graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$.
Vertex edit cost c_v, edge edit cost c_e.
Output: An approximation of GED between G_1 and G_2.

1: Initialize $\mathbf{X}_0$ and the corresponding cost C_0 randomly (or by a heuristic).
2: Let $i = 0$ and set i_{max}.
3: **while** Not converged and $i < i_{max}$ **do**
4: Compute $\mathbf{B}_{k+1} = \operatorname{argmin}_{\mathbf{B}\in\Pi(|V_1|,|V_2|)} (\mathbf{x}_i^\top \widehat{\Delta})\mathtt{vec}(\mathbf{B})$ with a LSAPE solver.
5: $C_{i+1} = \min\{C_i, C(\mathbf{B}_{k+1})\}$.
6: $\alpha_{i+1} = \operatorname{argmin}_{\alpha\in[0,1]} C(\mathbf{X}_i + \alpha(\mathbf{B}_{i+1} - \mathbf{X}_i))$.
7: $\mathbf{X}_{i+1} = \mathbf{X}_i + \alpha_{i+1}(\mathbf{B}_{i+1} - \mathbf{X}_i)$.
8: $i = i + 1$.
9: **end while**
10: C_{i+1} is the required approximation of the GED.

By adapting the integer projected fixed point (IPFP) algorithm [Leordeanu et al., 2009] designed for the quadratic assignment problem (QAP), the IPFP heuristic approximates the GED with the procedure presented in Algorithm 4.3 [Bougleux et al., 2016, Bougleux et al., 2015b]. The algorithm is first initialized randomly or by a heuristic such as bipartite (line 1), then updates by iterations (lines 3 to 8). In each iteration, a linear approximation is computed by a LSAPE solver where $\mathbf{x}_i^\top \widehat{\Delta}$ is regarded as the edit cost matrix (line 4). Then the local minimum of the cost and the corresponding binary solution is estimated by a line search (lines 6-7) [Bougleux et al., 2015b].

4.3 Stability of GED heuristics

The nature of the GED heuristics leads to a drop in computational stability, namely different trials may lead to different results. In the following, we analyze this instability by measuring the variability of the GED approximations over repeated trials. For instance, in the LSAPE-GED paradigm, the cost matrix $\mathbf{C}$ (Algorithm 4.1) may vary given vertex set with different orders, which affects the solution of the LSAPE problem, furthermore causing the instability. Likewise, in the LS-GED paradigm, the instability can be traced back to the initial procedure (Algorithm 4.2), where a random transformation may be assigned.

Low stability can degrade the performance of the GED heuristics, which implies a broader range of the confidence interval in a prediction task such as regression

Algorithm 4.4 Approximation of GED using mIPFP

Input: Graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$.
Vertex edit cost c_v, edge edit cost c_e. The number of solutions m.
Output: An approximation $C^\star$ of GED between G_1 and G_2.

Initialize cost $C^\star = C_0 = \infty$.
Let $j = 0$.
while $j < m$ **do**
    Approximate a new cost C_{j+1} with Algorithm 4.3.
    if $C^\star > C_{j+1}$ **then**
        $C^\star = C_{j+1}$.
    end if
    $j = j + 1$.
end while
$\texttt{ged}(G_i, G_j) = C^\star$.

and classification, or instability of the produced graphs in a pre-image problem. A straightforward method to mitigate this problem is repeating the GED computation. The minimum cost over repetitions is then chosen as the GED approximation. Strategies have been proposed to refine this method. Well-known ones are the $m$bipartite and mIPFP, which are the multi-start counterparts of bipartite and IPFP [Daller et al., 2018]. These two heuristics start several initial candidates simultaneously to acquire tighter upper bounds. The stability is concurrently ameliorated, which is examined in Figure 4.2 (a) and in the following analyses. Algorithm 4.4 presents the procedure of mIPFP.

Another factor that significantly influences the stability of GED heuristics turns out to be the relative values of vertex and edge edit costs. When vertex costs are markedly larger than edge costs, the GED stability often shows an observable improvement. This phenomenon is detailed in Figure 4.2 (b) and in the corresponding analyses.

In the remaining part of this section, we conduct experimental analyses on the GED stability. We first define a measure of the GED stability named *relative error*. Given a set of graphs $G_1, G_2 \ldots, G_N$, we compute the GED with a heuristic between each pair of graphs N_t times (trials). The relative error E_r is defined as

$$E_r = \frac{1}{N^2} \sum_{k=1}^{N_t} \frac{\sum_{i,j=1}^{N} \| (\texttt{ged}^{(k)}(G_i, G_j) - \texttt{ged}_0(G_i, G_j) \|}{\frac{1}{2} \left(\sum_{i,j=1}^{N} \texttt{ged}^{(k)}(G_i, G_j) + \texttt{ged}_0(G_i, G_j) \right)}, \tag{4.12}$$

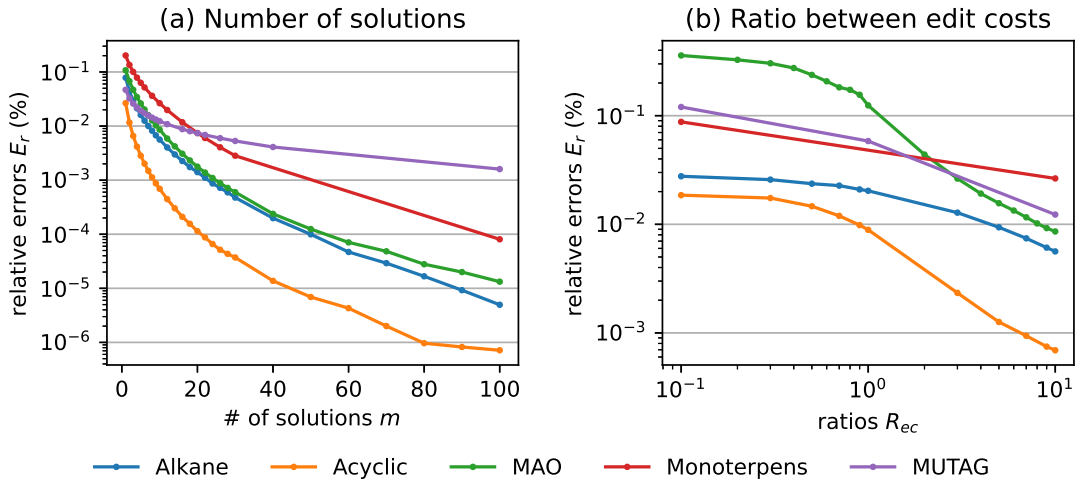

Figure 4.2 – The relative errors of mIPFP on five datasets with respect to the numbers of solutions and ratios between vertex and edge edit costs.

where $\mathrm{ged}^{(k)}(\mathrm{G}_i, \mathrm{G}_j)$ is the approximation of the GED in the k-th trial using Algorithm 4.4 and $\mathrm{ged}_0(\mathrm{G}_i, \mathrm{G}_j)$ is the exact GED between G_i and G_j. In practice, we replace ged_0 with the minimum approximation over all trials, namely $\mathrm{ged}_0(\mathrm{G}_i, \mathrm{G}_j) = \min_{1 \leqslant k \leqslant \mathrm{N}_t} \mathrm{ged}^{(k)}(\mathrm{G}_i, \mathrm{G}_j)$. The relative error E_r measures the average ratio between the offsets and the exact GEDs over trials and pairs of graphs. A smaller value indicates higher stability.

We analyse the stability with respect to two factors. The first one is the number of random initial candidates of the GED heuristic, namely "# of solutions". For mIPFP, it is equal to the parameter m in Algorithm 4.4. The second factor is the ratio between vertex and edge edit costs. Let c_{vfs}, c_{vfi}, c_{vfr}, c_{efs}, c_{efi}, $c_{efr} \in \mathbb{R}_+$ be the cost functions associated with, respectively, vertex substitutions, insertions, removals and edge substitutions, insertions, removals. Then the ratio is defined as

$$\mathrm{R}_{ec} = \frac{\mathtt{average}(c_{vfi}, c_{vfr}, c_{vfs})}{\mathtt{average}(c_{efi}, c_{efr}, c_{efs})}, \tag{4.13}$$

where $\mathtt{average}(\cdot)$ computes the average value of its inputs.

Figure 4.2 shows the effect of these factors on the relative error E_r defined in (4.12), considering the mIPFP algorithm on datasets *Alkane*, *Acyclic*, *MAO*, *Monoterpens*, and *MUTAG*. The first dataset is unlabeled, while the others contain symbolic labels. The left figure (a) exhibits how E_r drops with the increase of the "# of solutions" m. E_r drops rapidly when the solution number increases from 1 to 10, and

reaches at a relatively small value; the tendency mitigates afterwards. This result indicates that an adequately large number of solutions is necessary, thus a trade-off decision between stability and time complexity needs to be made for different applications.

The right figure (b) reveals the relation between E_r and the ratio R_{ec} between vertex and edge edit costs. Without loss of generality, the edit cost is set to be constant for each edit operation. The edge costs are set to be 1 and the vertex costs to be the ratio value (for insertions, removals, and substitutions). The removal costs of vertices (resp. edges) are set to 0 if vertices (resp. edges) are not labeled. E_r is relatively large when the ratio is smaller than 1, namely when edge costs are bigger than vertex costs, and drops with the increase of the ratio. We can observe that a larger ratio leads to higher stability. A possible cause of this phenomenon is that large edge costs amplify the arbitrariness of the edge edit operations. For graphs with n vertices, there are n^2 possible edges that can be inserted, removed, and substituted, which causes more uncertainty when constructing edit paths as well as computing their costs. Taking `IPFP` for instance, large edge costs lead to a big cost matrix $\mathbf{Q}$ in (4.10), implying the possibility of more variance on the value of the term $g\mathbf{x}^\top\mathbf{Q}\mathbf{x}$. Many edit costs given by domain experts are in accordance with this empirical rule, such as the ones in [Abu-Aisheh et al., 2017].

With the pre-knowledge of GED heuristics and their stability, we propose in the following a metric learning approach to optimize the edit costs, which is later applied to regression problems.

4.4 A metric learning approach to graph edit costs for regression

4.4.1 Related work

As stated in the overview, the choice of edit costs has a major impact on the computation of graph edit distance, and thus on the performance associated with the prediction task.

The first approach to design these costs is to set them manually, based on the knowledge on a given dataset/task (when such knowledge is available). This strategy leads, for instance, to the classical edit cost functions associated with the IAM dataset [Riesen and Bunke, 2008]. However, it is interesting to challenge these pre-

defined settings and experiment with how they can improve the prediction performance.

In order to fit a particular targeted property to predict, tuning the edit costs and thus the GED can be seen as a subproblem of metric learning. Metric learning consists in learning a dissimilarity (or similarity) measure given a training set composed of data instances and associated targeted properties. For the classical metric learning where each data instance is encoded by a real-valued vector, the problem consists in learning a dissimilarity measure, which decreases (resp. increases) where the vectors have similar (resp. different) targeted properties. Most metric learning studies focus on Euclidean data, while only a few addresses this problem on structured data [Bellet et al., 2013]. A complete review for general structured data representation is given in [Ontañón, 2020]. In the following, we will focus on existing studies to learn edit costs for graph edit distance.

A trivial approach to tune the edit costs is to use a grid search strategy among a predefined range. However, the complexity required to compute graph edit distance and the number of different edit costs forbid such an approach.

String edit distance constitutes a particular case of graph edit distance, associated with a lower complexity, where graphs are restricted to be only linear and sequential. In [Ristad and N.yianilos, 1998], the authors propose to learn edit costs using a stochastic approach. This method shows a performance improvement, hence demonstrating the interest to tune edit costs; it is however restricted to strings.

Another strategy is based on a probabilistic approach, as proposed by a series of papers [Neuhaus and Bunke, 2004, Neuhaus and Bunke, 2005, Neuhaus and Bunke, 2007]. By providing a probabilistic formulation for the common edition of two graphs, an Expectation-Maximization algorithm is used to derive weights applied to each edit operation. The tuning is then evaluated in an unsupervised manner. More especially in [Neuhaus and Bunke, 2005], the strategy consists in modifying the label space associated with vertices and edges such that edit operations occurring more often will be associated with lower edit costs. Conversely, higher values will be associated with edit operations occurring less often. The learning process was validated on two datasets. However, this approach is computationally too expensive when dealing with general graphs [Bellet et al., 2012].

In [Bellet et al., 2012], the authors propose an interesting way to evaluate whether a distance is a "good" one. This criterion is based on the following con-

cept:

> *a similarity function is* $(\epsilon, \gamma, \tau) - good$ *if a* $1 - \epsilon$ *proportion of examples are on average* 2γ *more similar to reasonable examples of the same class than to reasonable examples of the opposite class, where a* τ *proportion of examples must be reasonable.*

This principle is then derived to define an objective function to optimize. The matrix encoding the edit costs minimizing this objective function is then used to compute edit distances. However, this approach has only been adapted to strings and trees, but not to general graphs.

Another class of methods that address the problem of learning edit costs for GED is proposed in [Cortés and Serratosa, 2015, Cortés et al., 2019]. These methods propose to optimize the edit costs by maximizing the similarity between the computed mapping and a ground truth mapping between vertices of graphs. This framework requires thus a ground truth mapping, which is not available on most datasets such as the ones in the chemoinformatics domain.

4.4.2 Problem formulation

In this section, we propose an optimization procedure to learn edit costs in the context of regression tasks. Consider a dataset $\mathcal{G}$ of N graphs such that each graph $G_k = (V_k, E_k)$, for $k = 1, 2, \ldots, N$, where V_k represents the set of vertices of G_k labeled by a function $\ell_v : V \to L_v$, and E_k encodes the set of edges of G_k, namely $e_{ij} = (v_i, v_j) \in E_k$ if and only if an edge connects vertices v_i and v_j in G_k.

Recalling Section 2.3, the graph edit distance between two graphs is defined as the minimal cost associated with an optimal edit path. Given two graphs G_1 and G_2, an edit path between them is defined as a sequence of edit operations transforming G_1 into G_2. An edit operation e can correspond to a vertex substitution $e = (v_i \to v_j)$, removal $e = (v_i \to \varepsilon)$ or insertion $e = (\varepsilon \to v_j)$. Similarly, for edges, we have $(e_{ij} \to e_{kl}), (e_{ij} \to \varepsilon)$, and $(\varepsilon \to e_{kl})$. Each edit operation is associated with a cost characterizing the distortion induced by this edit operation on the graph. These costs can be encoded by a cost function c that associates a positive real value to each edit operation, depending on the elements being transformed.

In the remainder of this chapter, we will restrict ourselves to only constant cost functions. Therefore, we can associate each edit operation to a constant value. Let

c_{vs}, c_{vi}, c_{vr}, c_{es}, c_{ei}, $c_{er} \in \mathbb{R}_+$ be the cost values associated with, respectively, vertex substitutions, insertions, removals and edge substitutions, insertions, removals.

As shown in Section 2.3, given a mapping and considering constant cost functions, the cost associated with edit operations of an edit path represented by π is given by:

$$\mathrm{C}(\pi, \mathrm{G}_1, \mathrm{G}_2) = \mathrm{C}_v(\pi, \mathrm{G}_1, \mathrm{G}_2) + \mathrm{C}_e(\pi, \mathrm{G}_1, \mathrm{G}_2), \tag{4.14}$$

where $\mathrm{C}_v(\pi, \mathrm{G}_1, \mathrm{G}_2)$ is the cost associated with vertex operations (2.13), namely

$$\mathrm{C}_v(\pi, \mathrm{G}_1, \mathrm{G}_2) = \sum_{\substack{v \in \mathrm{V}_2 \\ \pi^{-1}(v)=\varepsilon}} c_{vi} + \sum_{\substack{v \in \mathrm{V}_1 \\ \pi(v)=\varepsilon}} c_{vr} + \sum_{\substack{v \in \mathrm{V}_1 \\ \pi(v)\neq\varepsilon}} c_{vs},$$

and $\mathrm{C}_e(\pi, \mathrm{G}_1, \mathrm{G}_2)$ is the one associated with edge operations (2.14), namely

$$\mathrm{C}_e(\pi, \mathrm{G}_1, \mathrm{G}_2) = \sum_{\substack{e=(v_i,v_j)\in \mathrm{E}_2 \mid \\ \pi^{-1}(v_i)=\varepsilon \vee \\ \pi^{-1}(v_j)=\varepsilon \vee \\ (\pi^{-1}(v_i),\pi^{-1}(v_j))\notin \mathrm{E}_1}} c_{ei} + \sum_{\substack{e=(v_i,v_j)\in \mathrm{E}_1 \mid \\ \pi(v_i)=\varepsilon \vee \\ \pi(v_j)=\varepsilon \vee \\ (\pi(v_i),\pi(v_j))\notin \mathrm{E}_2}} c_{er} + \sum_{\substack{e=(v_i,v_j)\in \mathrm{E}_1 \mid \\ \pi(v_i)\neq\varepsilon \wedge \\ \pi(v_j)\neq\varepsilon \wedge \\ (\pi(v_i),\pi(v_j))\in \mathrm{E}_2}} c_{es}.$$

Let n_{vs} be the number of vertex substitutions, i.e., the cardinality of the subset of V_1 being mapped onto V_2. This number is given by the number of terms of the first sum in (2.13), i.e., $n_{vs} = |\{v_i \in \mathrm{V}_1 \mid \pi(v_i) \neq \varepsilon\}|$. Similarly:

- The number of vertex removals is $n_{vr} = |\{v_i \in \mathrm{V}_1 \mid \pi(v_i) = \varepsilon\}|$;
- The number of vertex insertions is $n_{vi} = |\{v_i \in \mathrm{V}_2 \mid \pi^{-1}(v_i) = \varepsilon\}|$;
- The number of edge substitutions is $n_{es} = |\{e = (v_i, v_j) \in \mathrm{E}_1 \mid \pi(v_i) \neq \varepsilon \wedge \pi(v_j) \neq \varepsilon \wedge (\pi(v_i), \pi(v_j)) \in \mathrm{E}_2\}|$;
- The number of vertex removals is $n_{ei} = |\{e = (v_i, v_j) \in \mathrm{E}_1 \mid \pi(v_i) = \varepsilon \vee \pi(v_j) = \varepsilon \vee (\pi(v_i), \pi(v_j)) \notin \mathrm{E}_2\}|$;
- The number of vertex insertions is $n_{er} = |\{e = (v_i, v_j) \in \mathrm{E}_2 \mid \pi^{-1}(v_i) = \varepsilon \vee \pi^{-1}(v_j) = \varepsilon \vee (\pi^{-1}(v_i), \pi^{-1}(v_j)) \notin \mathrm{E}_1\}|$.

Then, let $\mathbf{x} \in \mathbb{N}^6$ encode the number of each edit operation as $\mathbf{x} = [n_{vi}, n_{vr}, n_{vs}, n_{ei}, n_{er}, n_{es}]^\top$. Note that these values depend on both graphs being compared and a given mapping between vertices. Similarly, we define a vector representation of the costs associated with each edit operation by $\mathbf{c} =$

$[c_{vi}, c_{vr}, c_{vs}, c_{ei}, c_{er}, c_{es}]^\top \in \mathbb{R}_+^6$. Given these representations, the cost associated with an edit path, as defined by (4.14), can be rewritten as:

$$\mathrm{C}(\pi, \mathrm{G}_1, \mathrm{G}_2, \mathbf{c}) = \mathbf{x}^\top \mathbf{c}. \tag{4.15}$$

Therefore, the graph edit distance between two graphs is defined as:

$$\mathtt{ged}(\mathrm{G}_1, \mathrm{G}_2, \mathbf{c}) = \underset{\pi,}{\operatorname{argmin}}\, \mathrm{C}(\pi, \mathrm{G}_1, \mathrm{G}_2, \mathbf{c}). \tag{4.16}$$

4.4.3 Learning the edit costs

Consider that each graph $\mathrm{G}_k \in \mathscr{G}$ is associated with a particular targeted property $y_k \in \mathscr{Y}$, namely the target in regression tasks (e.g. $\mathscr{Y} \subseteq \mathbb{R}$ for real-valued output regression). Furthermore, a distance $\mathrm{d}_{\mathscr{Y}} : \mathscr{Y} \times \mathscr{Y} \to \mathbb{R}_+$ is defined on this targeted property. Since this target space $\mathscr{Y}$ is often a vector space, we consider in this thesis two distances on $\mathscr{Y}$:

- The *Euclidean* distance: $\mathrm{d}_{\mathscr{Y}}(y_i, y_j) = \|y_i - y_j\|_2$.
- The *Manhattan* distance: $\mathrm{d}_{\mathscr{Y}}(y_i, y_j) = \|y_i - y_j\|_1$.

The main idea behind the proposed method is that the best metric in the GED space is the one best aligned to the target distances (i.e., $\mathrm{d}_{\mathscr{Y}}$). With this distance-preserving principle, we seek to learn the edit cost vector $\mathbf{c}$ by fitting the GEDs between graphs to the distances between their targeted properties. Ideally, we seek to preserve the GED between any two graphs G_i and G_j and the distance between their targeted properties. Considering the set of N available graphs $\mathrm{G}_1, \ldots, \mathrm{G}_\mathrm{N}$ and their corresponding targets $y_1, \ldots, y_\mathrm{N}$, we seek to have

$$\mathtt{ged}(\mathrm{G}_i, \mathrm{G}_j, \mathbf{c}) \approx \mathrm{d}_{\mathscr{Y}}(y_i, y_j) \qquad \text{for all } i, j = 1, 2, \ldots \mathrm{N}. \tag{4.17}$$

Let $\omega : \mathscr{G} \times \mathscr{G} \times \mathbb{R}_+^6 \to \mathbb{N}^6$ be the function that computes an optimal edit path between G_i and G_j according to the cost vector $\mathbf{c}$ and returns the vector $\mathbf{x}^\star \in \mathbb{R}_+^6$ of numbers of edit operations associated with this optimal edit path, namely $\mathbf{x}^\star = \omega(\mathrm{G}_i, \mathrm{G}_j, \mathbf{c})$. This function can be obtained by any method computing an exact or sub-optimal graph edit distance [Abu-Aisheh et al., 2017, Blumenthal et al., 2020].

For any pair of graphs $(\mathrm{G}_i, \mathrm{G}_j)$, let $\mathbf{x}_{i,j}$ be a vector encoding the number of each edit operation. Let $\mathbf{X} \in \mathbb{N}^{\mathrm{N}^2 \times 6}$ be the matrix of the numbers of edit operations for

Algorithm 4.5 Optimization of edit costs according to given targets

1: $\mathbf{c} \leftarrow$ random(6)
2: $\mathbf{X} \leftarrow [\omega(G_1, G_1, \mathbf{c})\ \ \omega(G_1, G_2, \mathbf{c})\ \ \cdots\ \ \omega(G_N, G_N, \mathbf{c})]^\top$
3: **while** not converged **do**
4: $\quad \mathbf{c} \leftarrow \operatorname{argmin}_{\mathbf{c}} ||\mathbf{Xc} - \mathbf{d}||_2^2$, subject to $\mathbf{c} > 0$
5: $\quad \mathbf{X} \leftarrow [\omega(G_1, G_1, \mathbf{c})\ \ \omega(G_1, G_2, \mathbf{c})\ \ \cdots\ \ \omega(G_N, G_N, \mathbf{c})]^\top$
6: **end while**

each pair of graphs, namely its $(i\mathrm{N} + j)$-th row is $\mathbf{x}_{i,j}^\top$. Then, $\mathbf{Xc}$ is the $\mathrm{N}^2 \times 1$ vector composed of edit distances computed according to $\mathbf{c}$ and $\mathbf{X}$ between all pairs of graphs. Let $\mathbf{d} \in \mathbb{R}_+^{\mathrm{N}^2}$ be a vector of the differences on targeted properties according to $\mathrm{d}_{\mathscr{Y}}$, with $\mathbf{d}(i\mathrm{N} + j) = \mathrm{d}_{\mathscr{Y}}(y_i, y_j)$. Therefore, the optimization problem can be rewritten as:

$$\underset{\mathbf{c}}{\operatorname{argmin}} \quad \mathscr{L}(\mathbf{Xc}, \mathbf{d}) \qquad \text{subject to} \quad \mathbf{c} > 0, \tag{4.18}$$

where $\mathscr{L}$ denotes a loss function and the constraint on $\mathbf{c}$ forces non-negative costs. Besides this constraint, one can also integrate a constraint to satisfy the triangular inequality, or one to ensure that a removal cost is equal to an insertion cost [Riesen, 2015].

In the case of a regression problem, $\mathscr{L}$ can be defined as the mean square error between computed graph edit distances and dissimilarities between the targeted properties. Therefore, the final optimization problem is:

$$\underset{\mathbf{c}}{\operatorname{argmin}} \quad ||\mathbf{Xc} - \mathbf{d}||_2^2 \qquad \text{subject to} \quad \mathbf{c} > 0. \tag{4.19}$$

Estimating $\mathbf{c}$ by solving this constrained optimization problem allows to linearly fit graph edit distances to a particular targeted property according to the edit paths initially given by ω. However, changing the edit costs may influence the optimal edit path, and consequently its description in terms of the number of edit operations. There is thus an interdependence between the function ω computing an optimal edit path according to $\mathbf{c}$, and the objective function optimizing $\mathbf{c}$ according to edit paths encoded within $\mathbf{X}$. To solve this interdependence, we propose an alternated optimization strategy, summarized in Algorithm 4.5. The two main steps of the algorithm are described next:

- Estimate $\mathbf{c}$ for fixed $\mathbf{X}$ (line 4): This optimization problem is a constrained

linear problem that can be resolved using off-the-shelf solvers, such as `CVXPY` [Diamond and Boyd, 2016] and `scipy` [Virtanen et al., 2020]. This optimization problem can also be viewed as a non-negative least squares problem [Lawson and Hanson, 1995]. For a given set of edit operations between each pair of graphs, this step linearly optimizes the constant costs to be applied by minimizing the difference between graph edit distances and distances between targets.

- Estimate $\mathbf{X}$ for fixed $\mathbf{c}$ (line 5): The modification performed on costs in the previous step may have an influence on the associated edit path. To address this point, the optimization of costs is followed by a re-computation of the optimal edit paths according to the newly computed $\mathbf{c}$ vector encoding the edit costs. This step can be achieved by any method computing graph edit distance. For the sake of computational time, one can choose an approximated version of GED [Bougleux et al., 2015a, Blumenthal et al., 2020].

This alternated optimization is repeated to compute both edit costs and edit operations. Since we do not have any theoretical proof of the convergence of this optimization scheme, we limit the number of iterations to 5 in our implementation, which turns out to be sufficient in the conducted experiments.

4.4.4 Experiments

We conducted experiments[1] on two well-known datasets in chemoinformatics, both composed of molecules and their boiling points, namely *Alkane* and *Acyclic* presented in Section 2.4. *Alkane* is composed of 150 acyclic molecules that are modeled as acyclic unlabeled graphs, while *Acyclic* is composed of 185 acyclic molecules that are represented as acyclic labeled graphs.

To evaluate the predictive power of different settings of edit costs, we used a k-nearest-neighbors regression [Altman, 1992] model, where k is the number of the neighbors considered to predict a property. The performances are estimated on ten different random splits. For each split, a test set representing 10% of the graphs in the dataset is randomly selected and used to measure the performance of the prediction. The remaining 90% are used to optimize the edit costs and the value of

[1] For the sake of reproducibility, the code is available at the following repository: `https://github.com/jajupmochi/graphkit-learn/tree/master/gklearn/experiments/thesis/ged/fit_distances`.

Table 4.1 – Results on each dataset in terms of RMSE for the 10 splits, measured on the training and on the test sets.

Dataset	Distance	Method	*bipartite*		*IPFP*	
			Train errors	**Test errors**	**Train errors**	**Test errors**
Alkane	Euclidean	random	11.77±2.59	15.54±5.58	9.001±1.08	9.56±2.39
		expert	9.75±0.75	13.45±3.98	7.53±0.23	8.26±1.15
		GH2020	10.92±1.08	12.61±2.47	8.50±0.97	9.19±2.35
		fitted	**5.93±0.39**	**6.88±3.23**	**5.83±0.25**	**5.03±0.93**
	Manhattan	random	17.30±6.36	22.00±12.66	14.13±6.09	15.82±6.40
		expert	9.63±0.42	10.83±1.71	8.14±0.80	7.95±1.28
		GH2020	10.92±1.08	12.61±2.47	8.50±0.97	9.19±2.35
		fitted	**6.10±0.23**	**5.32±0.91**	**5.56±0.22**	**5.97±0.90**
Acyclic	Euclidean	random	25.96±2.63	31.79±4.90	21.94±4.03	27.48±7.84
		expert	26.63±0.59	33.46±2.22	22.92±0.62	25.68±2.65
		GH2020	28.62±4.23	33.30±6.67	21.98±4.85	23.04±3.78
		fitted	**10.62±0.98**	**17.29±2.52**	**10.66±1.06**	**14.71±3.14**
	Manhattan	random	26.33±4.34	32.36±7.06	17.73±3.57	23.15±5.83
		expert	27.34±0.52	31.29±3.22	23.11±0.74	26.53±3.52
		GH2020	28.62±4.23	33.30±6.67	21.98±4.85	23.04±3.78
		fitted	**10.66±0.56**	**16.66±2.83**	**10.93±1.29**	**16.68±3.12**

k, where k is optimized through a 5-fold cross-validation (CV) procedure over the candidate values $\{3,5,7,9,11\}$. The number of iterations for the optimization of the edit costs is fixed to 5.

The proposed optimization procedure is compared to three other edit costs settings: the first one is a random set of edit costs. The second one is a predefined cost setting given in [Abu-Aisheh et al., 2017], namely the so-called expert costs with $c_{vi} = c_{vr} = c_{ei} = c_{er} = 3$, $c_{vs} = c_{es} = 1$. The third setting is obtained from a state-of-the-art method that optimizes the edit costs, as proposed recently in [Garcia-Hernandez et al., 2020] (denoted GH2020 in the following). We made some adaptations to this method to have a fair comparison:

- GH2020 was initially targeted at a binary classification problem and their objective function is an accumulation of a log loss function that measures whether the compared graphs are in the same class. To adapt it to regression problems, we use the RMSE between targets (e.g., boiling points) instead.

- GH2020 used extended reduced graphs (ErG). We use our dataset directly on

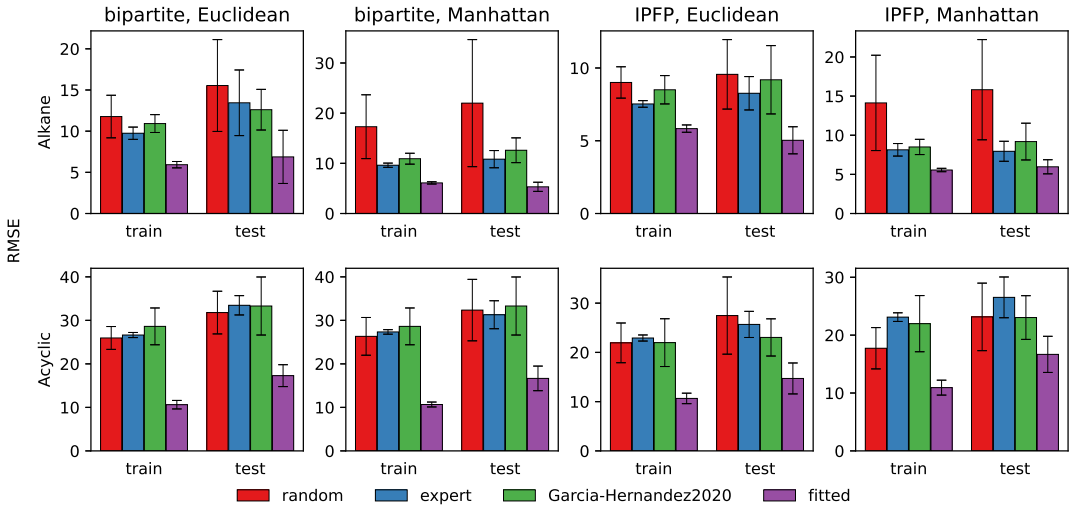

Figure 4.3 – Results on each dataset in terms of RMSE for the 10 splits, measured on the training and on the test sets.

this method.

- GH2020 defined an edit cost between each pair of vertex/edge attributes. We use constant costs instead.
- In [Garcia-Hernandez et al., 2020], only one edit cost over all costs was updated in each experiment, and 4 experiments were conducted. We update all constant edit costs simultaneously.

Table 4.1 shows the average root mean squared errors (RMSE) obtained for each cost settings over the 10 splits, estimated on the training set and on the test set. The ± sign gives the 95% confidence interval computed over the 10 repetitions. Figure 4.3 shows a different representation of the same results with error bars modeling the 95% confidence interval. As expected, a clear and significant gain in accuracy is obtained when using fitted costs on the two datasets, using both `bipartite` and `IPFP` heuristics and both *Euclidean* and *Manhattan* distances between target values. These promising results confirm the hypothesis that ad-hoc edit costs may help the graph edit distance catch better targeted properties that are associated with a graph, and thus improve the prediction accuracy while still operating in the graph space.

The fitted values of edit costs are summarized in Table 4.2. From these results, we can observe that insertion and removal costs are almost similar when using our optimization method, hence showing the symmetry of these operations. Also, one

Table 4.2 – Average and standard deviation of fitted edit costs values.

Dataset	Edit cost	Distance	c_{ni}	c_{nr}	c_{ns}	c_{ei}	c_{er}	c_{es}
Alkane	bipartite	Euclidean	26.45±0.48	26.24±0.60	-	0.13±0.06	0.14±0.09	-
		Manhattan	26.67±0.37	26.63±0.58	-	0.11±0.04	0.11±0.06	-
	IPFP	Euclidean	26.12±0.24	25.88±0.25	-	0.74±0.23	0.78±0.23	-
		Manhattan	25.94±0.38	25.71±0.44	-	0.89±0.30	0.77±0.29	-
Acyclic	bipartite	Euclidean	13.81±0.48	13.83±0.80	10.46±0.40	1.37±0.46	1.45±0.46	1.41±0.09
		Manhattan	13.76±0.39	14.14±0.57	10.28±0.44	1.44±0.20	1.45±0.19	1.45±0.07
	IPFP	Euclidean	11.61±0.45	11.68±0.43	11.07±0.53	4.49±0.30	4.46±0.24	4.48±0.18
		Manhattan	11.52±0.40	11.40±0.40	10.61±0.52	4.50±0.31	4.50±0.31	4.50±0.10

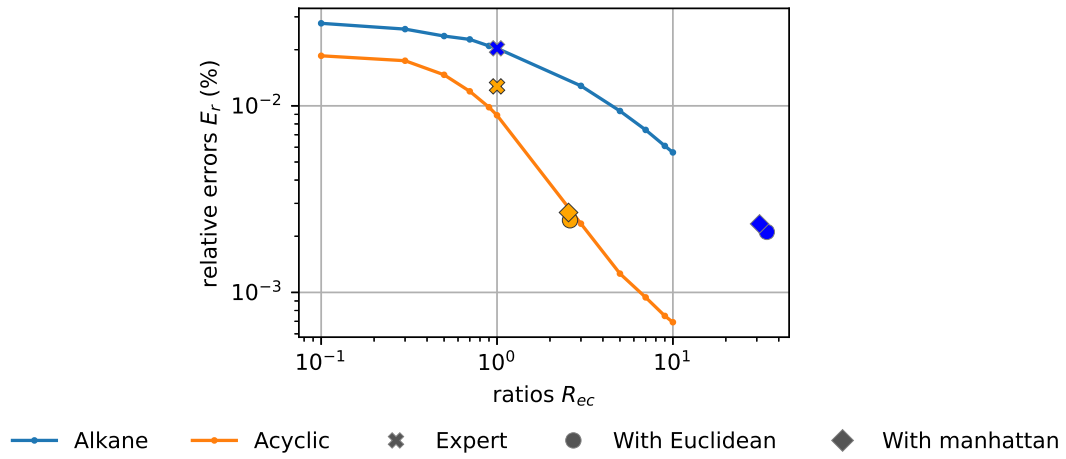

Figure 4.4 – The relative errors E_r of mIPFP on datasets *Alkane* and *Acyclic* with respect to the ratios R_{ec} between vertex and edge edit costs using different edit costs optimization methods.

can observe that removal and insertion costs are more important than substitution costs, which shows that the number of atoms is more important than the atom itself. This is coherent with the chemistry theory [Cherqaoui and Villemin, 1994]. Finally, we can note that costs associated with vertices are higher than the ones associated with edges.

We further examine the stability of the IPFP method when using these edit costs. Figure 4.4 demonstrates the relations between the relative errors E_r and the ratios R_{ec} between vertex and edge costs (See Section 4.3 and Figure 4.2 for more details). Therein the colors represent datasets (i.e., blue for *Alkane* and orange for *Acyclic*), and shapes represent different edit costs, with ✖, ●, and ◆ respectively for the expert costs, the optimized costs using the Euclidean distance, and the optimized costs using the Manhattan distance. For the expert costs, R_{ec}'s for the two datasets

are both 1, and E_r's are 0.02 for *Alkane* and 0.013 for *Acyclic*; when using the Euclidean distance, $R_{ec} = 34.21$, $E_r = 0.002$ for *Alkane*, and $R_{ec} = 2.6$, $E_r = 0.002$ for *Acyclic*; when using the Manhattan distance, $R_{ec} = 3.11$, $E_r = 2.33 \times 10^{-3}$ for *Alkane*, and $R_{ec} = 2.55$, $E_r = 2.68 \times 10^{-3}$ for *Acyclic*. It can be observed that using the optimized costs corresponds to much higher ratios and an order of magnitude lower relative errors than using the expert costs. Thus, an empirical conclusion can be derived, that the obtained optimized edit costs correspond to higher stability of GEDs.

4.5 Conclusion and future work

In this chapter, we first conducted analyses of the stability of GED heuristics, showing its strong connection with the number of random initial candidates of multi-start GED heuristics and the relation between vertex and edge edit costs. After that, we introduced a new principle to define optimal graph edit costs of a GED for a given task. Based on this principle, we defined the optimization problem of fitting the edit costs to a particular metric, measured for instance on a targeted property to predict. An alternated optimization strategy was proposed to solve this optimization problem. The conducted experiments on two well-known datasets showed that the optimization process leads to a GED with a better predictive power compared to other methods. All these observations confirm that the proposed method helps to fit edit costs and outperforms other methods. A further investigation indicates higher stability of GED computation given the optimized edit costs.

There are still several challenges to address in future work. First, a clear and complete comparison to other methods cited in the introduction and related work will be established. Second, we seek to examine other criteria than the distance-preserving criterion, such as the conformal map [Honeine and Richard, 2011]. Third, from a theoretical point of view, we are interested in establishing convergence proof on our alternated optimization strategy, and to extend these proofs to approximate computations of graph edit distances. Fourth, this scheme will be extended to classification problems and non-constant costs to be applicable in most application domains. Considering non-constant costs will need to optimize parametric functions rather than scalar values, hence complexifying the procedure.

The optimization problem defined in this chapter provides a new perspective to metric learning on graphs, allowing aligning graph space according to a target space. From this idea, we propose in the next chapter a graph pre-image strategy.

Chapter 5

Graph pre-image based on graph edit distances

Contents

5.1 Overview

Graph kernels provide an elegant strategy to deal with graphs in kernel space by implicitly embedding graphs into that space, where a given graph kernel function $k : \mathbb{G}^2 \to \mathbb{R}_+$ constructs a graph kernel space $\mathcal{H}$ and $\phi_{\mathcal{H}}(G)$ represents the (implicit) mapping of a graph G in $\mathcal{H}$. Graph pre-image is the reverse process of this embedding, aiming at constructing a graph $G^\star$ given an element ψ in the kernel space $\mathcal{H}$, such that $\phi(G^\star) = \psi$. Finding graph pre-images has many interesting applications, including molecule synthesis, drug design, and image reconstruction. Figure 5.1 illustrates the process of constructing a molecule by graph pre-image.

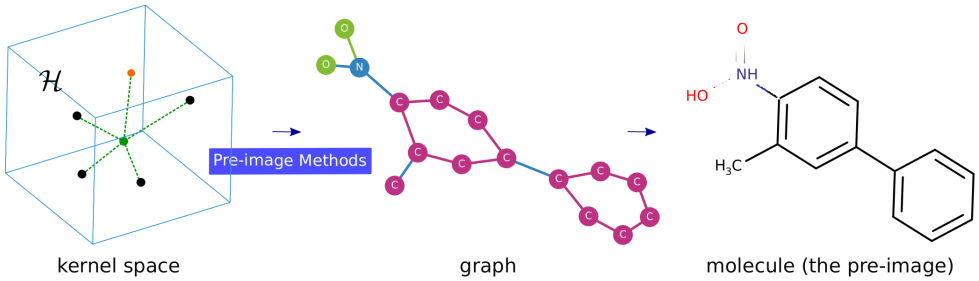

Figure 5.1 – An illustration of graph pre-image

The pre-image problem was originally defined on vectors and has been largely studied on Euclidean spaces. Finding the exact pre-image is challenging as most elements in the kernel space do not correspond to valid pre-images in the input space. Meanwhile, the reverse mapping of an element in kernel space to the input space does not exist in most cases. Methods to approximate pre-images have been therefore proposed. We refer interested readers to Section 2.2.3 for more details.

The graph pre-image problem inherits the difficulties of the traditional one on vectors and even more challenging to tackle due to the complexity of its inputs, namely the graphs. Graphs do not lie in a continuous space as vectors do. The number of vertices and edges in a graph are arbitrary integers, leading to a variation of graph sizes in a dataset. Moreover, each vertex and edge can be equipped with multiple symbolic and non-symbolic labels, possibly multi-dimensional. These structure features make it an involved problem to look for and construct a graph pre-image.

Some pioneer works have been proposed to deal with the graph pre-image problem. In [Bakır et al., 2004], the authors present a method based on a random search. To find a graph pre-image of an element Ψ in kernel space, an iterative procedure is

employed. The target pre-image is initialized by the pre-image of the nearest neighbor if Ψ. In each iteration, an initial set is constructed consisting of the graph pre-images of the k-nearest neighbors of Ψ and the target pre-image. For each graph in the initial set, l new candidate graphs are generated by randomly adding and deleting edges. After that, the target pre-image is updated by the candidate graph whose mapping in the kernel space is the closest to Ψ. This method is simple to implement, but has a very high computational complexity and is not applicable to continuous real-value labels, while the quality of the synthesized graph pre-images is not guaranteed as the random search is applied. Authors in [Akutsu and Fukagawa, 2005] and [Nagamochi, 2009] propose methods to infer a graph from path frequency. The former considers two types of structures: sequences are inferred from feature vectors corresponding to the spectrum kernel, while trees of bounded degree are inferred from feature vectors, which are composed of frequency of paths of fixed length under a fixed alphabet. The latter regards the problem as inferring a graph from a feature vector of path frequency with length 1. By formulating it as a loopless and connected detachment finding problem, the problem is solved by a method based on matroid intersection in discrete optimization. However, these methods are either restricted to applying a specific sub-structure of graphs, or ignoring vertex and edge labels, which are important information for graphs. All these studies do not fully benefit from discrete optimization that needs to be carried out for graph pre-image.

Benefiting from the ability of graph edit distances (GEDs) to construct graphs, we propose a generalized pre-image framework for graphs in this chapter, by bridging the gap between GEDs and any given graph kernel. The relationship between graph space and kernel space is studied. Within the proposed framework, three variants are proposed, depending on if the edit costs of the GEDs are given randomly, by experts, or optimized automatically. We emphasize particularly on the last variant, where the metric learning strategy is applied, aligning GEDs and distances of the elements in the kernel space. A method that optimizes the edit costs of the GED edit operations is proposed for this purpose, which is inspired by the similar procedure proposed in Section 4.4. The graph pre-image is then constructed in the aligned graph space, which is endowed with the GED measure with the optimized edit costs, in the spirit of metric learning in machine learning. we address a pre-image problem for median graph G of a graph set $\mathcal{G}$, where the median graph is approximated by the pre-image of the mean ψ of a set of elements in the kernel

space, which corresponds to the mappings of graphs in $\mathscr{G}$. In [Boria et al., 2019], the authors show that the value of an attribute of a vertex or an edge in the generalized median graph takes the form of the mean of the replacing values of that attribute in all graphs in $\mathscr{G}$. This form shows the underlying connection between G and ψ. Experimental results in Section 5.4 confirm this relevance. To solve the graph pre-image problem in this chapter, we revisit a recently developed iterative alternate minimization procedure that was proposed to generate median graphs [Boria et al., 2019]. This procedure can be replaced by any other generative median graph algorithms, such as the ones based on heuristic procedures [Musmanno and Ribeiro, 2016], graph embedding [Ferrer et al., 2010], combinatorial search and genetic algorithms [Bunke et al., 1999].

In the remainder of this chapter, we first formulate the problem (Section 5.2), then present the proposed method in two folds, namely the metric learning using distances in kernel spaces (Section 5.3.1) and the graph pre-image construction procedure (Section 5.3.2). After that, the experiments are conducted (Section 5.4), and the conclusion and future work are presented (Section 5.5).

5.2 Problem formulation

In this section, we formulate a novel pre-image procedure by generative graph edit distance methods. Given a space $\mathbb{G}$ of possibly labeled graphs, we consider a dataset $\mathscr{G}_N \subset \mathbb{G}$ of N graphs in which each graph is defined by $G_i = (V_i, E_i)$, for $i = 1, 2, \ldots, N$, with V_i being the vertex set of G_i labeled by a function $\ell_v : V \to L_v$, and E_i the edge set of G_i labeled by a function $\ell_e : V \to L_e$, as defined in Section 2.1. We denote the adjacent matrix of each graph G_i as A_i.

Following the definitions in Section 2.2 as shown in Figure 5.2, a given graph kernel function $k : \mathbb{G}^2 \to \mathbb{R}_+$ constructs a graph kernel space $\mathscr{H}$. Let $k(G_i, G_j) =< \phi_{\mathscr{H}}(G_i), \phi_{\mathscr{H}}(G_j) >$ be the inner product in $\mathscr{H}$, where $\phi_{\mathscr{H}}(G_i)$ represents the (implicit) mapping of G_i in $\mathscr{H}$. Given a $\psi \in \mathscr{H}$, the object of the exact graph pre-image problem is to find a graph $G^\star$ such that $\phi(G^\star) = \psi$, namely the pre-image of ψ. As ψ does not have a valid pre-image in general, the pre-image problem consists in estimating an approximate solution, namely $\widehat{G}$ such that $\phi(\widehat{G}) \approx \psi$. This definition is suitable for any special situations, such as the pre-image of the combination $\psi = \sum_i \alpha_i \phi(G_i)$ for $i = 1, 2, \ldots, N$, where $\alpha_i \in \mathbb{R}_+$ is a coefficient.

The graph edit distance is used as the dissimilarity measure in the graph space,

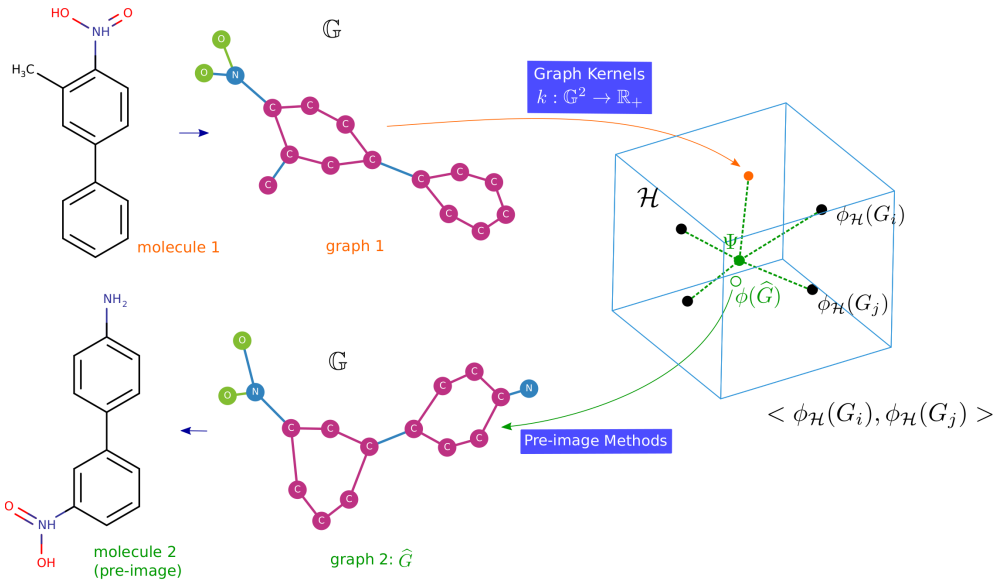

Figure 5.2 – Graph kernels and graph pre-images.

which is defined as the minimal cost associated with an optimal edit path (Section 2.3). An edit path consists of a sequence of edit operations transforming graph G_i to G_j, namely the removal, insertion, and substitutions of vertices and edges. A mapping π from G_i to G_j is defined by an edit path. A cost is assigned to each operation measuring the distortion induced by that operation. A cost function c encodes these costs by associating a positive real value to each edit costs, relying on the vertex or the edge being transformed.

In this chapter, we consider only constant edit costs. That is to say, we associate constant values c_{vr}, c_{vi}, c_{vs}, c_{er}, c_{ei}, $c_{es} \in \mathbb{R}_+$ with respectively vertex removals, insertions, substitutions and edge removals, insertions, substitutions. To construct an edit path, a coefficient is associated with each edit cost. Various settings can be assigned to these coefficients. For removal and insertion operations, one of the commonly used corresponding coefficients is the numbers of occurrences of these operations, namely

- The number of vertex insertions is $n_{vi} = |\{v_i \in V_2 \mid \pi^{-1}(v_i) = \varepsilon\}|$;
- The number of vertex removals is $n_{vr} = |\{v_i \in V_1 \mid \pi(v_i) = \varepsilon\}|$;
- The number of vertex insertions is $n_{er} = |\{e = (v_i, v_j) \in E_2 \mid \pi^{-1}(v_i) = \varepsilon \vee \pi^{-1}(v_j) = \varepsilon \vee (\pi^{-1}(v_i), \pi^{-1}(v_j)) \notin E_1\}|$;
- The number of vertex removals is $n_{ei} = |\{e = (v_i, v_j) \in E_1 \mid \pi(v_i) = \varepsilon \vee \pi(v_j) =$

$\varepsilon \vee (\pi(v_i), \pi(v_j)) \notin \mathrm{E}_2\}|$,

where ε represents a dummy element. For graphs with vertices and edges labeled by continuous values, one option of the coefficient of substituting elements is given by the sum of distance measures between the values of the original and the new labels over all substituted elements, namely

- The sum of distances of vertex substitution is

$$\omega_{vs} = \sum_{v_i \in \mathrm{V}_1, \pi(v_i) \in \mathrm{V}_2} \|\ell_{v_1}(v_i) - \ell_{v_2}(\pi(v_i))\|;$$

- The sum of distances of edge substitution is

$$\omega_{es} = \sum_{e=(v_i,v_j) \in \mathrm{E}_1, \pi(v_i) \neq \varepsilon \wedge \pi(v_j) \neq \varepsilon \wedge (\pi(v_i), \pi(v_j)) \in \mathrm{E}_2} \|\ell_{e_1}(e) - \ell_{e_2}(\pi(e)\|.$$

Let $\boldsymbol{c} = [c_{vr},\ c_{vi},\ c_{vs},\ c_{er},\ c_{ei},\ c_{es}]^\top$ be the edit costs vector and $\boldsymbol{\omega} = [n_{vr}, n_{vi}, \omega_{vs}, n_{er}, n_{ei}, \omega_{es}]^\top$ the weight vector. The cost associated with π is given by

$$\mathrm{C}(\mathrm{G}_i, \mathrm{G}_j, \pi, \mathbf{c}) = \boldsymbol{\omega}^\top \mathbf{c}. \tag{5.1}$$

The GED between G_i and G_j is defined as

$$\mathtt{ged}(\mathrm{G}_i, \mathrm{G}_j, \mathbf{c}) = \underset{\pi}{\operatorname{argmin}}\, \mathrm{C}(\mathrm{G}_i, \mathrm{G}_j, \pi, \mathbf{c}). \tag{5.2}$$

In the next section, we propose a general graph pre-image framework, based on generative graph edit distance methods. Setting $\mathbf{c}$ by different strategies provides three variants of this framework. Particularly, we propose to optimize $\boldsymbol{\omega}$ and $\mathbf{c}$, such that the GED between each pair of graphs in $\mathbb{G}$ is as close as possible to its corresponding distance in $\mathscr{H}$.

5.3 Proposed graph pre-image framework

The main motivation of this work is to address the pre-image problem by taking advantage of GED's ability to construct graphs. As the edit costs of GEDs constitute core properties that depict the performance of GEDs, we propose three variant methods under this framework, following the different strategies to choose the edit costs of GEDs:

- The first variant takes the **random edit costs**, requiring no prior knowledge of the data. The non-negativity property is required to constraint the choice of the edit costs. Besides, one can also integrate a constraint to satisfy the triangular inequality, or one to ensure that a removal cost is equal to an insertion cost [Riesen, 2015].

- The second variant uses the **expert edit costs**, which often come with the datasets. Some empirical costs can also be applied for a set of datasets with common structures or properties, such as molecules with symbolic vertex labels [Abu-Aisheh et al., 2017].

- The third variant uses the **optimized edit costs**, which we will address in detail in the following.

We propose to construct the graph pre-image by GED, with edit costs optimized under the guidance of the feature extracted in the kernel space. To achieve this goal, we borrow the basic idea of metric learning by building connections between graph and kernel spaces under the assumption that the corresponding elements in similar space structures possess similar properties and features. We propose to align the metrics of the two spaces by optimizing these edit costs such that GEDs approximate the distances in kernel space. Then, once GEDs and kernel distances are similar, we propose to recast the pre-image problem as a graph generation problem. An iterative alternate minimization method is adapted for this purpose, in which GEDs with the optimized edit cost distances are used. These two steps are detailed next, and the proposed method is summarized in Algorithm 5.1.

5.3.1 Learn edit costs by distances in kernel space

When computing GEDs, the choice of edit costs values is essential. However, these constants are normally determined by domain experts for a given dataset, instead of being tuned automatically. With our original idea of aligning the GEDs to the kernel metric, we propose to learn edit costs by distances of elements in kernel space.

On one hand, based on the features of graph kernels introduced in Section 5.2, the distance in graph kernel space $\mathcal{H}$ between two elements $\phi(G_i)$ and $\phi(G_j)$ is

$$d_{\mathcal{H}}(\phi(G_i),\phi(G_j)) = \sqrt{k(G_i,G_i) + k(G_j,G_j) - 2k(G_i,G_j)}, \tag{5.3}$$

Algorithm 5.1 The graph pre-image method with cost learning

Input: Graph dataset $\mathcal{G}_N$, graph kernel k,
thresholds of stopping criteria (r_{max}, i_{max}).

Initialize randomly $\boldsymbol{c}^{(0)} = [c_{vr}^{(0)}, c_{vi}^{(0)}, c_{vs}^{(0)}, c_{er}^{(0)}, c_{ei}^{(0)}, c_{es}^{(0)}]^\top$.
Compute kernel distances $\boldsymbol{d}_{\mathcal{H}}$ of all pairs of graphs in $\mathcal{G}_N$ with (5.3).
Let $r = 0$.
while termination criteria not met (see Section 5.3.1) **do**
    Estimate $\mathbf{W}^{(r)}$ by solving (5.7) using a GED heuristic (e.g. `bipartite` or `IPFP`).
    Estimate $\boldsymbol{c}^{(r+1)}$ by solving (5.8) using quadratic programming (e.g. `CVXPY`).
    $r = r + 1$.
end while
Find set-median $\widehat{G}^{(0)}$ by (5.11).
Let $i = 0$.
while $i < i_{max}$ **do**
    Compute transformation $\widehat{\pi}_p^{(i+1)}$ by (5.12) for $\widehat{G}^{(i)}$ with $\boldsymbol{c}^{(r+1)}$.
    Generate $\widehat{G}^{(i+1)}$ by (5.13) with $\widehat{\pi}_p^{(i+1)}$ and $\boldsymbol{c}^{(r+1)}$.
    $i = i + 1$.
end while
$\widehat{G}^{(i+1)}$ is the approximated graph pre-image.

where $k(G_i, G_j)$ is a graph kernel between graphs G_i and G_j. On the other hand, when transformation π is settled in (5.2), the GED between graphs G_i and G_j can be formalized by (5.1), namely $\texttt{ged}(G_i, G_j) = \boldsymbol{\omega}^\top \boldsymbol{c}$ (See (5.1) and (5.2)).

A major difficulty, which is not straightforward from (5.1), is that the GED is not linear in $\boldsymbol{c}$, because the weight vector $\boldsymbol{\omega}$ also depends on the $\boldsymbol{c}$ (since the number of times to perform the edit operations depends on their costs). Therefore, for a given pair of graphs, G_i and G_j, estimating the optimal edit costs vector for a fixed value of the metric $\texttt{ged}(G_i, G_j)$ (e.g. $\approx d_{\mathcal{H}}(\phi(G_i), \phi(G_j))$ as in the following) requires an alternate optimization strategy over $\boldsymbol{c}$ and $\boldsymbol{\omega}$.

Given a graph space $\mathbb{G}$ of attributed graphs and a kernel space $\mathcal{H}$, we propose to align their metrics, namely the GED in $\mathbb{G}$ and the distance in $\mathcal{H}$ between each pair of available graphs. In other words, we seek to learn the edit costs of the GED, such that the GED between each pair of graphs in $\mathbb{G}$ is as close as possible to its corresponding distance in $\mathcal{H}$ (see Figure 5.3). To achieve this goal, a least-squares optimization on the available graph set $\mathcal{G}_N = \{G_1, G_2, \ldots, G_N\} \subset \mathbb{G}$ is considered, namely

$$\underset{\boldsymbol{c},\boldsymbol{\omega}}{\operatorname{argmin}} \sum_{i,j=1}^{N} \| \texttt{ged}(G_i, G_j) - d_{\mathcal{H}}(\phi(G_i), \phi(G_j)) \|^2, \tag{5.4}$$

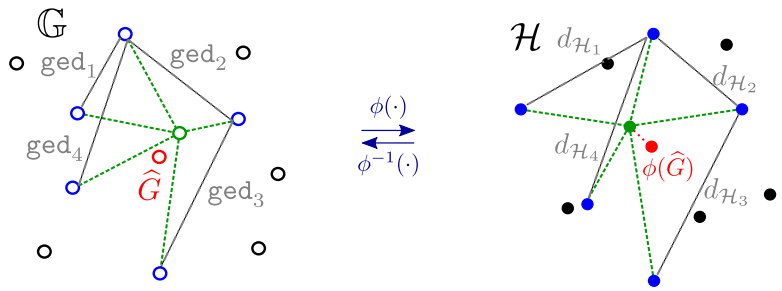

Figure 5.3 – Align GEDs in graph space $\mathbb{G}$ and distances in kernel space $\mathscr{H}$.

where ged is defined by the edit costs vector $\boldsymbol{c}$ and weight vector $\boldsymbol{\omega}$, as given in (5.1). Since an optimal $\boldsymbol{\omega}$ exists for each pair of graphs G_i and G_j in $\mathscr{G}_N$, we denote it by $\boldsymbol{\omega}(i,j)$. Moreover, to ensure that the minimum cost edit transformation π in (5.2) can be found, all edit costs need to be positive. The triangular inequality rule [Riesen, 2015] can also be affiliated, otherwise all substitutions will be replaced by a pair of deletion/insertion and the corresponding costs will be redundant [Brun et al., 2012]. With these constraints, the optimization problem becomes:

$$\begin{aligned} \underset{\boldsymbol{c},\mathbf{W}}{\text{argmin}} \quad & \|\mathbf{W}^\top \boldsymbol{c} - \boldsymbol{d}_{\mathscr{H}}\|^2 \\ \text{subject to} \quad & \boldsymbol{c} > \mathbf{0} \\ & c_{vr} + c_{vi} \geqslant c_{vs} \\ & c_{er} + c_{ei} \geqslant c_{es}, \end{aligned} \tag{5.5}$$

where $\mathbf{W}^\top$ is the N^2-by-6 matrix with rows $\boldsymbol{\omega}(i,j)^\top$ and $\boldsymbol{d}_{\mathscr{H}}$ the vector of N^2 entries $d_{\mathscr{H}}(\phi(G_i),\phi(G_j))$, for $i,j = 1,\ldots,N$, namely

$$\mathbf{W}^\top = \begin{bmatrix} \boldsymbol{\omega}(1,1)^\top \\ \vdots \\ \boldsymbol{\omega}(i,j)^\top \\ \vdots \\ \boldsymbol{\omega}(N,N)^\top \end{bmatrix}_{N^2\times 6} \quad \text{and} \quad \boldsymbol{d}_{\mathscr{H}} = \begin{bmatrix} d_{\mathscr{H}}(\phi(G_1),\, \phi(G_1)) \\ \vdots \\ d_{\mathscr{H}}(\phi(G_i),\, \phi(G_j)) \\ \vdots \\ d_{\mathscr{H}}(\phi(G_N),\, \phi(G_N)) \end{bmatrix}_{N^2\times 1}. \tag{5.6}$$

To solve this constrained optimization problem, we propose an alternative iterative optimization strategy over $\boldsymbol{c}$ and $\mathbf{W}$ (Lines 4 to 8 of Algorithm 5.1), after initializing $\mathbf{c}$ randomly or by expert costs:

- Step 1: for fixed $\mathbf{c}$, estimate $\mathbf{W}$ (line 5). The optimization problem (5.5) boils

down to

$$\underset{\mathbf{W}}{\operatorname{argmin}} \ \|\mathbf{W}^\top \boldsymbol{c} - \boldsymbol{d}_{\mathcal{H}}\|^2. \tag{5.7}$$

The weight matrix $\mathbf{W}$ is computed by any method computing graph edit distance. Heuristics to approximate GED can be used for the sake of computational complexity, such as the ones introduced in Chapter 4.

- Step 2: for fixed $\mathbf{W}$, estimate $\mathbf{c}$ (line 6). The constrained optimization problem (5.5) becomes

$$\begin{aligned} \underset{\boldsymbol{c}}{\operatorname{argmin}} \ & \|\mathbf{W}^\top \boldsymbol{c} - \boldsymbol{d}_{\mathcal{H}}\|^2 \\ \text{subject to} \ & \boldsymbol{c} > \mathbf{0} \\ & c_{vr} + c_{vi} \geqslant c_{vs} \\ & c_{er} + c_{ei} \geqslant c_{es}. \end{aligned} \tag{5.8}$$

  This optimization problem can be viewed as a constrained linear problem or a non-negative least squares problem [Lawson and Hanson, 1995]. Off-the-shelf solvers, such as `CVXPY` [Diamond and Boyd, 2016] and `scipy` [Virtanen et al., 2020] can be used to solve it. As a result, the difference between graph edit distances and distances between elements in the kernel space is minimized.

Termination criteria

We adapt the termination criteria in [Blumenthal et al., 2021]. Several termination criteria can be assigned to the aforementioned iterative process:

- Global convergence: the iteration stops if (5.4) is no longer improved, i.e., the difference between the sums in two adjacent iterations, namely the *residual*, is smaller than a threshold ϵ_g.
- Edit cost convergence: the iteration stops if the edit costs are no longer optimized, i.e., the difference between each optimized cost in two adjacent iterations is smaller than a threshold ϵ_{ec}.
- Maximum number of iterations: the iteration stops if a maximum number of iterations r_{max} is reached.
- Time limit: the iteration stops if a time limit t_{max} has been reached.

The global and edit cost convergences are always supposed to be met, which ensures that the local best edit costs are acquired. In this case, the alignment of the graph and kernel spaces can normally achieve good performance on given tasks. The choices of the thresholds ϵ_g and ϵ_{ec} play an important role in the convergence. On one hand, if the values are set too big, the iteration will stop quickly and the acquired edit costs may not be the best local; On the other hand, if they are set too small, a possible oscillation of the edit costs over iterations can be anticipated, which may lead to the failure of the convergence. The proper choices of the values of the thresholds depend on the absolute value of the residual and edit costs. In our experiments, ϵ_g and ϵ_{ec} are empirically set to 0.01, which is no more than one tenth of the corresponding absolute value (see Section 5.4.1 and Table 5.1 for more details).

The maximum number of iterations and the time limit thresholds, respectively r_{max} and t_{max}, are set in addition to achieve a trade-off between the quality of the optimized edit costs and time complexity. The design of the iteration procedure allows returning edit costs on the fly even before the convergence. Intuitively, the quality of the edit costs may not be optimized in this situation. However, the experiments show that the convergence can be achieved in 5 iterations in most cases for a well-designed problem, namely, there is no oscillation during the optimization. The time complexity of this amount of iterations is normally acceptable. To this end, we set r_{max} to 6 and t_{max} is unlimited in the experiment. The quality of the edit costs and the time complexity can be found in Section 5.4.3.

5.3.2 Generate graph pre-image

By aligning the distances in both graph and kernel spaces as we have done in Section 5.3.1, we propose to solve the pre-image problem by recasting it as a graph generation problem. Given a set $\mathcal{G}_\mathrm{N} \subset \mathbb{G}$ of N graphs and a graph kernel $k : \mathbb{G} \to \mathcal{H}$, the average of the set in the kernel space $\mathcal{H}$ is computed as $\psi = \sum_{i=1}^{\mathrm{N}} \alpha_i \phi(\mathrm{G}_i)$ with $\alpha_i = 1/\mathrm{N}$. The main objective is to estimate its pre-image, namely the graph $\widehat{\mathrm{G}}$ whose image $\phi(\widehat{\mathrm{G}})$ is as close as possible to ψ.

With the metric alignment principle thanks to (5.4), we get the relations $\mathtt{ged}(\mathrm{G}_i, \mathrm{G}_j) \approx d_{\mathcal{H}}(\phi(\mathrm{G}_i), \phi(\mathrm{G}_j))$, for all $\mathrm{G}_i, \mathrm{G}_j \in \mathcal{G}_\mathrm{N}$. Under the assumption that the corresponding elements in the similar space structures possess similar properties and features, estimating the pre-image $\widehat{\mathrm{G}}$ is equivalent to estimating the graph me-

dian, which can be tackled as the minimization of the sum of distances (SOD) from the graph median to all the graphs in $\mathcal{G}_N$, namely

$$\widehat{G} = \underset{G \in \mathbb{G}}{\operatorname{argmin}} \sum_{G_{p'} \in \mathcal{G}_N} \texttt{ged}(G, G_{p'}). \tag{5.9}$$

A first attempt to solve this problem is to restrict the solution to the set $\mathcal{G}_N$, namely

$$\widehat{G} = \underset{G_p \in \mathcal{G}_N}{\operatorname{argmin}} \sum_{G_{p'} \in \mathcal{G}_N} \texttt{ged}(G_p, G_{p'}). \tag{5.10}$$

By expanding the expression of the GED, we get

$$\begin{aligned}
\widehat{G} &= \underset{G_p \in \mathcal{G}_N}{\operatorname{argmin}} \sum_{G_{p'} \in \mathcal{G}_N} \texttt{ged}(G_p, G_{p'}) \\
&= \underset{G_p \in \mathcal{G}_N}{\operatorname{argmin}} \sum_{p'=1}^{N} \min_{\pi_{p'} \in \Pi(G_p, G_{p'})} c(\pi_{p'}, G_p, G_{p'}) \\
&= \underset{G_p \in \mathcal{G}_N}{\operatorname{argmin}} \sum_{p'=1}^{N} \min_{\pi_{p'} \in \Pi(G_p, G_{p'})} c_v(\pi_{p'}, \ell_{v_p}, \ell_{v_{p'}}) + \tfrac{1}{2} c_e(\pi_{p'}, A_p, \ell_{e_p}, A_{p'}, \ell_{e_{p'}}),
\end{aligned} \tag{5.11}$$

where cost $c(\pi_{p'}, G_p, G_{p'})$ consists of two parts, $c_v(\pi_{p'}, \ell_{v_p}, \ell_{v_{p'}})$ and $c_e(\pi_{p'}, A_p, \ell_{e_p}, A_{p'}, \ell_{e_{p'}})$, respectively the costs of vertex and edge transformations. This problem can be solved by computing GEDs between all pairs of graphs in $\mathcal{G}_N$. The time complexity is in $\mathcal{O}(aN^2)$, where a is the complexity of computing a GED between two graphs (for instance, by `bipartite` or `IPFP`). The resulting pre-image $\widehat{G}$, restricted to elements of the set $\mathcal{G}_N$, corresponds to the so-called set-median of $\mathcal{G}_N$ [Jiang et al., 2001].

Despite its simplicity, the set-median can only be chosen from the given dataset $\mathcal{G}_N$, which strongly limits the possibility of the pre-image. To obtain the pre-image from a bigger search space, we take advantage of recent advances in [Boria et al., 2019], where the proposed iterative alternate minimization procedure (IAM) allows generating new graphs. Next, we revisit this method and adapt it for the pre-image problem (lines 11 to 16 of Algorithm 5.1). Alternatively, other generative median graph algorithms can be also applied [Musmanno and Ribeiro, 2016, Ferrer et al., 2010, Bunke et al., 1999].

After initializing $\widehat{G}$ with the set-median, the proposed strategy alternates the optimization over all the $\widehat{\pi}_p$ (i.e., transformations from $\widehat{G}$ to G_p) and over the pre-image estimate $\widehat{G}$:

- Step 1: for fixed $\widehat{\mathrm{G}}$, estimate all $\widehat{\pi}_p$ by solving the following optimization problem:

$$\widehat{\pi}_p = \underset{\pi_p \in \Pi(\widehat{\mathrm{G}}, \mathrm{G}_p)}{\operatorname{argmin}} \; c(\pi_p, \widehat{\mathrm{G}}, \mathrm{G}_p) \qquad \forall p \in \{1, \ldots, \mathrm{N}\}. \tag{5.12}$$

  The resolution of (5.12) is carried out by solving the GED problem N times, between $\widehat{\mathrm{G}}$ and each $\mathrm{G}_p \in \mathscr{G}_\mathrm{N}$. Its time complexity is in $\mathscr{O}(a\mathrm{N})$, given $\mathscr{O}(a)$ the time complexity of evaluating GED between two graphs, such as with heuristics like `bipartite` or `IPFP`.

- Step 2: for fixed $\widehat{\pi}_p$, construct $\widehat{\mathrm{G}}$ by solving the following optimization problem:

$$\widehat{\mathrm{G}} = \underset{\substack{\varphi \in \mathscr{H}_v^{\widehat{n}} \\ \mathrm{A} \in \{0,1\}^{\widehat{n} \times \widehat{n}} \\ \Phi \in \mathscr{H}_e^{\widehat{n} \times \widehat{n}}}}{\operatorname{argmin}} \sum_{p=1}^{\mathrm{N}} c_v(\widehat{\pi}_p, \ell_v, \ell_{v_p}) + \tfrac{1}{2} c_e(\widehat{\pi}_p, \mathrm{A}, \ell_e, \mathrm{A}_p, \ell_{e_p}). \tag{5.13}$$

  In the computation of (5.13), the vertices and edges are updated separately. The new non-symbolic labels assigned for a vertex v or an edge e is given by the average values of the corresponding labels of the vertices substituted to v or the edges substituted to e, respectively. For each vertex v_i in $\widehat{\mathrm{G}}$, its label $\widehat{\ell}(v_i)$ is updated by

$$\widehat{\ell}(v_i) = \frac{1}{\sum_{p=1}^{\mathrm{N}} \delta_{\pi_p(v_i)}(\mathrm{V}_p)} \sum_{p=1}^{\mathrm{N}} \delta_{\pi_p(v_i)}(\mathrm{V}_p) \ell_p(\pi_p(v_i)), \tag{5.14}$$

  where the delta function $\delta_{\pi_p(v_i)}(\mathrm{V}_p) = 1$ if vertex $\pi_p(v_i) \in \mathrm{V}_p$ and 0 otherwise.

The iteration stops when no better pre-image can be generated. The obtained graph pre-image $\widehat{\mathrm{G}}$ can be viewed as the generalized median of $\mathscr{G}_\mathrm{N}$ [Jiang et al., 2001]. It is considered as the approximation of the pre-image in our method.

5.4 Experiments

In this section, we report experimental results. First, we introduce implementations applied for our algorithms. Then computational settings are presented. Finally, performances are exhibited and analyzed on the benchmark dataset *Letter*.

Table 5.1 – Parameter settings for experiments.

Parameters	Settings
Global threshold ϵ_g	0.01
Edit cost threshold ϵ_{ec}	0.01
Maximum number of iterations r_{max}	6
Time limit t_{max}	unlimited

5.4.1 Implementations and computational settings

To perform experiments, we implemented the proposed graph pre-image framework and Algorithm 5.1 in Python. To this end, the C++ library `GEDLIB` with its Python interface `gedlibpy` is used as the core implementation to compute graph edit distances and perform the IAM algorithm [Blumenthal et al., 2019].

In the original implementation of `GEDLIB`, a specific edit cost is implemented for each dataset. For example, an edit cost method `Letter` is designated for dataset *Letter*. However, this implementation uses an α coefficient to constrain relations between edit costs, and the constants for insertion and removal have to be equal. It limits the variability of edit costs and cannot fit into (5.5). The same problem arises for other graph datasets with non-symbolic labels. To burst these constraints, we implemented a general edit cost function `NonSymbolic` for graphs containing only non-symbolic vertex and/or edge labels. In this method, all edit costs can be freely set, which is more convenient for the optimization proposed in Section 5.3.1. We also implemented an edit cost method `Letter2` specifically for dataset *Letter* based on `NonSymbolic`. Library `gedlibpy` has been modified accordingly[1].

All experiments exhibited in this section are executed on a computer with 8 CPU cores of Intel(R) Core(TM) i7-7920HQ @ 3.10GHz, 32GB memory, and 64-bit operating system Ubuntu 16.04.3 LTS. Table 5.1 lists the parameter settings for experiments.

To estimate the graph edit distances, a multi-start counterpart of `IPFP`, namely m`IPFP` is applied in both procedures of producing the set-median and the gener-

[1]Links to implementations presented in Section 5.4.1 are listed below:
- `GEDLIB`: `https://github.com/dbblumenthal/gedlib`;
- `NonSymbolic` and `Letter2` methods:
`https://github.com/jajupmochi/gedlib/tree/master/src/edit_costs`;
- `gedlibpy` (modified): `https://github.com/jajupmochi/gedlibpy`;
- our pre-image algorithm:
`https://github.com/jajupmochi/graphkit-learn/tree/master/gklearn/preimage`.

alized median (see Section 5.3.2), where the number of solutions is set to $m = 40$ [Daller et al., 2018].

5.4.2 Experiments on real-world datasets

In this section, we evaluate our algorithm on *Letter* datasets[2]: *Letter-high*, *Letter-med*, and *Letter-low*. Each dataset consists of 2250 graphs representing distorted drawings of 15 capital letters of the Roman alphabet, where "*high*", "*med*", and "*low*" represent different distortions. Graph vertices are equipped with two non-symbolic labels "x" and "y" representing their positions. See Section 2.4 for more details.

The goal of this experiment is to compute, for a given kernel, the pre-image of the average of each class of letters, namely $\psi = \sum_{i=1}^{N} \alpha_i \phi(G_i)$ with $\alpha_i = 1/N$. A good estimator may be the median graph of the same set. Therefore, we propose to estimate the pre-image of ψ by using the median graph generated by our framework, using the random costs, the expert costs, and the optimized costs computed in Algorithm 5.1. Two graph kernels are applied in experiments, namely the shortest path kernel [Borgwardt and Kriegel, 2005] and the structural shortest path kernel [Ralaivola et al., 2005]. Both of them are able to deal with non-symbolic labels. See Section 3.3 for more details.

To exhibit and analyze the performance of proposed methods, we apply them on each class of each dataset, which corresponds to distortions of each letter at a given distortion level. In each class, all 150 graphs are chosen to compose the graph set $\mathcal{G}_N$.

5.4.3 Results and analyses

Designing evaluation measures of the performances of the generalized pre-image is an interesting topic. A straightforward method is to measure the distance between the generalized pre-image and the desired pre-image in the kernel space, which can be easily computed by (5.3). For datasets such as *Letter*, the visualization of each graph is possible, where the quality of the pre-image can be measured by the legibility. We analyze the performance of our methods with respect to both measures in the conducted experiments.

Table 5.2 exhibits results of our pre-image methods. First, we give results as a baseline of a method to generate median graphs (denoted "From median set"),

[2]Link: `http://graphkernels.cs.tu-dortmund.de`.

Table 5.2 – Running times and distances between the desired elements in kernel space and the mappings of the constructed pre-images computed using different methods for 2 graph kernels (shortest path and structural shortest path kernels).

Datasets	Graph Kernels	Algorithms	$d_{\mathcal{H}}$	Running Times (in seconds)		
				Optimization	Generation	Total
Letter-high	shortest path	From median set	**0.406**	-	-	-
		IAM (random costs)	0.467	-	142.59	142.59
		IAM (expert costs)	0.451	-	30.31	30.31
		IAM (optimized costs)	0.460	5968.92	26.55	5995.47
	structural sp	From median set	0.413	-	-	-
		IAM (random costs)	0.435	-	30.22	30.22
		IAM (expert costs)	**0.391**	-	29.71	29.71
		IAM (optimized costs)	**0.394**	24.79	25.60	50.39
Letter-med	shortest path	From median set	0.425	-	-	-
		IAM (random costs)	0.303	-	25.61	25.61
		IAM (expert costs)	**0.288**	-	26.93	26.93
		IAM (optimized costs)	**0.288**	23.72	24.79	48.52
	structural sp	From median set	0.380	-	-	-
		IAM (random costs)	0.286	-	24.77	24.77
		IAM (expert costs)	**0.248**	-	27.51	27.51
		IAM (optimized costs)	**0.248**	27.06	29.24	56.30
Letter-low	shortest path	From median set	0.139	-	-	-
		IAM (random costs)	**0.116**	-	26.47	26.47
		IAM (expert costs)	**0.116**	-	24.87	24.87
		IAM (optimized costs)	**0.116**	26.35	29.97	56.31
	structural sp	From median set	0.112	-	-	-
		IAM (random costs)	0.103	-	30.22	30.22
		IAM (expert costs)	**0.086**	-	29.43	29.43
		IAM (optimized costs)	0.104	21.95	24.59	46.53

where the median graph is directly chosen from the median set $\mathcal{G}_{\mathrm{N}}$ whose representation in kernel space is the closest to the true median's (ψ). After that, results of two sets of edit costs are presented, corresponding to the first two variants of the proposed framework. The first set of constants is randomly generated for each class of graphs, while the second set is given by domain experts, where $c_{vi} = c_{vr} = 0.675, c_{ei} = c_{er} = 0.425, c_{vs} = 0.75$ and $c_{es} = 0$ [Blumenthal et al., 2020]. It is worth noting that these expert values take into account prior knowledge of the data, such as setting c_{es} to 0 as graphs in *Letters* do not contain edge labels. Finally, results of the third variant of the proposed framework are presented, computed with the optimized costs using Algorithm 5.1. All three variants are denoted "IAM", as they are essentially Iterative Alternate Minimization (IAM) procedures, and they are distinguished from each other by edit costs. The average results over all classes are presented for all methods. Column "$d_{\mathcal{H}}$" gives the distances between the embed-

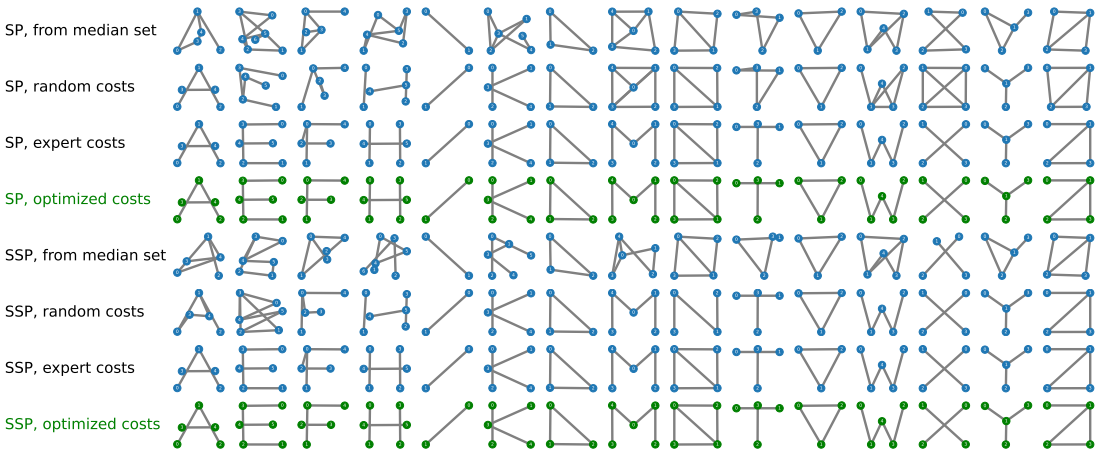

Figure 5.4 – Pre-images constructed by different algorithms for *Letter-high* with shortest path (SP) and structural shortest path (SSP) kernels.

ding of the computed pre-image and the element we want to approximate in the kernel space ($d_{\mathcal{H}}$). A lower distance indicates a better estimation of the pre-image. Columns "Running Times" give the time used to optimize edit costs and generate pre-images.

On all three datasets, almost all proposed methods applying our framework provide better pre-images than choosing from the median set with respect to the $d_{\mathcal{H}}$s of the generalized medians, no matter what edit costs are chosen. On *Letter-high* for the structural SP kernel, compared to $d_{\mathcal{H}}$ of pre-image choosing from median sets, $d_{\mathcal{H}}$ is respectively 5.33% and 4.60% smaller for the algorithm with expert and optimized costs. Moreover, $d_{\mathcal{H}}$ of the algorithm with optimized costs is 9.43% smaller than that with random costs and is almost the same as the algorithm with expert costs, which is also the case for the SP kernel. The advantage of the proposed framework is even more significant on *Letter-med* and *Letter-low*, where they dominate the performances. These results show that the algorithm with the optimized costs works better than the one with random costs to generate pre-images as median graphs, and can serve as a method to tune edit costs to help find expert costs, for both median generation problems using IAM and graph pre-image problems. Moreover, the running times to optimize edit costs and generate pre-images are acceptable in most cases.

Besides these improvements, the advantage of our methods can be evaluated from other aspects. Figure 5.4 presents the pre-images generated as the median graphs for each letter of the *Letter-high* dataset using the aforementioned methods,

which correspond to the first eight rows of Table 5.2, row by row. Vertices are drawn according to coordinates determined by their attributes "x" and "y". In this way, plots of graphs are able to display the letters that they represent, which are possible to be recognized by human eyes. When using the shortest path (SP) kernel (the first row to the fourth row), it can be seen that the pre-images chosen directly from the median set (the first row) are illegible in almost all cases, while the IAM with random costs provides more legible results, where letters A, K, Y can be easily recognized (the second row). When the expert and optimized costs are used, almost all letters are readable, despite that the pre-images of letter F are slightly different (the third and fourth rows). The same conclusion can be derived for the structure shortest path kernel as well (the fifth row to the eighth row).

This analysis indicates that even though the distances $d_{\mathcal{H}}$ are similar, the proposed pre-image algorithms are able to generate better pre-images, especially when edit costs are optimized. This phenomenon may benefit from the nature of the IAM algorithm. In the update procedure (5.13), the new non-symbolic label assigned for a vertex v is given by the average values of the corresponding labels of the vertices substituted to v [Boria et al., 2019]. It provides a "direction" to construct pre-images with respect to the features and structures of graphs. For instance, the "x" and "y" attributes on the vertices of the letter graphs represent the coordinates of the vertices. To this end, it makes sense to compute their average values as the new values of a vertex as the vertex will be re-positioned at the middle of all vertices substituted to it.

5.5 Conclusion and future work

In this chapter, we proposed a novel framework to estimate graph pre-images by taking advantage of GED's ability to construct graphs. Three variants were proposed, given the random, expert, and optimized costs. This last approach is based on the hypothesis that metrics in both kernel space and graph space can be aligned. We first proposed a method to align GEDs to distances in the kernel space. Within the procedure, the edit costs are optimized. Then the graph pre-image was generated by a new method to construct the graph generalized median, where we revisited the IAM algorithm. Our framework can generate better pre-images than other methods, as demonstrated on the *Letter* datasets.

Future work will focus on several aspects. First, establishing a convergence

proof of the optimization procedure is interesting. Second, our methods can be generalized to graphs with symbolic labels and to the construction of pre-images as arbitrary graphs rather than median graphs by improving both the IAM algorithm and the distance alignments. Third, consider non-constant costs to establish a more flexible scheme. Fourth, other criteria such as the conformal map will be examined to build connections between GEDs to distances in kernel space [Honeine and Richard, 2011].

Chapter 6

`graphkit-learn`: a Python library for graph machine learning

Contents

6.1 Overview

In previous chapters, we explored graph kernels, graph edit distance, and the graph pre-image problem, which constitute important and intriguing parts in the field of machine learning on graphs. As Python becomes a useful and popular platform-independent programming language in the machine learning community, it has included the implementation of a wide range of machine learning tools [Van Rossum and Drake Jr, 1995, Van Rossum and Drake, 2009]. To study the aforementioned problems on graphs and take advantage of the advances of Python, it is necessary to implement with this language the tools to deal with these problems as well.

Although several open-source libraries to solve these problems have been published, they all suffer from some shortcomings or are incompatible with our requirements, such as the lack of certain methods and the choice of a different programming language. In this chapter, we present a new open-source Python library named `graphkit-learn`, implementing methods to solve all three aforementioned problems on graphs. The library is publicly available to the community on GitHub:

`https://github.com/jajupmochi/graphkit-learn`,

and can be installed by `pip`:

`pip install graphkit-learn`.

The detailed description is carried out in each corresponding section in the following part of this chapter.

Through the implementation, the following underlying technologies are applied:

- `NumPy` is a fundamental package for scientific computing with Python, which offers numerous linear algebra operations [Harris et al., 2020].
- `SciPy` provides many user-friendly and efficient numerical routines, such as routines for numerical integration, interpolation, optimization, linear algebra, and statistics [Virtanen et al., 2020].
- `NetworkX` is for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks [Hagberg et al., 2008]. It is used to model graphs in `graphkit-learn`.
- `scikit-learn` is a machine learning tool for predictive data analysis, such as regression and classification problems [Pedregosa et al., 2011].

Technologies that are applied for specific modules in `graphkit-learn` will be introduced in the corresponding sections of this chapter.

The remainder of this chapter is organized as follows: The overall architecture of the library is first summarized in Section 6.2, then the graph data processing methods are introduced in Section 6.3. After that, the implementations of graph kernels, graph edit distances, and graph pre-image methods are respectively detailed in Sections 6.4, 6.5, and 6.6. Next, Section 6.7 categorizes the auxiliary functions. Finally, Section 6.8 concludes the chapter.

6.2 The overall architecture

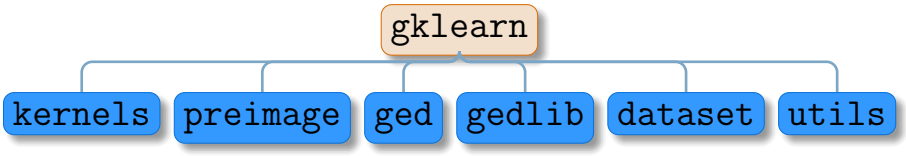

Figure 6.1 – The overall architecture of the `graphkit-learn` library.

As Figure 6.1 shows, `gklearn` is the module name of the `graphkit-learn` library, which is in total composed of six modules:

- The `kernels` module implements 9 graph kernels based on linear patterns and 2 on non-linear patterns examined in Chapter 3.
- The `preimage` module implements 2 graph pre-image methods including the one proposed in Chapter 5.
- The `ged` module implements the environment to deal with GED heuristics. The `LSAPE-GED` paradigm and the subordinate `bipartite` heuristic are implemented. A framework to define edit costs and a median graph estimator based on GED are provided.
- The `gedlib` module integrates the Python interface `gedlibpy` of the C++ library `GEDLIB` for GED computation [Blumenthal et al., 2019]. `GEDLIB` is modified for the integration. Upgrades are made on both `gedlibpy` and `GEDLIB` to provide more plentiful functions.
- The `dataset` module handles graph datasets, including fetching graph datasets from several online databases, loading graphs from various formats, and processing them.

- The `utils` collects auxiliary tools, such as graph manipulation tools, kernels between labels, parallelization tools, and the methods to perform model selection with cross-validation.

The following sections describe these contents in detail.

6.3 Graph data processing

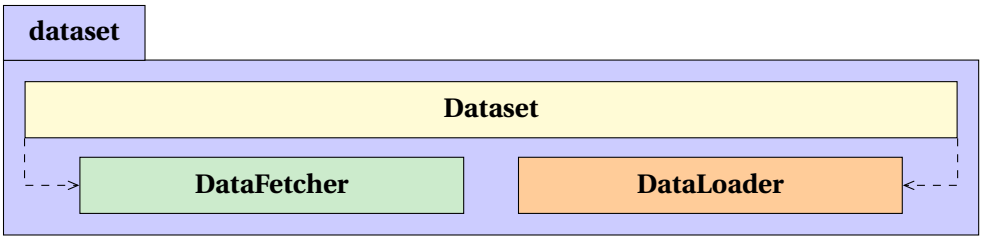

Figure 6.2 – Graph data processing module shown in a UML class diagram.

We first introduce the `dataset` module, which deals with graph data processing. A graph in `graphkit-learn` is represented by a `networkx.Graph` class if undirected or a `networkx.DiGraph` class if directed. Names of vertex and edge labels can be extracted at the same time. As shown in Figure 6.2, `dataset` includes mainly three classes:

`DataFetcher` is able to fetch graph data from databases publicly available online. `DataFetcher` supports several widely-used graph dataset formats, including the Graph eXchange Language file (GXL) [Winter et al., 2002], the structure data file (SDF)[1], and *TUDataset* format. For now, three benchmark databases can be fetched:

- *GREYC's Chemistry database*[2]: contains various chemical datasets of small molecules, each of which concerns either a classification or a regression problem.
- *TUDataset*[3]: collects benchmark datasets for the evaluation of graph kernels, which come from various domains such as bioinformatics, chemoinformatics, computer vision, social networks, and synthetic graphs [Morris et al., 2020]. New datasets are still being added to this database and `DataFetcher` is designed to be able to fetch them.

[1] See `http://www.gupro.de/GXL/Introduction/background.html`.
[2] Available at `https://brunl01.users.greyc.fr/CHEMISTRY`.
[3] Available at `www.graphlearning.io`.

Dataset	
.....	
+ set_labels()	// Sets the names of labels that will be processed.
+ get_dataset_infos()	// Extracts properties of the dataset, such as its size, the average vertex number and edge number, the average vertex degree, whether the graphs are directed, labeled symbolically and non-symbolically on their vertices and edges, etc.
+ print_dataset_infos()	// Prints out the properties of the dataset.
+ remove_labels()	// Removes irrelevant labels from the graphs.
+ cut_graphs()	// Cuts graphs into a given range.
+ trim_dataset()	// Removes graphs with no vertex or no edges.
...	

Figure 6.3 – User interfaces of `Dataset`.

- *IAM*[4]: consists of several graph datasets from chemoinformatics, image recognition, and social network domains. As most datasets in *IAM* are also included in *TUDataset*, we only fetch two datasets from this database, namely *Web* and *GREC*.

After `DataFetcher` obtains a dataset online and saves it locally, `DataLoader` is able to read graphs from the dataset file and transforms them to `NetworkX` graphs [Hagberg et al., 2008]. `NetworkX` supports rather comprehensive graph attributes, including symbolic and non-symbolic labels on vertices and edges, edge weights, directness, etc.

Taking advantage of `DataFetcher` and `DataLoader`, class `Dataset` can fetch and load graph dataset by a single Python statement, as given in line 3 of the following example:

```
1 from gklearn.dataset import Dataset
2
3 ds = Dataset('Alkane', # The input.
4     root='') # The root path to save the dataset.
5 graphs = ds.graphs
6 y = ds.targets
```

The input of `Dataset` can be a pre-defined dataset name, the path to the dataset files, or a list of `NetworkX` graphs. The variables `Dataset.graphs` and

[4] Available at `http://www.iam.unibe.ch/fki/databases/iam-graph-database`.

`Dataset.targets` are the loaded graphs and the corresponding targets. Moreover, `Dataset` provides a set of member functions to facilitate the manipulation of the dataset, which is shown in Figure 6.3.

6.4 Implementations of graph kernels

6.4.1 State of the art and motivation

Graph kernels have become a powerful tool in bridging the gap between machine learning and graph representations. Of particular interest are graph kernels based on linear patterns. Despite that non-linear patterns may encode more complex structural information than linear ones, the latter require much lower computational complexity on many occasions. Nevertheless, non-linear patterns normally include or imply linear ones. For example, the treelet pattern is non-linear as a whole, while treelets whose maximal size is less than 4 are linear (see Section 3.5.1). Thus, the linear patterns are the cornerstone of most graph kernels. In practice, it is intractable to compute non-linear or cyclic-based kernels on large graphs. Experiments conducted on kernels based on linear patterns in Section 3.7 demonstrate their relevance compared to kernels based on non-linear kernels. Moreover, they have been serving as a baseline for designing new kernels. These kernels have been constructed using either walk or path patterns, as detailed in Chapter 3.

Several open-source libraries have been published and are available online to compute graph kernels; however, only parts of the aforementioned kernels have been implemented so far and only limited types of graphs were tackled. Table 6.1 exhibits these libraries and shows the graph kernels each library implemented. Of particular interest is the `GraKel` library, which provides an object-oriented framework for graph kernel implementation, thus intriguing many followers [Siglidis et al., 2020]. As a result, it is essential to clarify the reasons that we incline to publish a new library rather than add implementations of new kernels into existing libraries (e.g. `GraKeL`). We would like to explain our motivations from the following aspects:

First, as a result of the advantages of graph kernels based on linear patterns, `graphkit-learn` **focuses mainly on graph kernels based on linear patterns**. Table 6.1 shows that our library is the only Python library available online which implements all graph kernels based on linear patterns. For example, `GraKeL` imple-

Table 6.1 – Available libraries implementing graph kernels based on linear patterns.

	Libraries	GraKeL[a]	pykernels[b]	ChemoKernel[c]	graphkernels[d]	graph-kernels[e]	graphkit-learn[f] (**our work**)
Kernels implemented	**Common walk**	✓	✗	✗	✓	✓	✓
	Marginalized	✗	✗	✓	✗	✗	✓
	Sylvester equation	✗	✗	✗	✗	✗	✓
	Conjugate gradient	✗	✓	✗	✗	✗	✓
	Fixed-point iterations	✗	✗	✗	✗	✗	✓
	Spectral decomposition	✗	✗	✗	✗	✗	✓
	Shortest path	✓	✓	✗	✓	✗	✓
	Structural shortest path	✗	✗	✗	✗	✗	✓
	Path up to length h	✗	✗	✓	✗	✗	✓
	Languages	Python	Python	C++	Python (C++ core)	C++, R (C++ core)	Python

a: `https://github.com/ysig/GraKeL`
b: `https://github.com/gmum/pykernels`
c: `https://github.com/bgauzere/ChemoKernel`
d: `https://github.com/BorgwardtLab/GraphKernels`
e: `https://github.com/BorgwardtLab/graph-kernels`
f: `https://github.com/jajupmochi/graphkit-learn`

mented only two kernels based on linear patterns, namely the common walk kernel and the shortest path kernel. These implementations provide a set of baseline benchmarks for graph kernels, which not only can be easily used for comparisons with newly-created kernels, but more importantly, can be computed in reasonable time in most cases.

Second, `graphkit-learn` **implements a more flexible and ease-of-use framework,** where symbolic and/or non-symbolic vertex and edge labels, as well as other parameters for kernel computation, can be taken into consideration by simply setting arguments of Python functions. As a result, all the graph kernels can be simply computed with a single Python statement. More importantly, this framework makes it possible to use simultaneously symbolic and non-symbolic labels in graph kernels, which enables graph kernels to tackle more types of graph datasets. Other Python libraries cannot do this. For example, `GraKeL` uses two different classes for the random walk kernel, `RandomWalkLabeled` for labeled graphs and `RandomWalk` for unlabeled graphs, and two different classes for the shortest path kernel, `ShortestPath` for symbolic labels and `ShortestPathAttr` for non-symbolic labels.

Third, `graphkit-learn` **implements three methods to accelerate the compu-**

Table 6.2 – Comparison of the implementations of graph kernels.

Kernels	Labeling				Directed	Edge weighted	Weighting	Implementations
	symbolic		non-symbolic					
	vertices	edges	vertices	edges				
Common walk	✓	✓	✗	✗	✓	✗	a priori	`CommonWalk`
Marginalized	✓	✓	✗	✗	✓	✗	✗	`Marginalized`
Sylvester equation	✗	✗	✗	✗	✓	✓	a priori	`SylvesterEquation`
Conjugate gradient	✓	✓	✓	✓	✓	✓	a priori	`ConjugateGradient`
Fixed-point iterations	✓	✓	✓	✓	✓	✓	a priori	`FixedPoint`
Spectral decomposition	✗	✗	✗	✗	✓	✓	a priori	`SpectralDecomposition`
Shortest path	✓	✗	✓	✗	✓	✓	✗	`ShortestPath`
Structural shortest path	✓	✓	✓	✓	✓	✗	✗	`StructuralSP`
Path kernel up to length h	✓	✓	✗	✗	✓	✗	✓	`PathUpToH`
Treelet	✓	✓	✗	✗	✓	✗	✓	`Treelet`
WL subtree	✓	✗	✗	✗	✓	✗	✗	`WLSubtree`

"Weighting" indicates whether the substructures can be weighted in order to obtain a similarity measure adapted to a particular prediction task.

tation of graph kernels, namely parallelization, Fast Computation of Shortest Path Kernel (FCSP) method, and the trie structure. Moreover, we have extended some of these methods to other graph kernels. Description and analyses of these methods are conducted in detail in Section 3.6.

Finally, `graphkit-learn` **integrates more machine learning tools for graphs,** such as graph edit distance methods and tools to solve the graph pre-image problems, which are closely related to graph kernels and depend on their implementations. This is the main reason the library is named "`graphkit-learn`", rather than names such as "`graphkernel-learn`".

In conclusion, even though libraries such as `GraKeL` are great contributions, we believe it is necessary to implement a new library that addresses the cornerstone graph kernels (i.e., kernels based on linear patterns), with a focus on improving the computational complexity and providing further contributions in graph edit distance and graph pre-image methods.

6.4.2 Implementation details of graph kernels

In this section, we detail the implementations of these kernels, which are collected under the `kernels` module. For comparison, two graph kernels based on non-linear patterns are also implemented, namely the Weisfeiler-Lehman (WL) subtree kernel [Shervashidze et al., 2011, Morris et al., 2017] and the treelet kernel [Gaüzère et al., 2015b, Bougleux et al., 2012, Gaüzère et al., 2012]. Our implementations provide the ability to address various types of graphs, including unlabeled graphs, vertex-labeled graphs, edge-labeled, and fully-labeled graphs, directed and

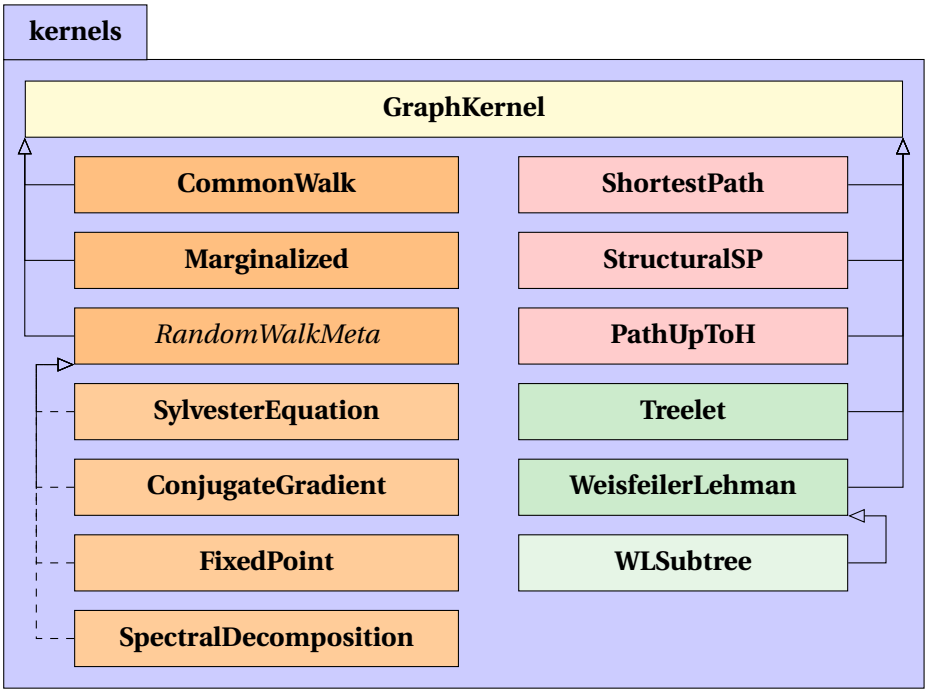

Figure 6.4 – Implemented graph kernels shown in a UML class diagram.

undirected graphs, and edge-weighted graphs. Only parts of these types have been tackled by other available libraries. Table 6.2 shows the types of graphs that each kernel can process.

Figure 6.4 shows the implementations of the graph kernels in a UML class diagram. Each graph kernel is implemented by a Python class inherited from the basic class `GraphKernel`. The corresponding relation between graph kernels and the Python class is presented in Table 6.2 (first and last columns).

Each kernel class is initialized with the parameters required to specify the computing methods and to serve as the tunable hyper-parameters of the kernel, according to the definition of the kernel (e.g., the maximum number of iterations of the marginalized kernel and the maximum path length of the path kernel up to length h). Meanwhile, the names of vertex and edge labels should also be provided if they exist. The computation of the graph kernel is then carried out by the member function `compute`, which takes any of the following three types of inputs:

- A list of `NetworkX` graph objects. In this case, a Gram matrix is returned whose entries are the evaluations of the kernel on pairs of graphs from the list. The parallelization is carried out over pairs of graphs if applied.

- A `NetworkX` graph object G_0 and a list of `NetworkX` graph objects $\mathcal{G}_N$. In this case, a list is returned where each entry is the graph kernel between G_0 and an entry in $\mathcal{G}_N$. The parallelization is carried out over $\mathcal{G}_N$ if applied.
- Two `NetworkX` graph objects. In this case, a real value is returned representing the graph kernel between these two graphs. No parallelization scheme is applied under this situation.

Auxiliary arguments can be provided to `compute`, namely:

- `parallel`: sets the parallelization scheme. If set to `imap_unordered`, then the parallelization is carried out by the `imap_unordered` method of Python's `multiprocessing.Pool` module; If set to `None`, then no parallelization scheme is applied. Its default value is `imap_unordered`.
- `n_jobs`: sets the number of jobs if parallelization is applied. Its default value is the number of all available CPU cores.
- `normalize`: decides whether to normalize the Gram matrix after computation. Its default value is `True`.

The specific implementation of each graph kernel is described next.

`CommonWalk` implements the common walk kernel [Gärtner et al., 2003]. Two computing methods are provided based on exponential series and geometric series as introduced by [Gärtner et al., 2003]. The direct product of labeled graphs is implemented for the convenience of computation.

`Marginalized` computes the marginalized kernel with the recursion algorithm [Kashima et al., 2003]. The users can set the argument `remove_totters=True` to remove tottering with the method introduced by [Mahé et al., 2004].

Four graph kernels derived from the generalized random walk kernel are implemented. The classes `SylvesterEquation`, `ConjugateGradient`, `FixedPoint`, and `SpectralDecomposition` compute respectively the Sylvester equation kernel, the conjugate gradient kernel, the fixed-point iterations kernel, and the spectral decomposition kernel, as introduced in [Vishwanathan et al., 2010]. For the conjugate gradient kernel and the fixed-point iterations kernel, the labels of vertices at the two ends of an edge are added to both sides of the corresponding edge labels.

The shortest path kernel is computed by `ShortestPath`. The Floyd-Warshall's algorithm [Floyd, 1962] is employed to transform the original graphs into shortest-paths graphs [Borgwardt and Kriegel, 2005], where the parallelization over graphs can be switched on. `StructuralSP` computes the structural shortest path kernel

introduced by [Ralaivola et al., 2005]. The shortest paths in each graph are found out a priori to reduce running time, where the parallelization over graphs can be applied. The *Fast Computation of Shortest Path Kernel* (FCSP) method is applied for both kernels. See Section 3.6.1 for details.

`PathUpToH` computes the path kernel up to length h. Two normalization kernels can be chosen, the Tanimoto kernel and the MinMax kernel, studied by [Suard et al., 2007]. The paths in each graph are extracted a priori for the sake of running time, where the parallelization over graphs can be applied. By default, the trie data structure is applied to store paths; nevertheless, it is recommended to choose the trie data structure according to h and structure properties of graphs (see Section 3.6.3).

Two graph kernels based on non-linear patterns are implemented. `Treelet` tackles the treelet kernel [Gaüzère et al., 2015b, Gaüzère et al., 2012]. A set recording the number of the occurrence of each treelet is constructed a priori, where parallelization can be applied over graphs. The `WLSubtree` class deals with the Weisfeiler-Lehman (WL) subtree kernel, which is derived from the `WeisfeilerLehman` class for the Weisfeiler-Lehman framework [Shervashidze et al., 2011]. When parallelization is applied, `WLSubtree` carries out operations on labels for each graph every time a kernel between a pair of graphs is evaluated; otherwise, the simultaneous operations on labels presented in Algorithm 3.1 is implemented.

Besides, user-defined vertex kernels and/or edge kernels of labeled graphs are supported in the shortest path kernels, the structural shortest path kernel, the conjugate gradient kernel, and the fixed-point iterations kernel. These kernels allow using simultaneously symbolic and non-symbolic labels, which enables graph kernels to tackle more types of graph datasets. The module `utils` contains several predefined kernels between labels of vertices or edges, which is detailed in Section 6.7. Moreover, edge weights can be included in the shortest path kernel, the structural shortest path kernel, the Sylvester equation kernel, and the spectral decomposition kernel.

6.4.3 Usage example

The following piece of code shows an example to compute the path kernel up to length h:

```
1 from gklearn.kernels import PathUpToH
2
```

```
gram_matrix, run_time = PathUpToH(
    node_labels=node_labels, # The list of node label names.
    edge_labels=edge_labels, # The list of edge label names.
    ds_infos={'directed': True}, # Dataset information required
   for computation.
    depth=3, # The longest length of paths.
    k_func='MinMax', # Or 'tanimoto'.
    compute_method='trie' # Or 'naive'.
    ).compute(graphs, # The list of graphs.
    parallel='imap_unordered', # Or None.
    n_jobs=1, # The number of jobs to run in parallel.
    normalize=True, # Whether to normalize the Gram matrix.
    verbose=True, # Whether to print out results.
    )
```

The Gram matrix and the time spent to compute it are returned.

6.5 Implementations of graph edit distance

6.5.1 State of the art and motivation

The graph edit distance (GED), as a natural and flexible dissimilarity measure between graphs, has been widely studied and used. However, it suffers from high computational complexity. To overcome this issue, various heuristics to estimate GED have been proposed over the years [Blumenthal et al., 2020]. To facilitate the use of these heuristics, several open-source libraries have been implemented and are available online. Table 6.3 lists the state-of-the-art ones, including:

- `GEDLIB`: a C++ library for (suboptimally) computing graph edit distances using various state-of-the-art methods [Blumenthal et al., 2019]. This library implements an integrated framework for GED computation, including an environment which encodes the information required to compute GED and user interfaces, the implementations of the complete list of GED heuristics examined in [Blumenthal et al., 2020], varied edit costs designated to different datasets, and a median graph estimator based on the extension of the method proposed in [Boria et al., 2019]. `GEDLIB` is one of the best-implemented libraries for GED.
- `gedlibpy`: a Python interface of the `GEDLIB` library implemented by Cython [Behnel et al., 2011]. Most of the functions provided by `GEDLIB` can be used in

Table 6.3 – Libraries available online implementing GEDs.

Libraries	Languages	Links
GEDLIB	C++	https://github.com/dbblumenthal/gedlib
gedlibpy	Python (C++ core)	https://github.com/Ryurin/gedlibpy
ged-toolbox	MATLAB	https://github.com/bgauzere/ged-toolbox
graph-lib	C++	https://github.com/bgauzere/graph-lib

Python through this interface.

- ged-toolbox: a MATLAB library implementing several GED heuristics based on bipartite matching and quadratic assignment [Gaüzère et al., 2014, Bougleux et al., 2015b].
- graph-lib: A C++ library implementing GED heuristics, such as bipartite and IPFP with their multi-start counterparts [Gaüzère et al., 2014, Bougleux et al., 2015b, Gaüzère et al., 2016, Daller et al., 2018].

None of these libraries are written in pure Python; therefore, they may cause compatibility problems when used by our library. To deal with these problems, we implement GED heuristics in graphkit-learn with two different strategies. In the ged module, a GED framework written in pure Python is provided, allowing it to take full advantage of the Python language and other parts of the library, while easy to modify and extend. Moreover, an extended compatible version of the gedlibpy is integrated into the gedlib module, providing the full state-of-the-art functions of GEDLIB, allowing speedup of computation thanks to the use of C++. These two strategies are detailed in the following sections.

6.5.2 The ged module

The overall architecture of the ged module is shown in Figure 6.5, which follows the one in the GEDLIB library. Four modules are included in ged: env, methods, edit_costs, and median.

The env module

The main class in env is GEDEnv, which deals with GED informations and provides general interfaces to compute GED. Figure 6.6 displays the major interfaces of GEDEnv, inlcuding:

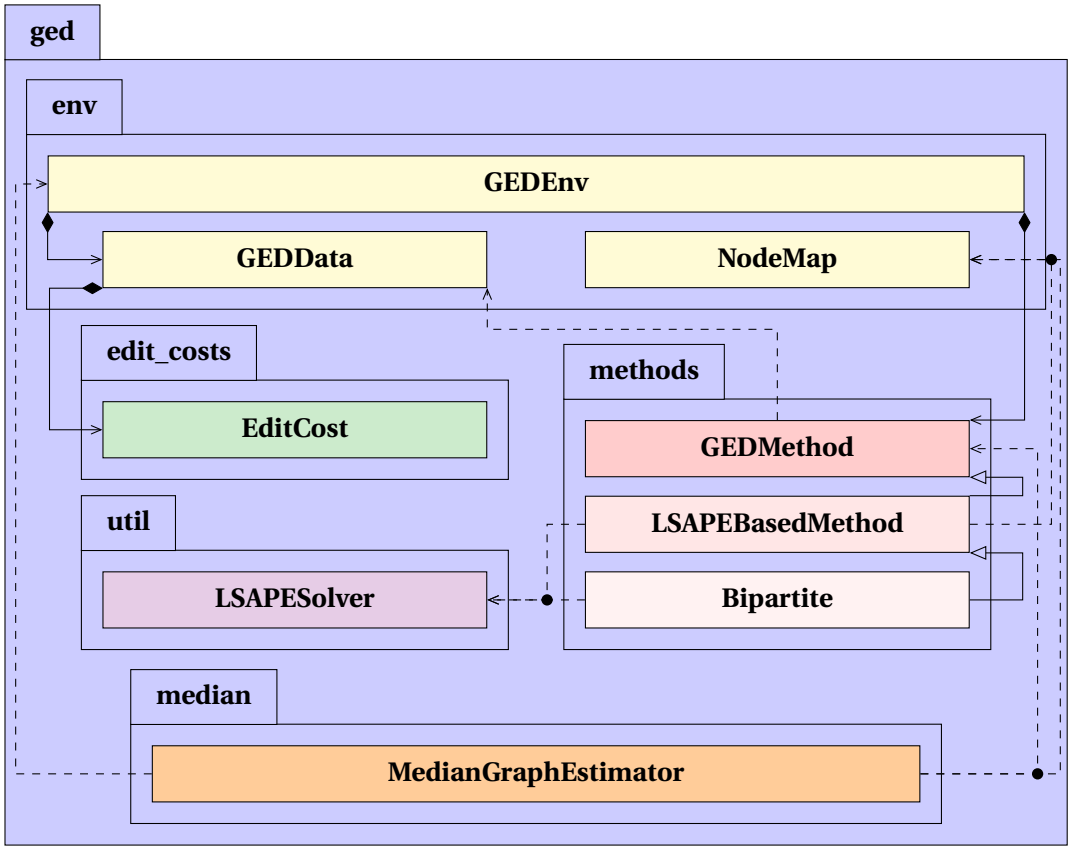

Figure 6.5 – Implemented graph edit distance tools shown in a UML class diagram.

- `add_nx_graph()`: adds a `NetworkX` graph into the environment.
- `set_edit_cost()`: adds edit cost functions to the environment, where the predefined cost functions can be selected.
- `init()`: initializes the environment.
- `set_method()`: selects a GED heuristic predefined in the `methods` module and sets parameters.
- `init_method`: initializes the selected GED heuristic.
- `run_method`: runs the GED heuristic between two specified graphs.

The results of the heuristics can be accessed by the getter member functions of `GEDEnv`.

Auxiliary classes are designed for the use of `GEDEnv`. `GEDData` provides interfaces to manipulate the data that is used for GED computation, such as the graphs and the edit costs, where the latter is acquired by invoking the `EditCost` class.

GEDEnv		
.....		
+ add_nx_graph() + set_method() ...	+ set_edit_cost() + init_method()	+ init() + run_method()

Figure 6.6 – The user interface of `GEDEnv`.

`NodeMap` models the vertex to vertex transformation given by an edit path between two graphs.

The `methods` module

The `methods` module contains heuristics to approximate GED. All heuristics are derived from the `GEDMethod` class, which provides a general interface for GED computation. `GEDMethod` initializes a GED heuristic with given parameters, runs the method, and returns the results.

The `LSAPE-GED` paradigm, based on the linear sum assignment problem with error-correction (see Section 4.2.1), is implemented by the `LSAPEBasedMethod` class, with the full capacity to realize any heuristic derived from it. The `LSAPESolver` class in the `ged.util` module provides the interface to solve the LSAPE problem. Notice that owing to the lack of LSAPE solver in Python, the `linear_sum_assignment` function in the `optimize` module of the `SciPy` library is actually used, which solves the linear sum assignment problem (LSAP). The heuristic `bipartite` is implemented by the `Bipartite` class, which is a derived class of `LSAPEBasedMethod`.

The `edit_costs` module

The `edit_costs` module includes implementation of varied edit cost functions. The `EditCost` class provides the interfaces to implement these functions and is used in the `GEDData` class. Figure 6.7 displays its main interfaces, including:

- `node_ins_cost_fun()`: computes the cost of inserting a vertex with a given label.
- `node_del_cost_fun()`: computes the cost of deleting a vertex with a given label.

EditCost		
.....		
+ node_ins_cost_fun() + edge_ins_cost_fun() ...	+ node_del_cost_fun() + edge_del_cost_fun()	+ node_rel_cost_fun() + edge_rel_cost_fun()

Figure 6.7 – The user interface of `EditCost`.

- `node_rel_cost_fun()`: computes the cost of relabeling a vertex with a given label by another label.
- `edge_ins_cost_fun()`: computes the cost of inserting an edge with a given label.
- `edge_del_cost_fun()`: computes the cost of deleting an edge with a given label.
- `edge_rel_cost_fun()`: computes the cost of relabeling an edge with a given label by another label.

The constant cost functions are implemented in the `Constant` class, which can be used for any data.

The `median` module

The `median` module contains the class `MedianGraphEstimator`, which estimates the generalized median graph of a given graph set by the extent of the algorithm proposed in [Boria et al., 2019]. After initialization and parameter settings, the algorithm can be executed by the member function `run()`. Then the results can be collected by the getter member functions. Classes `GEDEnv` and `NodeMap` are needed. Compared to [Boria et al., 2019], `MedianGraphEstimator` allows the change of the number of vertices while updating the median graph, which is consistent with the implementation in the `GEDLIB` library.

In the implementation of graph pre-image methods, `MedianGraphEstimator` is used to perform the graph generation function. See Section 6.6.

6.5.3 The `gedlib` module

The `gedlib` module is adapted from the `gedlibpy` library, which provides interfaces to the C++ library `GEDLIB`. The interfaces are written in `Cython` [Behnel et al., 2011] and the C++ code is encapsulated into a dynamic library file that comes along with `graphkit-learn`. Comparing to `gedlibpy`, the `gedlib` module differs in the following three aspects:

First, to keep consistent with the interfaces in the `ged` module, a Python class named `GEDEnv` is defined to hold all functions derived from the `GEDLIB` library. The member functions of `GEDEnv` are redesigned to match their counterparts in the `ged` module. As a result, the same code to approximate GEDs can be used to invoke both Python and C++ implementations by simply importing the proper module.

Second, interfaces in `gedlib` are enriched, making it possible to implement more tasks, such as the estimation of generalized median graphs. Principal additional member functions are exhibited in Figure 6.8.

gedlib.GEDEnv	
.....	
+ get_induced_cost()	// Returns the induced cost between the two indicated graphs.
+ compute_induced_cost()	// Computes the edit cost between two graphs induced by a node map.
+ get_median_node_label()	// Computes median node label.
+ get_median_edge_label()	// Computes median edge label.
+ get_nx_graph()	// Get graph with a given id in the form of the `NetworkX` Graph.
+ get_node_cost()	// Returns the edit cost between two node labels.
...	

Figure 6.8 – The additional user interfaces of `gedlib.GEDEnv`.

Third, two edit costs are designed and added to the `GEDLIB` library, namely

- `NonSymbolic` is designed for graphs containing only non-symbolic vertex and/or edge labels, where all edit costs can be set freely.
- `Letter2` is designed for the `Letter` dataset, which is a special case of `NonSymbolic`.

The Python interfaces are revised accordingly.

6.5.4 Usage example

The following piece of code shows an example to compute the GED between two graphs with the `GED` module, where `bipartite` is used:

```
1  from gklearn.ged.env import GEDEnv
2
3  ged_env = GEDEnv() # Initailize GED environment.
4  ged_env.set_edit_cost('CONSTANT', # Edit cost function.
5      edit_cost_constants=[3, 3, 1, 3, 3, 1] # The constant edit
       costs.
6  ged_env.add_nx_graph(graph1, '') # Add graph1.
7  ged_env.add_nx_graph(graph2, '') # add graph2.
8  listID = ged_env.get_all_graph_ids() # Get list IDs of graphs.
9  ged_env.init() # Initialize the GED environment.
10 ged_env.set_method('BIPARTITE'), # Set GED method.
11 ged_env.init_method() # Initialize the GED method.
12 ged_env.run_method(listID[0], listID[1]) # Run method.
13
14 pi_forward = ged_env.get_forward_map(listID[0], listID[1]) # Get
       the forward map.
15 pi_backward = ged_env.get_backward_map(listID[0], listID[1]) #
       Get the backward map.
16 dis = ged_env.get_upper_bound(listID[0], listID[1]) # Get the GED
       between two graphs.
```

In this example, `pi_forward` consists of mapping each vertex in `graph1` to a vertex in `graph2` according to the edit path, while `pi_backward` is the reverse mapping from `graph2` to `graph1`. Finally, `dis` is the evaluated GED.

This piece of code can be directly used to invoke the `gedlib` module to compute GEDs, as long as replacing the import statement (line 1) by:

```
1  from gklearn.gedlib import librariesImport
2  from gklearn.gedlib.gedlibpy import GEDEnv
```

6.6 Implementations of graph pre-image methods

The graph pre-image methods are implemented in the `preimage` module. To our knowledge, there is no attempt of realizing these methods in other libraries publicly available. Two generative models, the random iterative model [Bakır et al., 2004] and the median pre-image model [Jia et al., 2021] are respectively implemented

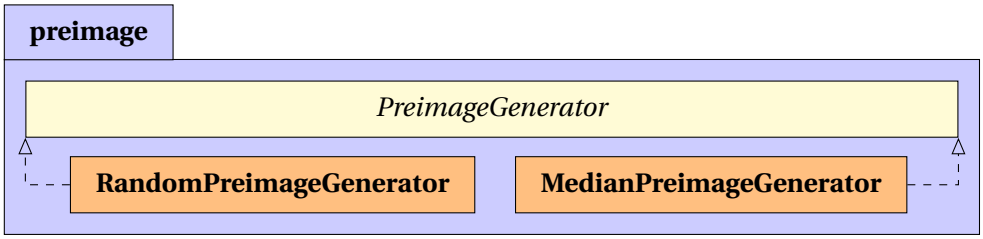

Figure 6.9 – Implemented pre-image methods shown in a UML class diagram.

by classes `RandomPreimageGenerator` and `MedianPreimageGenerator` (see Chapter 5 for detail). These two classes are derived from the abstract class `PreimageGenerator`, which defines the basic getter and setter interfaces for the generator.

Both generators provide two main interfaces, namely the member functions `set_options()` and `run()`. The former one chooses the parameters used to construct the graph pre-image. For `RandomPreimageGenerator`, it sets the graph kernel method and setting used to construct the kernel space, as well as the parameters controlling the iterative procedure, such as the maximum number of iterations and number of random graphs generated in each iteration. For `MedianPreimageGenerator`, it determines the parameters for the graph kernel, the graph edit distance, the iterative procedure, and the termination criteria used to construct the graph pre-image. Instances of these settings are exhibited in the example codes at the end of this section.

`MedianPreimageGenerator` takes advantage of classes `GraphKernel`, `GEDEnv` and `MedianGraphEstimator` in the aforementioned modules. The number of vertices can be optimized through the update procedure. The following codes display an example of generating graph pre-image by `MedianPreimageGenerator`. Parameters are first determined.

```
### Set parameters.

# Parameters for MedianPreimageGenerator (our method).
mpg_options = {'fit_method': 'k-graphs', # How to fit edit costs.
    "k-graphs" means use all graphs in the median set while
   fitting.
          'init_ecc': [4, 4, 2, 1, 1, 1], # Initialize edit costs.
          'ds_name': ds_name, # Name of the dataset.
          'parallel': True, # Whether the parallel scheme is to be
    used.
          'time_limit_in_sec': 0, # Maximum time limit to compute
```

```
    the pre-image. If set to 0 then there is no limit.
9         'max_itrs': 100, # Maximum iteration limit to optimize
    edit costs. If set to 0 then there is no limit.
10        'max_itrs_without_update': 3, # If the time that edit
    costs are not updated is more than this number, then the
    optimization stops.
11        'epsilon_residual': 0.01, # In optimization, the
    residual is only considered changed if the change is bigger
    than this number.
12        'epsilon_ec': 0.1, # In optimization, the edit costs are
     only considered changed if the changes are bigger than this
    number.
13        'verbose': 2 # Whether or not to print out results.
14              }
15
16 # Parameters for graph kernel computation.
17 kernel_options = {'name': 'PathUpToH', # Use path kernel up to
    length h.
18         'normalize': True # Whether to use a normalized Gram
    matrix to optimize edit costs.
19                }
20
21 # Parameters for GED computation.
22 ged_options = {'method': 'IPFP', # Use IPFP huristic.
23         'initialization_method': 'RANDOM', # or 'NODE', etc.
24         'initial_solutions': 10, # When bigger than 1 after
    multiplied by "ratio_runs_from_initial_solutions", then the
    method is considered mIPFP.
25         'edit_cost': 'CONSTANT', # Use CONSTANT cost.
26         'attr_distance': 'euclidean', # The distance between non
    -symbolic node/edge labels is computed by euclidean distance.
27         'ratio_runs_from_initial_solutions': 1,
28         'threads': 1 # parallel threads. Do not work if
    mpg_options['parallel'] = False.
29              }
30
31 # Parameters for MedianGraphEstimator (Boria's method).
32 mge_options = {'init_type': 'MEDOID', # How to initial median (
    compute set-median). "MEDOID" is to use the graph with the
    smallest SOD.
33         'random_inits': 10, # Number of random initializations
    when 'init_type' == 'RANDOM'.
```

```
        'time_limit': 600, # Maximum time limit to compute the generalized median. If set to 0 then no limit.
        'verbose': 2, # Whether to print out results.
        'refine': False # whether to refine the final SODs or not.
            }
```

We exhibit as many parameters as possible to facilitate users to understand their meanings and exhibit the flexibility of our implementation. In practice, most of them are equipped with default values or can be computed automatically. After these settings, the generation process can be carried out as follows:

```
### Run median preimage generator.

from gklearn.preimage import MedianPreimageGenerator
# Create a median preimage generator instance.
mpg = MedianPreimageGenerator()
# Add dataset.
mpg.dataset = dataset
# Set parameters.
mpg.set_options(**mpg_options.copy())
mpg.kernel_options = kernel_options.copy()
mpg.ged_options = ged_options.copy()
mpg.mge_options = mge_options.copy()
# Run.
mpg.run()
# Get results.
graph = mpg.gen_median
```

6.7 Auxiliary tools

To facilitate the use of the aforementioned modules, we implement a set of auxiliary tools in the `utils` module.

The first group of tools consists of graph manipulation functions applied in the graph kernels, GEDs, and graph pre-images, including:

- `getSPLengths` computes the shortest paths between each pair of vertices in a graph.
- `getSPGraph` transforms a given graph to its corresponding shortest-paths graph by Floyd transformation [Floyd, 1962].

- `get_shortest_paths` returns all shortest paths in a graph.
- `untotterTransformation` transforms a given graph according to [Mahé et al., 2004] to filter out tottering patterns for the marginalized kernel.
- `direct_product_graph` returns the direct / tensor product graph of two given directed graphs [Gärtner et al., 2003].
- `compute_gram_matrices_by_class` computes the Gram matrix of each set of graphs belonging to the same class in a given dataset and graph kernel.
- `find_paths` finds all paths no longer than a certain length that start from a given source node in a graph. A recursive depth-first search is applied.
- `find_all_paths` finds all paths no longer than a certain length in a given graph. A recursive depth-first search is applied.
- `compute_distance_matrix` converts a Gram matrix to a distance matrix.

The second group of tools computes kernels between labels, all included in the sub-module `kernels`. Among them, functions `deltakernel` computes the Kronecker delta function between symbolic labels, while `gaussiankernel`, `polynomialkernel`, and `linearkernel` compute respectively the Gaussian kernel, the polynomial kernel, and the linear kernel between non-symbolic labels. Functions `kernelsum` and `kernelproduct` are respectively the sum and product of kernels between symbolic and non-symbolic labels.

The third group provides concise wrappers of parallelization methods using Python's `Pool` module, especially for graph kernel computations. The `parallel` module envelopes these wrappers.

In the fourth group, a complete model selection and evaluation procedure is implemented in the module `model_selection_precomputed`, for the convenience of use. All work in it is carried out by function `model_selection_for_precomputed_kernel`. This function first pre-processes the input dataset, then computes Gram matrices and performs the model evaluation with machine learning methods from the `scikit-learn` library [Pedregosa et al., 2011]. Support Vector Machines (SVM) are applied for classification tasks and kernel ridge regression for regression [Schölkopf and Smola, 2002].

As shown in Figure 6.10, a two-layer nested cross-validation (CV) is applied to select and evaluate models, where the outer CV randomly splits the dataset into 10

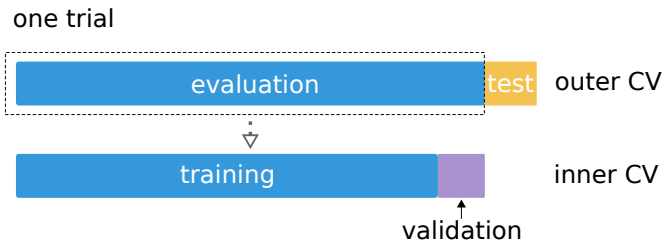

Figure 6.10 – Exhibition of a two-layer cross validation (re-presenting Figure 3.17).

folds with 9 as evaluation set, and the inner CV then randomly splits the evaluation set to 10 folds with 9 as the training set and 1 as the validation set. The hyper-parameters are optimized on the evaluation set by grid search. The whole procedure is performed 30 times, and the average performance is computed over these trials. The kernel parameters are tuned within this procedure. This design allows the users to perform model selection in a single Python statement. The following code shows a demo to use this function:

```
from gklearn.utils import model_selection_for_precomputed_kernel
from gklearn.kernels import untilhpathkernel
import numpy as np

# Set parameters.
datafile = 'DATA_FOLDER/MUTAG/MUTAG_A.txt'
param_grid_precomputed = {
    'depth': np.linspace(1, 10, 10),
    'k_func': ['MinMax', 'tanimoto'],
    'compute_method': ['trie']}
param_grid = {'C': np.logspace(-10, 10, num=41, base=10)}

# Perform model selection and classification.
model_selection_for_precomputed_kernel(
    datafile, # The path of dataset file.
    untilhpathkernel, # The graph kernel used for estimation.
    param_grid_precomputed, # The parameters used to compute gram
    matrices.
    param_grid, # The penalty parameters used for penalty items.
    'classification', # Or 'regression'.
    NUM_TRIALS=30, # The number of the random trials of the outer
    CV loop.
    ds_name='MUTAG', # The name of the dataset.
    n_jobs=1,
    verbose=True)
```

The "`param_grid_precomputed`" and the "`param_grid`" arguments specify grids of hyper-parameter values used for grid search in the cross-validation procedure. The results are automatically saved.

More demos and examples can be found in the `notebooks` directory and the `gklearn.examples` module of the library.

6.8 Conclusion and Future Work

In this chapter, we presented the Python library `graphkit-learn` for machine learning methods on graphs. It is the first library that provides a thorough coverage of graph kernels based on linear patterns (9 kernels based on linear patterns and 2 on non-linear patterns for comparison). The first attempt to implement a well-structured collection of graph edit distance computing methods was provided, with state-of-the-art paradigms and heuristics. Graph pre-image methods are included, building the foundation for solving this problem. Graph dataset processing and other auxiliary functions extend the ability of the library.

Future work includes implementations of other non-linear kernels, a more thorough test of graph kernels on a wider range of benchmark datasets, a C++ implementation bound to Python interface for faster computation. Moreover, more graph edit distance methods and tools to solve the graph pre-image problem will be included. Furthermore, we will integrate more machine learning tools for graphs into the library, such as graph neural networks applied for kernels, GEDs, and pre-image problems. Finally, we encourage interested users and authors of graph machine learning problems to commit their implementations to the library.

Chapter 7

Conclusions and future work

Contents

7.1 Conclusion

In this thesis, we explored machine learning methods on graphs, focusing on graph kernels, graph edit distances, and the graph pre-image problem. On one hand, we provided thorough examination and improvements on graph kernels based on linear patterns. On the other hand, we studied the stability property of graph edit distances (GEDs) and proposed metric learning algorithms to optimize GEDs for regression problems. Based on these advances, we proposed a new strategy to deal with the pre-image problem on graphs. We detail each work in the following.

- In Chapter 3, we provided a survey on graph kernels, with an emphasis on the ones based on linear patterns and two kernels based on non-linear patterns for comparison. The theoretical foundations of these kernels were well-examined, including their mathematical expression, their computational complexities, their strengths and weaknesses, types of graphs that they can be used on, relationships between themselves, as well as their connections with other classic and state-of-the-art kernels from literature. Moreover, experiments were conducted on various types of graphs, both synthesized and real-world ones, where the prediction performance and the time complexity of each graph kernel were analyzed and compared. Experiments showed that graph kernels based on linear patterns, such as the path kernel up to length h, can achieve a comparable performance with the ones based on non-linear patterns. As a conclusion, we gave the recommendation of the choice of the proper kernels according to the properties of the given graph data. This work provides a baseline for the analyses of graph kernels and cornerstones to develop new kernels.

  To better utilize these kernels, we proposed three strategies to reduce their computational complexity and memory usage. The first one was the FCSP method, where the sub-kernels between vertices and edges were computed a priori. The second one was parallelization, which was conducted between pairs of graphs and the cross-validation procedure. The third one was the use of the trie structure, which stores the path sub-patterns to reduce memory occupation. Conducted experiments proved the ability of these methods to reduce the time and space complexity of graph kernels.

- In Chapter 4, the graph edit distance, a dissimilarity measure between graphs,

was examined. Based on the revisiting of two paradigms `LSAPE-GED` (resp. `LS-GED`) and two representative heuristics lying in these paradigms, i.e., `bipartite` (resp. `IPFP`), we studied the stability of GED heuristics, which is the first time in literature. We found out empirically the effects of two factors on the stability, namely the choice of edit costs and the repeated time to compute the GED. Experiments were carried out with `IPFP` on several datasets, showing that the stability of `IPFP` decreases as the ratio between vertex and edge edit costs increases. The same phenomenon was observed when the parameter ratio is replaced by the repeated time. As these factors may affect the stability and further the task performances of the GED, they should be carefully taken care of.

Moreover, we developed a metric learning algorithm to optimize the GED for regression problems. The edit costs were tuned by aligning GEDs and distances between targets. With the distance-preserving principle, we proposed an alternate iterative procedure, where two steps are carried out alternately: the update of edit costs obtained by solving a constrained linear problem; and a re-computation of the optimal edit paths according to the newly computed costs were performed alternately. The optimized edit costs were analyzed to have an insight into the structure of the graph space with respect to the targets. Experiments showed that the optimized costs produced significant improvement on the prediction performance, where a k-nearest neighbors regression was performed.

- Inspired by this metric learning procedure, we proposed a new strategy to tackle the pre-image problem on graphs in Chapter 5. To construct an approximation of a graph pre-image, we first aligned the graph space to the kernel space by minimizing the differences between distances in both spaces. The distances in graph space can be measured by graph edit distances, while the distances in kernel space were computed by a graph kernel. A linear optimization problem was established for the minimization, where the edit costs were optimized according to the distances in kernel space. To this end, the aforementioned two-step alternate iterative optimization procedure over edit costs and edit paths was used, with the distances in kernel space replacing the distances between targets. After that, using the optimized edit costs, a graph generative algorithm was used for the generation of the graph pre-image. Specif-

ically, we proposed a method to generate the desired graph pre-image, by adapting a recently introduced method for the construction of the generalized median graph, where an iterative alternate minimization procedure to generate median graphs is applied. The conducted experiments demonstrated that our method outperforms the pre-image from the dataset and the random edit costs, and was competitive with the expert costs. Our method allows to automatically construct good graph pre-images even without expert edit costs, which is a solid foundation for the future study of the graph pre-image problem.

- To facilitate the use and examination of the aforementioned work and carried out experiments, we released the open-source Python library `graphkit-learn` online, which was described in detail in Chapter 6. This library includes machine learning tools on graphs, mainly graph kernels, graph edit distances, and graph pre-image methods. To support these tools, a graph dataset fetching, loading, and processing module is included. Auxiliary tools are provided, such as the ones for the model selection and evaluation process.

7.2 Future work

Plenty of intriguing future work can be conducted for the graph pre-image problem, beginning with the work on graph kernels and graph edit distance methods used for that problem:

- The most interesting work for graph kernels includes **investigating the relation between kernels based on linear patterns and the state-of-the-art kernels, especially the ones based on non-linear patterns.** Novel kernels are developed ceaselessly with respect to various strategies and sub-patterns. Besides the *treelet* kernel and the *WL* kernel examined in this thesis, widely-known sub-patterns include subtree patterns, cyclic patterns, quantum walks, etc. Experiments in Chapter 3 have exhibited promising performance of some kernels based on linear patterns, so how much advantage did these newly-developed kernels gain? By answering this question, we look forward to digging into the properties of different types of sub-patterns, and revealing the necessity of applying these patterns when facing varied graph types

and tasks, considering the trade-off between performance and computational complexity. Meanwhile, graph kernels constructed by strategies other than sub-patterns have been investigated as well, such as the ones based on propagation, message passing, and deep frameworks. Integrating kernels based on linear patterns into these strategies constitutes another intriguing research direction. These strategies and sub-patterns have been well studied in several recent survey papers [Borgwardt et al., 2020, Kriege et al., 2020, Ghosh et al., 2018], demonstrating that graph kernels are still of great interest to the community. A deeper investigation of the taxonomy proposed in these surveys will help better understand the strengths and weaknesses of each type of kernels and that of graph kernels as a whole, compared to other graph machine learning methods. Promising future development of new graph kernels can be anticipated derived from these researches.

- Another future research topic will **concentrate on the computational efficiency and scalability of graph kernels**. On one side, we will explore the feasibility of improving our methods to reduce the computational complexity proposed in Section 3.6, and of adapting them onto state-of-the-art graph kernels. On the other side, many methods have been proposed over the decades to lower the computational complexity of graph kernels, such as graph kernels based on iterative label refinement [Shervashidze and Borgwardt, 2009] and the use of hashing functions [Morris et al., 2016, Hido and Kashima, 2009]. We would like to take advantage of these strategies to generalize them into graphs of various types while enhancing their efficiency. Underlying principles of these strategies, such as hashing, can be borrowed to improve our methods.

- **The future work for the GED** consists of a thorough study of the stability of heuristics. For the edit cost optimization method, the convergence proof will be conducted. We are also looking forward to extending this scheme to classification problems and non-constant edit costs. To achieve these goals, a more sophisticated cost updating procedure is required, where the parameters to optimize include the edit costs between each pair of labels on vertices and edges. This work has the potential to capture information brought by labels. Recent advances on learning edit costs, such as genetic algorithms [Garcia-Hernandez et al., 2020] and neural networks [Martineau et al., 2020], can be applied for this procedure. Criteria other than distance-preserving can

be considered for optimization as well, such as the conformal map that can be used to align the graph space and target space by preserving inner product measures, thus leading to a redesigning of the edit cost updating procedure.

- **The future work for the graph pre-image problem will be carried out in three folds.** First, for the current methods, the convergence proof will be conducted. Meanwhile, the generalization of the proposed method can be included, such as on graphs with symbolic labels, on the construction of user-defined pre-images, and on the non-constant edit costs. In general, a unified framework can be constructed for the graph pre-image problem based on metric learning with tuned edit costs. Any cost learning method can be embraced by this framework, such as the ones introduced in Section 4.4.1. Second, inquiring into the distance measures on graphs and parameter updating criteria can help to catch their influence on the produced pre-images. Examples of these distances include the chemical distance and the Chartrand-Kubiki-Shultz (CKS) distance, as introduced in Section 1.2.2. Meanwhile, the aforementioned conformal map is a promising candidate for the space-aligning criteria. Third, it is worth exploring the advantages of many emerging tools, such as knowledge graphs and graph neural networks, to tackle the graph pre-image problem, where the former can integrate prior expertise from a model-driven perspective, while the latter capitalizes on neural networks employed on graphs.

- All the aforementioned work was on kernel framework; while the graph neural networks (GNNs) have sprung up in recent years and have achieved state-of-the-art performance on many tasks [Wu et al., 2020, Zhang et al., 2020, Zhou et al., 2018a]. GNNs provide an elegant framework to bridge the gap between graphs and machine learning methods, which have a strong background connection to the Weisfeiler-Lehman propagation [Morris et al., 2019, Xu et al., 2018]. Their properties have recently been increasingly investigated [Balcilar et al., 2021]. We would like to **revisit our work within the GNN framework**, in the sense of examining the similarities and differences, pros and cons, and performances between GNN and graph kernel frameworks.

- GNNs that endow the ability to construct graphs, namely **generative GNNs, provide new perspectives to solve the graph pre-image problem**. For instance, a graph autoencoder model takes a graph as input and maps it into

a feature space with a probabilistic encoder model, then a probabilistic decoder model is trained to sample realistic graphs, where the last step can be regarded as a pre-image procedure [Simonovsky and Komodakis, 2018, Grover et al., 2019, Kipf and Welling, 2016]. Other approaches, such as autoregressive generative models [Liao et al., 2019, You et al., 2018] and generative adversarial networks (GANs) [De Cao and Kipf, 2018, Wang et al., 2018], can generate new samples, which would be useful in the generative procedure when designing graph pre-image methods.

- To reinforce these work, **enriching the** `graphkit-learn` **library** is necessary. Among those include the implementations of state-of-the-art graph kernels, graph edit distances, and pre-image algorithms. The integration with other tools will also be considered, such as graph neural networks applied for these problems.

Part III

Appendix

Annexe A

Synthèse de la thèse en français

Contents

Ce chapitre présente un résumé de cette thèse en français. Premièrement, nous présentons le contexte général de ce travail. Ensuite, les contributions principales portant sur la distance d'édition, les noyaux sur graphes et le problème de calcul de la pré-image sont présentées. Enfin, nous concluons sur les contributions apportées ainsi que sur les perspectives ouvertes par ces travaux.

A.1 Contexte de la représentation graphique

Les graphes permettent de modéliser un large éventail de données du monde réel. Constitués de sommets et d'arêtes les reliant, les graphes sont capables d'encoder des éléments ainsi que leurs relations, ce qui permet de saisir les informations structurelles sous-jacentes des données. En outre, chaque sommet et chaque arête peut être doté d'attributs discrets et/ou continus, ce qui permet de représenter une grande variété d'informations pour décrire les données.

Grâce à la complexité et à l'expressivité de la structure des graphes, de nombreuses applications dans divers domaines utilisent cette représentation. En chémoinformatique et en bioinformatique, les molécules, y compris les biomacromolécules, peuvent être naturellement représentées par des graphes [Trinajstic, 2018, Huber et al., 2007]. Un sommet est associé à un atome, et est étiqueté par le type d'atome et parfois d'autres attributs tels que sa position dans l'espace, tandis qu'une arête peut représenter la connexion entre les deux atomes, généralement étiquetée par la valence. Des applications biologiques et biomédicales peuvent alors se baser sur ces représentations, comme pour la conception et la découverte de médicaments [Vamathevan et al., 2019], l'analyse des effets biologiques et chimiques, ainsi que l'identification biométrique [Kisku et al., 2011]. Dans l'analyse des réseaux sociaux, de grands graphes sont utilisés pour modéliser ces réseaux et les interactions entre utilisateurs [Scott, 2011, Wasserman et al., 1994]. Par exemple, un profil d'utilisateur peut être codé par un sommet étiqueté par ses informations détaillées, et la relation entre les profils d'utilisateurs peut être codée par des arêtes. Cette représentation peut ensuite être utilisée pour des applications telles que l'exploration de données sur le web [Russell, 2013, Rettinger et al., 2012] et la publicité [Guo et al., 2020]. En vision par ordinateur, les images 2D et 3D peuvent être représentées par des graphes. Dans les applications 2D, une application courante est la reconnaissance de l'écriture manuscrite, où les lignes tracées sont représentées par des arêtes et les points de connexion entre lignes

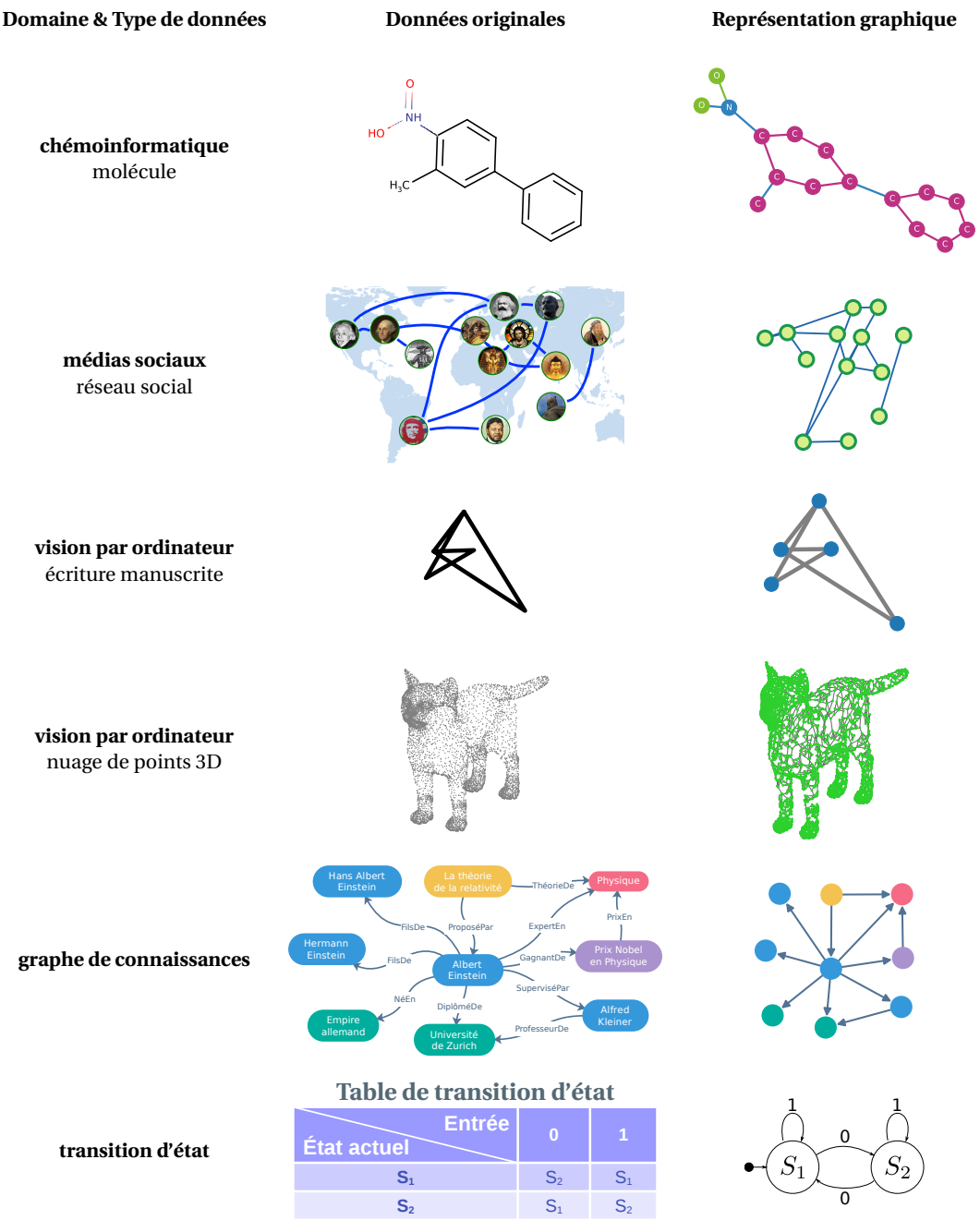

FIGURE A.1 – Exemples de données et leur représentation graphique.

sont représentés par des sommets étiquetés par leurs positions dans le plan [Riesen and Bunke, 2008]. Cette représentation permet de représenter directement l'information. Une stratégie similaire peut être utilisée pour des applications telles que la reconnaissance d'objets [Nene et al., 1996]. D'autres approches pour modéliser une image comme un graphe incluent l'encodage de chaque pixel par un

sommet et les arêtes par la connexion aux pixels adjacents [Wang, 2015]. En considérant des données 3D, un nuage de points peut être représenté intuitivement par un graphe, chaque point 3D étant un sommet [de Oliveira Rente et al., 2018]. Le graphe de connaissances, ainsi que son dérivé sous forme d'ontologie, constituent un autre domaine reposant sur des représentations sous forme de graphes [Ji et al., 2020]. Comme son nom l'indique, un graphe de connaissances est un graphe construit par des informations provenant d'un ensemble de connaissances du domaine. Un grand nombre d'applications entrent dans cette catégorie, comme la recherche d'information [Reinanda et al., 2020], le traitement automatique du langage naturel [Nastase et al., 2015], le Web sémantique [Ding et al., 2007], et la modélisation des processus de l'industrie [Cao et al., 2018]. D'autres applications des graphes incluent la transition d'état [Conte et al., 2004], les graphes temporels [Michail, 2016], etc. La modélisation et les applications inter-domaines ont été largement explorées. FIGURE A.1 présente plusieurs exemples de données sous forme de graphes dans différents domaines. La molécule provient du jeu de données *MUTAG* [Debnath et al., 1991] ; les images utilisées pour les avatars du réseau social proviennent de Wikipedia[1] ; la lettre manuscrite "A" provient du jeu de données *Letterhigh* [Riesen and Bunke, 2008] ; le nuage de points est généré à partir d'un modèle de chat tridimensionnel à l'aide de la bibliothèque `Open3D`[2] [Zhou et al., 2018b] ; le graphe de connaissances est construit sur les données de [Ji et al., 2020].

Toutes ces applications sont fondées sur les réponses à deux questions fondamentales pour la modélisation sous forme de graphe :

- **Question 1 : Comment acquérir des caractéristiques et propriétés intrinsèques des jeux de données de graphes ?**
- **Question 2 : Comment construire des graphes dotés des caractéristiques et propriétés souhaitées ?**

Par exemple, lorsqu'une molécule de médicament est représentée sous forme de graphe, l'analyse de l'influence d'une sous-structure sur sa fonction (par exemple, sa capacité anticancéreuse) relève de la question 1, tandis que la question 2 peut impliquer la conception de nouveaux médicaments pour une propriété donnée. Dans cette thèse, nous visons à concevoir des méthodes efficaces pour aborder ces deux questions fondamentales.

[1] `https://www.wikipedia.org`.
[2] Disponible à l'adresse `https://free3d.com/3d-model/cat-v1--522281.html`.

A.2 Les propriétés des graphes et le paradigme du noyau

A.2.1 De l'apprentissage humain à l'apprentissage automatique

Comme dans la plupart des applications en science des données, l'extraction de caractéristiques et l'analyse de propriétés à partir de données sous forme de graphes sont généralement effectuées à partir des connaissances des experts du domaine depuis de nombreuses années. Cependant, ces méthodes souffrent de plusieurs problèmes majeurs. Tout d'abord, pour concevoir certaines méthodes, la connaissance du domaine est nécessaire a priori, et les choix effectués en amont peuvent avoir une grande influence sur les performances. Deuxièmement, en raison de leur spécificité à un certain domaine et un jeu de donnés particulier, la capacité de généralisation peut être limitée. Enfin, ces méthodes n'ont généralement pas la capacité d'apprendre à partir de (nouvelles) données, ce qui entraîne une potentielle perte d'informations [George and Hautier, 2020].

Avec l'avènement de l'ère dite du *Big Data*, les méthodes d'apprentissage automatique sont de plus en plus utilisées pour analyser des quantités massives de données. Kevin P. Murphy [Murphy, 2012] définit l'apprentissage automatique comme suit :

> *L'apprentissage automatique est un ensemble de méthodes permettant de détecter automatiquement des motifs dans les données, puis d'utiliser les motifs découverts pour prédire les propriétés de données futures ou pour prendre d'autres types de décisions en situation d'incertitude (comme la planification de la collecte de données supplémentaires).*

Ces dernières années, l'apprentissage automatique est devenu un outil efficace dans de nombreuses tâches du monde réel, telles que les problèmes de régression et de classification. Les algorithmes d'apprentissage automatique se concentrent sur les données elles-mêmes, en extrayant des informations pertinentes. Les avantages de l'apprentissage automatique incitent fortement à leur adaptation sur des données sous forme de graphes [Chami et al., 2020], incluant l'apprentissage automatique de données moléculaires [Wu et al., 2018] et la découverte de nouveaux médicaments [Giguère et al., 2015]. Cependant, les algorithmes d'apprentissage automatique ont été généralement définis sur des espaces vectoriels, ce qui permet d'utiliser les propriétés d'algèbre linéaire. Cependant, il est difficile de vectoriser de nombreux types

de données en raison de leurs structures complexes, comme les chaînes de caractères, les arbres et les graphes, les structures de graphes étant l'une des plus difficiles car elles peuvent être vues comme une généralisation des séquences (et donc des chaînes de caractères) et des arbres.

Malgré le pouvoir d'expression structurelle permis par les graphes, leur complexité structurelle devient un talon d'Achille lorsque l'on souhaite appliquer des méthodes d'apprentissage automatique sur données sous forme de graphe. Pour exploiter la puissance combinée de ces deux puissants outils, il est essentiel de représenter la structure des graphes sous des formes qui peuvent être utilisées par les méthodes d'apprentissage automatique les plus courantes, tout en limitant la perte d'informations lors de la transformation des graphes. Étant donné qu'une partie des algorithmes d'apprentissage automatique reposent sur des mesures de (dis)similarité entre les données, le problème peut premièrement se résumer à calculer une mesure de la (dis)similarité entre les graphes, ce qui peut être interprété via le problème de l'appariement de nœuds de graphes [Conte et al., 2004, Foggia et al., 2014, Livi and Rizzi, 2013, Yan et al., 2016, Wills and Meyer, 2020]. Les mesures de similarité de graphes basées sur l'appariement de nœuds peuvent être grossièrement regroupées en deux grandes catégories : la similarité exacte et la similarité inexacte [Conte et al., 2004]. La première nécessite une correspondance stricte entre les deux graphes appariés, comme pour le problème d'isomorphisme de graphes [Kobler et al., 2012]. Malheureusement, la similarité exacte ne peut pas être calculée en temps polynomial par ces méthodes ; elle n'est donc pas utilisable sur une grande partie des données issues du monde réel. De plus, elle fournit une comparaison binaire et ne permet pas de quantifier la différence entre deux graphes. Pour cette raison, les mesures de similarité inexactes sont couramment appliquées aux graphes. Une façon intuitive d'aborder le problème de la dissimilarité des graphes est de définir les distances et les métriques directement sur l'espace des graphes.

A.2.2 Sur les espaces des graphes

Un *espace des graphes* est généralement considéré comme un ensemble de graphes, doté d'une certaine distance ou métrique. La pierre angulaire est la définition d'une distance appropriée entre deux graphes. Idéalement, nous cherchons des distances qui sont des métriques valides, c'est-à-dire qui satisfont aux conditions de

non-négativité, d'identité, de symétrie et d'inégalité triangulaire. C'est le cas de la *distance chimique*, qui vise à minimiser les différences entre les arêtes appariées de deux graphes [Kvasnička et al., 1991], et de la *distance Chartrand-Kubiki-Shultz (CKS)*, qui utilise les distances des plus courts chemins entre chaque paire de sommets [Chartrand et al., 1998]. Cependant, ces distances sont coûteuses en termes de calcul [Bento and Ioannidis, 2019].

Les pseudométriques permettent d'alléger ces contraintes. Une famille de méthodes intuitive et largement utilisée est la *distance d'édition* [Garey and Johnson, 1979, Sanfeliu and Fu, 1983], qui mesure la dissimilarité entre deux graphes par le coût associé à la transformation d'un graphe en un autre. Des heuristiques sont proposées pour approximer la distance d'édition avec une complexité de calcul acceptable en pratique [Blumenthal et al., 2020]. Une extension de la distance chimique est proposée dans [Jain, 2016], qui peut prendre en compte les attributs des arêtes. Cependant, elle est limitée à la norme de Frobenius et nécessite d'autres relaxations pour être calculée sur des données réelles. D'autres distances, telles que la *distance du sous-graphe commun maximal* [Bunke and Shearer, 1998, Bunke, 1997] et la *distance de réaction* [Koca et al., 2012], souffrent des mêmes problèmes que les deux précédentes [Bento and Ioannidis, 2019].

La littérature sur les propriétés théoriques des espaces des graphes est limitée [Jain, 2016], ce qui inclut les espaces des graphes dotés de la distance du sous-graphe commun maximal [Hurshman and Janssen, 2015] et ceux dotés d'un *noyau d'alignement optimal* [Jain and Obermayer, 2009], ce dernier étant étendu aux espaces de motifs sous forme d'arbres [Feragen et al., 2010, Feragen et al., 2011, Feragen et al., 2012]. Dans [Jain, 2016], les auteurs proposent un *Théorème de Représentation des Graphes* qui induit un espace d'orbite dans lequel les graphes peuvent être considérés comme des points. Un travail connexe est réalisé dans [Kolaczyk et al., 2020].

La limite des distances métriques est bien résumée dans [Grattarola et al., 2019] :

> *L'utilisation de distances métriques, comme les distances d'alignement de graphes [Jain, 2016], ne fait qu'atténuer le problème (que les distances de graphes spécifiques à une application ne satisfont souvent pas la propriété d'identité ou l'inégalité triangulaire [Livi and Rizzi, 2013, Wilson et al., 2014]), car elles sont coûteuses à calculer et donc inutiles pour des applications pratiques. Par conséquent, une approche commu-*

nément utilisée consiste à projeter les graphes dans un espace géométrique plus conventionnel, tel qu'un espace euclidien.

Comme corroboré dans [Grattarola et al., 2019] et bien d'autres, cette limite pousse les chercheurs à étudier les méthodes de projection de graphes et des méthodes à noyaux sur graphes.

A.2.3 Projection des graphes et noyaux sur graphes

Les méthodes de projection de graphes et les noyaux sur graphes peuvent être assimilées à des méthodes de similarité inexacte. Ces stratégies consistent à projeter les graphes dans un espace où les calculs peuvent être facilement effectués, comme un calcul de moyenne ou la mise en œuvre de méthodes de classification ou de régression.

Les méthodes de projection classique de graphes permettent de projeter les graphes dans des espaces euclidiens à dimension finie, dans lesquels les vecteurs sont calculés (explicitement dans de nombreux cas) en encodant certaines informations contenues dans les graphes. Les techniques utilisées par les méthodes de plongement des graphes incluent la factorisation de matrices, les marches aléatoires et l'apprentissage profond [Goyal and Ferrara, 2018, Cai et al., 2018]. En raison de la perte en précision de la représentation d'un graphe sous forme d'un vecteur, une perte d'information est anticipée. En revanche, les noyaux se basent sur un plongement implicite en représentant les graphes dans un espace de caractéristiques de dimension éventuellement infinie, ce qui réduit les restrictions sur les informations encodées. Les deux stratégies sont illustrées dans la FIGURE A.2.

En effet, en tant que généralisation du produit scalaire, les noyaux peuvent être vus comme des mesures de similarité naturelle entre les données. En utilisant l'astuce des noyaux, on peut calculer un produit scalaire dans l'espace des caractéristiques sans décrire explicitement chaque représentation dans cet espace [Schölkopf and Smola, 2002]. Les noyaux ont été largement utilisés en apprentissage automatique avec des méthodes bien connues, telles que les machines à vecteurs de support (SVM) [Cortes and Vapnik, 1995]. Par conséquent, la définition de noyaux entre les graphes est une approche intéressante pour rapprocher l'apprentissage automatique et les données représentées sous forme de graphes. Cette méthode fournit également des solutions pour répondre à la **Question 1** posée dans la Section A.1.

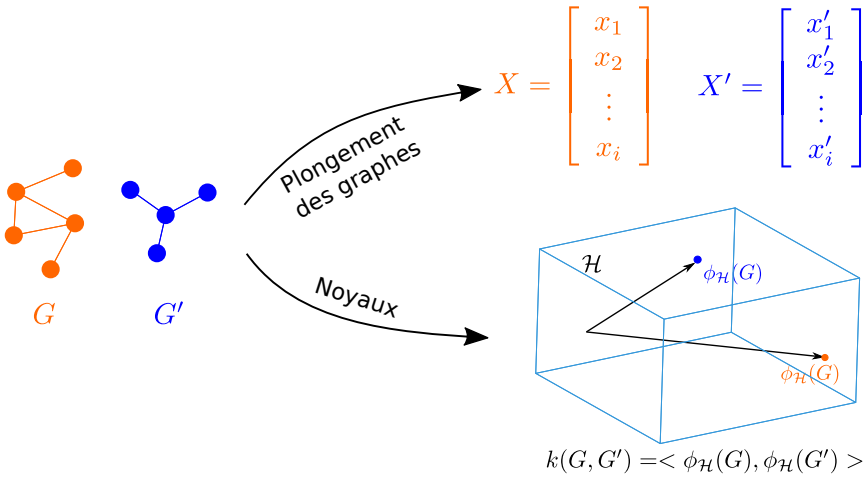

FIGURE A.2 – Comparaison illustrative entre la projection des graphes et les noyaux sur deux graphes arbitraires G et G′. Pour les méthodes de projection, les deux graphes sont représentés par deux vecteurs, X et X′. Pour les méthodes à noyaux, les deux graphes sont implicitement projetés dans un espace de Hilbert $\mathscr{H}$ par une fonction $\phi_{\mathscr{H}}(\cdot)$, donnant $\phi_{\mathscr{H}}$(G) et $\phi_{\mathscr{H}}$(G′) ; de plus, leur produit scalaire $< \phi_{\mathscr{H}}(G), \phi_{\mathscr{H}}(G') >$ est facilement calculé en utilisant une fonction noyau k(G, G′).

Les noyaux sur graphes peuvent être calculés en utilisant des informations globales et/ou locales dans les graphes, au moyen de stratégies variées. Parmi celles-ci, on trouve les noyaux basés sur des sous-structures, les noyaux de propagation de l'information et les noyaux profonds sur graphes [Ghosh et al., 2018]. Les noyaux basés sur des sous-motifs/structures présentent un intérêt particulier. Pour comparer des graphes et analyser leurs propriétés, le principe de similarité a été largement étudié [Johnson and Maggiora, 1990]. Il stipule que les molécules ayant des sous-structures similaires ont des propriétés similaires. Ce principe peut être généralisé à d'autres domaines où les données sont modélisées sous forme de graphes. Il fournit un support théorique pour construire des noyaux sur graphes en étudiant les sous-structures des graphes, qui sont également appelées motifs.

A.3 Construire des graphes par la résolution du problème de la pré-image

Les noyaux sur graphes permettent de combler le fossé entre les structures de graphes et les méthodes à noyaux, grâce à un plongement implicite des graphes dans un espace à noyau. En contrepartie, L'inverse du plongement (implicite) ob-

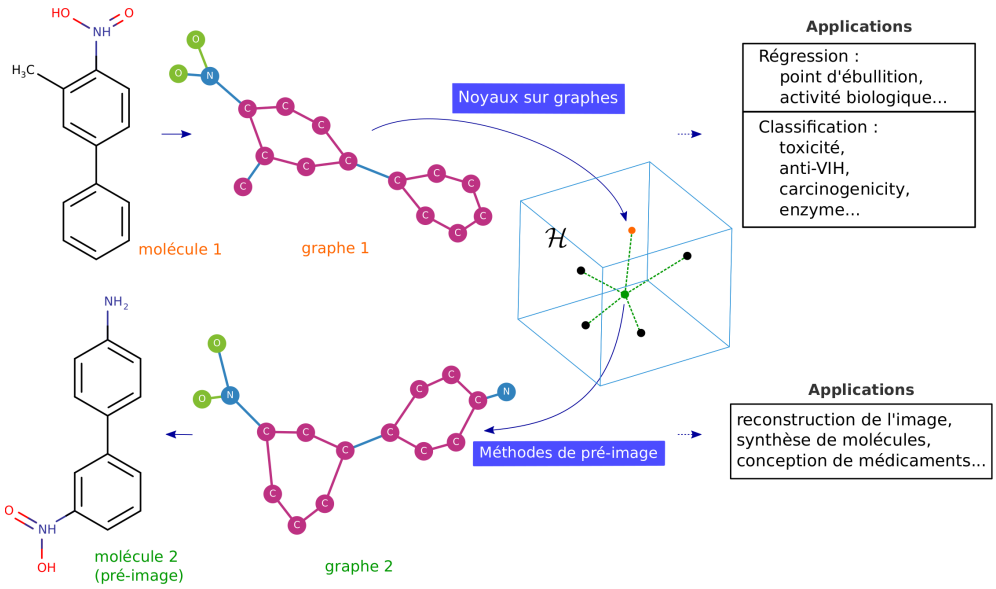

FIGURE A.3 – Relation entre les noyaux sur graphes et les pré-images des graphes. Le premier relie les graphes à un espace à noyau $\mathcal{H}$, tandis que le second effectue la transformation inverse, en reliant les éléments de l'espace à noyau aux graphes.

tenu via les noyaux sur graphes est ce qu'on appelle le problème de la pré-image des graphes. Le problème de la pré-image consiste à estimer l'application inverse de celle associée aux noyaux, c'est-à-dire de l'espace à noyau vers l'espace de départ, l'espace des graphes. Le problème de la pré-image du graphe consiste à reconstruire un graphe associé à un point particulier de l'espace à noyau, ce qui implique de reconstruire un graphe ayant certaines caractéristiques et propriétés souhaitées. La résolution du problème de la pré-image des graphes apporte des réponses à la **Question 2** soulevée dans la Section A.1. La Figure A.3 illustre la relation entre les noyaux sur graphes et la pré-image des graphes. L'estimation des pré-images des graphes se situe à la frontière entre le problème général de la pré-image et le problème de la génération de graphes.

La pré-image est une application inverse non linéaire des éléments de l'espace à noyau vers l'espace de départ. Le problème de la pré-image a été principalement étudié sur les espaces euclidiens, avec de nombreuses applications, incluant le débruitage et l'extraction de caractéristiques avec l'analyse en composantes principales à noyau [Honeine, 2012] et avec la factorisation de matrices non négatives à noyau [Zhu and Honeine, 2017]. Il est également étroitement lié au problème de la réduction de la dimension. Le défi de trouver la pré-image réside dans le fait que l'application inverse n'existe pas en général et que la plupart des élé-

ments dans l'espace à noyau ne possèdent pas de pré-images valides dans l'espace d'entrée. Par conséquent, diverses méthodes ont été développées pour approcher une solution. Nous invitons les lecteurs intéressés à consulter le tutoriel [Honeine and Richard, 2011].

La résolution du problème de la pré-image pour les graphes ouvre la porte à de nombreuses applications intéressantes, telles que la synthèse de molécules et la conception de médicaments. Cependant, la recherche de la pré-image sous forme de graphe hérite des difficultés du problème de la pré-image traditionnelle. De plus, contrairement aux entrées considérées par le problème traditionnel de la pré-image (c'est-à-dire les vecteurs) qui se trouvent généralement dans des espaces continus, les graphes sont des structures où les nombres de sommets et d'arêtes ne peuvent être que des entiers naturels. Les nombres de sommets et d'arêtes dans un graphe peuvent être arbitraires et une arête peut exister entre n'importe quelle paire de sommets. En outre, plusieurs étiquettes et attributs peuvent être associés à chaque sommet et à chaque arête d'un graphe. Compte tenu de ces caractéristiques structurelles, le problème de la pré-image des graphes est plus difficile à résoudre. Plusieurs travaux pionniers pour construire des pré-images des graphes ont été proposés [Bakır et al., 2004, Akutsu and Fukagawa, 2005, Nagamochi, 2009] ; cependant, ils sont limités à des noyaux sur graphes particuliers ou des types de graphes spécifiques, tels que les chaînes de caractères. Il est donc intéressant de proposer des méthodes plus généralistes.

Apprentissage de métrique pour la pré-image des graphes

Les noyaux sur graphes ne fonctionnent pas directement dans l'espace des graphes ; il est donc difficile de les utiliser pour construire des pré-images des graphes. Il est nécessaire de définir des outils et des métriques qui opèrent dans l'espace des graphes. Dans cette catégorie se trouve la distance d'édition entre graphes. En ajustant la relation entre l'espace des graphes et l'espace à noyau, on peut construire des pré-images de graphes en fonction des informations contenues dans l'espace à noyau. L'apprentissage de métrique convient parfaitement à cet objectif et devrait permettre d'optimiser la distance d'édition afin de se rapprocher de l'espace à noyaux en termes de mesure de dissimilarité.

L'apprentissage de métrique consiste à apprendre une mesure de (dis)similarité à partir d'un ensemble d'apprentissage composé d'instances de données et de pro-

priétés à prédire associées. Pour l'apprentissage de métrique classique, où chaque instance de données est encodée par un vecteur à valeurs réelles, le problème consiste à apprendre une mesure de dissimilarité qui diminue (resp. augmente) lorsque les vecteurs ont des propriétés similaires (resp. différentes). De nombreuses études sur l'apprentissage de métrique se concentrent sur les données euclidiennes, alors que seulement quelques-unes abordent ce problème sur des données structurées [Bellet et al., 2013]. Une revue complète de la représentation générale des données structurées est donnée dans [Ontañón, 2020]. Lorsqu'il s'agit de distances d'édition de graphes, une stratégie d'apprentissage de métrique supervisée peut aider à trouver des coûts d'édition optimisés pour une tâche particulière, en s'appuyant sur les informations contenues dans l'espace cible.

Dans l'esprit de l'apprentissage de métrique, le positionnement multidimensionnel (*MultiDimensionnal Scaling* ou MDS) cherche à projeter les données dans un espace de faible dimension en préservant les distances entre chaque paire d'observations [Cox and Cox, 2008]. Des méthodes basées sur ce principe ont été proposées pour résoudre le problème de la pré-image de données vectorielles [Kwok and Tsang, 2004]. Dans le Chapitre 5, nous proposons de résoudre le problème de la pré-image des graphes en considérant l'apprentissage de métrique dans les espaces des graphes, via l'utilisation de la distance d'édition de graphes [Jia et al., 2021].

A.4 Contributions

Cette thèse étudie les métriques dans les espaces de graphes et à noyaux, afin de connecter ces espaces dans la perspective de la résolution du problème de la pré-image.

Les premières contributions se concentrent sur les noyaux sur graphes, en mettant l'accent sur ceux basés sur des motifs linéaires, et plusieurs autres basés sur des motifs non linéaires pour comparaison. Une étude approfondie et une comparaison de ces noyaux sont proposées, tant sur le plan théorique que sur le plan expérimental. En ce qui concerne les aspects théoriques, nous examinons leurs expressions mathématiques et mettons en évidence leurs relations, nous étudions leurs complexités de calcul, ainsi que les forces et les faiblesses de chaque noyau. De plus, nous établissons des connexions avec d'autres noyaux de la littérature. Dans l'analyse expérimentale exhaustive menée dans cette thèse, chaque noyau est appliqué

sur divers jeux de données synthétiques et également issus du monde réel. Chaque jeu de données présente différents types de graphes, y compris des graphes étiquetés et non étiquetés, des graphes avec différents nombres de sommets, des graphes avec différents degrés moyens de sommets, des graphes linéaires et non linéaires, etc. Une analyse approfondie des performances, y compris la comparaison entre ces types de graphes, est effectuée en tenant compte de la précision de la prédiction et du temps de calcul. Cet examen rigoureux permet de fournir des suggestions pour choisir les noyaux en fonction du type de graphes à disposition.

La complexité de calcul étant un point faible des noyaux sur graphes, nous proposons plusieurs stratégies pour résoudre ce problème critique, incluant la parallélisation, la structure de données *trie* et la méthode FCSP (Fast Computation of Shortest Path) que nous étendons à d'autres noyaux et à la comparaison des arêtes. Toutes les stratégies proposées permettent de gagner des ordres de grandeur en termes de temps de calcul et d'utilisation de la mémoire. Des expériences sont réalisées pour démontrer leur pertinence.

Les secondes contributions se concentrent sur une métrique dans l'espace des graphes, à savoir la distance d'édition entre graphes (GED). Lors du calcul de la GED par des heuristiques, la précision de l'approximation peut varier et donc influencer sur la performance de la tâche. En se basant sur le réexamen de deux heuristiques représentatives, à savoir `bipartite` et `IPFP`, une étude de la stabilité du calcul de la GED est réalisée sur des ensembles de données réelles bien connues. Un critère pour mesurer la stabilité est défini. Une alternative à initialisations multiples de `bipartite` et `IPFP` introduite dans [Daller et al., 2018], à savoir m`bipartite` et m`IPFP`, permet d'obtenir de meilleures approximations. Nous examinons comment la stabilité du calcul change avec le nombre de solutions utilisées. Les effets d'un autre facteur sur la stabilité sont également étudiés, à savoir le rapport entre les coûts d'édition sur les sommets et sur les arêtes. Les raisons qui induisent ces influences sont expliquées, et des conseils pour choisir les valeurs appropriées de ces facteurs sont proposés.

Une stratégie visant à optimiser les coûts d'édition de la GED en fonction d'une tâche particulière est proposée, ce qui permet d'éviter l'utilisation de coûts prédéfinis. L'idée est d'aligner la métrique dans l'espace des graphes (à savoir, la GED) sur l'espace cible dans l'esprit de l'apprentissage de métrique. Avec ce principe de préservation de la distance, une procédure d'optimisation itérative est proposée, en alternant une mise à jour des coûts d'édition par la résolution d'un problème linéaire

sous contraintes et un nouveau calcul des chemins d'édition optimaux en fonction des nouveaux coûts calculés. Les coûts d'édition résultant de la procédure d'optimisation peuvent ensuite être analysés pour mieux comprendre comment l'espace des graphes est structuré. La pertinence de la méthode proposée est démontrée sur deux tâches de régression, montrant que les coûts optimisés conduisent à une erreur de prédiction plus faible par rapport à l'état de l'art où les coûts sont définis aléatoirement, par des experts, ou encore par optimisation.

Notre troisième contribution, qui bénéficie de ces études sur les espaces des graphes et à noyaux, porte sur la résolution du problème de la pré-image des graphes. Nous proposons une nouvelle méthode de pré-image pour les graphes, en reliant la distance d'édition entre graphes (GED) et les noyaux sur graphes. En empruntant les méthodes d'apprentissage de métrique susmentionné, les coûts d'édition de la GED sont ajustés en fonction des distances correspondantes dans l'espace à noyau. La stratégie d'optimisation itérative alternée sur les coûts d'édition et les chemins d'édition optimaux est utilisée. En conséquence, les métriques des deux espaces sont alignées, ce qui permet de construire la pré-image du graphe par des méthodes de construction de graphe basées sur les GED avec les coûts d'édition optimisés. Plus précisément, le problème de pré-image est abordé par le calcul du graphe médian, en se basant sur l'hypothèse que, grâce à l'alignement des deux métriques, le graphe médian d'un ensemble de graphes correspond à la pré-image de la moyenne de leurs projections dans l'espace à noyau. Ainsi, le graphe pré-image correspondant peut être approximé via le calcul du graphe médian. Nous profitons des avancées récentes dans le domaine de la GED pour résoudre ce problème, où nous revisitons une procédure de minimisation itérative alternée introduite dans [Boria et al., 2019] pour générer des graphes médians. Les expériences effectuées montrent que notre méthode peut générer de meilleures pré-images que les méthodes existantes, et que les coûts d'édition optimisés donnent de meilleurs résultats que les coûts aléatoires, et sont compétitifs avec les coûts experts.

La dernière contribution de ce travail est l'implémentation d'une bibliothèque Python open-source d'outils d'apprentissage automatique pour les graphes, qui est disponible publiquement sur GitHub[3]. La bibliothèque se compose principalement de quatre parties. Dans la première partie, tous les noyaux sur graphes basés sur des motifs linéaires et deux noyaux basés sur des motifs non linéaires (à savoir le noyau de sous-arbres de Weisfeiler-

[3]Le lien GitHub est `https://github.com/jajupmochi/graphkit-learn`.

Lehman (WL) [Shervashidze et al., 2011, Morris et al., 2017] et le noyau de treelets [Gaüzère et al., 2015b, Bougleux et al., 2012, Gaüzère et al., 2012]) sont implémentés, ainsi que les stratégies permettant de réduire la complexité de calcul de ces noyaux. L'implémentation de chaque noyau est capable de traiter différents types de graphes. La deuxième partie met en œuvre un ensemble de méthodes pour le calcul de GED basé sur la bibliothèque C++ `GEDLIB` [Blumenthal et al., 2019, Blumenthal et al., 2020]. Le paradigme `LSAPE-GED`, qui utilise des transformations du problème d'affectation à somme linéaire avec correction d'erreur (LSAPE), et l'heuristique `bipartite` sont inclus. Un estimateur de graphe médian et un module d'apprentissage de métrique pour les coûts d'édition sont également inclus dans cette bibliothèque. En outre, un module définissant l'interface Python de `GEDLIB` est intégré pour améliorer la vitesse de calcul, qui est basé sur la bibliothèque `GEDLIBPY`. La troisième partie inclut des méthodes de calcul de pré-image de graphes, dont la méthode précitée et celle basée sur la génération aléatoire [Bakır et al., 2004]. La quatrième partie constitue des modules divers. Un module permet de récupérer, charger et manipuler des ensembles de données de graphes à partir de bases de données publiques ; un autre module adapte le calcul des noyaux sur graphes au pipeline de la bibliothèque `scikit-learn` [Pedregosa et al., 2011] et effectue automatiquement la sélection et la validation de modèles.

A.5 Structure de la thèse

Le reste de la thèse est organisé comme suit :

Le Chapitre 2 introduit les préliminaires nécessaires à cette thèse. La Section 2.1 présente les concepts de base et les définitions issues de la théorie des graphes. La Section 2.2 présente le contexte mathématique des méthodes à noyaux, des noyaux basés sur la décomposition, des noyaux sur graphes basés sur les motifs et de la pré-image des graphes. Les concepts relatifs à la distance d'édition entre graphes sont ensuite présentés dans la Section 2.3. Enfin, la Section 2.4 présente et catégorise les jeux de données de graphes utilisés dans la thèse.

Le Chapitre 3 présente les améliorations et les analyses des noyaux sur graphes. Les noyaux sur graphes basés sur des motifs linéaires et leurs connexions avec d'autres noyaux de la littérature sont d'abord examinés en détail dans les Sections 3.2, 3.3, et 3.4. Ensuite, deux noyaux sur graphes basés sur des motifs non linéaires sont présentés dans la Section 3.5. La Section 3.6 détaille ensuite trois stra-

tégies permettant de traiter la complexité de calcul des noyaux sur graphes. Des expériences et des analyses complètes de ces noyaux sur différents types de jeux de données de graphes sont présentées dans la Section 3.7. Enfin, la Section 3.8 conclut ce travail.

Le Chapitre 4 se concentre sur la distance d'édition de graphes. Tout d'abord, la Section 4.2, présente les heuristiques d'approximation de la GED, en mettant l'accent sur deux heuristiques représentatives. Ensuite, une étude sur la stabilité du calcul de la GED est réalisée dans la Section 4.3. Puis, une approche d'apprentissage de métrique pour estimer les coûts d'édition entre graphes pour la régression est proposée et évaluée expérimentalement dans la Section 4.4. La Section 4.5 conclut ce travail.

Le Chapitre 5 propose une méthode pour calculer la pré-image des graphes basée sur la GED. La Section 5.1 présente les méthodes de pré-image et de génération de graphes les plus récentes. La Section 5.2 fournit la formulation du problème et la Section 5.3 présente la méthode proposée en deux volets, l'apprentissage des coûts d'édition des GED en fonction des distances dans l'espace à noyau (Section 5.3.1) et le calcul de la pré-image du graphe (Section 5.3.2). La Section 5.4 présente des expériences et des analyses de ces travaux. Enfin, la Section 5.5 conclut le travail.

Le Chapitre 6 présente les détails d'implémentation de notre bibliothèque Python pour l'apprentissage automatique avec les graphes : `graphkit-learn`. Une brève introduction est donnée dans la Section 6.1 et l'architecture globale est décrite dans la Section 6.2. Ensuite, de la Section 6.3 à la Section 6.7, nous présentons les détails de la bibliothèque `graphkit-learn`, dont les implémentations du traitement des données de graphes, les noyaux sur graphes, des méthodes de calcul de distances d'édition entre graphes, des méthodes de pré-image des graphes, et des outils auxiliaires. Une comparaison avec d'autres bibliothèques est également présentée. Des exemples de codes pour utiliser la bibliothèque illustrent les sections correspondantes. La Section 6.8 présente la conclusion et les travaux futurs.

Le chapitre 7 conclut la thèse et offre des perspectives sur les travaux futurs.

A.6 Conclusion

Dans cette thèse, nous avons exploré les méthodes d'apprentissage automatique sur les graphes, en nous concentrant sur les noyaux sur graphes, la distance d'édition entre graphes et le problème de la pré-image des graphes. D'une part, nous avons

fourni un examen approfondi et des améliorations sur les noyaux sur graphes basés sur des motifs linéaires. D'autre part, nous avons étudié la stabilité des méthodes de calcul de distances d'édition entre graphes (GED) et proposé des algorithmes d'apprentissage de métrique pour optimiser la GED dans un cadre de problèmes de régression. Sur la base de ces avancées, nous avons proposé une nouvelle stratégie pour traiter le problème des pré-images sur les graphes. Nous détaillons chaque travail dans la partie suivante.

- Dans le chapitre 3, nous avons présenté une étude des noyaux sur graphes, en mettant l'accent sur les noyaux basés sur des motifs linéaires et sur deux noyaux basés sur des motifs non linéaires à des fins de comparaison, à savoir le noyau de treelet et le noyau de sous-arbre de Weisfeiler-Lehman. Les fondements théoriques de ces noyaux ont été examinés en détail, y compris leur expression mathématique, leur complexité de calcul, leurs forces et faiblesses, les types de graphes sur lesquels ils peuvent être utilisés, les relations entre eux, ainsi que leurs connexions avec d'autres noyaux classiques et de pointe de la littérature. De plus, des expériences ont été menées sur différents types de graphes, à la fois synthétiques et réels, où les performances de prédiction et la complexité en temps de chaque noyau sur graphe ont été analysées et comparées. Les expériences montrent que les noyaux sur graphes basés sur des motifs linéaires, tels que le noyau de chemin jusqu'à la longueur h, peuvent atteindre des performances comparables à celles des noyaux basés sur des motifs non linéaires. En conclusion, nous avons recommandé le choix des noyaux appropriés en fonction des propriétés des données du graphe donné. Ce travail fournit une base pour les analyses des noyaux sur graphes et des éléments pour développer de nouveaux noyaux.

  Pour mieux utiliser ces noyaux, nous avons proposé trois stratégies pour réduire leur complexité de calcul. La première était la méthode FCSP, où les sous-noyaux entre les sommets et les arêtes étaient calculés a priori. La seconde était la parallélisation, qui a été effectuée entre des paires de graphes et la procédure de validation croisée. La troisième est l'utilisation de la structure trie. Elle stocke les sous-motifs de chemin pour réduire l'occupation de la mémoire. Les expériences menées montrent la capacité de ces méthodes afin de réduire la complexité en termes de temps de calcul et d'espace mémoire des noyaux sur graphes.

- Les noyaux sur graphes fournissent une mesure de similarité dans un espace à noyau. Dans le chapitre 4, la distance d'édition entre graphes, une mesure de dissimilarité entre les graphes, a été présentée. En se basant sur la révision de deux paradigmes, `LSAPE-GED` et `LS-GED`, et de deux heuristiques représentatives de ces paradigmes, `bipartite` et `IPFP`, nous avons étudié la stabilité des heuristiques GED, ce qui est une première dans la littérature. Nous avons découvert empiriquement les effets de deux facteurs sur la stabilité, à savoir le choix des coûts d'édition et le nombre de répétitions du calcul de la GED. Des expériences sont menées avec `IPFP` sur plusieurs jeux de données. En conclusion, nous avons observé que plus le rapport (ratio) entre les coûts d'édition des sommets et des arêtes augmente, plus la stabilité de `IPFP` diminue. Le même phénomène a été observé lorsque le paramètre ratio est remplacé par le nombre de répétitions du calcul. Comme ces facteurs peuvent affecter la stabilité et les performances de la GED, ils doivent être soigneusement pris en compte.

  De plus, nous avons développé un algorithme d'apprentissage de métrique pour optimiser la GED pour des problèmes de régression. Les coûts d'édition sont ajustés en alignant les GED et les distances entre les propriétés à prédire. Avec le principe de préservation de la distance, nous avons proposé une procédure itérative, où les deux étapes sont exécutées en alternance : la mise à jour des coûts d'édition est obtenue en résolvant un problème linéaire sous contraintes ; et un nouveau calcul des chemins d'édition optimaux en fonction des nouveaux coûts calculés. Les coûts d'édition optimisés ont été analysés pour avoir un aperçu de la structure de l'espace des graphes par rapport aux propriétés à prédire. Les expériences ont montré que les coûts optimisés permettent d'améliorer de manière significative les performances en prédiction, où un algorithme de k-plus proches voisins a été appliqué.

- Inspirés par cette procédure d'apprentissage de métrique, nous avons proposé au chapitre 5 une nouvelle stratégie pour aborder le problème du calcul de la pré-image sur les graphes. Le calcul de pré-image de graphes peut être vu comme la transformation inverse de celle appliquée par les noyaux sur graphes. Pour construire une approximation d'une pré-image de graphe, nous avons d'abord aligné l'espace des graphes et celui à noyau en minimisant les différences entre les distances dans les deux espaces. Les distances

dans l'espace des graphes peuvent être mesurées par les distances d'édition entre graphes, tandis que les distances dans l'espace à noyau ont été calculées via les valeurs de noyaux sur graphe. Un problème d'optimisation linéaire a été établi pour la minimisation des coûts d'édition ont été optimisés en fonction des distances dans l'espace à noyau. À cette fin, la procédure susmentionnée d'optimisation alternée en deux étapes, sur les coûts d'édition et les chemins d'édition, a été mise en œuvre, les distances dans l'espace à noyau remplaçant les distances entre les propriétés à prédire. Ensuite, en utilisant les coûts d'édition optimisés, un algorithme de génération de graphe a été utilisé pour construire la pré-image. Plus précisément, nous avons adapté une contribution récente sur la construction du graphe médian généralisé pour générer la pré-image de graphe souhaitée, où une procédure itérative de minimisation alternée pour générer des graphes médians est appliquée. Les expériences ont montré la pertinence de notre approche. Elle a surpassé la pré-image issu du jeu de de données et les coûts d'édition définis de manière aléatoire. De plus, elle a été compétitive avec les résultats obtenus en prenant directement les coûts des experts. Notre méthode permet de construire automatiquement de bonnes pré-images de graphes même sans coûts d'édition experts, ce qui constitue une base solide pour l'étude future du problème des pré-images de graphes.

- Pour faciliter l'utilisation et l'examen des travaux mentionnés précédemment et la réalisation des expériences, nous avons publié en ligne la bibliothèque open-source Python `graphkit-learn`, qui a été décrite en détail au Chapitre 6. Cette bibliothèque inclut des outils d'apprentissage automatique sur les graphes, principalement des noyaux sur graphes, des algorithmes de calcul de la distance d'édition entre graphes et des méthodes de calcul de pré-image de graphes. Pour soutenir ces outils, un module pour récupérer, pour charger et pour traiter des données de graphes est inclus. Des outils auxiliaires tels que celui pour le processus de sélection et d'évaluation des modèles sont également implémentés.

A.7 Perspectives

Nous détaillons les futurs travaux pour chaque partie de notre travail comme suit :

- La prochaine étape pour les noyaux sur graphe comprend la comparaison avec les noyaux de l'état de l'art, en particulier ceux basés sur des motifs non linéaires. Des expériences sur un ensemble plus complet de jeux de données pourraient être intéressantes. En outre, nous explorerons la possibilité d'améliorer nos méthodes pour réduire la complexité de calcul et les adapter à d'autres noyaux sur graphes. Les analyses de relations et la comparaison des performances avec les réseaux de neurones sur graphes font également partie des perspectives envisagées.

- Concernant la GED, les futurs travaux consistent en une étude approfondie de la stabilité des heuristiques. Pour la méthode d'optimisation des coûts d'édition, une comparaison avec d'autres méthodes et la preuve de la convergence seront effectuées. Nous envisageons également d'étendre cette méthode aux problèmes de classification et aux coûts d'édition sous forme de fonctions non constantes.

- Pour le problème des pré-images de graphes, la preuve de convergence et la comparaison avec d'autres méthodes seront également effectuées. De plus, la généralisation de la méthode proposée peut être incluse, par exemple sur les graphes avec des étiquettes symboliques, sur la construction de pré-images définies par l'utilisateur, et sur les coûts d'édition non constants. En outre, d'autres mesures de distance sur les graphes et d'autres critères peuvent être utilisées et comparées. Une direction plus ambitieuse consiste à tirer parti d'autres outils pour aborder le problème des pré-images de graphes, tels que les modèles de réseaux de neurones génératifs de graphes et les graphes de connaissances.

- Le dernier axe de travail concerne la bibliothèque `graphkit-learn`, y compris les implémentations des noyaux sur graphes les plus récents, des distances d'édition entre graphes et des algorithmes de pré-image. L'intégration avec d'autres outils sera également envisagée, tels que les réseaux de neurones de graphes appliqués à ces problèmes.

Bibliography

[Abrahamsen and Hansen, 2009] Abrahamsen, T. J. and Hansen, L. K. (2009). Input space regularization stabilizes pre-images for kernel pca de-noising. In 2009 IEEE International Workshop on Machine Learning for Signal Processing, pages 1–6. IEEE.

[Abu-Aisheh et al., 2017] Abu-Aisheh, Z., Gaüzère, B., Bougleux, S., Ramel, J.-Y., Brun, L., Raveaux, R., Héroux, P., and Adam, S. (2017). Graph edit distance contest: Results and future challenges. Pattern Recognition Letters, 100:96–103.

[Akutsu and Fukagawa, 2005] Akutsu, T. and Fukagawa, D. (2005). Inferring a graph from path frequency. In Annual Symposium on Combinatorial Pattern Matching, pages 371–382. Springer.

[Altman, 1992] Altman, N. S. (1992). An introduction to kernel and nearest-neighbor nonparametric regression. The American Statistician, 46(3):175–185.

[Amdahl, 1967] Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. In Proceedings of the April 18-20, 1967, spring joint computer conference, pages 483–485.

[Aziz et al., 2013] Aziz, F., Wilson, R. C., and Hancock, E. R. (2013). Backtrackless walks on a graph. IEEE Transactions on Neural Networks and Learning Systems, 24(6):977–989.

[Bach, 2008] Bach, F. R. (2008). Graph kernels between point clouds. In Proceedings of the 25th international conference on Machine learning, pages 25–32.

[Bai et al., 2017] Bai, L., Rossi, L., Cui, L., Zhang, Z., Ren, P., Bai, X., and Hancock, E. (2017). Quantum kernels for unattributed graphs using discrete-time quantum walks. Pattern Recognition Letters, 87:96–103.

[Bai et al., 2015] Bai, L., Rossi, L., Torsello, A., and Hancock, E. R. (2015). A quantum jensen–shannon graph kernel for unattributed graphs. Pattern Recognition, 48(2):344–355.

[Bajema and Merlin, 1987] Bajema, K. and Merlin, R. (1987). Raman scattering by acoustic phonons in fibonacci gaas-aias superlattices. Physical Review B, 36(8):4555.

[Bakır et al., 2004] Bakır, G. H., Zien, A., and Tsuda, K. (2004). Learning to find graph pre-images. In Joint Pattern Recognition Symposium, pages 253–261. Springer.

[Balcilar et al., 2021] Balcilar, M., Renton, G., Héroux, P., Gaüzère, B., Adam, S., and Honeine, P. (2021). Analyzing the expressive power of graph neural networks in a spectral perspective. In International Conference on Learning Representations.

[Behnel et al., 2011] Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D. S., and Smith, K. (2011). Cython: The best of both worlds. Computing in Science & Engineering, 13(2):31–39.

[Bellet et al., 2012] Bellet, A., Habrard, A., and Sebban, M. (2012). Good edit similarity learning by loss minimization. Machine Learning, 89(1-2):5–35.

[Bellet et al., 2013] Bellet, A., Habrard, A., and Sebban, M. (2013). A survey on metric learning for feature vectors and structured data. arXiv preprint arXiv:1306.6709.

[Bento and Ioannidis, 2019] Bento, J. and Ioannidis, S. (2019). A family of tractable graph metrics. Applied Network Science, 4(1):107.

[Blumenthal et al., 2021] Blumenthal, D. B., Boria, N., Bougleux, S., Brun, L., Gamper, J., and Gaüzère, B. (2021). Scalable generalized median graph estimation and its manifold use in bioinformatics, clustering, classification, and indexing. Information Systems, page 101766.

[Blumenthal et al., 2020] Blumenthal, D. B., Boria, N., Gamper, J., Bougleux, S., and Brun, L. (2020). Comparing heuristics for graph edit distance computation. The VLDB Journal, 29(1):419–458.

[Blumenthal et al., 2019] Blumenthal, D. B., Bougleux, S., Gamper, J., and Brun, L. (2019). Gedlib: A c++ library for graph edit distance computation. In

International Workshop on Graph-Based Representations in Pattern Recognition, pages 14–24. Springer.

[Borgwardt et al., 2020] Borgwardt, K., Ghisu, E., Llinares-López, F., O'Bray, L., and Rieck, B. (2020). Graph kernels: State-of-the-art and future challenges. arXiv preprint arXiv:2011.03854.

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

[Borgwardt et al., 2005] Borgwardt, K. M., Ong, C. S., Schönauer, S., Vishwanathan, S., Smola, A. J., and Kriegel, H.-P. (2005). Protein function prediction via graph kernels. Bioinformatics, 21(suppl_1):i47–i56.

[Boria et al., 2019] Boria, N., Bougleux, S., Gaüzère, B., and Brun, L. (2019). Generalized median graph via iterative alternate minimizations. In International Workshop on Graph-Based Representations in Pattern Recognition, pages 99–109. Springer.

[Boser et al., 1992] Boser, B. E., Guyon, I. M., and Vapnik, V. N. (1992). A training algorithm for optimal margin classifiers. In Proc. fifth annual workshop on Computational learning theory, pages 144–152. ACM.

[Bougleux and Brun, 2016] Bougleux, S. and Brun, L. (2016). Linear sum assignment with edition. arXiv preprint arXiv:1603.04380.

[Bougleux et al., 2015a] Bougleux, S., Brun, L., Carletti, V., Foggia, P., Gaüzère, B., and Vento, M. (2015a). Graph edit distance as a quadratic assignment problem. Pattern Recognition Letters.

[Bougleux et al., 2015b] Bougleux, S., Brun, L., Carletti, V., Foggia, P., Gaüzere, B., and Vento, M. (2015b). A quadratic assignment formulation of the graph edit distance. arXiv preprint arXiv:1512.07494.

[Bougleux et al., 2012] Bougleux, S., Dupé, F.-X., Brun, L., and Mokhtari, M. (2012). Shape similarity based on a treelet kernel with edition. In Gimel'farb, G. and et al., editors, Structural, Syntactic, and Statistical Pattern Recognition, pages 199–207, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Bougleux et al., 2020] Bougleux, S., Gaüzère, B., Blumenthal, D. B., and Brun, L. (2020). Fast linear sum assignment with error-correction and no cost constraints. Pattern Recognition Letters, 134:37–45.

[Bougleux et al., 2016] Bougleux, S., Gaüzère, B., and Brun, L. (2016). Graph edit distance as a quadratic program. In 2016 23rd International Conference on Pattern Recognition (ICPR), pages 1701–1706.

[Bougleux et al., 2017] Bougleux, S., Gaüzère, B., and Brun, L. (2017). A hungarian algorithm for error-correcting graph matching. In International Workshop on Graph-Based Representations in Pattern Recognition, pages 118–127. Springer.

[Bouhamidi and Jbilou, 2008] Bouhamidi, A. and Jbilou, K. (2008). A note on the numerical approximate solutions for generalized sylvester matrix equations with applications. Applied Mathematics and Computation, 206(2):687–694.

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

[Brun, 2018] Brun, L. (2018). Greyc chemistry dataset. accessed October 30, 2018.

[Brun et al., 2012] Brun, L., Gaüzère, B., and Fourey, S. (2012). Relationships between graph edit distance and maximal common structural subgraph. In SSPR/SPR, pages 42–50.

[Bunke, 1997] Bunke, H. (1997). On a relation between graph edit distance and maximum common subgraph. Pattern Recognition Letters, 18(8):689–694.

[Bunke, 1999] Bunke, H. (1999). Error correcting graph matching: On the influence of the underlying cost function. IEEE transactions on pattern analysis and machine intelligence, 21(9):917–922.

[Bunke and Allermann, 1983] Bunke, H. and Allermann, G. (1983). Inexact graph matching for structural pattern recognition. Pattern Recognition Letters, 1(4):245–253.

[Bunke et al., 1999] Bunke, H., Münger, A., and Jiang, X. (1999). Combinatorial search versus genetic algorithms: A case study based on the generalized median graph problem. Pattern recognition letters, 20(11-13):1271–1277.

[Bunke and Shearer, 1998] Bunke, H. and Shearer, K. (1998). A graph distance metric based on the maximal common subgraph. Pattern recognition letters, 19(3-4):255–259.

[Cai et al., 2018] Cai, H., Zheng, V. W., and Chang, K. C.-C. (2018). A comprehensive survey of graph embedding: Problems, techniques, and applications. IEEE Transactions on Knowledge and Data Engineering, 30(9):1616–1637.

[Cao et al., 2018] Cao, Q., Zanni-Merk, C., and Reich, C. (2018). Ontologies for manufacturing process modeling: A survey. In International Conference on Sustainable Design and Manufacturing, pages 61–70. Springer.

[Carletti et al., 2015] Carletti, V., Gaüzère, B., Brun, L., and Vento, M. (2015). Approximate graph edit distance computation combining bipartite matching and exact neighborhood substructure distance. In International Workshop on Graph-Based Representations in Pattern Recognition, pages 188–197. Springer.

[Chami et al., 2020] Chami, I., Abu-El-Haija, S., Perozzi, B., Ré, C., and Murphy, K. (2020). Machine learning on graphs: A model and comprehensive taxonomy. arXiv preprint arXiv:2005.03675.

[Chartrand et al., 1998] Chartrand, G., Kubicki, G., and Schultz, M. (1998). Graph similarity and distance in graphs. Aequationes Mathematicae, 55(1-2):129–145.

[Cherqaoui and Villemin, 1994] Cherqaoui, D. and Villemin, D. (1994). Use of a neural network to determine the boiling point of alkanes. Journal of the Chemical Society, Faraday Transactions, 90(1):97–102.

[Cherqaoui et al., 1994] Cherqaoui, D., Villemin, D., Mesbah, A., Cense, J.-M., and Kvasnicka, V. (1994). Use of a neural network to determine the normal boiling points of acyclic ethers, peroxides, acetals and their sulfur analogues. Journal of the Chemical Society, Faraday Transactions, 90(14):2015–2019.

[Conte et al., 2004] Conte, D., Foggia, P., Sansone, C., and Vento, M. (2004). Thirty years of graph matching in pattern recognition. International journal of pattern recognition and artificial intelligence, 18(03):265–298.

[Cormen et al., 2009] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). Introduction to algorithms. MIT press.

[Cortes et al., 2004] Cortes, C., Haffner, P., and Mohri, M. (2004). Rational kernels: Theory and algorithms. Journal of Machine Learning Research, 5(Aug):1035–1062.

[Cortes and Vapnik, 1995] Cortes, C. and Vapnik, V. (1995). Support-vector networks. Machine learning, 20(3):273–297.

[Cortés et al., 2019] Cortés, X., Conte, D., and Cardot, H. (2019). Learning edit cost estimation models for graph edit distance. Pattern Recognition Letters, 125:256–263.

[Cortés and Serratosa, 2015] Cortés, X. and Serratosa, F. (2015). Learning graph-matching edit-costs based on the optimality of the oracle's node correspondences. Pattern Recognition Letters, 56:22–29.

[Cox and Cox, 2008] Cox, M. A. and Cox, T. F. (2008). Multidimensional scaling. In Handbook of data visualization, pages 315–347. Springer.

[Cristianini et al., 2002] Cristianini, N., Shawe-Taylor, J., Elisseeff, A., and Kandola, J. (2002). On kernel-target alignment. In Advances in neural information processing systems, pages 367–373.

[Daller et al., 2018] Daller, É., Bougleux, S., Gaüzère, B., and Brun, L. (2018). Approximate graph edit distance by several local searches in parallel. In 7th International Conference on Pattern Recognition Applications and Methods.

[De Cao and Kipf, 2018] De Cao, N. and Kipf, T. (2018). Molgan: An implicit generative model for small molecular graphs. arXiv preprint arXiv:1805.11973.

[de Oliveira Rente et al., 2018] de Oliveira Rente, P., Brites, C., Ascenso, J., and Pereira, F. (2018). Graph-based static 3d point clouds geometry coding. IEEE Transactions on Multimedia, 21(2):284–299.

[de Vries, 2013] de Vries, G. K. (2013). A fast approximation of the weisfeiler-lehman graph kernel for rdf data. In Joint European Conference on Machine Learning and Knowledge Discovery in Databases, pages 606–621. Springer.

[Debnath et al., 1991] Debnath, A. K., Lopez de Compadre, R. L., Debnath, G., Shusterman, A. J., and Hansch, C. (1991). Structure-activity relationship of mutagenic

aromatic and heteroaromatic nitro compounds. correlation with molecular orbital energies and hydrophobicity. Journal of medicinal chemistry, 34(2):786–797.

[Diamond and Boyd, 2016] Diamond, S. and Boyd, S. (2016). Cvxpy: A python-embedded modeling language for convex optimization. The Journal of Machine Learning Research, 17(1):2909–2913.

[Dijkstra, 1959] Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. Numerische mathematik, 1(1):269–271.

[Ding et al., 2007] Ding, L., Kolari, P., Ding, Z., and Avancha, S. (2007). Using ontologies in the semantic web: A survey. In Ontologies, pages 79–113. Springer.

[Dobson and Doig, 2003] Dobson, P. D. and Doig, A. J. (2003). Distinguishing enzyme structures from non-enzymes without alignments. Journal of molecular biology, 330(4):771–783.

[Farhi and Gutmann, 1998] Farhi, E. and Gutmann, S. (1998). Quantum computation and decision trees. Physical Review A, 58(2):915.

[Feragen et al., 2013] Feragen, A., Kasenburg, N., Petersen, J., de Bruijne, M., and Borgwardt, K. (2013). Scalable kernels for graphs with continuous attributes. In Advances in Neural Information Processing Systems, pages 216–224.

[Feragen et al., 2010] Feragen, A., Lauze, F., Lo, P., de Bruijne, M., and Nielsen, M. (2010). Geometries on spaces of treelike shapes. In Asian Conference on Computer Vision, pages 160–173. Springer.

[Feragen et al., 2012] Feragen, A., Lo, P., de Bruijne, M., Nielsen, M., and Lauze, F. (2012). Toward a theory of statistical tree-shape analysis. IEEE transactions on pattern analysis and machine intelligence, 35(8):2008–2021.

[Feragen et al., 2011] Feragen, A., Nielsen, M., Hauberg, S., Lo, P., de Bruijne, M., and Lauze, F. (2011). A geometric framework for statistics on trees. Technical report, Technical report, Department of Computer Science, University of Copenhagen.

[Ferrer et al., 2010] Ferrer, M., Valveny, E., Serratosa, F., Riesen, K., and Bunke, H. (2010). Generalized median graph computation by means of graph embedding in vector spaces. Pattern Recognition, 43(4):1642–1655.

[Fey and Lenssen, 2019] Fey, M. and Lenssen, J. E. (2019). Fast graph representation learning with PyTorch Geometric. In ICLR Workshop on Representation Learning on Graphs and Manifolds.

[Floyd, 1962] Floyd, R. W. (1962). Algorithm 97: shortest path. Communications of the ACM, 5(6):345.

[Foggia et al., 2014] Foggia, P., Percannella, G., and Vento, M. (2014). Graph matching and learning in pattern recognition in the last 10 years. International Journal of Pattern Recognition and Artificial Intelligence, 28(01):1450001.

[Fredkin, 1960] Fredkin, E. (1960). Trie memory. Communications of the ACM, 3(9):490–499.

[Garcia-Hernandez et al., 2020] Garcia-Hernandez, C., Fernández, A., and Serratosa, F. (2020). Learning the edit costs of graph edit distance applied to ligand-based virtual screening. Current topics in medicinal chemistry, 20(18):1582–1592.

[Garey and Johnson, 1979] Garey, M. R. and Johnson, D. S. (1979). Computers and intractability, volume 174. freeman San Francisco.

[Gärtner, 2003a] Gärtner, T. (2003a). Exponential and geomteric kernels for graphs. NIPS Workshop on Unreal Data : Principles of Modeling Nonvectorial Data, 2003.

[Gärtner, 2003b] Gärtner, T. (2003b). A survey of kernels for structured data. ACM SIGKDD Explorations Newsletter, 5(1):49–58.

[Gärtner et al., 2003] Gärtner, T., Flach, P., and Wrobel, S. (2003). On graph kernels: Hardness results and efficient alternatives. Learning Theory and Kernel Machines, pages 129–143.

[Gaüzère et al., 2016] Gaüzère, B., Bougleux, S., and Brun, L. (2016). Approximating graph edit distance using gnccp. In Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR), pages 496–506. Springer.

[Gaüzère et al., 2014] Gaüzère, B., Bougleux, S., Riesen, K., and Brun, L. (2014). Approximate graph edit distance guided by bipartite matching of bags of walks. In Joint IAPR International Workshops on Statistical Techniques in Pattern

Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR), pages 73–82. Springer.

[Gaüzère et al., 2012] Gaüzère, B., Brun, L., and Villemin, D. (2012). Two new graphs kernels in chemoinformatics. Pattern Recognition Letters, 33(15):2038–2047.

[Gaüzère et al., 2015a] Gaüzère, B., Brun, L., and Villemin, D. (2015a). Graph kernels in chemoinformatics. In Dehmer, M. and Emmert-Streib, F., editors, Quantitative Graph Theory Mathematical Foundations and Applications, pages 425–470. CRC Press.

[Gaüzére et al., 2012] Gaüzére, B., Brun, L., Villemin, D., and Brun, M. (2012). Graph kernels based on relevant patterns and cycle information for chemoinformatics. In Proceedings of the 21st International Conference on Pattern Recognition (ICPR2012), pages 1775–1778. IEEE.

[Gaüzère et al., 2015b] Gaüzère, B., Grenier, P.-A., Brun, L., and Villemin, D. (2015b). Treelet kernel incorporating cyclic, stereo and inter pattern information in chemoinformatics. Pattern Recognition, 48(2):356 – 367.

[George and Hautier, 2020] George, J. and Hautier, G. (2020). Chemist versus machine: Traditional knowledge versus machine learning techniques. Trends in Chemistry.

[Ghosh et al., 2018] Ghosh, S., Das, N., Gonçalves, T., Quaresma, P., and Kundu, M. (2018). The journey of graph kernels through two decades. Computer Science Review, 27:88–111.

[Gibert et al., 2012] Gibert, J., Valveny, E., and Bunke, H. (2012). Graph embedding in vector spaces by node attribute statistics. Pattern Recognition, 45(9):3072–3083.

[Giguère et al., 2015] Giguère, S., Laviolette, F., Marchand, M., Tremblay, D., Moineau, S., Liang, X., Biron, É., and Corbeil, J. (2015). Machine learning assisted design of highly active peptides for drug discovery. PLoS Comput Biol, 11(4):e1004074.

[Giscard and Rochet, 2018] Giscard, P.-L. and Rochet, P. (2018). Enumerating simple paths from connected induced subgraphs. Graphs and Combinatorics, 34(6):1197–1202.

[Giscard and Wilson, 2017] Giscard, P.-L. and Wilson, R. C. (2017). The all-paths and cycles graph kernel. arXiv preprint arXiv:1708.01410.

[Goyal and Ferrara, 2018] Goyal, P. and Ferrara, E. (2018). Graph embedding techniques, applications, and performance: A survey. Knowledge-Based Systems, 151:78–94.

[Grattarola et al., 2019] Grattarola, D., Zambon, D., Livi, L., and Alippi, C. (2019). Change detection in graph streams by learning graph embeddings on constant-curvature manifolds. IEEE Transactions on neural networks and learning systems, 31(6):1856–1869.

[Grenier et al., 2013] Grenier, P.-A., Brun, L., and Villemin, D. (2013). Treelet kernel incorporating chiral information. In International Workshop on Graph-Based Representations in Pattern Recognition, pages 132–141. Springer.

[Grohe et al., 2017] Grohe, M., Kersting, K., Mladenov, M., and Schweitzer, P. (2017). Color refinement and its applications. Van den Broeck, G.; Kersting, K.; Natarajan, S, page 30.

[Grover et al., 2019] Grover, A., Zweig, A., and Ermon, S. (2019). Graphite: Iterative generative modeling of graphs. In International conference on machine learning, pages 2434–2444. PMLR.

[Guo et al., 2020] Guo, Q., Zhuang, F., Qin, C., Zhu, H., Xie, X., Xiong, H., and He, Q. (2020). A survey on knowledge graph-based recommender systems. arXiv preprint arXiv:2003.00911.

[Hagberg et al., 2008] Hagberg, A., Swart, P., and S Chult, D. (2008). Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States).

[Hamilton, 2020] Hamilton, W. L. (2020). Graph representation learning. Synthesis Lectures on Artifical Intelligence and Machine Learning, 14(3):1–159.

[Harris et al., 2020] Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del R'10, J. F., Wiebe, M., Peterson, P., G'erard-Marchant, P., Sheppard, K., Reddy, T., Weckesser,

W., Abbasi, H., Gohlke, C., and Oliphant, T. E. (2020). Array programming with NumPy. Nature, 585(7825):357–362.

[Haussler, 1999] Haussler, D. (1999). Convolution kernels on discrete structures. Technical report, Technical report, Department of Computer Science, University of California at Santa Cruz.

[Hido and Kashima, 2009] Hido, S. and Kashima, H. (2009). A linear-time graph kernel. In 2009 Ninth IEEE International Conference on Data Mining, pages 179–188. IEEE.

[Honeine, 2012] Honeine, P. (2012). Online kernel principal component analysis: a reduced-order model. IEEE Transactions on Pattern Analysis and Machine Intelligence, 34(9):1814 – 1826.

[Honeine et al., 2013] Honeine, P., Noumir, Z., and Richard, C. (2013). Multiclass classification machines with the complexity of a single binary classifier. Signal Processing, 93(5):1013 – 1026.

[Honeine and Richard, 2009] Honeine, P. and Richard, C. (2009). Solving the pre-image problem in kernel machines: A direct method. In 2009 IEEE International Workshop on Machine Learning for Signal Processing, pages 1–6. IEEE.

[Honeine and Richard, 2011] Honeine, P. and Richard, C. (2011). Preimage problem in kernel-based machine learning. IEEE Signal Processing Magazine, 28(2):77–88.

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

[Huber et al., 2007] Huber, W., Carey, V. J., Long, L., Falcon, S., and Gentleman, R. (2007). Graphs in molecular biology. BMC bioinformatics, 8(6):1–14.

[Hurshman and Janssen, 2015] Hurshman, M. and Janssen, J. (2015). On the continuity of graph parameters. Discrete Applied Mathematics, 181:123–129.

[Jaakkola et al., 1999] Jaakkola, T. S., Diekhans, M., and Haussler, D. (1999). Using the fisher kernel method to detect remote protein homologies. In ISMB, volume 99, pages 149–158.

[Jain, 2016] Jain, B. J. (2016). On the geometry of graph spaces. Discrete Applied Mathematics, 214:126–144.

[Jain and Obermayer, 2009] Jain, B. J. and Obermayer, K. (2009). Structure spaces. Journal of Machine Learning Research, 10(11).

[Ji et al., 2020] Ji, S., Pan, S., Cambria, E., Marttinen, P., and Yu, P. S. (2020). A survey on knowledge graphs: Representation, acquisition and applications. arXiv preprint arXiv:2002.00388.

[Jia et al., 2021] Jia, L., Gaüzère, B., and Honeine, P. (2021). A graph pre-image method based on graph edit distances. In Proceedings of the IAPR Joint International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (S+SSPR), Venice, Italy.

[Jiang et al., 2001] Jiang, X., Munger, A., and Bunke, H. (2001). An median graphs: properties, algorithms, and applications. IEEE Transactions on pattern analysis and machine intelligence, 23(10):1144–1151.

[Johnson and Maggiora, 1990] Johnson, M. A. and Maggiora, G. M. (1990). Concepts and applications of molecular similarity. Wiley.

[Kang et al., 2012] Kang, U., Tong, H., and Sun, J. (2012). Fast random walk graph kernel. In Proceedings of the 2012 SIAM International Conference on Data Mining, pages 828–838. SIAM.

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

[Kaspar and Horst, 2010] Kaspar, R. and Horst, B. (2010). Graph classification and clustering based on vector space embedding, volume 77. World Scientific.

[Kersting et al., 2014] Kersting, K., Mladenov, M., Garnett, R., and Grohe, M. (2014). Power iterated color refinement. In Twenty-Eighth AAAI Conference on Artificial Intelligence.

[Kipf and Welling, 2016] Kipf, T. N. and Welling, M. (2016). Variational graph auto-encoders. arXiv preprint arXiv:1611.07308.

[Kisku et al., 2011] Kisku, D. R., Gupta, P., and Sing, J. K. (2011). Graphs in biometrics. In Cases on ICT Utilization, Practice and Solutions: Tools for Managing Day-to-Day Issues, pages 151–180. IGI Global.

[Kobler et al., 2012] Kobler, J., Schöning, U., and Torán, J. (2012). The graph isomorphism problem: its structural complexity. Springer Science & Business Media.

[Koca et al., 2012] Koca, J., Kratochvil, M., Kvasnicka, V., Matyska, L., and Pospichal, J. (2012). Synthon model of organic chemistry and synthesis design, volume 51. Springer Science & Business Media.

[Kolaczyk et al., 2020] Kolaczyk, E. D., Lin, L., Rosenberg, S., Walters, J., Xu, J., et al. (2020). Averages of unlabeled networks: Geometric characterization and asymptotic behavior. The Annals of Statistics, 48(1):514–538.

[Kondor and Lafferty, 2002] Kondor, R. I. and Lafferty, J. (2002). Diffusion kernels on graphs and other discrete input spaces. In ICML, volume 2, pages 315–322.

[Kriege et al., 2014] Kriege, N., Neumann, M., Kersting, K., and Mutzel, P. (2014). Explicit versus implicit graph feature maps: A computational phase transition for walk kernels. In 2014 IEEE International Conference on Data Mining, pages 881–886. IEEE.

[Kriege, 2019] Kriege, N. M. (2019). Deep weisfeiler-lehman assignment kernels via multiple kernel learning. arXiv preprint arXiv:1908.06661.

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

[Kriege et al., 2020] Kriege, N. M., Johansson, F. D., and Morris, C. (2020). A survey on graph kernels. Applied Network Science, 5(1):1–42.

[Kriege et al., 2017] Kriege, N. M., Neumann, M., Morris, C., Kersting, K., and Mutzel, P. (2017). A unifying view of explicit and implicit feature maps for structured data: systematic studies of graph kernels. arXiv preprint arXiv:1703.00676.

[Kuhn, 1955] Kuhn, H. W. (1955). The hungarian method for the assignment problem. Naval research logistics quarterly, 2(1-2):83–97.

[Kvasnička et al., 1991] Kvasnička, V., Pospíchal, J., and Baláž, V. (1991). Reaction and chemical distances and reaction graphs. Theoretica chimica acta, 79(1):65–79.

[Kwok and Tsang, 2004] Kwok, J.-Y. and Tsang, I.-H. (2004). The pre-image problem in kernel methods. IEEE transactions on neural networks, 15(6):1517–1525.

[Lamberti et al., 2008] Lamberti, P., Majtey, A., Borras, A., Casas, M., and Plastino, A. (2008). Metric character of the quantum jensen-shannon divergence. Physical Review A, 77(5):052311.

[Lawson and Hanson, 1995] Lawson, C. L. and Hanson, R. J. (1995). Solving least squares problems. SIAM.

[Leordeanu et al., 2009] Leordeanu, M., Hebert, M., and Sukthankar, R. (2009). An integer projected fixed point method for graph matching and map inference. Advances in neural information processing systems, 22:1114–1122.

[Li et al., 2012] Li, B., Zhu, X., Chi, L., and Zhang, C. (2012). Nested subtree hash kernels for large-scale graph classification over streams. In 2012 IEEE 12th International Conference on Data Mining, pages 399–408. IEEE.

[Li et al., 2016] Li, W., Saidi, H., Sanchez, H., Schäf, M., and Schweitzer, P. (2016). Detecting similar programs via the weisfeiler-leman graph kernel. In International conference on software reuse, pages 315–330. Springer.

[Liao et al., 2019] Liao, R., Li, Y., Song, Y., Wang, S., Nash, C., Hamilton, W. L., Duvenaud, D., Urtasun, R., and Zemel, R. S. (2019). Efficient graph generation with graph recurrent attention networks. arXiv preprint arXiv:1910.00760.

[Livi and Rizzi, 2013] Livi, L. and Rizzi, A. (2013). The graph matching problem. Pattern Analysis and Applications, 16(3):253–283.

[Mahé et al., 2004] Mahé, P., Ueda, N., Akutsu, T., Perret, J.-L., and Vert, J.-P. (2004). Extensions of marginalized graph kernels. In Proc. the twenty-first international conference on Machine learning, page 70. ACM.

[Majtey et al., 2005] Majtey, A., Lamberti, P., and Prato, D. (2005). Jensen-shannon divergence as a measure of distinguishability between mixed quantum states. Physical Review A, 72(5):052310.

[Martineau et al., 2020] Martineau, M., Raveaux, R., Conte, D., and Venturini, G. (2020). Learning error-correcting graph matching with a multiclass neural network. Pattern Recognition Letters, 134:68–76.

[Mercer, 1909] Mercer, B. (1909). Xvi. functions of positive and negative type, and their connection the theory of integral equations. Phil. Trans. R. Soc. Lond. A, 209(441-458):415–446.

[Michail, 2016] Michail, O. (2016). An introduction to temporal graphs: An algorithmic perspective. Internet Mathematics, 12(4):239–280.

[Mitchell et al., 1997] Mitchell, T. M. et al. (1997). Machine learning.

[Morgan, 1965] Morgan, H. L. (1965). The generation of a unique machine description for chemical structures-a technique developed at chemical abstracts service. Journal of Chemical Documentation, 5(2):107–113.

[Morris et al., 2017] Morris, C., Kersting, K., and Mutzel, P. (2017). Glocalized weisfeiler-lehman graph kernels: Global-local feature maps of graphs. In 2017 IEEE International Conference on Data Mining (ICDM), pages 327–336.

[Morris et al., 2020] Morris, C., Kriege, N. M., Bause, F., Kersting, K., Mutzel, P., and Neumann, M. (2020). Tudataset: A collection of benchmark datasets for learning with graphs. In ICML 2020 Workshop on Graph Representation Learning and Beyond (GRL+ 2020).

[Morris et al., 2016] Morris, C., Kriege, N. M., Kersting, K., and Mutzel, P. (2016). Faster kernels for graphs with continuous attributes via hashing. In Data Mining (ICDM), 2016 IEEE 16th International Conference on, pages 1095–1100. IEEE.

[Morris et al., 2019] Morris, C., Ritzert, M., Fey, M., Hamilton, W. L., Lenssen, J. E., Rattan, G., and Grohe, M. (2019). Weisfeiler and leman go neural: Higher-order graph neural networks. In Proceedings of the AAAI Conference on Artificial Intelligence, volume 33, pages 4602–4609.

[Munkres, 1957] Munkres, J. (1957). Algorithms for the assignment and transportation problems. Journal of the society for industrial and applied mathematics, 5(1):32–38.

[Murphy, 2012] Murphy, K. P. (2012). Machine Learning: A Probabilistic Perspective. MIT Press.

[Murray et al., 2018] Murray, R. M. et al. (2018). Python Control Systems Library.

[Musmanno and Ribeiro, 2016] Musmanno, L. M. and Ribeiro, C. C. (2016). Heuristics for the generalized median graph problem. European Journal of Operational Research, 254(2):371–384.

[Nagamochi, 2009] Nagamochi, H. (2009). A detachment algorithm for inferring a graph from path frequency. Algorithmica, 53(2):207–224.

[Nastase et al., 2015] Nastase, V., Mihalcea, R., and Radev, D. R. (2015). A survey of graphs in natural language processing. Natural Language Engineering, 21(5):665–698.

[Nene et al., 1996] Nene, S. A., Nayar, S. K., Murase, H., et al. (1996). Columbia object image library (coil-100).

[Neuhaus and Bunke, 2004] Neuhaus, M. and Bunke, H. (2004). A probabilistic approach to learning costs for graph edit distance. Proceedings ICPR, 3(C):389–393.

[Neuhaus and Bunke, 2005] Neuhaus, M. and Bunke, H. (2005). Self-organizing maps for learning the edit costs in graph matching. IEEE transactions on systems, man, and cybernetics, 35(3):503–14.

[Neuhaus and Bunke, 2007] Neuhaus, M. and Bunke, H. (2007). Automatic learning of cost functions for graph edit distance. Information Sciences, 177(1):239–247.

[Neuhaus et al., 2006] Neuhaus, M., Riesen, K., and Bunke, H. (2006). Fast suboptimal algorithms for the computation of graph edit distance. In Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR), pages 163–172. Springer.

[Nikolentzos et al., 2018] Nikolentzos, G., Meladianos, P., Limnios, S., and Vazirgiannis, M. (2018). A degeneracy framework for graph similarity. In IJCAI, pages 2595–2601.

[Nikolentzos and Vazirgiannis, 2018] Nikolentzos, G. and Vazirgiannis, M. (2018). Message passing graph kernels. arXiv preprint arXiv:1808.02510.

[Ontañón, 2020] Ontañón, S. (2020). An overview of distance and similarity functions for structured data. Artificial Intelligence Review, pages 1–43.

[Orsini et al., 2015] Orsini, F., Frasconi, P., and De Raedt, L. (2015). Graph invariant kernels. In Proceedings of the Twenty-fourth International Joint Conference on Artificial Intelligence, pages 3756–3762.

[Pedregosa et al., 2011] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. Journal of Machine Learning Research, 12:2825–2830.

[Ralaivola et al., 2005] Ralaivola, L., Swamidass, S. J., Saigo, H., and Baldi, P. (2005). Graph kernels for chemical informatics. Neural networks, 18(8):1093–1110.

[Ramon and Gärtner, 2003] Ramon, J. and Gärtner, T. (2003). Expressivity versus efficiency of graph kernels. In Proceedings of the first international workshop on mining graphs, trees and sequences, pages 65–74.

[Reinanda et al., 2020] Reinanda, R., Meij, E., de Rijke, M., et al. (2020). Knowledge graphs: An information retrieval perspective. Foundations and Trends® in Information Retrieval, 14(4):1–158.

[Rettinger et al., 2012] Rettinger, A., Lösch, U., Tresp, V., d'Amato, C., and Fanizzi, N. (2012). Mining the semantic web. Data Mining and Knowledge Discovery, 24(3):613–662.

[Rieck et al., 2019] Rieck, B., Bock, C., and Borgwardt, K. (2019). A persistent weisfeiler-lehman procedure for graph classification. In International Conference on Machine Learning, pages 5448–5458.

[Riesen, 2015] Riesen, K. (2015). Structural pattern recognition with graph edit distance. In Advances in computer vision and pattern recognition. Springer.

[Riesen and Bunke, 2008] Riesen, K. and Bunke, H. (2008). Iam graph database repository for graph based pattern recognition and machine learning. In Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition and Structural and Syntactic Pattern Recognition, pages 287–297. Springer.

[Ristad and N.yianilos, 1998] Ristad, E. S. and N.yianilos, P. (1998). Learning string-edit distance. IEEE Transactions on Pattern Analysis and Machine Intelligence, 20(5):522–532.

[Rossi et al., 2013] Rossi, L., Torsello, A., and Hancock, E. R. (2013). A continuous-time quantum walk kernel for unattributed graphs. In International Workshop on Graph-Based Representations in Pattern Recognition, pages 101–110. Springer.

[Russell, 2013] Russell, M. A. (2013). Mining the social web: data mining Facebook, Twitter, LinkedIn, Google+, GitHub, and more. " O'Reilly Media, Inc.".

[Sanfeliu and Fu, 1983] Sanfeliu, A. and Fu, K.-S. (1983). A distance measure between attributed relational graphs for pattern recognition. IEEE transactions on systems, man, and cybernetics, (3):353–362.

[Schieber et al., 2017] Schieber, T. A., Carpi, L., Díaz-Guilera, A., Pardalos, P. M., Masoller, C., and Ravetti, M. G. (2017). Quantification of network structural dissimilarities. Nature communications, 8:13928.

[Schneider et al., 2020] Schneider, P., Walters, W. P., Plowright, A. T., Sieroka, N., Listgarten, J., Goodnow, R. A., Fisher, J., Jansen, J. M., Duca, J. S., Rush, T. S., et al. (2020). Rethinking drug design in the artificial intelligence era. Nature Reviews Drug Discovery, 19(5):353–364.

[Schölkopf et al., 2001] Schölkopf, B., Herbrich, R., and Smola, A. J. (2001). A generalized representer theorem. In Proc. 14th Annual Conference on Computational Learning Theory and 5th European Conference on Computational Learning Theory, COLT/EuroCOLT, pages 416–426, London, UK. Springer-Verlag.

[Schölkopf and Smola, 2002] Schölkopf, B. and Smola, A. J. (2002). Learning with kernels: support vector machines, regularization, optimization, and beyond. MIT press.

[Schomburg et al., 2004] Schomburg, I., Chang, A., Ebeling, C., Gremse, M., Heldt, C., Huhn, G., and Schomburg, D. (2004). Brenda, the enzyme database: updates and major new developments. Nucleic acids research, 32(suppl_1):D431–D433.

[Scott, 2011] Scott, J. (2011). Social network analysis: developments, advances, and prospects. Social network analysis and mining, 1(1):21–26.

[Shawe-Taylor and Cristianini, 2004] Shawe-Taylor, J. and Cristianini, N. (2004). Kernel methods for pattern analysis. Cambridge university press.

[Shervashidze and Borgwardt, 2009] Shervashidze, N. and Borgwardt, K. (2009). Fast subtree kernels on graphs. In Advances in neural information processing systems, pages 1660–1668.

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

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

[Siglidis et al., 2020] Siglidis, G., Nikolentzos, G., Limnios, S., Giatsidis, C., Skianis, K., and Vazirgiannis, M. (2020). Grakel: A graph kernel library in python. Journal of Machine Learning Research, 21(54):1–5.

[Simonovsky and Komodakis, 2018] Simonovsky, M. and Komodakis, N. (2018). Graphvae: Towards generation of small graphs using variational autoencoders. In International Conference on Artificial Neural Networks, pages 412–422. Springer.

[Suard et al., 2007] Suard, F., Rakotomamonjy, A., and Bensrhair, A. (2007). Kernel on bag of paths for measuring similarity of shapes. In ESANN, pages 355–360.

[Sugiyama and Borgwardt, 2015] Sugiyama, M. and Borgwardt, K. (2015). Halting in random walk kernels. In Advances in neural information processing systems, pages 1639–1647.

[Sugiyama et al., 2017] Sugiyama, M., Ghisu, M. E., Llinares-López, F., and Borgwardt, K. (2017). graphkernels: R and python packages for graph comparison. Bioinformatics, 34(3):530–532.

[Togninalli et al., 2019] Togninalli, M., Ghisu, E., Llinares-López, F., Rieck, B., and Borgwardt, K. (2019). Wasserstein weisfeiler-lehman graph kernels. In Advances in Neural Information Processing Systems, pages 6439–6449.

[Trinajstic, 2018] Trinajstic, N. (2018). Chemical graph theory. Routledge.

[Vamathevan et al., 2019] Vamathevan, J., Clark, D., Czodrowski, P., Dunham, I., Ferran, E., Lee, G., Li, B., Madabhushi, A., Shah, P., Spitzer, M., et al. (2019). Applications of machine learning in drug discovery and development. Nature Reviews Drug Discovery, 18(6):463–477.

[Van Rossum and Drake, 2009] Van Rossum, G. and Drake, F. L. (2009). Python 3 Reference Manual. CreateSpace, Scotts Valley, CA.

[Van Rossum and Drake Jr, 1995] Van Rossum, G. and Drake Jr, F. L. (1995). Python reference manual. Centrum voor Wiskunde en Informatica Amsterdam.

[Virtanen et al., 2020] Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., et al. (2020). Scipy 1.0: fundamental algorithms for scientific computing in python. Nature methods, 17(3):261–272.

[Vishwanathan et al., 2010] Vishwanathan, S. V. N., Schraudolph, N. N., Kondor, R., and Borgwardt, K. M. (2010). Graph kernels. Journal of Machine Learning Research, 11(Apr):1201–1242.

[Wale et al., 2008] Wale, N., Watson, I. A., and Karypis, G. (2008). Comparison of descriptor spaces for chemical compound retrieval and classification. Knowledge and Information Systems, 14(3):347–375.

[Wang et al., 2018] Wang, H., Wang, J., Wang, J., Zhao, M., Zhang, W., Zhang, F., Xie, X., and Guo, M. (2018). Graphgan: Graph representation learning with generative adversarial nets. In Proceedings of the AAAI conference on artificial intelligence, volume 32.

[Wang, 2015] Wang, X. (2015). Graph based approaches for image segmentation and object tracking. PhD thesis, Ecully, Ecole centrale de Lyon.

[Wasserman et al., 1994] Wasserman, S., Faust, K., et al. (1994). Social network analysis: Methods and applications, volume 8. Cambridge university press.

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

[West et al., 2001] West, D. B. et al. (2001). Introduction to graph theory, volume 2. Prentice hall Upper Saddle River.

[Wills and Meyer, 2020] Wills, P. and Meyer, F. G. (2020). Metrics for graph comparison: A practitioner's guide. Plos one, 15(2):e0228728.

[Wilson et al., 2014] Wilson, R. C., Hancock, E. R., Pekalska, E., and Duin, R. P. (2014). Spherical and hyperbolic embeddings of data. IEEE transactions on pattern analysis and machine intelligence, 36(11):2255–2269.

[Winter et al., 2002] Winter, A., Kullbach, B., and Riediger, V. (2002). An overview of the gxl graph exchange language. In Software Visualization, pages 324–336. Springer.

[Wolpert and Macready, 1997] Wolpert, D. H. and Macready, W. G. (1997). No free lunch theorems for optimization. IEEE transactions on evolutionary computation, 1(1):67–82.

[Wu et al., 2020] Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., and Philip, S. Y. (2020). A comprehensive survey on graph neural networks. IEEE Transactions on Neural Networks and Learning Systems.

[Wu et al., 2018] Wu, Z., Ramsundar, B., Feinberg, E. N., Gomes, J., Geniesse, C., Pappu, A. S., Leswing, K., and Pande, V. (2018). Moleculenet: a benchmark for molecular machine learning. Chemical science, 9(2):513–530.

[Xu et al., 2018] Xu, K., Hu, W., Leskovec, J., and Jegelka, S. (2018). How powerful are graph neural networks? arXiv preprint arXiv:1810.00826.

[Xu et al., 2014] Xu, L., Wang, W., Alvarez, M., Cavazos, J., and Zhang, D. (2014). Parallelization of shortest path graph kernels on multi-core cpus and gpus. Proceedings of the Programmability Issues for Heterogeneous Multicores (MultiProg), Vienna, Austria.

[Yan et al., 2016] Yan, J., Yin, X.-C., Lin, W., Deng, C., Zha, H., and Yang, X. (2016). A short survey of recent advances in graph matching. In Proceedings of the 2016 ACM on International Conference on Multimedia Retrieval, pages 167–174.

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

[Yanardag and Vishwanathan, 2015b] Yanardag, P. and Vishwanathan, S. (2015b). A structural smoothing framework for robust graph comparison. In Advances in Neural Information Processing Systems, pages 2134–2142.

[Yao and Holder, 2014] Yao, Y. and Holder, L. (2014). Scalable svm-based classification in dynamic graphs. In 2014 IEEE international conference on Data Mining, pages 650–659. IEEE.

[You et al., 2018] You, J., Ying, R., Ren, X., Hamilton, W., and Leskovec, J. (2018). Graphrnn: Generating realistic graphs with deep auto-regressive models. In International Conference on Machine Learning, pages 5708–5717. PMLR.

[Zeng et al., 2009] Zeng, Z., Tung, A. K., Wang, J., Feng, J., and Zhou, L. (2009). Comparing stars: On approximating graph edit distance. Proceedings of the VLDB Endowment, 2(1):25–36.

[Zhang et al., 2020] Zhang, Z., Cui, P., and Zhu, W. (2020). Deep learning on graphs: A survey. IEEE Transactions on Knowledge and Data Engineering.

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

[Zhou et al., 2018a] Zhou, J., Cui, G., Zhang, Z., Yang, C., Liu, Z., Wang, L., Li, C., and Sun, M. (2018a). Graph neural networks: A review of methods and applications. arXiv preprint arXiv:1812.08434.

[Zhou et al., 2018b] Zhou, Q.-Y., Park, J., and Koltun, V. (2018b). Open3D: A modern library for 3D data processing. arXiv:1801.09847.

[Zhu and Honeine, 2017] Zhu, F. and Honeine, P. (2017). Online kernel nonnegative matrix factorization. Signal Processing, 131:143 – 153.