Graph Kernels Based on Linear Patterns: Theoretical and Experimental Comparisons

Linlin Jia, Benoit Gaüzère, Paul Honeine

Graph Kernels Based on Linear Patterns: Theoretical and Experimental Comparisons

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

[a]*LITIS, INSA Rouen Normandie, Rouen, France*
[b]*LITIS, Université de Rouen Normandie, Rouen, France*

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 from the similarity between corresponding patterns. Kernels based on linear patterns constitute a good trade-off between accuracy performance 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. Then, 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, cyclic and acyclic graphs. Finally, for regression and classification tasks, performance and computational complexity of kernels are compared and analyzed, and 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.

Keywords: Graph kernels, Walks, Paths, Kernel methods, Graph representation

---

*Corresponding author
Email address: `linlin.jia@insa-rouen.fr` (Linlin Jia)

1. Introduction

In recent years, machine learning has becoming a considerably effective and efficient tool in multiple real-world tasks, such as regression and classification problems. Machine learning algorithms have been defined on vector spaces, allowing to take advantage of the easiness in linear algebra operations. While many data can be represented by vectors, such as images by unfolding them into vectors, it turns out that many data types are too complex to be vectorized.

Graphs are able to model a wide range of real-world data, by encoding elements as well as the relationship between them. Considering for example a molecule represented by a graph: its vertices stand for the molecule atoms, while its edges model the relationship between them, such as bond types between two atoms. Due to these properties, graph representation has broad applications in wide domains, such as 2D and 3D image analysis, document processing, biometric identification, image database, video analysis, biological and biomedical applications, bioinformatics, chemoinformatics, web data mining, etc., where it models structures such as molecules, social networks, and state transition (Conte et al., 2004).

Therefore, 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 et al., 2012). Unfortunately, the exact similarity is a binary measure which can only take values from $\{0, 1\}$, and cannot be computed in polynomial time by these methods; Hence it is neither useful nor practical for real-world data.

Inexact similarity measures are commonly applied for graphs, in which category
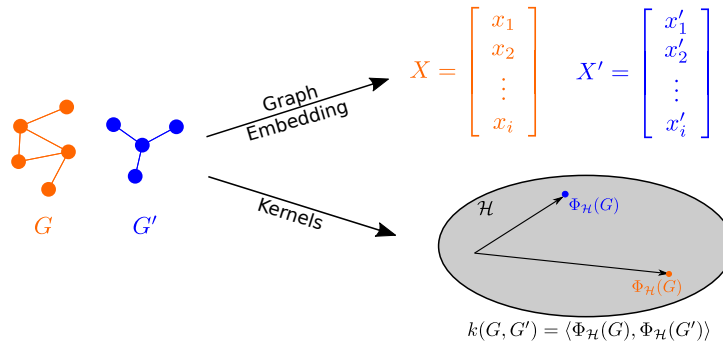
$$X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_i \end{bmatrix} \qquad X' = \begin{bmatrix} x'_1 \\ x'_2 \\ \vdots \\ x'_i \end{bmatrix}$$

$$k(G, G') = \langle \Phi_{\mathcal{H}}(G), \Phi_{\mathcal{H}}(G') \rangle$$

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 $k(G, G')$.

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. The two strategies are detailed next, with an illustration of this embedding in Figure 1.

Graph embedding explicitly computes vectors that encode some information of the graphs. Riesen et al. (2007) proposed a method where a set of prototype graphs is selected as a baseline. Each graph is then compared to each prototype by means of a dissimilarity measure. The dissimilarities of a graph to each prototype compose a finite-dimension feature vector of real numbers. In this way graphs are transformed into vectors.

Kernels allow an implicit embedding by representing graphs in a possibly infinite-dimension feature space which relaxes the limitations on the encoded information. Indeed, as generalizations of the scalar inner product, kernels are natural similarity measures between data, expressed as inner products between elements in some feature space. By employing the kernel trick (Schölkopf and Smola, 2002), 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 machines, such as Support Vector Machines (SVM). Therefore, defining kernels between graphs is a powerful design to bridge the gap between machine learning
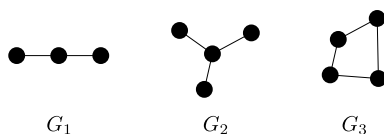
3

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.

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 considered in the literature, 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, as given by G_1. However, when a substructure contains vertices that have more than two neighbors, as given by G_2, linear patterns are insufficient to completely describe the structure. This is where non-linear patterns become useful, with either non-linear (acyclic) patterns (as in G_2) or cyclic patterns, which contain cycles (as in G_3).

Despite that non-linear patterns may encode more complex structural information than linear patterns, the latter are of great interest. Linear patterns tend to require much lower computational complexity than non-linear patterns. Nevertheless, while linear patterns explore less complex structures, non-linear patterns normally include or imply them. For example, the treelet pattern is non-linear as a whole, while treelets whose maximal size is less than 4 are linear (Gaüzere et al., 2012). This property is important especially for big graphs, as it may be much more time consuming to construct non-linear kernels and cyclic kernels on big graphs. Therefore in this article, we focus on studying and comparing the existing 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; while a walk can contain the same vertex more than once, a path is a walk without repeated vertices. Based on the principle of comparing

all possible walks in two graphs, Gärtner et al. (2003) constructed the common walk kernel and proposed closed forms to compute two special cases of this kernel. In the meantime, Kashima et al. (2003) introduced randomness into the walk-based graph kernels by generating walks with marginal distributions. Prior knowledge is able to be brought into this kernel, named the marginalized kernel, by choosing parameters of these distributions. More recently, Vishwanathan et al. (2010) built a framework for kernels with the so-called generalized random walk kernel, and offered four methods to reduce the computational complexity of some special cases. Both the common walk kernel and the marginalized kernel are special cases of this generalized kernel.

Due to their structures, walks may bring artifacts to graph kernels due to tottering and halting (see Section 3.1.4 for details on these shortcomings). Graph kernels based on paths, on the other hand, are relieved from these issues. Borgwardt and Kriegel (2005) proposed the shortest path kernel, based on the comparison of each pair of vertices of two graphs by checking whether their weighted shortest paths have the same length. Ralaivola et al. (2005) constructed the structural shortest path kernel, where the vertices and edges on two shortest paths are compared successively, rather than the lengths of the shortest paths. While these two kernels are both based on shortest paths, Suard et al. (2007) proposed a path kernel that compares all paths in the two graphs. In practice, only paths shorter than a given length are considered.

In this article, we thoroughly study graph kernels based on linear patterns, with an emphasis on the aforementioned kernels, and compare 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. In this library, every kernel is able to tackle different

types of graphs, and several computation methods are provided for kernels.

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

2. Preliminaries

This section introduces notations and terminologies required to study graph kernels.

2.1. Basic concepts of graph theory

In the following, we define notations that will be used in this paper. Definitions that are only used in specific sections will be given in the corresponding parts. 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. For a subset A of a set X, the indicator function $\mathbf{1}_A : X \rightarrow \{0, 1\}$ is defined as

$$
\mathbf{1}_A(x) = \begin{cases} 1 & \text{if } x \in A; \\ 0 & \text{if } x \notin A. \end{cases}
$$

Definition 1 (graph). *A graph G is defined by an ordered pair of disjoint sets (V, E) such that V corresponds to a finite set of vertices and $E \subset V \times V$ corresponds to a set of edges. If $(u, v) \in E$, then u is adjacent to v. We denote the number of graph vertices as n, i.e., $n = |V|$, and the number of graph edges as m, i.e., $m = |E|$.*

Definition 2 (labeled and unlabeled graph). *A labeled graph G is a graph that has additionally a set of labels L along with a labeling function ℓ that assigns a label to each edge and/or vertex. In edge-labeled graphs, the labeling function $\ell_e : E \rightarrow L$ assigns labels to edges only. In vertex-labeled graphs, the labeling function $\ell_v : V \rightarrow L$ assigns labels to vertices only. In fully-labeled graphs, the labeling function $\ell_f : V \cup E \rightarrow L$ assigns labels to both vertices and edges. Unlabeled graphs have no such labeling function.*

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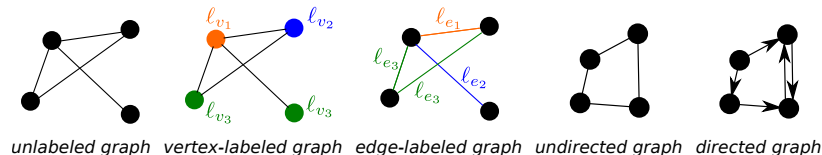
Figure 3: The different types of graphs. In vertex- and edge-labeled graphs, vertices and edges with different labels are distinguished by color in the figure.

Definition 3 (symbolic and non-symbolic labels). *Labels defined in Definition 2 are 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.

Definition 4 (Kronecker delta function). *The Kronecker delta function between two labels ℓ_i and ℓ_j is defined as*

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

where $\ell_i, \ell_j \in L$. For the sake of conciseness, it is denoted as the delta function.

Definition 5 (Gaussian kernel). *The Gaussian kernel between $\ell_i, \ell_j \in L$ is defined as*

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

where δ is the tunable bandwidth parameter.

Definition 6 (directed and undirected graph). *A directed graph is a graph whose edges are directed from one vertex to another, where the edge set E consists of ordered pairs of vertices (u, v). An ordered pair (u, v) is said to be an edge directed from*

u to v, namely an edge beginning at u and ending at v. In contrast, a graph where the edges are bidirectional is called an undirected graph, i.e., if $(u, v) \in E$, 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.

Definition 7 (neighbors and degree). *In a graph $G = (V, E)$, a neighbor of a vertex $v \in V$ is a vertex u that meets the condition $(u, v) \in E$. The degree of a vertex $v \in V$ is the number of these neighbors, namely $|\{u \in V \mid (u, v) \in E\}|$. The degree of the graph, denoted by d, is the largest vertex degree of all its vertices. If G is directed, then $|\{u^- \in V \mid (u^-, v) \in E\}|$ is called the indegree of vertex v, and $|\{u^+ \in V \mid (v, u^+) \in E\}|$ is the outdegree of v.*

Definition 8 (adjacency matrix). *The adjacency matrix of an n-vertices 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.*

Definition 9 (walks, paths and cycles). *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. If each vertex appears only once except for v_1, then the cycle is called a simple cycle. Note that when $h = 0$, a walk or path is a single vertex without edges.*

Definition 10 (label sequences). *The (contiguous) label sequence of a length h walk/path W of a fully-labeled graph is defined as $s = (\ell_v(v_1), \ell_e((v_1, v_2)), \ell_v(v_2), \ell_e((v_2, v_3)), \ldots, \ell_v(v_{h+1}))$. For a vertex-labeled or edge-labeled graph, the label sequence of W is constructed by removing all edge labels $\ell_e((v_i, v_j))$ or vertex labels $\ell_v(v_i)$ in s, respectively.*

2.2. Kernel methods

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 11 (positive semi-definite kernel). *A positive semi-definite kernel defined on $\mathcal{X}$ is a symmetric bilinear function $k : \mathcal{X}^2 \rightarrow \mathbb{R}$ that fulfills the following condition*

$$\sum_{i=1}^{n} \sum_{j=1}^{n} c_i \, c_j \, k(x_i, x_j) \geq 0, \tag{3}$$

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 $\lim_{n \rightarrow \infty} k_n$ 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 that are introduced in this paper. The first one is applied for all six graph kernels discussed in this paper, and the second one is applied for both the common walk kernel and the generalized random walk kernel.

Mercer's theorem states that any positive semi-definite kernel k corresponds to an inner product in some Hilbert space $\mathcal{H}$ (Mercer, 1909). In other words, for all $(x_i, x_j) \in \mathcal{X}^2$, we have

$$k(x_i, x_j) = \langle \Phi_{\mathcal{H}}(x_i), \Phi_{\mathcal{H}}(x_j) \rangle_{\mathcal{H}}, \tag{4}$$

where $\Phi_{\mathcal{H}} : \mathcal{X} \rightarrow \mathcal{H}$ 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 $X = \{x_1, x_2, \ldots, x_N\}$ be the finite dataset of N samples available for training the machine learning method,

with associated data labels $\{y_1, y_2, \ldots, y_N\}$ where $y_i \in \{-1, +1\}$ for binary classification and $y_i \in \mathbb{R}$ for regression (extensions to multiclass classification and vector output is straightforward (Honeine et al., 2013)). It turns out that one does not need to have access to the raw data X for training and evaluating, 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 the minimization of a regularized cost function, 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 et al., 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 $[\omega_1 \ldots \omega_N]^\top = (K + \lambda I)^{-1}[y_1 \ldots y_N]^\top$ (Murphy, 2012). The 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 et al., 1992).

Kernel-based methods provide an elegant and powerful framework in machine learning for any input space, without the need to exhibit the data or optimize in that space, as long as one can define a kernel on it. Besides conventional kernels, such as the Gaussian and polynomial kernels for vector spaces, novel kernels can be engineered by combining other valid kernels, such as using additive or multiplicative rules (Definition 11). Of particular interest in kernel engineering are R-convolution kernels (Haussler, 1999), which provide the foundation of kernels based on bags of patterns. Since they can be regarded as the cornerstones to engineer graph kernels using graph patterns, we provide next some details on the R-convolution kernels.

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 $x_i \in X$ has a composite structure, namely described by its "parts" $(x_{i1}, x_{i2}, \ldots, x_{iD}) \in X_1 \times X_2 \times \cdots \times X_D$, for some positive integer D. Since multiple decompositions could exist, let $R(x)$ denotes all possible decompositions of x, namely

$$R(x) = \left\{ \boldsymbol{x} \in X_1 \times X_2 \times \cdots \times X_D \;\middle|\; \boldsymbol{x} \text{ is a decomposition of } x \right\}.$$

For each decomposition space 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 X is defined as:

$$k(x_i, x_j) = \sum_{\substack{(x_{i1}, \ldots, x_{iD}) \in R(x_i) \\ (x_{j1}, \ldots, x_{jD}) \in R(x_j)}} \prod_{d=1}^{D} k_d(x_{id}, x_{jd}). \tag{5}$$

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. In this section, we shall focus on studying their mathematical representations and comparing 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 walk based graph kernels have been proposed.

Table 1: Characteristics of graph kernels based on linear patterns.

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

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

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

3.1.1. Common walk kernel

A 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.

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

In (Gärtner et al., 2003), the fully-labeled direct product graph is employed to reduce the computational complexity, within two types of kernels: kernels based on label pairs and kernels based on contiguous label sequences; the latter one can deal with labels on vertices and/or edges. In this case, the direct product is defined as following:
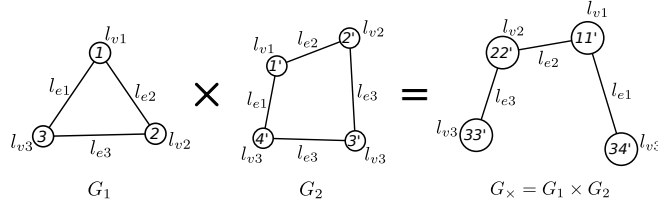
Figure 4: Direct product of fully-labeled graphs

Definition 12 (direct product graph). *Consider two graphs, $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. Their direct product graph, denoted $G_\times = G_1 \times G_2$, is defined by its vertex and edge sets as:*

$$\begin{cases} V_\times(G_1 \times G_2) = \Big\{ (v_1, v_2) \in V_1 \times V_2 \mid \ell_v(v_1) = \ell_v(v_2) \Big\} \\ E_\times(G_1 \times G_2) = \Big\{ ((u_1, u_2), (v_1, v_2)) \in V^2(G_1 \times G_2) \mid \\ \qquad (u_1, v_1) \in E_1 \wedge (u_2, v_2) \in E_2 \wedge \ell_e(u_1, v_1) = \ell_e(u_2, v_2) \Big\}, \end{cases} \tag{6}$$

where $\ell_v(\cdot)$ and $\ell_e(\cdot)$ are the labeling functions defined in Definition 2. 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 $G_\times$ 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 13 (common walk kernel applying direct product). *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*

13

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

$$k_\times(G_1, G_2) = \sum_{i,j=1}^{|V_\times|} \left[\sum_{h=1}^{\infty} \lambda_h A_\times^h \right]_{ij}, \tag{7}$$

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, then $[A_\times^h]_{ij}$ 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 $e^{\beta A_\times} = I + \beta A_\times/1! + \beta^2 A_\times^2/2! + \beta^3 A_\times^3/3! + ...$ Computing this exponentiation 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.

- Another one 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 computational complexity of the common walk kernel composes of two parts. The first one is the direct product transform. One needs $O(n^2)$ to span all vertices of two graphs pairwise to construct vertices of the direct product graph and $O(n^4)$ for edges. The second part, computing the kernel, requires cubic computational complexity by applying either exponential or geometric series. Since the computation is based on

14

direct product graphs, which could potentially have n^2 vertices, thus the computational complexity is $O(n^6)$, which is also the computational complexity of the common walk kernel (Gärtner et al., 2003).

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 within polynomial time. 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 to take all walks into count for large-scale graphs. Furthermore, it 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), as described next. From the start vertex of each walk, a conditional probability is given to each vertex to choose the next vertex to go from its neighbors, which is called the transition probability, and a termination probability to finish generating the walk. The marginalized kernel is constructed as:

$$k(G_1, G_2) = \sum_{w_1 \in W(G_1)} \sum_{w_2 \in W(G_2)} k_W(w_1, w_2) p_{G_1}(w_1) p_{G_2}(w_2), \tag{8}$$

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 $w = (v_1, v_2, \ldots, v_h)$, then

$$p_G(w) = p_0(v_1) \prod_{i=2}^{h} p_t(v_i|v_{i-1}) p_q(v_h), \tag{9}$$

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

15

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 computed recursively 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

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

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. Thus, the computational complexity boils down to $O(rn^4)$, where r is the number of iterations.

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 common walk kernels and marginalized kernels are special cases of this kernel. Besides, it is proven in the same paper that certain rational kernels (Cortes et al., 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 $v_{i_1}, v_{i_2}, v_{i_3}, \ldots$ according to a conditional probability $p(i_{k+1} \mid i_k) = A_{i_{k+1},i_k}$, where

16

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 12. It is worth noting that the direct product graph of unlabeled graphs often has much more edges than the labeled one, as illustrated in Figure 5 when taking the graphs in Figure 4 as unlabeled. The generalized random walk kernel between two graphs G_1 and G_2 is defined in (Vishwanathan et al., 2010) as

$$k(G_1, G_2) = \sum_{h=0}^{\infty} f(h) q_\times^\top W_\times^h p_\times. \tag{10}$$

In this expression, $f(h)$ is the weight chosen for random walks of length h a priori; $p_\times = p_{0_1} \otimes p_{0_2}$ 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 $G_\times$, (10) 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 (10).

The complexity of the direct computation is $O(N^2 n^6)$. Four methods are presented to accelerate the computation (Vishwanathan et al., 2010):
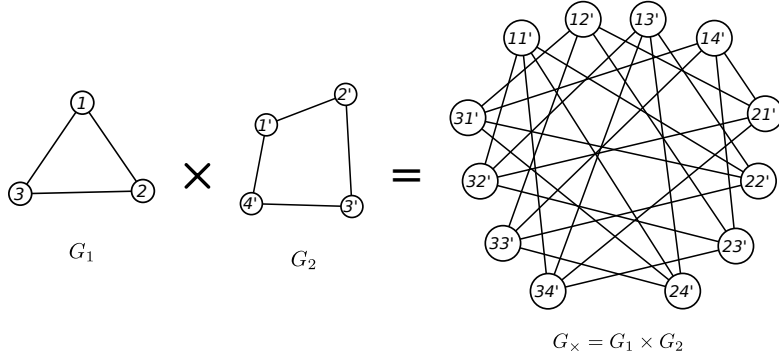
Figure 5: Direct product of unlabeled graphs (compared to the labeled ones in Figure 4)

- The *Sylvester equation method* is based on the generalized Sylvester equation:

$$M = \sum_{i=1}^{d} S_i M T_i + M_0,$$

where M is the matrix to be solved, d is the number of different edge labels, and S_i, T_i, M_0 are given matrices. For graphs with symbolic edge labels, when $f(h) = \lambda_h$, the kernel in (10) 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

$$M = \sum_{i=1}^{d} \lambda^i A_2 M^i A_1^\top + M_0,$$

where $vec(M_0) = p_\times$ 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 $d > 1$. Compared to the direct computation of the generalized random walk kernel, 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 pre-computed for each graph. Besides, computing $q_\times^\top vec(M)$ requires $O(N^2 n^2)$ time. Thus for N unlabeled graphs, the complexity of computing the corresponding Gram matrix is $O(N^2 n^3)$, which is reduced compared to the direct computation.

18

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 from *Python Control Systems Library* (Murray et al., 2018), can only take d as 1, which means the solver is limited to edge-unlabeled graphs.labels.

- The second one 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 $O(rn^4)$ 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 $O(rn^4)$ for r iterations.

- The *spectral decomposition* method applies the decomposition $W_\times = P_\times D_\times P_\times^{-1}$, 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 $O(N^2 n^2 + Nn^3)$ for an $N \times N$ Gram matrix.

The generalized random walk kernel provides a quite flexible framework for walk based kernels (see (10)). 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 (10). The spectral decomposition method works on any $f(h)$ that makes (10) converge, but is only efficient for unlabeled graphs.
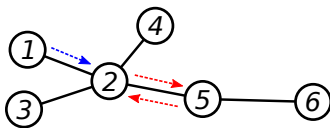
Figure 6: A tottering example: walk (v_1, v_2, v_5, v_2) has tottering between vertices v_2 and v_5, *i.e.*, (v_2, v_5, v_2).

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.

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. Graph kernels based on walks often suffer from 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 6 shows, a tottering brings unnecessary structure to the pattern and may influence the performance of graph kernels.

Mahé et al. (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

$$\begin{cases} V' = V \cup E \\ E' = \left\{ (v, (v, t)) \mid v \in V, (v, t) \in E \right\} \cup \left\{ ((u, v), (v, t)) \mid (u, v), (v, t) \in E, u \neq t \right\} \end{cases} \quad (11)$$

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 $O((n+m)^2)$, 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 walks with longer lengths tend to become statistically trivial to kernels. It is as if the common kernel 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 dominates 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.

Graph kernels based on paths avoid these two problems.

3.2. Graph kernels based on paths

Several graph kernels based on paths have been proposed to overcome walk issues.

3.2.1. Shortest path kernel

The shortest path kernel is built on the comparison of shortest paths between any pair of vertices in the 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 following:

Definition 14 (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*

$$k_{sp}(G_1, G_2) = \sum_{e_1 \in E_1} \sum_{e_2 \in E_2} k_w(e_1, e_2), \tag{12}$$

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

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, namely $\delta_{\ell_{v_1} \ell_{v_2}}$ for vertices v_1 and v_2; while the kernel for non-symbolic vertex attributes is not given for general cases.

In this paper, we only 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. For instance, label enrichment, such as the number of edges, can be applied to speed up kernel computation. Considering k shortest paths between each pair of vertices rather than only one, where k is an integer determined a priori, can introduce more information but with higher computational complexity (Borgwardt and Kriegel, 2005).

The Floyd-Warshall's algorithm, required in the shortest-path kernel to perform the Floyd-transformation, can be done in $O(n^3)$. 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 $O(1)$, then pairwise comparison of all edges in two shortest-paths graphs requires a computational complexity of $O(n^4)$, which is then also the computational complexity to compute the shortest path kernel.

Comparing 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 main shortcomings in theory and practice are:

- It simplifies the graph structure by Floyd transformation and only considers information concerning shortest distances. Only attributes of start and end vertices of shortest paths are considered, while information about intermediate vertices and edges is missing.

- 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).

- 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.

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.

To construct this kernel, the first step is to obtain all shortest paths between all vertices in each graph, 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:

$$k_p(p, p') = k_v(\ell_v(v_1), \ell_v(v'_1)) \prod_{i=2}^{n} k_e(\ell_e(v_{i-1}, v_i), \ell_e(v'_{i-1}, v'_i)) k_v(\ell_v(v_i), \ell_v(v'_i)), \qquad (13)$$

where v_i and v'_i, for $i = 1, 2, \ldots, n$, are respectively the vertices on paths p and p', $\ell_v(\cdot)$ and $\ell_e(\cdot)$ are label functions of vertices and edges, respectively. The 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 as the Gaussian kernel for non-symbolic labels. These two kernel functions are multiplied if both symbolic and non-symbolic labels exist.

The structural shortest path kernel can then be derived from the shortest-paths kernel k_p given in (13). Here we use the simple and straightforward mean average kernel:

$$k_{ssp}(G_1, G_2) = \frac{1}{n_1} \frac{1}{n_2} \sum_{p_i \in P_1} \sum_{p_j \in P_2} k_p(p_i, p_j), \qquad (14)$$

23

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 $O(n^2 \log n + nm)$ (Bajema and Merlin, 1987). The computational complexity to match all paths in two graphs is $O(hn^4)$, where h is the average length of shortest paths. Hence the complexity of the kernel computation is $O(hn^4 + nm)$.

Compared to the shortest path kernel, the structural shortest path kernel involves more structural information. However, since both kernels adopt the shortest path, the structure of real paths is 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

$$k_{path}(G_1, G_2) = \sum_{p \in P(G_1) \cup P(G_2)} \phi_p(G_1)\phi_p(G_2), \tag{15}$$

where $P(G)$ is the set of all possible 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 follows:

$$k_{path}^t(G_1, G_2) = \frac{k_{path}(G_1, G_2)}{k_{path}(G_1, G_1) + k_{path}(G_2, G_2) - k_{path}(G_1, G_2)},$$

where $k_{path}(G_1, G_2)$ is the kernel defined as (15) 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

$$k_{path}^m(G_1, G_2) = \frac{\sum_{p \in P(G_1) \cup P(G_2)} \min(\phi_p(G_1), \phi_p(G_2))}{\sum_{p \in P(G_1) \cup P(G_2)} \max(\phi_p(G_1), \phi_p(G_2))}.$$

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. These two kernels are implemented in our library, and their performance is discussed in this paper.

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 to solve this problem 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 $O(h^2 n^2 d^{2h})$.

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 is a significant drawback of this kernel, especially for the computational complexity in large-scale graphs.

4. Experiments

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 methods to accelerate computation of kernels, and propose an analysis on applying these methods by choosing proper parameters. Finally, we perform each graph kernel on each dataset and analyze the performance 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. Real-world datasets

We examine 7 well-known benchmark datasets, where the first two are for the regression task of estimating the boiling points of molecules and the others are classification tasks.

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
				symbolic		non-symbolic								
	linear	non-linear	cyclic	vertices	edges	vertices	edges							
Acyclic	✓	✓	✗	3	✗	✗	✗	✗	183	8.15	7.15	1.47	-	R
Alkane	✓	✓	✗	2	✗	✗	✗	✗	150	8.87	7.87	1.75	-	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

"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").

Alkane is a dataset of 150 alkanes with only carbon atoms, which are represented by acyclic unlabeled graphs (Cherqaoui and Villemin, 1994).

Acyclic has 183 acyclic molecules with hetero atoms (Cherqaoui et al., 1994).

MAO is a Monoamine Oxidase dataset of 68 cycle-included molecules labeled by two classes: 38 of them act as antidepressant drugs by inhibiting the monoamine oxidase and 30 do not (Brun, 2018).

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

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

Letter-med involves graphs of 2250 distorted letter drawings of 15 capital letters of the Roman alphabet (Riesen and Bunke, 2008). The "med" suffix indicates that the distortion is of medium strength. The task is to classify each graph to the proper letter.

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

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.

4.2. Computational settings

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.

Two machines are used to execute the experiments. The first one is a cluster with 28 CPUs of Intel(R) Xeon(R) E5-2680 v4 @ 2.40GHz, 252GB memory, and 64-bit operating system CentOS Linux release 7.3.1611, referred as *CRIANN*. Another machine, referred as *laptop*, is a laptop with 8 CPUs of Intel(R) Core(TM) i7-7920HQ @ 3.10GHz, 32GB memory, and 64-bit operating system Ubuntu 16.04.3 LTS. When comparisons on different machines are required, both *CRIANN* and *laptop* are used; otherwise only *CRIANN* is used. All results were run with Python 3.5.2.

4.3. Methods to accelerate computation

4.3.1. Fast computation of shortest path kernel implementation

To compute the shortest path kernel between two graphs, G_1 and G_2, shortest paths between all pairs of vertices in both graphs are compared. Each time we compare two shortest paths, both their corresponding pairs of vertices are compared once, which causes significant redundancy during the vertex comparison procedure. If G_1 has n_1 vertices and G_2 has n_2 vertices, then there are at most n_1^2 shortest paths in G_1 and n_2^2 shortest paths in G_2, thus $n_1^2 n_2^2$ comparisons between shortest paths and $2n_1^2 n_2^2$ comparisons between vertices are required to compute the kernel. Each pair of vertices is compared $2n_1 n_2$ times in average.

Xu et al. (2014) propose the Fast Computation of Shortest Path Kernel (FCSP) to reduce this redundancy. Instead of comparing vertices directly during the procedure of comparing shortest paths, FCSP compares all pairs of vertices between two graphs at the beginning, and then stores the comparison results in an $n_1 \times n_2$ matrix named

shortest path adjacency matrix. Finally when comparing shortest paths, it retrieves comparison results of corresponding vertices from the matrix. This method reduces vertex comparison to at most $n_1 n_2$ times, which is only $1/(2n_1 n_2)$ of the direct implementation. FCSP can reduce the time complexity up to several orders of magnitudes, with an additional memory usage of only size $O(n_1 n_2)$.

In our implementation, we apply this vertex comparing method to the shortest path kernel, as recommended in Xu et al. (2014). Moreover, we also extend this strategy and apply it to the structural shortest path kernel, which allows more redundancy to be reduced since this kernel requires comparisons between all vertices on a pair of shortest paths. If average length of shortest paths in G_1 and G_2 is h, the new method is at most $n_1 n_2 h$ times faster than direct comparison.

Furthermore, we extend this strategy to edge comparison, when computing the structural shortest kernel. If G_1 has m_1 edges and G_2 has m_2 edges, then it requires $m_1 m_2$ times for edge label comparison, compared to $n_1^2 n_2^2 h$ times by the original method.

4.3.2. Parallelization

Parallelization may significantly reduce the computational complexity. The basic concept of parallelization is to split a set of computation tasks into several pieces, and then carry them out separately on multiple computation units such as CPUs or GPUs. We implement parallelization with Python's `multiprocessing.Pool` module on all the graph kernels in two ways: To compute Gram matrices of graph kernels, parallelization is performed on pairs of graphs; during cross validation procedure, parallelization is carried out on the set of trials.

Many factors may influence the efficiency of parallelization, such as the number of computing cores, the transmission bandwidth between these cores, the method to split the data, the computational complexity to tackle one piece of data, etc.

Figure 7 reveals the influence of parallelization and CPU numbers on computational time. For each machine, *CRIANN* and *laptop*, we give the computational time of the Gram matrix and of model selection for the shortest path kernel on each dataset. Moreover, we also give the ratio between runtimes to compute the Gram matrix on
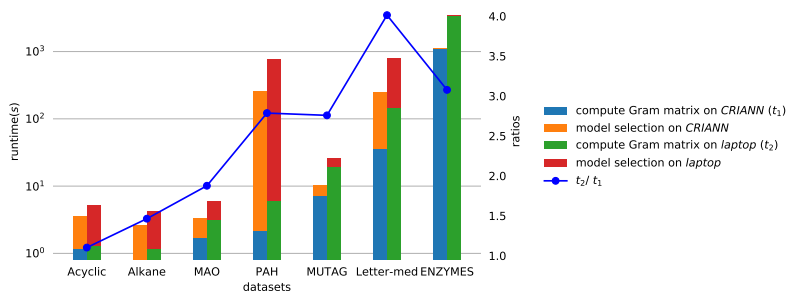
28

Figure 7: Left y-axis: Time (in seconds) to compute Gram matrices (bottom of each pillar) and perform model selection (top of each pillar) of the shortest path kernel on each dataset on *CRIANN* (left pillar) and *laptop* (right pillar) with parallelization. Right y-axis: The blue dots are the ratios between time to compute Gram matrices of each dataset on *laptop* and on *CRIANN*.

laptop and on *CRIANN*. The values of this ratio for large-scale datasets are around 3.5, which turns out to be the inverse ratio of number of CPUs of the two machines (8 versus 28). It is worth noting that the *Letter-med* dataset has the largest number of graphs (but with the relatively "small" graphs), and *Enzymes* has the "average-largest" graphs (the second one being the PAH dataset).

Ideally, parallelization is more efficient when more computing cores are applied. However this efficiency can be suppressed by the parallel procedure required to distribute data to computing cores and collect returned results. Paralleling relatively small graphs to a large number of computing cores can be more time consuming than non-parallel computation. For instance, Figure 7 shows that it takes almost the same time to compute the Gram matrix of the small dataset *Acyclic* on both machines, indicating that the time efficiency raised by applying more CPU cores on machine *CRIANN* is nearly neutralized by the cost to allocate these cores. Choosing appropriate chunksize is an essential way to tackle this problem. Chunksize describes how many data are grouped together as one piece to be distributed to a single computing core.

In Figure 8, runtimes to compute Gram matrices of the shortest path kernel with different chunksize settings are compared, on 28 and 14 CPU cores. With both CPU core numbers, for almost all datasets, when chunksizes are too small, the computing time becomes slightly high, as the parallel procedure costs too much time; as chunksizes
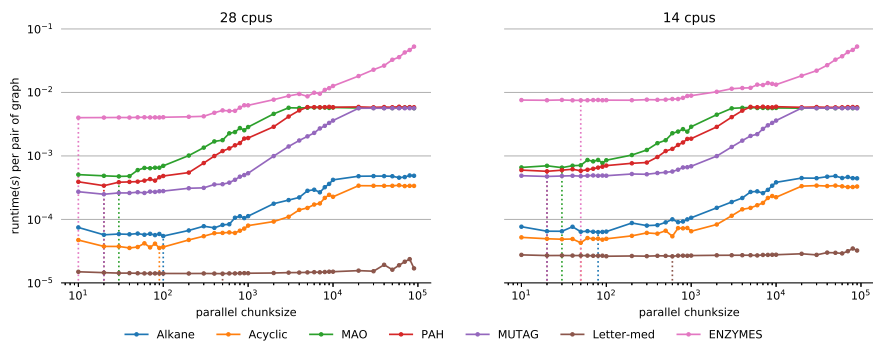
29

Figure 8: Time to compute the Gram matrices of the shortest path kernel on all datasets on machine *CRIANN* with parallelization and different chunksize settings.

become bigger, the computing time gets smaller, and then reaches the minimum; after that, the computing time can become much bigger as chunksizes continue growing, due to the waste of computational resources. The vertical dot line shows the chunksize corresponding to the minimum time of each dataset, which varies due to the time and memory consumed to compute Gram matrices. Computation with wise chunksize choices could be more than 20 times faster than the worst case. In our experiments, for convenience of implementations and comparisons, the chunksize to compute an $N \times N$ Gram matrix on a $n_\heartsuit$-core CPU is set to 100 if $N^2 > 100 n_\heartsuit$; and $N^2 / n_\heartsuit$ otherwise. The value 100 is chosen since the runtime is close enough to the minima on all the dataset.

4.4. Performance analysis

In this section, we compare and analyse performance of the kernels for all datasets. Table 3 gathers performance of all these kernels on all datasets. The column 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 will be computed, and average time consumption and its confidence are obtained over the hyper-parameters 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. The column t_{all} exhibits the total time consumed to compute Gram ma-
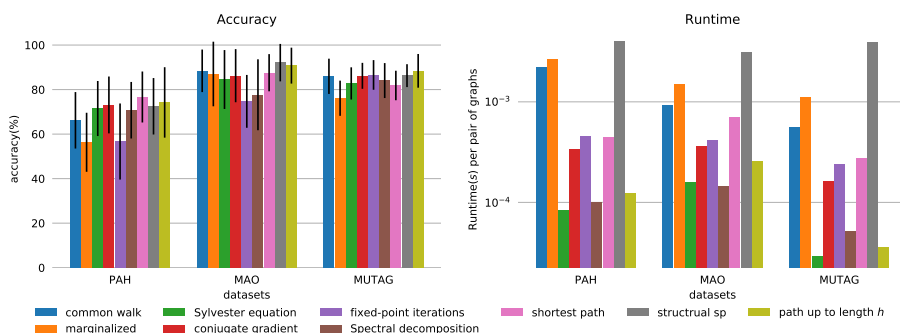
Figure 9: Comparison of accuracy and runtime of all kernels on unlabeled (*PAH*) and labeled datasets (*MAO, Mutag*). Accuracy is the mean value on 30 trials (pillars), with confidence interval around it (error bars).

trix/matrices as well as to perform model selection for each kernel. For regression tasks (*Acyclic* and *Alkane*), the performances are given in terms of boiling points errors; for other datasets, the performances exhibit the classification accuracy in percentage.

4.4.1. Labeled and unlabeled graphs

To study the influence of labeling on performance of graph kernels, we examine 3 datasets that have similar properties (*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 9 exhibits the accuracy of each kernel and the average time to compute each kernel between a pair of graphs. We can see that, for 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. However, there is no significant relationship between the datasets and the runtime of the kernels, which implies that the influence of the graph labeling on runtime is minor on these datasets compared to other factors, such as the average number of graph vertices.

4.4.2. Graphs with symbolic and non-symbolic labels

Non-symbolic labels are able to introduce continuous attributes to graphs. From 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 struc-

31

Table 3: Results of all graph kernels based on linear patterns on all datasets.

| Datasets	Kernels	Train Perf	Valid Perf	Test Perf	Parameters	t_{gm}	t_{all}
Alkane	Common walk	6.48±0.48	11.13±1.79	12.97±14.17	method: geo, γ: 0.08, α: 1e-10	1.93"/2.62"±0.71"	149.00"
	Marginalized	41.48±1.93	40.73±2.04	48.05±15.72	iter: 16, p_q: 0.7, α: 1e-3	3.77"/2.75"±1.23"	264.94"
	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"
	Conjugate gradient	9.21±1.00	12.30±1.92	13.54±11.22	λ: 0.01, α: 3.16e-9	0.61"/0.64"±0.04"	20.91"
	Fixed-point iterations	13.54±0.72	16.44±1.46	13.28±7.27	λ: 1e-3, α: 3.16e-8	0.76"/0.63"±0.07"	18.32"
	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"
	Shortest path	7.80±0.18	8.72±0.22	8.45±1.64	α: 3.16e-4	0.73"	**2.68"**
	Structural SP	5.89±0.17	10.33±0.28	**7.21±1.68**	α: 0.01	1.49"	3.26"
	Path up to length h	0.85±0.03	8.23±1.39	8.61±12.84	h: 9, k_func: MinMax, α: 0.01	0.48"/0.50"±0.04"	42.47"
Acyclic	Common walk	7.22±0.26	12.16±0.92	14.15±5.30	method: geo, γ: 0.02, α: 1.00e-10	1.53"/1.94"±0.38"	142.26"
	Marginalized	11.27±0.40	17.85±1.59	17.75±3.77	iter: 19, p_q: 0.3, α: 1e-5	5.48"/3.47"±1.59"	321.16"
	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"
	Conjugate gradient	9.04±0.46	12.78±1.06	13.98±5.55	λ: 0.01, α: 3.16e-9	0.83"/0.79"±0.10"	25.82"
	Fixed-point iterations	11.77±0.23	13.54±0.44	11.41±2.46	λ: 1e-3, α: 1e-8	0.77"/0.70"±0.05"	19.40"
	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"
	Shortest path	6.22±0.25	9.61±0.56	9.09±2.64	α: 3.16e-9	0.88"	**3.41"**
	Structural SP	3.76±0.15	13.02±1.16	13.60±4.80	α: 1e-3	1.52"	3.85"
	Path up to length h	1.90±0.14	6.93±0.43	**6.44±1.65**	h: 2, k_func: MinMax, α: 3.16e-3	0.52"/0.50"±0.04"	45.70"
MAO	Common walk	97.66±1.04	90.80±3.18	88.43±9.55	method: geo, γ: 0.1, C: 3.16e+5	2.14"/6.11"±3.98"	965.02"
	Marginalized	97.34±0.66	90.55±3.14	87.00±14.51	iter: 7, p_q: 0.6, C: 3.16e+6	3.47"/3.93"±1.78"	3836.35"
	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"
	Conjugate gradient	96.93±1.34	87.89±2.96	86.24±11.93	λ: 0.1, C: 3.16e+5	0.85"/0.76"±0.10"	68.57"
	Fixed-point iterations	80.78±1.71	77.68±3.24	74.71±11.87	λ: 1e-3, C: 3.16e+8	0.96"/0.85"±0.10"	18.58"
	Spectral decomposition	79.50±1.78	79.33±1.92	77.67±15.93	λ: 1e-7, C: 3.16e+4	**0.34"**/1.38"±1.06"	55.28"
	Shortest path	97.30±0.77	88.10±2.47	87.58±8.35	C: 3.16e+3	1.65"	**3.36"**
	Structural SP	98.04±0.62	88.64±2.85	**92.10±8.41**	C: 3.16e+2	7.27"	9.06"
	Path up to length h	98.12±0.63	91.02±2.66	**93.38±10.07**	h: 9, k_func: MinMax, C: 10	0.94"/0.65"±0.19"	40.01"
PAH	Common walk	76.51±1.86	73.07±3.07	66.20±12.68	method: geo, γ: 0.1, C: 3.16e+4	9.87"/32.19"±21.25"	1340.08"
	Marginalized	63.53±1.58	63.61±1.63	56.33±13.26	iter: 13, p_q: 0.1, C: 3.16e-8	11.67"/9.40"±4.46"	673.91"
	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"
	Conjugate gradient	75.81±1.45	71.94±2.48	73.10±12.77	λ: 0.1, C: 3.16e+4	1.50"/1.24"±0.10"	50.35"
	Fixed-point iterations	63.49±2.04	63.58±2.17	56.67±17.09	λ: 1e-4, C: 3.16e+1	2.03"/1.60"±0.38"	**23.50"**
	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"
	Shortest path	78.77±1.39	75.77±2.19	**76.67±11.45**	C: 3.16e+2	1.97"	237.00"
	Structural SP	77.52±1.52	75.00±2.30	72.50±12.68	C: 3.16e+2	17.91"	617.92"
	Path up to length h	76.98±1.15	73.61±2.48	70.20±10.66	h: 1, k_func: MinMax, C: 10	0.53"/0.55"±0.04"	39.43"
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"
	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"
	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"
	Conjugate gradient	92.19±0.76	87.14±1.60	86.18±5.83	λ: 1e-3 C: 3.16e+4	2.84"/2.73"±0.09"	74.39"
	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"
	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"
	Shortest path	98.23±0.40	84.39±2.35	81.84±6.63	C: 1e+3	4.89"	**7.58"**
	Structural SP	100.00±0.00	84.66±1.57	86.26±5.14	C: 1e+9	68.85"	71.18"
	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"
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"
	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"
	Sylvester equation	39.14±0.31	36.26±0.65	37.27±1.93	λ: 0.1, C: 1e+4	13.76"/13.63"±0.50"	29832.59"
	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"
	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"
	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"
	Shortest path	98.96±0.07	93.87±0.29	93.72±1.12	C: 10	36.98"	**255.48"**
	Structural SP	99.10±0.08	94.84±0.23	**94.88±1.24**	C: 10	41.92"	257.96"
	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"
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"
	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"
	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"
	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"
	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"
	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"
	Shortest path	100.00±0.00	68.86±1.91	**70.09±4.20**	C: 1e+6	704.54"	**717.35"**
	Structural SP	-	-	-	-	-	> 12d
	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"
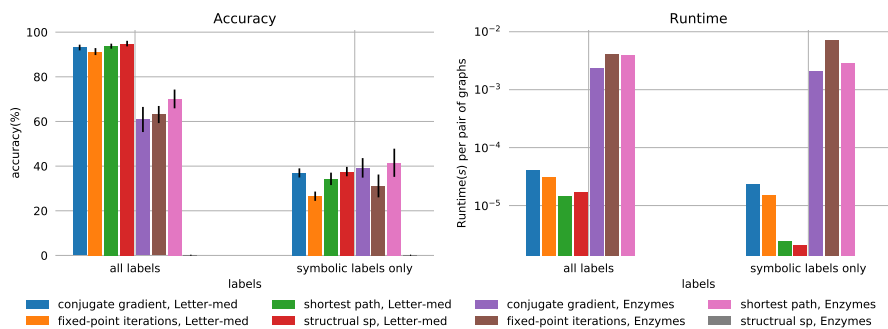
Figure 10: Comparison of accuracy and runtime of graph kernels on datasets with and without non-symbolic labels. The last pillar was removed due to its high computational time.

tural 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 on these 2 datasets, then we remove the non-symbolic labels from the datasets and compute the performance.

Figure 10 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 3 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.

4.4.3. 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
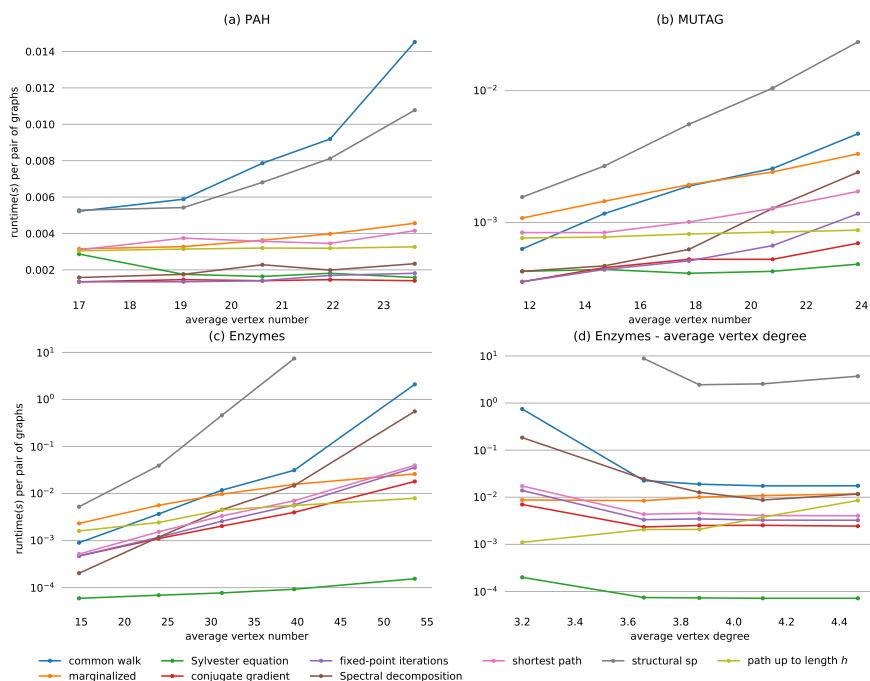
33

Figure 11: Comparison of runtime of each kernel on datasets with different average vertex numbers.

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. Runtime to compute graph kernels of these subsets is compared

Figure 11(a)(b)(c) shows 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 growths the fastest, the runtime for the Sylvester equation kernel and the path kernel up to length h remain relatively stable, while the increase rate of runtime 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

34

phenomenon is more observable for small datasets with a more narrow range of vertex numbers, such as *PAH* shown in Figure 11(a).

4.4.4. 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 4.4.3, we choose a dataset with relatively wide range of vertex degrees, *Enzymes*, order it based on the vertex degree, split it into 5 subsets. Figure 11(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 is directly affected by the average vertex degrees m, namely the structural shortest path kernel and the path kernel up to length h. As a result, Figure 11(d) shows that the runtime for the path kernel up to length h increases as the growth of the average vertex degree. 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 than the others.

4.4.5. Linear and non-linear graphs

Graph kernels based on linear patterns are specific to linear structures in graphs-datasets. In this section, we examine the performance of these graph kernels on non-linear structures. Datasets *MAO* and *Mutag* are employed, since the former contains more linear structures and less cycles, while the latter, composed of aromatic and heteroaromatic compounds, contains more cycles and less linear structures. Besides of that, they share comparable properties (*e.g.* vertex labels, $\bar{n}$, $\bar{m}$ and d in Table 2). From Figure 12, we can see that graph kernels based on linear patterns allow to explore non-linear structures, without a significant impact on the accuracy.

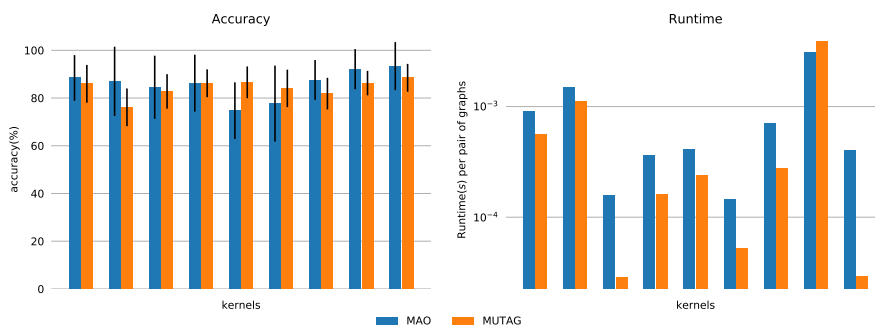Figure 13 compares the runtime to compute Gram matrices and classification errors

Figure 12: Comparison of accuracy and runtime of all kernels on datasets with different amounts of non-linear substructures. Each pair of pillars represents the result of a kernel in Table 1.
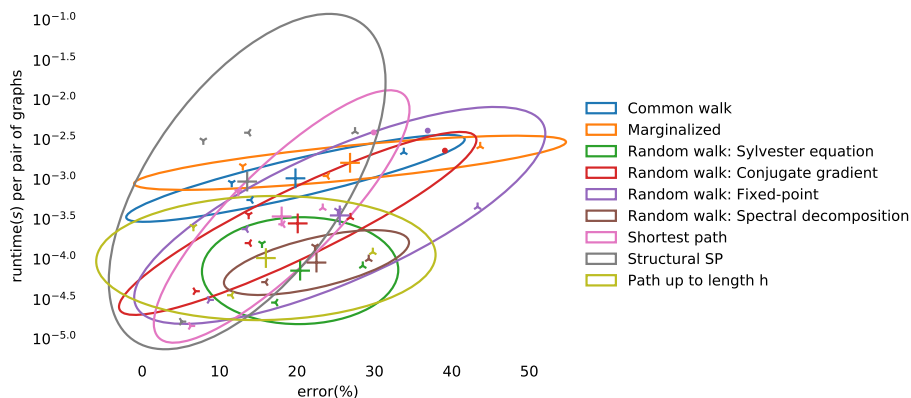

Figure 13: Comparison of runtime versus accuracy error of all graph kernels based on linear patterns. Colors correspond to different kernels, tri-dot markers depict the results on different datasets. The ellipses represent 80% confidence intervals of each kernel, with their centers marked by "+".

of all kernels on all classification datasets. Note that for datasets *Letter-med* and *Enzymes*, kernels that cannot tackle non-symbolic labels are omitted. By portraying the confidence ellipses and their centers, we provide general conclusions on these graph kernel. From a global viewpoint, all kernels provide a good accuracy on all datasets. Examining the center of the ellipse of each kernel, we can see that the marginalized kernel has the worse runtime-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 decom-

36

position kernel and the path kernel up to length h have the good compromise between time complexity and accuracy. Based on this analysis, before choosing a graph kernel, one can have a rough expectation of its performances.

5. Conclusion

In this paper, an extensive analysis of the 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.

The application and generalization of some implementation tricks were rather helpful to reduce real-time computational complexity. Fast computation of the shortest path kernel was extended to any graph kernel that requires comparing vertices and/or edges explicitly. Parallelization was implemented for all kernels. Both strategies significantly reduced the computational time when dealing with real-world datasets, especially large ones. We implemented all graph kernels and applied these tricks to them. The code is available on GitHub as an open-source Python library named `py-graph`[1].

---

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

References

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: Proceedings of the fifth annual workshop on Computational learning theory, ACM. pp. 144–152.

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.

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.

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: Proceedings of the 20th international conference on machine learning (ICML-03), pp. 321–328.

Kobler, J., Schöning, U., Torán, J., 2012. The graph isomorphism problem: its structural complexity. Springer Science & Business Media.

Mahé, P., Ueda, N., Akutsu, T., Perret, J.L., Vert, J.P., 2004. Extensions of marginalized graph kernels, in: Proceedings of 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.

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 (SPR) and Structural and Syntactic Pattern Recognition (SSPR), Springer. pp. 287–297.

Riesen, K., Neuhaus, M., Bunke, H., 2007. Graph embedding in vector spaces by means of prototype selection, in: International Workshop on Graph-Based Representations in Pattern Recognition, Springer. pp. 383–393.

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.

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.

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 .