Graph kernels based on linear patterns: Theoretical and experimental comparisons

Linlin Jia, Benoit Gaüzère, Paul Honeine

▶ **To cite this version:**

Linlin Jia, Benoit Gaüzère, Paul Honeine. Graph kernels based on linear patterns: Theoretical and experimental comparisons. Expert Systems with Applications, 2022, 189, pp.116095. ⟨10.1016/j.eswa.2021.116095⟩. ⟨hal-03410508⟩

HAL Id: hal-03410508

https://normandie-univ.hal.science/hal-03410508v1

Submitted on 5 Jan 2024

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

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

Distributed under a Creative Commons CC BY-NC 4.0 - Attribution - Non-commercial use - International License

Version of Record: https://www.sciencedirect.com/science/article/pii/S0957417421014299
Manuscript_727adaf75a5ea21be0c10eccf91f8863

Graph kernels based on linear patterns: theoretical and experimental comparisons

Linlin Jia[a,*], Benoit Gaüzère[a] and Paul Honeine[b]

[a]LITIS, Institut National des Sciences Appliquées Rouen Normandie, 685 Avenue de l'Université, 76800 Saint-Étienne-du-Rouvray, France
[b]LITIS, Université de Rouen Normandie, Avenue de l'Université, 76800 Saint-Étienne-du-Rouvray, France

ARTICLE INFO

Keywords:
Graph kernels
Walks
Paths
Kernel methods
Graph representation

ABSTRACT

Graph kernels are powerful tools to bridge the gap between machine learning and data encoded as graphs. Most graph kernels are based on the decomposition of graphs into a set of patterns. The similarity between two graphs is then deduced to the similarity between corresponding patterns. Kernels based on linear patterns constitute a good trade-off between accuracy and computational complexity. In this work, we propose a thorough investigation and comparison of graph kernels based on different linear patterns, namely walks and paths. First, all these kernels are explored in detail, including their mathematical foundations, structures of patterns and computational complexity. After that, experiments are performed on various benchmark 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. Finally, for regression and classification tasks, accuracy and computational complexity of these kernels are compared and analyzed, in the light of baseline kernels based on non-linear patterns. Suggestions are proposed to choose kernels according to the types of graph datasets. This work leads to a clear comparison of strengths and weaknesses of these kernels. An open-source Python library containing an implementation of all discussed kernels is publicly available on GitHub to the community, thus allowing to promote and facilitate the use of graph kernels in machine learning problems.

1. Introduction

Machine learning algorithms have been conventionally defined on vector spaces, allowing to take advantage of the easiness in linear algebra operations. However, it is challenging to vectorize many data types due to their complex structures. Graphs are able to model a wide range of real-world data, by encoding elements as well as the relationship between them. Due to these properties, graph representation has broad applications in wide domains, such as 2D and 3D image analysis, document processing, bioinformatics, chemoinformatics, web data mining, etc., where it models structures such as molecules, social networks, and state transition (Conte, Foggia, Sansone and Vento, 2004).

It is natural to raise the problem of applying machine learning methods for graph data, in order to unleash the power of these two powerful tools. To achieve this goal, 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. When machine learning algorithms rely on (dis-) similarity measures between data, the problem boils down to measuring the similarity between graphs. Graph similarity measures can be roughly grouped in 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, Schöning and Torán, 2012). Unfortunately, the exact similarity cannot be computed in polynomial time by these methods; hence it is not practical for real-world data.

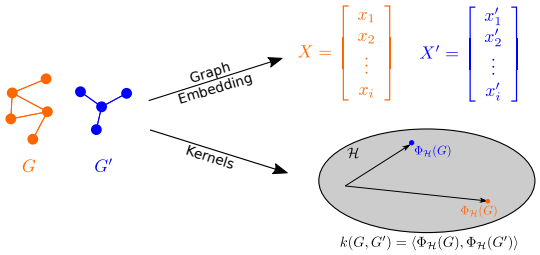

Figure 1: Illustrative comparison between graph embedding and kernels, for two arbitrary graphs G and G'. Through graph embedding, the two graphs are represented by two vectors, X and X'. By kernels, the two graphs are implicitly embedded by a function $\Phi_{\mathcal{H}}(\cdot)$ into a Hilbert space $\mathcal{H}$, yielding $\Phi_{\mathcal{H}}(G)$ and $\Phi_{\mathcal{H}}(G')$; moreover, their inner product $\langle \Phi_{\mathcal{H}}(G), \Phi_{\mathcal{H}}(G') \rangle$ is easily computed using a kernel function $k(G, G')$.

Inexact similarity measures are commonly applied for graphs, in which category graph embedding and graph kernels lie. 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.

Graph embedding explicitly computes vectors that encode some information of the graphs; while kernels allow an implicit embedding by representing graphs in a possibly infinite-dimension feature space which relaxes the limitations on the encoded information. The two strategies are illustrated in Figure 1. Indeed, as generalizations of the scalar inner product, kernels are natural similarity measures be-

*Corresponding author
linlin.jia@insa-rouen.fr (L. Jia); benoit.gauzere@insa-rouen.fr (B. Gaüzère); paul.honeine@univ-rouen.fr (P. Honeine)
ORCID(s): 0000-0002-3834-1498 (L. Jia)

© 2021 published by Elsevier. This manuscript is made available under the CC BY NC user license
https://creativecommons.org/licenses/by-nc/4.0/

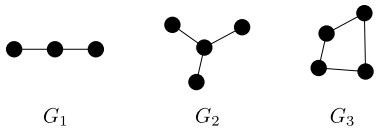

Figure 2: Different types of graph patterns. G_1, G_2, G_3 are examples of linear patterns, non-linear (acyclic) patterns and cyclic patterns, respectively.

tween 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. Kernels have been widely applied in machine learning, with well-known popular methods, such as Support Vector Machines. Therefore, defining kernels between graphs is a powerful design to bridge the gap between machine learning and data encoded as graphs.

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 2. 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. Linear patterns require lower computational complexity than non-linear patterns in some occasions. Nevertheless, non-linear patterns normally include or imply the linear ones. For example, the treelet pattern is non-linear as a whole, while treelets whose maximal size is less than 4 are linear (Gaüzere, Brun and Villemin, 2012). Moreover, it could be intractable to compute non-linear or cyclic based kernels on large graphs. Therefore in this article, we focus on studying and comparing graph kernels based on different linear patterns.

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. Kernels based on walks proposed include the common walk kernel (Gärtner, Flach and Wrobel, 2003), the marginalized kernel (Kashima, Tsuda and Inokuchi, 2003) and the generalized random walk kernel Vishwanathan, Schraudolph, Kondor and Borgwardt (2010). Meanwhile, the shortest path kernel (Borgwardt and Kriegel, 2005), the structural shortest path kernel (Suard, Rakotomamonjy and Bensrhair, 2007) and the path kernel up to length h (Ralaivola, Swamidass, Saigo and Baldi, 2005) are constructed based on paths, which are relieved from artifacts brought by walks due to tottering and halting (see Section 3.1.4). More recently, many developments have been carried out to enhance these graph kernels (Aziz, Wilson and Hancock, 2013; Xu, Wang, Alvarez, Cavazos and Zhang, 2014; Sugiyama and Borgwardt, 2015).

The main contributions of this article are studying graph kernels based on linear patterns, with an emphasis on the aforementioned kernels, and comparing them theoretically and experimentally. Among them, the generalized random walk kernel is split into four different kernels due to the computing methods they use. 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. In the exhaustive experimental analysis conducted in this paper, each kernel is applied on various datasets exhibiting different types of graphs, and a thorough performance analysis 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. Finally, all the implementations are publicly available as an open-source Python library on GitHub[1]. In this library, every kernel is able to tackle different types of graphs, and several computation methods are provided for kernels. Moreover, we propose several advanced methods to reduce the computational complexity of the implemented graph kernels, by both accelerating computation and reducing storage requirements.

The paper is organized as follows: Section 2 introduces preliminaries for graph and kernels in machine learning. Section 3 presents detailed discussions on each graph kernel. Experiments and analyses are performed in Section 4. Finally, Section 5 concludes this work.

2. Preliminaries

2.1. Basic concepts of graph theory

In the following, we define notations that will be used in this paper. For more details, we refer interested readers to (West et al., 2001). First, we clarify definitions of different types of graphs. Figure 3 shows types of graphs mentioned below. Let $|\cdot|$ denote the cardinality of a set, namely the number of its elements. The indicator function $\mathbf{1}_A : X \to \{0, 1\}$ is defined as $\mathbf{1}_A(x) = 1$ if $x \in A$, and 0 otherwise.

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 $E \subset V \times V$ corresponds to a set of edges. u is adjacent to v if $(u, v) \in E$. We denote the number of graph vertices as n, *i.e.*, $n = |V|$, and the number of graph edges as m, *i.e.*, $m = |E|$. 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 $\ell_e : E \to L$ assigns labels to edges only; in vertex-labeled graphs, the labeling function $\ell_v : V \to L$ assigns labels to vertices only; in fully-labeled graphs, the labeling

[1] The GitHub link is *https://github.com/jajupmochi/py-graph*.

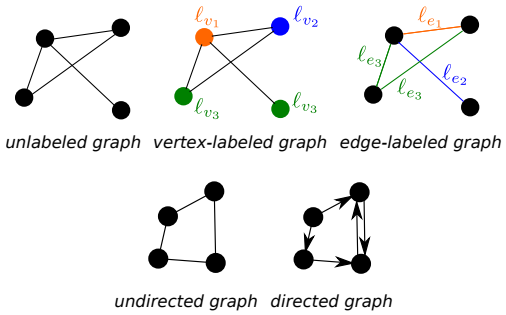

Figure 3: The different types of graphs. In vertex- and edge-labeled graphs, vertices and edges with different labels are distinguished by color in the figure.

function $\ell_f : V \cup E \to L$ assigns labels to both vertices and edges. Unlabeled graphs have no such labeling function. Labels can be either symbolic or non-symbolic, for vertices and/or edges. A symbolic label is a discrete symbol, such as the type of atoms or chemical bonds; a non-symbolic label is a continuous value. Due to this difference, symbolic labels are considered equal as long as they are the same and unequal otherwise (namely the Kronecker delta function; see below), while non-symbolic labels are compared by continuous measures, for instance, the Gaussian kernel (see 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. The Kronecker delta function between two labels ℓ_i and ℓ_j is defined as $k(\ell_i, \ell_j) = 1$ if $\ell_i = \ell_j$, and 0 otherwise. For the sake of conciseness, it is denoted as the delta function $\delta_{\ell_i \ell_j}$. The Gaussian kernel between labels ℓ_i and ℓ_j of L is defined as $k(\ell_i, \ell_j) = \exp\left(-\frac{\|\ell_i - \ell_j\|^2}{2\delta^2}\right)$, where δ is the tunable bandwidth parameter.

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 $(u, v) \in E$, then $(v, u) \in E$.

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. In a graph $G = (V, E)$, a neighbor of a vertex $v \in V$ is a vertex u that meets the condition $(u, v) \in E$. If G is undirected, the degree of a vertex $v \in V$ is the number of these neighbors, namely $|\{u \in V \mid (u, v) \in E\}|$; if G is directed, then $|\{u^- \in V \mid (u^-, v) \in E\}|$ is called the indegree of vertex v, $|\{u^+ \in V \mid (v, u^+) \in E\}|$ is 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. The adjacency matrix of an n-vertex graph $G = (V, E)$ is an $n \times n$ matrix $A(G) = [a_{ij}]$, where $a_{ij} = \mathbf{1}_E((v_i, v_j))$, namely $a_{ij} = 1$ if $(v_i, v_j) \in E$ and 0 otherwise. For a graph $G = (V, E)$, a walk of length h is a sequence of vertices $W = (v_1, v_2, \ldots, v_{h+1})$ where $(v_i, v_{i+1}) \in E$ 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. 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

In this section, formal definitions of a kernel and Gram matrix are first 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 paper to assess the relevance of graphs. For more details on kernel methods, we refer interested readers to (Shawe-Taylor and Cristianini, 2004; Schölkopf and Smola, 2002). Let $\mathcal{X}$ denotes the input space.

Definition 1 (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 $\sum_{i=1}^{n} \sum_{j=1}^{n} c_i\, c_j\, k(x_i, x_j) \geq 0$, 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 one is applied for all six graph kernels discussed in this paper, 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 $(x_i, x_j) \in \mathcal{X}^2$:

$$k(x_i, x_j) = \langle \Phi_{\mathcal{H}}(x_i), \Phi_{\mathcal{H}}(x_j) \rangle_{\mathcal{H}}, \tag{1}$$

where $\Phi_{\mathcal{H}} : \mathcal{X} \to \mathcal{H}$ is an embedding function. The positive semi-definiteness of the 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 paper.

Kernel based methods in machine learning take advantage of Mercer's theorem, in order to transform conventional

linear models into non-linear ones, by replacing classical inner products between data with a non-linear kernel. Let $X = \{x_1, x_2, \dots, x_N\}$ be the finite dataset of N samples available for training the machine learning method, with associated data labels $\{y_1, y_2, \dots, y_N\}$ where $y_i \in \{-1, +1\}$ for binary classification and $y_i \in \mathbb{R}$ for regression (extensions to multiclass classification and vector output is straightforward (Honeine, Noumir and Richard, 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. A Gram matrix K associated to a kernel k for a training set X is an $N \times N$ matrix defined as $K_{i,j} = k(x_i, x_j)$, for all $(x_i, x_j) \in \mathcal{X}^2$.

Kernel-based machine learning relies on regularized cost functions, namely $\arg\min_{\psi \in \mathcal{H}} \sum_{i=1}^{N} c(y_i, \psi(x_i)) + \lambda \|\psi\|^2$, for some cost function c and positive regularization parameter λ. The generalized representer theorem (Schölkopf, Herbrich and Smola, 2001) states that the optimal solution has the form $\psi(x) = \sum_{i=1}^{N} \omega_i k(x_i, x)$, where k is the kernel inducing the Hilbert space $\mathcal{H}$. The coefficients ω_i's are obtained using the Gram matrix. 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 $[\omega_1 \dots \omega_N]^\top = (K + \lambda I)^{-1}[y_1 \dots y_N]^\top$ (Murphy, 2012). Support Vector Machines (SVM) for classification considers 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, Guyon and Vapnik, 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.

2.3. Pattern-based kernels

R-convolution kernels propose a way to measure similarity between two objects by measuring the similarities between their substructures (Haussler, 1999). Suppose each sample $x_i \in \mathcal{X}$ has a composite structure, namely described by its "parts" $(x_{i1}, x_{i2}, \dots, x_{iD}) \in \mathcal{X}_1 \times \mathcal{X}_2 \times \cdots \times \mathcal{X}_D$, for some positive integer D. Since multiple decompositions could exist, let $R(x)$ denotes all possible decompositions of x, namely $R(x) = \left\{ \boldsymbol{x} \in \mathcal{X}_1 \times \mathcal{X}_2 \times \cdots \times \mathcal{X}_D \;\middle|\; \boldsymbol{x} \text{ is a decomposition of } x \right\}$. 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:

$$k(x_i, x_j) = \sum_{\substack{(x_{i1}, \dots, x_{iD}) \in R(x_i) \\ (x_{j1}, \dots, x_{jD}) \in R(x_j)}} \prod_{d=1}^{D} k_d(x_{id}, x_{jd}). \tag{2}$$

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 more easy to compare them. The kernels differ mainly in the ways of composition and the similarity measures used to compare the substructures.

3. Graph Kernels Based on Linear Patterns

The fundamental class of graph kernels based on bags of patterns are based on walks and paths as elementary decompositions. This section studies their mathematical representations and compares their computational complexities. Table 1 provides an insight into the characteristics of theses kernels.

3.1. Graph kernels based on walks

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

3.1.1. 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). Despite that sometimes it is referred to as the random walk kernel (Vishwanathan et al., 2010; Borgwardt and Kriegel, 2005), there is actually no stochastic process applied.

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 then defined as following:

Definition 2 (direct product graph). *The direct product graph of two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, denoted $G_\times = G_1 \times G_2$, is defined by*

$$\begin{cases} V_\times(G_1 \times G_2) = \left\{ (v_1, v_2) \in V_1 \times V_2 \;\middle|\; \ell_v(v_1) = \ell_v(v_2) \right\} \\ E_\times(G_1 \times G_2) = \Big\{ ((u_1, u_2), (v_1, v_2)) \in V^2(G_1 \times G_2) \;\Big|\; \\ \qquad (u_1, v_1) \in E_1 \wedge (u_2, v_2) \in E_2 \wedge \ell_e(u_1, v_1) \\ \qquad = \ell_e(u_2, v_2) \Big\}, \end{cases} \tag{3}$$

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 $G_\times$, and an edge between two vertices in graph $G_\times$

Table 1
Characteristics of graph kernels based on linear patterns, and two on non-linear patterns

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

The "Computational 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, Neumann, Kersting and Mutzel, 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 (4) is set to γ^h when constructing the kernel by geometric series (see Section 3.1.1).

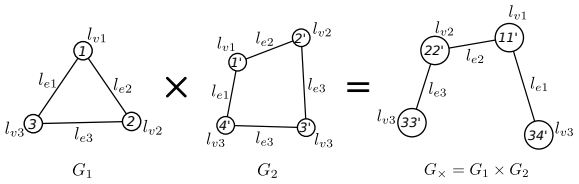

Figure 4: Direct product of fully-labeled graphs

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 4 illustrates the direct product $G_1 \times G_2$ of two fully-labeled graphs G_1 and G_2, where $\{\ell_{v_i} \mid i = 1, 2, \dots\}$ denotes the set of vertex labels and $\{\ell_{e_i} \mid i = 1, 2, \dots\}$ 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 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 is then designed this way.

Definition 3 (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*

$$k_\times(G_1, G_2) = \sum_{i,j=1}^{|V_\times|} \left[\sum_{h=1}^{\infty} \lambda_h A_\times^h \right]_{ij}, \tag{4}$$

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 calculated component-wise in linear time. Diagonalizing matrix $A_\times$ has roughly a cubic computational 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 $I + \gamma^1 A^1 + \gamma^2 A^2 + \dots$ The limit of this series can be computed by inverting the matrix $I - \gamma A$, which is roughly of cubic computational 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, excessively restraining the effect of long walk sequences on the kernel (see Section 3.1.4). 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.

3.1.2. Marginalized kernel

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

$$k(G_1, G_2) = \sum_{w_1 \in W(G_1)} \sum_{w_2 \in W(G_2)} k_W(w_1, w_2) \cdot p_{G_1}(w_1) p_{G_2}(w_2), \tag{5}$$

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. $p_G(w)$ is the probability of traversing walk w in G. If $w = (v_1, v_2, \dots, v_h)$, then

$$p_G(w) = p_0(v_1) \prod_{i=2}^{h} p_t(v_i|v_{i-1}) p_q(v_h), \tag{6}$$

where $p_0(v)$ is the initial probability distribution, indicates the probability that walk w starts from vertex v; $p_t(v_i|v_{i-1})$ is the transition probability on vertex v_{i-1}, describes the probability of choosing v_i as the next vertex of v_{i-1}; 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, $p_t(v_i|v_{i-1})$ to be uniform over all neighbors of vertex v_{i-1}, and p_q to be a constant.

Kashima et al. (2003) proposed an efficient method to compute this kernel as

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

where $s(v_1, v_1') = p_0(v_1) p_0'(v_1') k_v(v_1, v_1')$ and $R_h(v_1, v_1')$ is updated recursively in r iterations with $R_h(v_1, v_1') = r_1(v_1, v_1') + \sum_{v_i \in V} \sum_{v_j' \in V'} t(v_i, v_j', v_1, v_1') R_{h-1}(v_i, v_j')$, where

$$\begin{cases} r_1(v_1, v_1') = q(v_1, v_1') = R_1(v_1, v_1'), \\ q(v_h, v_h') = p_q(v_h) p_q(v_h') \\ t(v_i, v_i', v_{i-1}, v_{i-1}') = p_t(v_i|v_{i-1}) p_t'(v_i'|v_{i-1}') k_v(v_i, v_i') \\ \qquad k_e(e_{v_{i-1}, v_i}, e'_{v_{i-1}', v_i'}). \end{cases}$$

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

3.1.3. Generalized random walk kernel

The generalized random walk kernel, as a unified framework for random walk kernels, was proposed by 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, Haffner and Mohri, 2004) also boils 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 marginalized kernels. Then, a random walk generates a sequence of vertices $v_{i_1}, v_{i_2}, v_{i_3}, \dots$ according to a conditional probability $p(i_{k+1} \mid i_k) = A_{i_{k+1}, i_k}$, where A is the normalized adjacency matrix of the graph. This probability plays a similar role as the transition probability p_t of the marginalized kernel, which 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 $p_t(i_{k+1} \mid i_k) = w_{i_{k+1}} / \sum_j w_j, j \in N(i_k)$, where $N(i_k)$ is the set of neighbors of the vertex i_k. The edge weight here is a special label which represents the transition probability from one vertex to another rather than a property of the edge itself. Finally, like termination probability p_q of marginalized kernels, a stopping probability distribution $q = (q_{i_1}, q_{i_2}, \dots), n_{i_k} \in V$ is associated with a graph $G = (V, E)$ over all its vertices, which models the phenomenon where a random walk stops at vertex n_{i_k}. Both the initial probability and the stopping probability are practically set as uniform distribution.

Like 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 2. It is worth noting that the direct product graph of unlabeled graphs often has much more edges than the labeled one. The generalized random walk kernel between two graphs G_1 and G_2 is defined in (Vishwanathan et al., 2010) as

$$k(G_1, G_2) = \sum_{h=0}^{\infty} f(h) q_\times^\top W_\times^h p_\times. \tag{7}$$

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 $G_\times$, respectively, where operator $\otimes$ denotes the Kronecker product and $W_\times \in \mathbb{R}^{n \times n}$ is the weight matrix.

Assuming the initial and the stopping probability distributions to be uniform and setting $W_\times$ as the unnormalized adjacency matrix of $G_\times$, (7) can be transformed to the common walk kernel. Meanwhile, applying $f(h) = 1$ and $W_\times$ as a specific form, the marginalized kernel can be recovered from (7).

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

The *Sylvester equation method* is based on the generalized Sylvester equation $M = \sum_{i=1}^{d} S_i M T_i + M_0$. For graphs with symbolic edge labels, when $f(h) = \lambda_h$, the kernel in (7) can be computed by $q_\times^\top vec(M)$, with $vec(\cdot)$ the column-stacking operator and M the solution of the generalized Sylvester equation $M = \sum_{i=1}^{d} \lambda\, {}^iA_2 M\, {}^iA_1^\top + M_0$, where d is the number of different edge labels, $vec(M_0) = p_\times$ and ${}^iA$ is the normalized adjacency matrix of graph G filtered by the i-th edge label ${}^i\ell_e$ of G; namely, ${}^iA_{jk} = A_{jk}$ 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 computational 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 size of n^2 and can be precomputed for each graph. Besides, computing $q_\times^\top vec(M)$ requires $\mathcal{O}(N^2n^2)$ time. Thus for N unlabeled graphs, the complexity of computing the corresponding Gram matrix is $\mathcal{O}(N^2n^3)$, 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.

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

The third method, *fixed-point iterations*, rewrites $(I - \lambda W_\times)x = p_\times$ as $x = p_\times + \lambda W_\times x$, and computes x by finding a fixed point of the equation by iterating $x_{t+1} = p_\times + \lambda W_\times x_t$. The worst-case computational complexity is $\mathcal{O}(rn^4)$ 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 $\mathcal{O}(N^2n^2 + Nn^3)$ for N graphs.

The generalized random walk kernel provides a quite flexible framework for walk based kernels (see (7)). However, the problems lie in the methods to compute the kernel as introduced above. Despite of the improvement on computational 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 (through possibly edge-weighted). 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 (7). The spectral decomposition method works on any $f(h)$ that makes (7) converge, but is only efficient for unlabeled graphs.

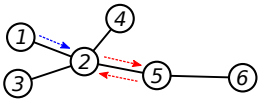

Figure 5: A tottering example: walk (v_1, v_2, v_5, v_2) has tottering between vertices v_2 and v_5, *i.e.*, (v_2, v_5, v_2).

For conciseness in this paper, 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.

In contrast with the classic walk patterns, a series of graph kernels have been proposed based on quantum walks. Bai, Rossi, Torsello and Hancock (2015) and Rossi, Torsello and Hancock (2013) have introduced a quantum graph kernel using the quantum Jensen-Shannon divergence and continunous-time quantum walks. Bai, Rossi, Cui, Zhang, Ren, Bai and Hancock (2017) and Bai, Rossi, Cui, Cheng and Hancock (2019) have extended this kernel to discrete-time quantum walks and for complete weighted graphs, while Minello, Rossi and Torsello (2019) have considered both directed and undirected graphs and integrated node-level topological information while constructing the kernel. The quantum walk kernels are based on the computation of the mutual information between a pair of graphs and their composite graph such as quantum Jensen-Shannon divergence, rather than the similarities between isomorphic subpatterns as R-convolution kernels. Due to this reason, we do not examine quantum walk kernels in this paper.

3.1.4. Problems raised by walks

There are two problems that may lead to worse 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 Figure 5 shows, a tottering brings unnecessary structure to the pattern and may worsen the performance of graph kernels.

Mahé, Ueda, Akutsu, Perret and Vert (2004) propose a technique to avoid this problem for the marginalized kernel. It first transforms each graph $G = (V, E)$ to $G' = (V', E')$,

with

$$\begin{cases} V' = V \cup E \\ E' = \left\{ (v,(v,t)) \;\middle|\; v \in V, (v,t) \in E \right\} \\ \qquad \cup \left\{ ((u,v),(v,t)) \;\middle|\; (u,v),(v,t) \in E, u \neq t \right\} \end{cases} \tag{8}$$

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 $\ell'_e(e') = \ell_e(v'_2)$. Then it computes the marginalized kernel between transformed graphs. This extension is able to remove tottering from walks for the marginalized kernel, hence enhances the performance of the kernel. However, this improvement is only minor according to the experiments (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 worse case computational complexity of the kernel is $\mathcal{O}((n+m)^2)$, which is not practical for graphs with high average vertex degree. For all these reasons, experiments conducted in Section 4 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 3.1.1, 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 is degenerated 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.

3.2. Graph kernels based on paths

3.2.1. 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 there is an edge between all vertices which 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 4 (shortest path kernel). *Let $S_1 = (V_1, E_1)$ and $S_2 = (V_2, E_2)$ 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*

$$k_{sp}(G_1, G_2) = \sum_{e_1 \in E_1} \sum_{e_2 \in E_2} k_w(e_1, e_2), \tag{9}$$

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 paper, 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 S contains n^2 edges. Assuming that vertex kernels and edge kernels are computed in $\mathcal{O}(1)$, then pairwise comparison of all edges in two shortest-paths graphs requires a computational complexity of $\mathcal{O}(n^4)$, which is also the complexity to compute the shortest path kernel.

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

1) 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.

2) It cannot deal with graphs whose edges bear 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).

3) 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 which is a product of two kernels: the delta function for symbolic vertex labels and the Gaussian kernel for non-symbolic vertex attributes.

3.2.2. 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 the 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:

$$k_p(p, p') = k_v(\ell_v(v_1), \ell_v(v'_1)) \cdot \prod_{i=2}^{n} k_e(\ell_e(v_{i-1}, v_i), \ell_e(v'_{i-1}, v'_i)) k_v(\ell_v(v_i), \ell_v(v'_i)), \tag{10}$$

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 (10). Here we use the simple and straightforward mean average kernel:

$$k_{ssp}(G_1, G_2) = \frac{1}{n_1} \frac{1}{n_2} \sum_{p_i \in P_1} \sum_{p_j \in P_2} k_p(p_i, p_j), \tag{11}$$

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 computational 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 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.

3.2.3. 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

$$k_{ph}(G_1, G_2) = \sum_{p \in P(G_1) \cup P(G_2)} \phi_p(G_1) \phi_p(G_2), \tag{12}$$

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

Based on the definitions of $\phi_p(G)$, different types of path kernels can be constructed. The Tanimoto kernel, based on the binary feature map, is defined as

$$k^t_{ph}(G_1, G_2) = \frac{k_{ph}(G_1, G_2)}{k_{ph}(G_1, G_1) + k_{ph}(G_2, G_2) - k_{ph}(G_1, G_2)},$$

where $k_{ph}(G_1, G_2)$ is the kernel defined as (12) corresponding to the binary feature map. When $\phi_p(G)$ takes the form of the counting feature map, then the MiniMax kernel can be constructed as

$$k^m_{ph}(G_1, G_2) = \frac{\sum_{p \in P(G_1) \cup P(G_2)} \min(\phi_p(G_1), \phi_p(G_2))}{\sum_{p \in P(G_1) \cup P(G_2)} \max(\phi_p(G_1), \phi_p(G_2))}.$$

These two kernels are related to the Tanimoto similarity measure in the chemistry literature, and provide normalization for the path kernel. While the MiniMax 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 up 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 direct saving, especially when label set is small (Fredkin, 1960). Thus, the path kernel between two graphs is computed in $\mathcal{O}(h^2 n^2 d^{2h})$.

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.

In order to analyze and compare the performance of the aforementioned graph kernels, we implemented them in Python. Several methods are applied to accelerate the computation of the kernels in the implementation.

4. Experiments and Results

In this section, we first introduce several benchmark datasets corresponding to different types of graphs. These types include labeled and unlabeled graphs, with symbolic and non-symbolic attributes, different average vertex numbers, different average vertex degrees, linear, non-linear and cyclic patterns. Then we introduce the computational settings. Finally, we perform each graph kernel on each dataset and analyze the accuracy and computational complexity according to the types of graphs, offer advice to choose graph kernels based on the type of datasets and discuss which ones work on particular graphs.

4.1. Datasets and settings

We examine 11 well-known benchmark datasets for both regression and classification tasks. *Alkane* (Cherqaoui

Table 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
Letter-med	✓	✓	✓	✗	✗	2	✗	✗	2250	4.67	3.21	1.35	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
D&D	✓	✓	✓	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").

and Villemin, 1994), *Acyclic* (Cherqaoui, Villemin, Mesbah, Cense and Kvasnicka, 1994), *PAH* (Brun, 2018), *Mutag* (Debnath, Lopez de Compadre, Debnath, Shusterman and Hansch, 1991), *AIDS* (Riesen and Bunke, 2008), *NCI1* and *NCI109* (Wale, Watson and Karypis, 2008) are chemical compounds; *MAO* (Brun, 2018), *Enzymes* (Schomburg, Chang, Ebeling, Gremse, Heldt, Huhn and Schomburg, 2004; Borgwardt, Ong, Schönauer, Vishwanathan, Smola and Kriegel, 2005) and *D&D* (Dobson and Doig, 2003) are enzymes and proteins; *Letter-med* (Riesen and Bunke, 2008) involves graphs of distorted letter drawings. See (Kersting, Kriege, Morris, Mutzel and Neumann, 2016) for more details. These datasets are chosen as they come from different fields, such as bioinformatics and handwriting recognition, and have different properties that allow to provide an extensive analysis of graph kernels. Table 2 outlines the properties of these datasets.

The criteria used for prediction are SVM for classification and kernel ridge regression for regression. A two-layer nested cross validation (CV) method is applied to select and evaluate models as follows. In the outer CV, the whole dataset is first randomly split into 10 folds, nine of which serve for model validation and one for an unbiased estimate of the accuracy. Then, in the inner CV, the validation set is split into 10 folds, nine of which are used for training, and the remaining split is used for evaluating 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.

The machine used to execute the experiments is a cluster with 28 CPU cores of Intel(R) Xeon(R) E5-2680 v4 @ 2.40GHz, 252GB memory, and 64-bit operating system CentOS Linux release 7.3.1611. All results were run with Python 3.5.2.

4.2. Performance analysis

Tables 3 and 4 gather the performances of all these kernels on all datasets for regression and classification tasks, respectively. 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 of that of the fastest kernels. For even larger datasets, such as *NCI1*, *NCI109* and *D&D*, 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 better considered baselines to test the performance of new 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 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 more structure information into consideration than the former one, thus 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 *D&D*, whose average number of vertices is 284.32.

Tables 3 and 4 exhibit additionally performance of

Table 3
Results of all graph kernels on datasets for regression tasks

Datasets	Kernels	Train Perf	Valid Perf	Test Perf	Parameters	t_{gm}	t_{all}	
Alkane	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*
	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*
Acyclic	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*
	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-9.5, ..., 1e10]. Ranges of all the parameters can be found in the demos of our Python library. t_{gm} is the time to compute Gram matrix/matrices in seconds. Note for kernels which need to tune hyper-parameters that are required 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.

two well-known graph kernels based on non-linear patterns, namely the treelet kernel (Gaüzere, Grenier, Brun and Villemin, 2015) (see also (Bougleux, Dupé, Brun and Mokhtari, 2012; Gaüzere et al., 2012)) and the Weisfeiler-Lehman (WL) subtree kernel (Shervashidze, Schweitzer, Leeuwen, Mehlhorn and Borgwardt, 2011) (see also (Morris, Kersting and Mutzel, 2017)). Several graph kernels based on linear patterns, especially paths, provide competitive or even higher accuracies than these two kernels. On *MAO* dataset, the common walk kernel achieves 93% accuracy (Table 4, Line 1), which is comparable to the accuracy of the WL subtree kernel (93.05%, Table 4, Line 11) and is higher than that of the treelet kernel (91.19%, Table 4, Line 10). The shortest path kernel achieves the highest accuracy on dataset *Enzymes* (70.09%, Table 4, Line 49), which is about 20% higher than the treelet kernel and the WL subtree kernel (Table 4, Lines 51 and 52). The structural shortest path kernel has the equivalent accuracy as the WL subtree kernel on *PAH* (Table 4, 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 *D&D*. On the *AIDS* dataset, all exhibited kernels have comparable accuracies (Table 4, 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 (see (Morris, Kriege, Kersting and Mutzel, 2016) for instance), 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 will remain the same, and the aforementioned analyses still stand.

According to average graph vertex numbers $\bar{n}$, datasets of classification tasks in Table 2 can be classified into small graphs (including *Letter-med*), big graphs (including *D&D*) and medium graphs (including all the rests). Figures 6(a)(b)(c) exhibit the time complexity and classification accuracies of all kernels on these datasets. For small datasets (*Letter-med*), the kernels based on shortest paths, the conjugate gradient kernel and the 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 *Letter-med* dataset (Figure 6(a)). As sizes of graphs grow, kernels based on walks, paths and non-linear patterns may all have good trade-offs between computational complexity and accuracy for certain datasets (Figure 6(b)). Accuracies of the latter two groups of kernels are higher in general. On the big dataset *D&D*, the path kernel up to length h performs better than the WL subtree kernel (Figure 6(c)).

Figure 6(d) compares the average of computational complexity and classification accuracies over all classification datasets of each kernel. Note that for datasets *Letter-med* and *Enzymes*, kernels that cannot tackle non-symbolic labels are omitted. We provide general conclusions on these graph kernel. From a global viewpoint, all kernels provide a good accuracy on all datasets. We can see that the marginalized kernel has the worse 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 good compromise between time complexity and ac-

Table 4
Results of all graph kernels for classification tasks (accuracy in percentage)

Datasets	Kernels	Train Perf	Valid Perf	Test Perf	Parameters	t_{gm}	t_{all}	
MAO	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"	*1*
	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"	*2*
	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"	*3*
	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"	*4*
	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"	*5*
	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"	*6*
	Shortest path	97.43±0.76	88.51±2.10	87.81±7.38	C: 3.16e+3	1.79"	**3.82"**	*7*
	Structural SP	96.70±0.76	90.79±2.44	91.62±9.16	C: 1e+3	7.63"	9.60"	*8*
	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"	*9*
	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"	*10*
	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"	*11*
PAH	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"	*12*
	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"	*13*
	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"	*14*
	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"	*15*
	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"**	*16*
	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"	*17*
	Shortest path	79.53±1.26	76.66±2.55	69.40±11.57	C: 3.16e+2	2.30"	329.48"	*18*
	Structural SP	77.39±1.85	74.22±2.50	**74.50±13.39**	C: 3.16e+2	20.91"	776.99"	*19*
	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"	*20*
	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"	*21*
	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"**	*22*
Mutag	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"	*23*
	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"	*24*
	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"	*25*
	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"	*26*
	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"	*27*
	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"	*28*
	Shortest path	98.23±0.40	84.39±2.35	81.84±6.63	C: 1e+3	4.89"	**7.58"**	*29*
	Structural SP	100.00±0.00	84.66±1.57	86.26±5.14	C: 3.16e+9	68.85"	71.18"	*30*
	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"	*31*
	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"	*32*
	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"	*33*
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"	*34*
	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"	*35*
	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"	*36*
	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"	*37*
	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"	*38*
	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"	*39*
	Shortest path	98.96±0.07	93.87±0.29	93.72±1.12	C: 10	36.98"	**255.48"**	*40*
	Structural SP	99.10±0.08	94.84±0.23	**94.88±1.24**	C: 10	41.92"	**257.96"**	*41*
	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"	*42*
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"	*43*
	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"	*44*
	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"	*45*
	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"	*46*
	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"	*47*
	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"	*48*
	Shortest path	100.00±0.00	68.86±1.91	**70.09±4.20**	C: 1e+6	704.54"	717.35"	*49*
	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"	*50*
	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"	*51*
	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"**	*52*
AIDS	Shortest path	99.91±0.02	99.13±0.11	99.26±0.55	C: 10	892.26"	**994.27"**	*53*
	Structural SP	99.80±0.03	98.90±0.10	98.84±0.63	C: 3.16	8021.98"	8125.28"	*54*
	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"	*55*
	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"	*56*
	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"	*57*
NCI1	Structural SP	92.75±0.13	80.13±3.86	79.88±1.71	C: 3.16e+2	132848.44"	135483.96"	*58*
	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"	*59*
	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"**	*60*
	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"	*61*
NCI109	Structural SP	89.16±0.21	78.89±0.40	79.04±1.80	C: 10	134539.59"	141844.22"	*62*
	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"	*63*
	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"**	*64*
	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"	*65*
D&D	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"**	*66*
	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"	*67*

Same legends as Table 3. For the large-scale datasets, graph kernels are neglected if their time or memory consumption are much higher than other kernels.

curacy. 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. 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.

4.2.1. Labeled and unlabeled graphs

To study the influence of labeling on performance of graph kernels, we examine 3 datasets that have similar prop-

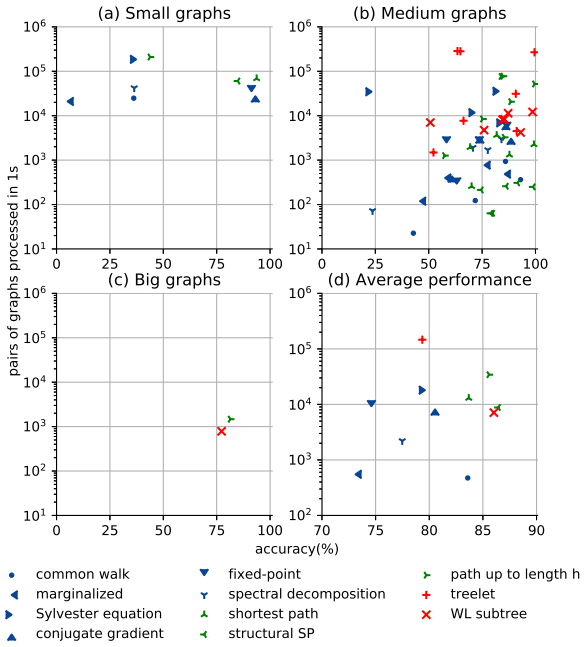

Figure 6: Comparison of computational complexity versus accuracies of all graph kernels on small (a), medium (b) and big (c) datasets, as well as the average performance of each kernel over all datasets (d). Markers correspond to different kernels; Colors blue, green and red depict graph kernels based on walks, paths and non-linear patterns, respectively.

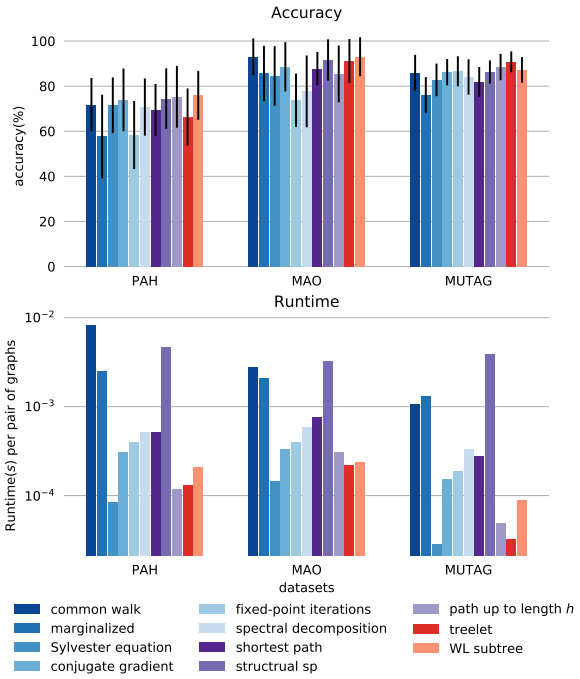

Figure 7: Comparison of accuracy and runtime of all kernels on unlabeled (*PAH*) and labeled datasets (*MAO, Mutag*). Accuracy is the mean value on 30 trials (pillars), with confidence interval around it (error bars).

erties (*e.g.* $\bar{n}$, $\bar{m}$ and d in Table 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 7 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 accuracy 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 on *MUTAG* is the shortest and on *PAH* is the longest due to the different average number of graph vertices of each graph. The runtime for each dataset are also consistent with computational complexity of each kernels in Table 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.

4.2.2. 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-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. From available datasets, *Letter-med* and *Enzymes* contain non-symbolic vertex labels. We compute accuracy and time complexity of each kernel aforementioned on these 2 datasets, then we remove the non-symbolic labels from the datasets and compute the performance again.

Figure 8 shows that, with non-symbolic labels, 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 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 4 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 into 2 groups: graphs containing non-symbolic labels (including *Letter-med*, *Enzymes*, *AIDS*) and those with only symbolic labels (including all the rests). Figure 9 exhibits

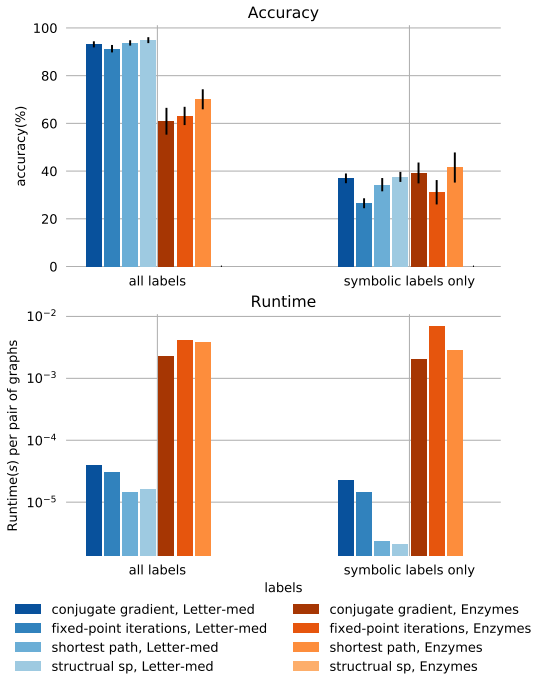

Figure 8: Comparison of accuracy and runtime of graph kernels on datasets with and without non-symbolic labels. The last pillar was removed due to its high computational time.

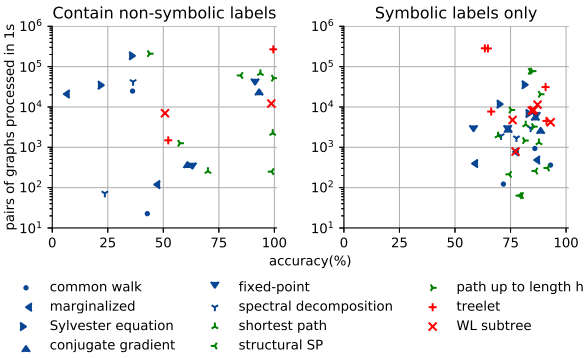

Figure 9: Comparison of computational complexity versus accuracies of all graph kernels on graphs with and without non-symbolic labels. Markers correspond to different kernels; Colors blue, green and red depict graph kernels based on walks, paths and non-linear patterns, respectively.

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, 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.

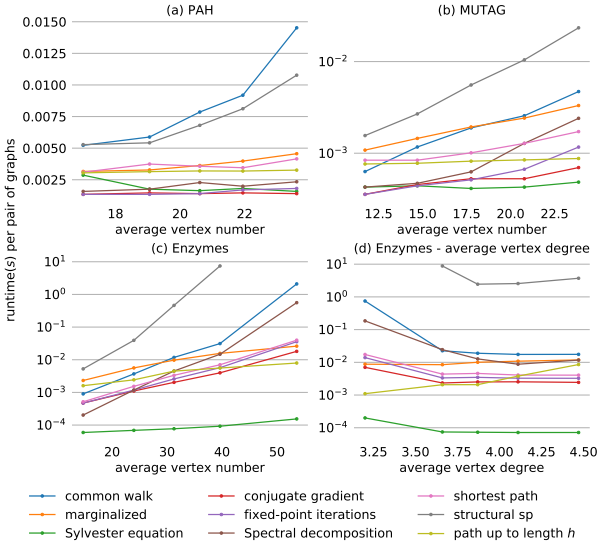

Figure 10: Comparison of runtime of each kernel on datasets with different average vertex numbers and average vertex degrees.

4.2.3. 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 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.

Figure 10(a)(b)(c) show the evolution in the runtime to compute Gram matrices with the growth of average vertex numbers. The runtimes for the common walk kernel and the structural shortest path kernel growth the fastest, the runtimes for 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 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 growth. This phenomenon is more observable for small datasets with a more narrow range of vertex numbers, such as *PAH* shown in Figure 10(a).

4.2.4. Graphs with different average vertex degrees

As vertex numbers, the vertex degrees play 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 4.2.3, we choose a dataset with relatively wide range of vertex degrees, *Enzymes*, order it based on the vertex degree and split it into 5 subsets. Figure 10(d) reveals the relationship between the runtime to compute the graph kernel and the average vertex degree of each subset.

Table 1 displays 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 10(d) shows that the runtime for the path kernel up to length h increases as the average vertex degree grows. Runtime for most of the other kernels, however, is high for the first subset (when $m = 3.2$), and stays stable after. 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 thans 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 substructures compared to graph kernels based on non-linear patterns, even though these kernels are not build for the latter structure. 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.

5. Conclusion

In this paper, an extensive analysis of graph kernels based on linear patterns was performed. Although graph kernels based on linear patterns are 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.

The computational complexity – a major issue in designing and working with graph kernels – was extensively addressed in this paper. The average vertex numbers and average vertex degrees restrict kernels' scale abilities. Time complexity of all kernels are 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 trivial influence on the time complexity, which remained low on all datasets.

Acknowledgment

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

References

Aziz, F., Wilson, R.C., Hancock, E.R., 2013. Backtrackless walks on a graph. IEEE Transactions on Neural Networks and Learning Systems 24, 977–989.

Bai, L., Rossi, L., Cui, L., Cheng, J., Hancock, E.R., 2019. A quantum-inspired similarity measure for the analysis of complete weighted graphs. IEEE transactions on cybernetics 50, 1264–1277.

Bai, L., Rossi, L., Cui, L., Zhang, Z., Ren, P., Bai, X., Hancock, E., 2017. Quantum kernels for unattributed graphs using discrete-time quantum walks. Pattern Recognition Letters 87, 96–103.

Bai, L., Rossi, L., Torsello, A., Hancock, E.R., 2015. A quantum jensen–shannon graph kernel for unattributed graphs. Pattern Recognition 48, 344–355.

Bajema, K., Merlin, R., 1987. Raman scattering by acoustic phonons in fibonacci gaas-aias superlattices. Physical Review B 36, 4555.

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

Borgwardt, K.M., Ong, C.S., Schönauer, S., Vishwanathan, S., Smola, A.J., Kriegel, H.P., 2005. Protein function prediction via graph kernels. Bioinformatics 21, i47–i56.

Boser, B.E., Guyon, I.M., Vapnik, V.N., 1992. A training algorithm for optimal margin classifiers, in: Proc. fifth annual workshop on Computational learning theory, ACM. pp. 144–152.

Bougleux, S., Dupé, F.X., Brun, L., Mokhtari, M., 2012. Shape similarity based on a treelet kernel with edition, in: Gimel'farb, G., et al. (Eds.), Structural, Syntactic, and Statistical Pattern Recognition, Springer Berlin Heidelberg, Berlin, Heidelberg. pp. 199–207.

Brun, L., 2018. Greyc chemistry dataset. URL: https://brunl01.users.greyc.fr/CHEMISTRY/index.html. accessed October 30, 2018.

Cherqaoui, D., Villemin, D., 1994. Use of a neural network to determine the boiling point of alkanes. Journal of the Chemical Society, Faraday Transactions 90, 97–102.

Cherqaoui, D., Villemin, D., Mesbah, A., Cense, J.M., 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, 2015–2019.

Conte, D., Foggia, P., Sansone, C., Vento, M., 2004. Thirty years of graph matching in pattern recognition. International journal of pattern recognition and artificial intelligence 18, 265–298.

Cortes, C., Haffner, P., Mohri, M., 2004. Rational kernels: Theory and algorithms. Journal of Machine Learning Research 5, 1035–1062.

Debnath, A.K., Lopez de Compadre, R.L., Debnath, G., Shusterman, A.J., 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, 786–797.

Dijkstra, E.W., 1959. A note on two problems in connexion with graphs. Numerische mathematik 1, 269–271.

Dobson, P.D., Doig, A.J., 2003. Distinguishing enzyme structures from non-enzymes without alignments. Journal of molecular biology 330, 771–783.

Floyd, R.W., 1962. Algorithm 97: shortest path. Communications of the ACM 5, 345.

Fredkin, E., 1960. Trie memory. Communications of the ACM 3, 490–499.

Gärtner, T., Flach, P., Wrobel, S., 2003. On graph kernels: Hardness results and efficient alternatives. Learning Theory and Kernel Machines , 129–143.

Gaüzere, B., Brun, L., Villemin, D., 2012. Two new graphs kernels in chemoinformatics. Pattern Recognition Letters 33, 2038–2047.

Gaüzere, B., Grenier, P.A., Brun, L., Villemin, D., 2015. Treelet kernel incorporating cyclic, stereo and inter pattern information in chemoinformatics. Pattern Recognition 48, 356 – 367.

Haussler, D., 1999. Convolution kernels on discrete structures. Technical Report. Technical report, Department of Computer Science, University of California at Santa Cruz.

Honeine, P., Noumir, Z., Richard, C., 2013. Multiclass classification machines with the complexity of a single binary classifier. Signal Processing 93, 1013 – 1026.

Johnson, M.A., Maggiora, G.M., 1990. Concepts and applications of molecular similarity. Wiley.

Kashima, H., Tsuda, K., Inokuchi, A., 2003. Marginalized kernels between labeled graphs, in: Proc. of the 20th international conference on machine

learning (ICML-03), pp. 321–328.
Kersting, K., Kriege, N.M., Morris, C., Mutzel, P., Neumann, M., 2016. Benchmark data sets for graph kernels. URL: http://graphkernels.cs.tu-dortmund.de.
Kobler, J., Schöning, U., Torán, J., 2012. The graph isomorphism problem: its structural complexity. Springer Science & Business Media.
Kriege, N., Neumann, M., Kersting, K., Mutzel, P., 2014. Explicit versus implicit graph feature maps: A computational phase transition for walk kernels, in: 2014 IEEE International Conference on Data Mining, IEEE. pp. 881–886.
Mahé, P., Ueda, N., Akutsu, T., Perret, J.L., Vert, J.P., 2004. Extensions of marginalized graph kernels, in: Proc. the twenty-first international conference on Machine learning, ACM. p. 70.
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, 415–446.
Minello, G., Rossi, L., Torsello, A., 2019. Can a quantum walk tell which is which? a study of quantum walk-based graph similarity. Entropy 21, 328.
Morris, C., Kersting, K., Mutzel, P., 2017. Glocalized weisfeiler-lehman graph kernels: Global-local feature maps of graphs, in: 2017 IEEE International Conference on Data Mining (ICDM), pp. 327–336.
Morris, C., Kriege, N.M., Kersting, K., Mutzel, P., 2016. Faster kernels for graphs with continuous attributes via hashing, in: Data Mining (ICDM), 2016 IEEE 16th International Conference on, IEEE. pp. 1095–1100.
Murphy, K.P., 2012. Machine Learning: A Probabilistic Perspective. MIT Press.
Murray, R.M., et al., 2018. Python Control Systems Library. URL: http://python-control.readthedocs.io/en/latest/index.html.
Ralaivola, L., Swamidass, S.J., Saigo, H., Baldi, P., 2005. Graph kernels for chemical informatics. Neural networks 18, 1093–1110.
Riesen, K., 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, Springer. pp. 287–297.
Rossi, L., Torsello, A., Hancock, E.R., 2013. A continuous-time quantum walk kernel for unattributed graphs, in: International Workshop on Graph-Based Representations in Pattern Recognition, Springer. pp. 101–110.
Schölkopf, B., Herbrich, R., 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, Springer-Verlag, London, UK. pp. 416–426.
Schölkopf, B., Smola, A.J., 2002. Learning with kernels: support vector machines, regularization, optimization, and beyond. MIT press.
Schomburg, I., Chang, A., Ebeling, C., Gremse, M., Heldt, C., Huhn, G., Schomburg, D., 2004. Brenda, the enzyme database: updates and major new developments. Nucleic acids research 32, D431–D433.
Shawe-Taylor, J., Cristianini, N., 2004. Kernel methods for pattern analysis. Cambridge university press.
Shervashidze, N., Schweitzer, P., Leeuwen, E.J.v., Mehlhorn, K., Borgwardt, K.M., 2011. Weisfeiler-lehman graph kernels. Journal of Machine Learning Research 12, 2539–2561.
Suard, F., Rakotomamonjy, A., Bensrhair, A., 2007. Kernel on bag of paths for measuring similarity of shapes., in: ESANN, pp. 355–360.
Sugiyama, M., Borgwardt, K., 2015. Halting in random walk kernels, in: Advances in neural information processing systems, pp. 1639–1647.
Vishwanathan, S.V.N., Schraudolph, N.N., Kondor, R., Borgwardt, K.M., 2010. Graph kernels. Journal of Machine Learning Research 11, 1201–1242.
Wale, N., Watson, I.A., Karypis, G., 2008. Comparison of descriptor spaces for chemical compound retrieval and classification. Knowledge and Information Systems 14, 347–375.
West, D.B., et al., 2001. Introduction to graph theory. volume 2. Prentice hall Upper Saddle River.
Xu, L., Wang, W., Alvarez, M., Cavazos, J., 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 .

Paul Honeine received the Dipl.-Ing. degree in mechanical engineering in 2002 and the M.Sc. degree in industrial control in 2003, both from the Faculty of Engineering, the Lebanese University, Lebanon. In 2007, he received the Ph.D. degree in Systems Optimisation and Security from the University of Technology of Troyes, France, and was a Postdoctoral Research associate with the Systems Modeling and Dependability Laboratory, from 2007 to 2008. From September 2008 till August 2015, he was an assistant Professor at the University of Technology of Troyes, France. Since September 2015, he is full professor at the LITIS Lab of the University of Rouen (Normandie Université), France. His research interests include non-stationary signal analysis and classification, nonlinear and statistical signal processing, sparse representations, machine learning. Of particular interest are applications to (wireless) sensor networks, biomedical signal and image processing, hyperspectral imagery and nonlinear adaptive system identification.

Benoit Gaüzère received his PhD from GREYC laboratory in 2013. This PhD was conducted under the supervision of Pr. Luc Brun and Pr. Didier Villemin on the definition of graph kernels for chemoinformatics. Then he spent one year as post doc in MIVIA research lab, University of Salerno, working on knowledge based models for tracking tasks. Since September 2015, he is now associate professor in LITIS, Normandy University. His current research domains consist in bridging the gap between graph based pattern recognition and machine learning.

Linlin Jia received the B.E. degree in information engineering, in 2014, and the M.E. degree in software engineering, in 2017, both from Xi'an Jiaotong University, China. Since 2017, he has been a Ph.D. candidate in computer science in Laboratoire d'Informatique, de Traitement de l'Information et des Systèmes (LITIS) in INSA Rouen Normandie, France, with a PH.D. thesis "machine learning and patterns recognition in chemoinformatics.", focusing on graph kernels in machine learning and pre-image problems.