Pattern Anal Applic (2013) 16:253–283
DOI 10.1007/s10044-012-0284-8

SURVEY

The graph matching problem

Lorenzo Livi · Antonello Rizzi

Received: 9 September 2011 / Accepted: 7 July 2012 / Published online: 21 August 2012
© Springer-Verlag London Limited 2012

Abstract In this paper, we propose a survey concerning the state of the art of the *graph matching problem*, conceived as the most important element in the definition of inductive inference engines in graph-based pattern recognition applications. We review both methodological and algorithmic results, focusing on inexact graph matching procedures. We consider different classes of graphs that are roughly differentiated considering the complexity of the defined labels for both vertices and edges. Emphasis will be given to the understanding of the underlying methodological aspects of each identified research branch. A selection of inexact graph matching algorithms is proposed and synthetically described, aiming at explaining some significant instances of each graph matching methodology mainly considered in the technical literature.

Keywords Graph-based pattern recognition · Inexact graph matching · Graph edit distance · Graph kernels · Graph embedding

1 Introduction

The graph matching problem is a research field characterized by both theoretical and practical issues. This problem has received a great amount of research efforts in the last 30 years, mainly because many pattern recognition problems have been formulated through graphs that are complex combinatorial objects able to model both relational and semantic information in data. They are flexible modeling structures with a vast scientific literature also in many applied contexts, but they lack a strong and well-established mathematical framework for some important operations. For example, the *similarity* of two (real) vectors can be easily defined, but it is not so easy to say how similar two given graphs are. Conversely, the whole set of pattern recognition and machine learning methodologies are well established and tested on standard domains, where basic concepts, like distance between simple patterns, are well defined. Thus, in the past few years, the research challenge was to be able to import the whole set of learning and recognition tools in the domain of graphs. This goal was achieved in two ways: defining a measure of *dissimilarity* directly in the graphs domain, and through a *representation* of them in a suitable space. The numerous matching procedures proposed in the technical literature can be classified into two well-defined families, those of *exact* and *inexact matching*. The first one relies on a *boolean* evaluation of the (dis)similarity of the graphs, while the latter is a more complex problem where the challenge is in computing *how much* they differ. In this survey, our interest will be focused on inexact graph matching related issues, because they are of great interest in a vast range of modern scientific disciplines and applied fields.

Our objective is to both describe the main methodological approaches identified by us in the literature and some of the algorithms, providing a compact, yet clear, taxonomy of these. Some algorithms are simply cited and the related experimental results are not treated in this survey, postponing to other references for a deeper analysis. Of course, the list of presented algorithms does not pretend to be exhaustive. Considering this aim, we will show also some formal definitions and results, limiting the exposition at the essentiality. We will see that for each presented algorithm, it is possible to identify a set of important parameters that by definition influence and, in

L. Livi (✉) · A. Rizzi
SAPIENZA University of Rome, Rome, Italy
e-mail: livi@diet.uniroma1.it

the same time permit, the applicability of these methods to different domains. The exposition of both methods and algorithms aims at homogenizing as much as possible the different notations and viewpoints that can be found in the original contributions.

This article is structured as follows: the preliminary definitions and a brief description of the context is given in Sect. 1. The state of the art methodologies are exposed in Sect. 2. In Sect. 3 some of the most important algorithms are described, followed, in Sect. 3.5, by an analysis concerning their peculiarities. In Sect. 4, we will draw our conclusions together with some interesting new research directions.

1.1 Preliminary definitions

In this section, we will give some basic preliminary definitions, mainly regarding labeled (or attributed) graphs, which is the more general way to define a graph, without assuming restrictions to both vertices and edges label characterization. The definitions are extremely general and can be found in many references [11, 31], or in some of the reviews in [1, 24].

The set of real numbers $\mathbb{R}$ is assumed to be equipped with the number zero, i.e., $\mathbb{R} = \mathbb{R} \cup \{0\}$. In general, the calligraphic form $\mathcal{X}$ denotes a set, $\underline{\mathbf{x}}$ a vector, $\mathbf{A}$ a matrix and $f(\cdot)$ a function (not its evaluation). The element (i, j) of a matrix $\mathbf{A}$ can be referred to as A_{ij} or $[\mathbf{A}]_{ij}$.

Definition 1 (*Labeled graph*) A labeled graph is a tuple $G = (\mathcal{V}, \mathcal{E}, \mu, \nu)$, where

- $\mathcal{V}$ is the (finite) set of vertices (also referred to as nodes),
- $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is the set of edges,
- $\mu : \mathcal{V} \rightarrow \mathcal{L}_{\mathcal{V}}$ is the vertex labeling function with $\mathcal{L}_{\mathcal{V}}$ the vertex-labels set, and
- $\nu : \mathcal{E} \rightarrow \mathcal{L}_{\mathcal{E}}$ is the edge labeling function with $\mathcal{L}_{\mathcal{E}}$ the edge-labels set.

Both $\mu(\cdot)$ and $\nu(\cdot)$ are assumed to be total functions. The items of $\mathcal{E}$ can be denoted with $e_{ij} = (v_i, v_j)$, meaning an edge from vertex v_i to vertex v_j. We can also denote edges with $e = (v, u)$ or $e_i = (v, u),\ i = 1, \ldots, |\mathcal{E}|$, with $v \neq u$, without using any index on the vertices set. If $\mathcal{E}$ is a symmetric relation, $(v, u) \in \mathcal{E} \Leftrightarrow (u, v) \in \mathcal{E}, \forall v, u \in \mathcal{V}$, then the graph G is called an *undirected* graph, conversely it is referred to as a *directed* graph. If $\mathcal{L}_{\mathcal{V}} = \mathcal{L}_{\mathcal{E}} = \emptyset$ then G is referred to as an *unlabeled* graph. In any case the vertices set $\mathcal{V}$ is assumed to be indexed, i.e., $\{v_i\}_{i=1}^{n}$ where $|\mathcal{V}| = n$, so that we can distinguish them. The same is valid for $\mathcal{E}$. If $\mathcal{L}_{\mathcal{E}} \subseteq \mathbb{R}$, then G is usually called a *weighted* graph. An unweighted graph can be seen as a weighted one with $\nu(e) = 1, \forall e \in \mathcal{E}$. If $\mathcal{L}_{\mathcal{E}} \neq \emptyset \wedge \mathcal{L}_{\mathcal{V}} = \emptyset$, the graph G is referred to as *edge-labeled*. If $\mathcal{L}_{\mathcal{E}} = \emptyset \wedge \mathcal{L}_{\mathcal{V}} \neq \emptyset$, the graph G is referred to as *vertex-labeled*. Finally, if both sets are non-empty we can refer to G as a *fully labeled* or simply labeled graph. The notations $\mathcal{V}(G)$ and $\mathcal{E}(G)$ will refer to the set of vertices and edges of the graph G. If it is not explicitly defined, a graph is assumed to be labeled.

Definition 2 (*Walks*) A walk w of length k in a graph G is a sequence of vertices $w = (v_1, \ldots, v_{k+1})$ with $e_{i,i+1} = (v_i, v_{i+1}) \in \mathcal{E},\ i = 1 \rightarrow k$.

A *path* in a graph G is a walk in which $v_i \neq v_j \Leftrightarrow i \neq j$. A *cycle* in a directed graph G is a path with $(v_{k+1}, v_1) \in \mathcal{E}$. A graph G is called *connected* if there is at least one walk between any two vertices.

Definition 3 (*Subgraph*) Let $G_1 = (\mathcal{V}_1, \mathcal{E}_1, \mu_1, \nu_1)$ and $G_2 = (\mathcal{V}_2, \mathcal{E}_2, \mu_2, \nu_2)$ be two labeled graphs. Graph G_1 is a subgraph of G_2, written as $G_1 \subseteq G_2$, if these conditions hold

- $\mathcal{V}_1 \subseteq \mathcal{V}_2$,
- $\mathcal{E}_1 \subseteq \mathcal{E}_2$,
- $\mu_1(v) = \mu_2(v),\ \forall v \in \mathcal{V}_1$, and
- $\nu_1(e) = \nu_2(e),\ \forall e \in \mathcal{E}_1$.

Conversely, graph G_2 is called a supergraph of G_1.

If the second condition is replaced by $\mathcal{E}_1 = \mathcal{E}_2 \cap (\mathcal{V}_1 \times \mathcal{V}_1)$, then we refer to *induced subgraph* by the set of vertices $\mathcal{V}_1$, denoted with $G_1 = G_2[\mathcal{V}_1]$.

Definition 4 (*Neighborhood subgraph*) Given a graph $G = (\mathcal{V}, \mathcal{E}, \mu, \nu)$ and a vertex $v \in \mathcal{V}$, the neighborhood subgraph $G_{|v}$ of v in G is defined as

$$\mathcal{V}_v = \{v\} \cup \{u : (u, v) \in \mathcal{E} \vee (v, u) \in \mathcal{E}\},$$
$$\mathcal{E}_v = \mathcal{E} \cap \{\mathcal{V}_v \times \mathcal{V}_v\},$$
$$\mu_v = \mu_{|\mathcal{V}_v},$$
$$\nu_v = \nu_{|\mathcal{E}_v}.$$

Sometimes, especially from the computational point of view, it is useful to represent the edges labels as a matrix.

Definition 5 (*Edge-labels matrix*) Let G be a graph with $|\mathcal{V}| = n$.The edge-labels matrix is a square matrix $\mathbf{L}_{(\mathcal{E})}^{n \times n}$ with

$$[\mathbf{L}_{(\varepsilon)}]_{ij} = \begin{cases} \nu(e_{ij}) & \text{if } e_{ij} \in \mathcal{E}, \\ \zeta & \text{otherwise.} \end{cases}$$

where the special label ζ means “no label”.

The *adjacency matrix* of G is denoted with $\mathbf{A}^{n \times n}$, and if G is weighted, we have the *weighted adjacency matrix* $A_{ij} = \nu(e_{ij})$, usually denoted as $\mathbf{W}$, which can be thought of as a special case ($L_{\mathcal{E}} \subseteq \mathbb{R}$ and $\zeta = 0$) of the edge-labels matrix shown in Definition 5. For undirected graphs,

matrices $\mathbf{A}, \mathbf{L}_{(\mathcal{E})}$ and $\mathbf{W}$ are symmetric. The *transition matrix* of G is denoted with $\mathbf{T}^{n \times n}$, and is defined as $\mathbf{T} = \mathbf{D}^{-1}\mathbf{A}$, where $D_{ii} = \deg(v_i) = \sum_j A_{ij}$ is a diagonal matrix of vertices degree.

Definition 6 (*Random walks*) A random walk on G is a stochastic process generating sequences of vertices $v_{i_1}, v_{i_2}, \ldots$ according to the conditional probability $P(i_{k+1}|i_1, \ldots, i_k) = T_{i_k, i_{k+1}}$.

The k-th power of this matrix, $\mathbf{T}^k$, describes k-length random walks on G. The component T^k_{ij} gives the probability of a transition from vertex v_i to vertex v_j via a random walk of length k. Similarly, A^k_{ij} gives the number of k-length walks.

1.2 Graph representations and applications

Research on inductive modeling has defined many automatic systems able to cope with patterns defined on $\mathbb{R}^n$ [117]. However, many recognition problems coming from interesting practical applications deal directly with *structured patterns*, such as images [29, 89], audio/video signals [45, 101], chemical compounds [14] and metabolic networks [121], for instance. Usually, to take advantage of the existing data driven modeling systems, each pattern of a structured domain $\mathcal{S}$ is transformed to an $\mathbb{R}^m$ feature vector by adopting a suitable *preprocessing* function $\phi : \mathcal{S} \rightarrow \mathbb{R}^m$. The design of these functions is a challenging problem, mainly due to the implicit *semantic and informative gap* between $\mathcal{S}$ and $\mathbb{R}^m$. A key element to design an automatic system dealing with these recognition problems is the *information granulation* of the input set $\mathcal{S}$ [78, 102]. Granular computing and modeling [5] is a novel paradigm concerned with the analysis of complex data, usually characterized by the need of different levels of representation. The key aspect, and founding concept, of the granular modeling approach is the grouping of low-level atomic elements into semantically relevant groups, called *information granules*. Hence, granular computing consists in finding the *correct* level of information granulation, i.e., a way to map a raw data level domain into a higher semantic level, and in defining a proper inductive inference directly into this symbolic domain. Labeled graphs enter predominantly in this context, because they are general enough to be able to model information granules and their mutual spatio-temporal relations via vertices and edges, respectively, together with their assigned labels. That is, they are able to represent both topological and semantic information of data in a single structure.

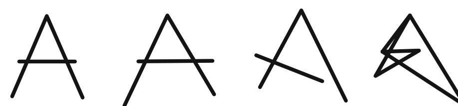

Fig. 2 Letter example

A variegated recent repository of labeled graphs is the IAM graphs database [97]. It consists of different datasets, from different real scientific contexts, such as recognition of characters and molecules. The graph-based representation can be very intuitive and effective when dealing with molecules. For example, in [64] the recognition of *mutagenic* compounds is carried out employing graphs as patterns representing the chemical dataset. The representation of molecules as graphs is straightforward. Indeed, the atoms are the vertices and the covalent bonds become the edges. Vertices are labeled with the corresponding chemical symbol and edges by the valence of the linkage. In [14] data on proteins are considered for recognition. Labeled graphs are constructed considering the secondary structure elements of the proteins and their spatial relations. Indeed, each vertex is connected to the three nearest neighbors in the space. Both vertices and edges are equipped with complex composite type labels, describing both biological and spatial information of data. Figure 1, taken from [14, Figure 2], shows a simple illustration of the graphs' elaboration process.

Another example of graph-based representation comes from the recognition of letters, largely described in [89]. Graphs are employed to represent distorted letter drawings. For example, Fig. 2 shows different levels of distortions applied to the "A" letter. Labeled graphs are constructed representing straight lines by undirected and unlabeled edges and ending points of lines by vertices. Each vertex is

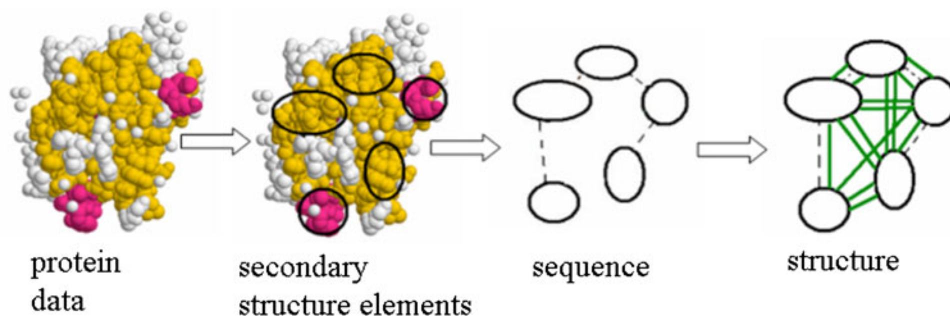

Fig. 1 Graphical representation of data on proteins

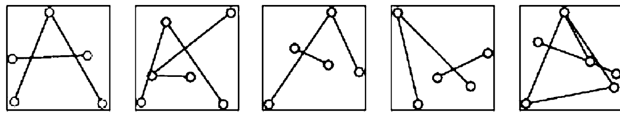

Fig. 3 Graphs representing a (distorted) "A" Letter

labeled with a two-dimensional attribute giving its position relative to a reference coordinate system (usually the 2D plane). Figure 3 shows a sample graph representation, taken from a pattern of the IAM Letter dataset, of a distorted "A" letter.

There are many other fields of application where labeled graphs can be, and have been, applied as a powerful and general representation tool. For example, just to mention a few, we can cite applications to Web content-based information retrieval [110], smart grids modeling [33] and complex networks analysis [10, 21]. Obviously, from the computational viewpoint, there are different interests, not limited to the graph matching problem. Generally speaking, it is of interest to represent data as labeled graphs when both topological and semantic (i.e., labels) information are relevant for the task at hand.

1.3 Exact matching

The exact matching between graphs is characterized by the fact that the mapping between the vertices of the two graphs must be *edge-preserving*, in the sense that if two vertices in the first graph are adjacent, they are mapped to two vertices in the second graph that are adjacent as well. If we consider labeled graphs, we also need to match the labels of both vertices and edges. When this relation is *bijective*, we are talking about the well-known *graph isomorphism* problem.

1.3.1 Graph isomorphism

The strictest form of exact matching between graphs is the graph isomorphism. Informally, it consists in deciding if two given graphs are equivalent in terms of structure and labels. The definition of the problem, considering labeled graphs, is the following:

Definition 7 (*Labeled Graph Isomorphism*) Let $G_1 = (\mathcal{V}_1, \mathcal{E}_1, \mu_1, \nu_1)$ and $G_2 = (\mathcal{V}_2, \mathcal{E}_2, \mu_2, \nu_2)$ be two graphs. A graph isomorphism between G_1 and G_2 is a bijection $f : \mathcal{V}_1 \rightarrow \mathcal{V}_2$ satisfying

- $\mu_1(v) = \mu_2(f(v)),\ \forall v \in \mathcal{V}_1$,
- $\forall e_1 = (u, v) \in \mathcal{E}_1$there exists an edge $e_2 = (f(u), f(v)) \in \mathcal{E}_2$ such that $\nu_1(e_1) = \nu_2(e_2)$,
- $\forall e_2 = (u, v) \in \mathcal{E}_2$ there exists an edge $e_1 = (f^{-1}(u), f^{-1}(v)) \in \mathcal{E}_1$ such that $\nu_1(e_1) = \nu_2(e_2)$.

Two graphs are called *isomorphic* if there exists an isomorphism $f(\cdot)$ between them. Definition 7 is an extension of the classical formulation of the problem to the case of labeled graphs. To establish an isomorphism, one has to map each vertex from the first graph to a vertex of the second graph such that the edge structure is preserved and the vertex and edge labels are consistent to each other. This problem is known to be in NP, neither known to be in P nor NP-complete [42]. We will refer to the isomorphism relation with the notation $G \simeq G'$.

There are well-known special cases where the graph isomorphism problem can be solved efficiently, i.e., polynomial time. For example, checking for the isomorphism between *planar* graphs is known to be solvable in linear time [55].

Another type of exact graph matching is the *graph homomorphism* [54]. This is a weaker form of matching in which adjacent vertices on the first graph must be mapped to adjacent vertices in the second graph, but the correspondence can be many to one.

1.4 Inexact and error-tolerant graph matching

In many real-world applications, especially in the fields of machine learning and pattern recognition, it is more interesting to take into account both structural and labels-related differences between graphs. This need comes from the motivation that graphs that represent patterns from the same *class* may differ only in small parts, due, for example, to external noises. Some of the state-of-the-art methodologies are general enough to be applied to a wide range of graphs, but we think that it is operatively better to distinguish between two main categories: the ones that works well on graphs with simple labels and the ones that are better on (possibly) fully labeled graphs. This distinction, also done in [41], is motivated by the fact that it is possible to formulate very specific and more efficient algorithms that rely on the particular domain of application and structural definition of the graph. There are two other well-done and interesting works with a general point of view about the algorithmic problems concerning the graph matching related issues [23, 89].

One intuitive way to deal with an *imprecise* graph matching consists in evaluating how much two graphs share. This issue can be addressed via the notion of subgraph isomorphism.

Definition 8 (*Subgraph isomorphism*) Let G_1, G_2 be two graphs. An injective function $f : \mathcal{V}_1 \rightarrow \mathcal{V}_2$ is called a subgraph isomorphism from G_1 to G_2 if there exists a subgraph $G \subseteq G_2$ such that $f(\cdot)$ is a graph isomorphism between G_1 and G.

A subgraph isomorphism exists between two graphs if the larger, say G_2, of the two can be reduced into a smaller

graph by removing some vertices and/or edges, and this reduced version is isomorphic to G_1. The subgraph isomorphism problem is known to be NP-complete [24]. Another possibility for the computation of the matching degree between graphs comes from the maximum common subgraph (MCS) problem [73]. Unfortunately, also this problem is known to be NP-hard [42].

Definition 9 (*MCS*) Let G_1, G_2 be two graphs. A graph $G = (\mathcal{V}, \mathcal{E}, \mu, \nu)$ is called a common subgraph of G_1 and G_2 *if* $G \simeq G_1$ and $G \simeq G_2$. A common subgraph G is called maximum (MCS), denoted with G_{MCS}, if there exists no other common subgraph of G_1 and G_2 with more vertices than G.

One naive method to establish a dissimilarity measure, using the MCS, between two graphs is the well-known *MCS distance* [19], defined as

$$d_{\mathrm{MCS}}(G_1, G_2) = 1 - \frac{|\mathcal{V}_{\mathrm{MCS}}|}{\max\{|\mathcal{V}_1|, |\mathcal{V}_2|\}}. \quad (1)$$

If two graphs are isomorphic, their MCS distance is zero, and if they do not share anything, their MCS distance is one [19]. The MCS distance between two graphs is uniquely defined; conversely, the maximum common subgraph is not unique.

2 Methodologies

Given two labeled graphs, G_1 and G_2, the general objective is to be able to match these two structures considering both structural and semantic information, i.e., the information provided by the labels. As in [41], we will refer to the first considered graph as *data graph* and to the second one as the *model graph*. The challenge is in obtaining an estimation of how much the data graph *resembles* the model graph. This generic problem can be formulated in two ways:

- compute the *similarity* or
- compute the *dissimilarity* of these two graphs.

The difference between these two approaches may seem small, although there are theoretical and practical implications. The first approach is based on the representation of graphs in a suitable *implicitly induced vector space*. The second one has two incarnations. The first one aims to estimate the amount of *distortions* needed to transform the data graph into the model graph. This estimation is carried out directly in the domain of the graphs. The second incarnation is again built with the aim of representing the graph in an *explicit embedding* space, where the commonalities between the input graphs should be reflected by their mutual distance in this space. Of course, there are also hybridized formulations. The establishment of a (dis)similarity measure between graphs permits performing recognition and learning tasks with standard tools, such as the k-NN classifier, (fuzzy) neural networks or kernel machines [117]. As in many graph-based problems, one of the main limitation is the computational cost of these procedures. A straightforward consequence is the adoption of some feasible *approximate solution* for both time and space requirements. The objective is to obtain a good trade-off between what is left in the approximation and what is gained in terms of resources. Unfortunately, it is not so easy to achieve this trade-of only by theoretical analysis, leaving the final judgment to the mandatory experimentations.

In the actual scientific literature, we can clearly distinguish three mainstream approaches for the inexact graph matching problem:

1. *Graph edit distance* [17, 40, 83, 84, 87, 88, 89, 90, 94, 98, 109, 129]: these methods match the graphs directly in their domain and, in general, are applicable to a wide class of graphs.
2. *Graph kernels* [14, 43, 50, 63, 66, 75, 79, 85, 86, 123]: they are based on the notion of similarity between two discrete objects that is evaluated on an implicitly induced *feature space*. Being able to define a kernel function for graphs permits importing the whole class of kernel machines on this domain.
3. *Graph embedding* [28, 29, 34, 35, 59, 74, 92, 96, 99, 100, 101, 103, 104, 105, 115]: these methods are based on the embedding of the graph to obtain a general (and usually relative to the data) vector representation. These methods can be seen as a generalization of the graph kernels approach.

2.1 Graph edit distance

The first and very important concept to introduce here is the graph edit distance (GED) measure [17,109], which can be thought as a reformulation of the well-known edit distance for strings, such as the *Levenshtein* distance [72], in the graphs domain. A GED is a measure of dissimilarity between graphs, defined directly in their domain $\mathcal{G}$ as a nonnegative function $d : \mathcal{G} \times \mathcal{G} \rightarrow \mathbb{R}^+$. The problem of defining such a dissimilarity measure can be formulated is this way: given two graphs G_1 and G_2, we calculate the amount of both structural and label's distortions needed to transform G_1 into G_2. To be able to talk about distortions in graphs, we need to introduce the concept of *edit path*.

Definition 10 (*Edit path*) Let G_1, G_2 be two graphs. Any bijection $f : \hat{\mathcal{V}}_1 \rightarrow \hat{\mathcal{V}}_2$, where $\hat{\mathcal{V}}_1 \subseteq \mathcal{V}_1$ and $\hat{\mathcal{V}}_2 \subseteq \mathcal{V}_2$, is called an edit path between G_1 and G_2.

To be able to construct an edit path between two graphs, we need to define the basic *edit operations* that are valid for both vertices and edges. For simplicity, in the following list, we will refer to vertex operations only.

- Substitution: $u \rightarrow v$, with $u \in \mathcal{V}_1$ and $v \in \mathcal{V}_2$. The substitution is in fact a vertex label substitution.
- Deletion: $u \rightarrow \varepsilon$.
- Insertion: $\varepsilon \rightarrow u$.

Other types of basic edit operations can be defined for application-specific purposes, such as vertex merge and splitting [3]. Let o be a complete edit path, i.e., a path that completely transform the data graph into the model graph. We can denote it as a sequence of basic edit operations $o = (o_i, \ldots, o_k) = (u \rightarrow v, \ldots, \varepsilon \rightarrow w)$. Each edit operation o_i has an associated edit cost, denoted with $c(o_i)$. For all pairs of graphs there exist at least one edit path, i.e., by removing all vertices from the data graph and inserting all vertices of the model graph, but this approach is not much informative about the structural dissimilarity of the two graphs.

Definition 11 (*Edit cost function*) The edit cost function is a nonnegative function of the form

$$c : \mathcal{O} \rightarrow \mathbb{R}^+$$

that also satisfies the following inequalities to avoid unnecessary edit operations:

$$c(u \rightarrow w) \leq c(u \rightarrow v) + c(v \rightarrow w)$$
$$c(u \rightarrow \varepsilon) \leq c(u \rightarrow v) + c(v \rightarrow \varepsilon)$$
$$c(\varepsilon \rightarrow v) \leq c(\varepsilon \rightarrow u) + c(u \rightarrow v)$$

$\mathcal{O}$ is the set of all edit paths. It is intuitive to understand that edit paths without unnecessary edit operations are to be considered preferable in this edit model [89]. It is worth stressing that the definition of such edit costs $c(o_i)$ is a crucial task for the inexact graph matching based on GED. Now, we are ready to define the graph edit distance [17, 36, 109].

Definition 12 (*Graph edit distance*) Let G_1, G_2 be two graphs, and let $c(o)$ denote the cost of an edit path o from G_1 to G_2. Let $\mathcal{O}$ be the finite set of edit paths from G_1 to G_2, then the edit distance between G_1 and G_2 is defined as

$$d_{\mathrm{GED}}(G_1, G_2) = \min_{o \in \mathcal{O}} \sum_i c(o_i). \tag{2}$$

The set $\mathcal{O}$ in general is infinite, but with proper observations of some of these given in Definition 11, the number of allowable edit operations can be reduced. We can consider only $|\mathcal{V}_1|$ deletion of vertices from G_1, $|\mathcal{V}_2|$ insertion of vertices from G_2, the $|\mathcal{V}_1| \cdot |\mathcal{V}_2|$ vertices substitutions from G_1 to G_2 and the corresponding $|\mathcal{E}_1| + |\mathcal{E}_2| + |\mathcal{E}_1| \cdot |\mathcal{E}_2|$ operations on edges [89]. In general GED is not symmetric, but if the cost function satisfies the conditions of positive definiteness and symmetry as well as the triangle inequality at the level of single edit operations o_i, the resulting edit distance is known to be a metric [17].

Figure 4, taken from [98, Figure 1], shows a possible sequence of edit operations needed to transform the graph G_1 into the graph G_2.

2.1.1 Exact computation of GED

Computing the *exact* edit distance between two graphs G_1 and G_2 is equal to finding the minimum of Eq. 2. The A* algorithm [17] evaluates all edit paths traversing a search tree in a greedy strategy, choosing, from the current set of edit path candidates, the one with the minimum edit cost. In general, this approach is known to be exponential both in space and time in the number of involved vertices, and thus is practically applicable only to very small graphs [89]. All modern strategies are based on suboptimal solutions of this problem. It was observed that the gain in accuracy, when using the exact procedure, is not justified with the big gap in terms of computation time and resources demand [89]. Its usage is very limited in the domains where the maximum accuracy is mandatory. The A* algorithm is explained in Sect. 3.1.1.

2.2 Graph kernels

In this section, we introduce the concept of kernel function and its application to the domain of graphs, namely the *graph kernels*. Graphs are the most general example of discrete structures, and thus these methodologies can be applied, with proper observations, to any discrete structure, such as strings and automata. Graph kernels rely on the representation of a graph in an implicitly defined feature space, where eventually they are analyzed. Therefore, graph kernels are the key for the application of kernel machines (e.g., *Support Vector Machines* [15, 25]) to the domain of labeled graphs $\mathcal{G}$.

The needed mathematical background is briefly introduced here, starting from kernel functions and the reproducing property in a *Hilbert space*.

Fig. 4 Example of edit path

Definition 13 (*Positive definite kernel function*) Let $\mathcal{X}$ be a generic input space and $k : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$ be a continuous function on the product *space* $\mathcal{X} \times \mathcal{X}$. The function $k(\cdot,\cdot)$ is called a positive definite kernel on $\mathcal{X} \times \mathcal{X}$ if it is symmetric, $k(x,z) = k(z,x)$, $\forall x,z \in \mathcal{X}$, and positive definite, that is $\forall n \in \mathbb{N}, x_1,\ldots,x_n \in \mathcal{X}$ and $c_1,\ldots,c_n \in \mathbb{R}$, it follows that

$$\sum_{i,j\in\{1,\ldots,n\}} c_i c_j k(x_i,x_j) \geq 0.$$

Actually, if the $\geq$ operator is used in Definition 13, the kernel is called positive semi-definite. As many other authors [6], we will avoid the specification. A kernel function provides a way to express the similarity between elements of a (generic) input set. When $\mathcal{X}$ coincides with $\mathbb{R}^n$ there are many different types of kernel functions [111, 113], some of which are listed in Table 1.

We introduce now the definition of *inner product spaces*, a powerful generalization of Euclidean spaces to *vector spaces* geometry, where notions such as *angles* and *length* of vectors (and functions) can be formally defined.

Definition 14 (*Inner product space*) An inner product space $(\mathcal{X}, \langle\cdot,\cdot\rangle)$ is a vector (or linear) space $\mathcal{X}$ along with a function $\langle\cdot,\cdot\rangle : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$ called the inner product, such that

1. $\forall x,y \in \mathcal{X}$ holds $\langle x,y\rangle = \langle y,x\rangle$ (symmetry),
2. $\forall x,y,z \in \mathcal{X}$ and scalar $\alpha \in \mathbb{R}$ holds $\langle \alpha x,y\rangle = \alpha\langle x,y\rangle$ and $\langle x+z,y\rangle = \langle x,y\rangle + \langle z,y\rangle$(linearity),
3. $\forall x \in \mathcal{X}$ holds $x \neq \mathbf{0} \Rightarrow \langle x,x\rangle > 0$ (positiveness).

Every inner product space is a *normed vector space* with the norm $x_2 = \sqrt{\langle x,x\rangle}$, and thus a *metric space* with $d_2(x,y) = \|x-y\|_2 = \sqrt{\langle x-y,x-y\rangle} = \sqrt{\langle x,x\rangle + \langle y,y\rangle - 2\langle x,y\rangle}$. An Hilbert space $\mathcal{H}$ is an inner product space that is also complete (i.e., each Cauchy sequence is convergent in it) with respect to the induced metric by the inner product.

Now, we introduce the important reproducing property of kernel functions, being a fundamental pillar in pattern recognition and machine learning contexts [111, 113].

Table 1 Some kernel functions

Kernel	Formula
Linear	$\langle \underline{\mathbf{x}},\underline{\mathbf{z}}\rangle = \underline{\mathbf{x}}\cdot\underline{\mathbf{z}}$
Polynomial	$(\langle \underline{\mathbf{x}},\underline{\mathbf{z}}\rangle + v)^{\mathrm{d}}$
RBF	$\exp(-\gamma \parallel \underline{\mathbf{x}}-\underline{\mathbf{z}} \parallel^2), \gamma > 0$
General Gaussian	$\exp(-(\underline{\mathbf{x}}-\underline{\mathbf{z}})^{\mathrm{T}}\mathbf{C}(\underline{\mathbf{x}}-\underline{\mathbf{z}}))$
Normalized	$\frac{k(\underline{\mathbf{x}},\underline{\mathbf{z}})}{\sqrt{k(\underline{\mathbf{x}},\underline{\mathbf{x}})\cdot k(\underline{\mathbf{z}},\underline{\mathbf{z}})}}$
Weighted sum	$\sum_i w_i k_i(\underline{\mathbf{X}},\underline{\mathbf{Z}}),\ w_i \geq 0$

Definition 15 (*Reproducing kernels Hilbert space*) Let $\mathcal{H}$ be a set of functions of the form $f : \mathcal{X} \rightarrow \mathbb{R}$. A kernel function $k : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$ is called a *reproducing kernel if*

- $\forall x \in \mathcal{X}$ the function $k(x,\cdot) \in \mathcal{H}$, and
- $\forall x,z \in \mathcal{X}$ and $\forall f(\cdot) \in \mathcal{H}$ the reproducing property holds, i.e., $\langle k(x,\cdot), f(\cdot)\rangle = f(x)$. In particular, $\langle k(x,\cdot), k(z,\cdot)\rangle = k(x,z)$.

$\mathcal{H}$ is called the reproducing kernel Hilbert space (RKHS).

Let $\Phi : \mathcal{X} \rightarrow \mathcal{H}$ be a mapping function that assigns to each pattern $x \in \mathcal{X}$ a function on the domain $\mathcal{X}$, that is $\Phi(x)(\cdot) = k(x,\cdot)$. It is possible to construct a feature space, i.e., an RKHS $\mathcal{H}$, that contains the image of the input patterns of $\mathcal{X}$ under $\Phi(\cdot)$, and where an inner product operator $\langle\cdot,\cdot\rangle_{\mathcal{H}}$ can be evaluated such that $k(x,z) = \langle\Phi(x),\Phi(z)\rangle_{\mathcal{H}}$, $\forall x,z \in \mathcal{X}$, holds [111, 113]. As a consequence, we can state that every reproducing kernel is a positive definite kernel, since:

$$\sum_{i,j\in\{1,\ldots,n\}} c_i c_j k(x_i,x_j) = \left\langle \sum_{i=1}^{n} c_i k(x_i,\cdot), \sum_{i=1}^{n} c_i k(x_i,\cdot) \right\rangle$$
$$= \left\| \sum_{i=1}^{n} c_i k(x_i,\cdot) \right\|^2 \geq 0. \quad (3)$$

The RKHS $\mathcal{H}$ is a Hilbert space of functions. It is possible to formulate what is called a *Mercer kernel* [111, Section 2.2.4], with very similar properties to the ones of reproducing kernels, but the associated Hilbert space, denoted as $\mathcal{H}_k$, is now a sequences space (e.g., an l_2 space) and not a functions space. To be a valid Mercer kernel, $k(\cdot,\cdot)$ must satisfy the well-known *Mercer's conditions* [2, 80], i.e., it must be continuous, symmetric and positive definite. It is possible to show that for each valid Mercer kernel $k : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$ there is at least one mapping function $\Phi : \mathcal{X} \rightarrow \mathcal{H}_k$ such that $k(x,z) = \langle\Phi(x),\Phi(z)\rangle_{\mathcal{H}_k}$, $\forall x,z \in \mathcal{X}$. The inner product is evaluated in $\mathcal{H}_k$, that is dependent on $k(\cdot,\cdot)$ and is most of the times unknown [111, 113]. In the remainder of the paper, we will omit the subscript to the inner product operator.

A notable example of valid kernel is the Gaussian RBF kernel, defined as $k(\underline{\mathbf{x}},\underline{\mathbf{z}}) = \exp(-\gamma \cdot d(\underline{\mathbf{x}},\underline{\mathbf{z}})^2)$, where $d(\cdot,\cdot)$ is a metric distance, $\gamma = 1/2\sigma^2$ and $\underline{\mathbf{x}},\underline{\mathbf{z}}$ are assumed to be real vectors. Of course, the Gaussian RBF kernel can be applied to any generalized input set $\mathcal{X}$. The only requirement is that a suitable (symmetric) distance function $d : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$ must be defined. Anyway, the associated Hilbert space $\mathcal{H}_k$ is an infinite dimensional feature space in this case. A larger class of kernel functions related to positive definite kernels are the *conditionally positive definite kernels*, where $\sum_{i=1}^{n} c_i = 0$ holds. Every positive definite kernel is also a conditionally positive definite

kernel. For the closure property under pointwise addition and multiplication of kernel functions [6], it is possible to derive other kernels that are useful in specific contexts. Other examples of positive definite kernels are the linear, polynomial and (for a restricted range of its parameters) the hyperbolic tangent (also called sigmoid) kernel function. For a more in-depth analysis on kernel functions, together with their applications in kernel machines, see [111, 113].

Both reproducing and Mercer kernel functions are of fundamental importance in machine learning and pattern recognition applications with kernel machines, and the relation $k(x,z) = \langle \Phi(x), \Phi(z) \rangle,\ \forall x, z \in \mathcal{X}$, is referred to as the *kernel trick* in those contexts [2]. Indeed, the term kernel trick is usually intended for the necessity to define only a valid kernel function $k(\cdot,\cdot)$, tailored to the specific nature of $\mathcal{X}$, without any necessity to define explicitly the function $\Phi(\cdot)$ in closed form. In Sect. 3.2, we will show many applications of this property on graph matching related problems, i.e., when $\mathcal{X} = \mathcal{G}$.

To help the reader understand the mechanism behind the kernel trick, we report a simple example that shows explicitly the embedding space $\mathcal{H}_k$. Let $\mathcal{X}$ be $\mathbb{R}^2$. Suppose that the employed valid kernel function is the polynomial kernel with $v = 0$ and $d = 2$, i.e., $k(\underline{\mathbf{x}}, \underline{\mathbf{z}}) = (\underline{\mathbf{x}} \cdot \underline{\mathbf{z}})^2$, where the operator $\cdot$ is the dot product. In this case, it is easy to find a Hilbert space $\mathcal{H}_k$, and a mapping function $\Phi : \mathbb{R}^2 \to \mathcal{H}_k$ such that $(\underline{\mathbf{x}} \cdot \underline{\mathbf{z}})^2 = \langle \Phi(\underline{\mathbf{x}}), \Phi(\underline{\mathbf{z}}) \rangle$. We define the inner product as the dot product, i.e., as $\langle \Phi(\underline{\mathbf{x}}), \Phi(\underline{\mathbf{z}}) \rangle = \Phi(\underline{\mathbf{x}}) \cdot \Phi(\underline{\mathbf{z}})$. Expanding the kernel function, we have $(\underline{\mathbf{x}} \cdot \underline{\mathbf{z}})^2 = (x_1z_1 + x_2z_2)^2 = x_1^2z_1^2 + 2x_1z_1x_2z_2 + x_2^2z_2^2$, that can be also expressed as

$$(\underline{\mathbf{x}} \cdot \underline{\mathbf{z}})^2 = \begin{bmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{bmatrix} \begin{bmatrix} z_1^2 \sqrt{2}z_1z_2 & z_2^2 \end{bmatrix}. \tag{4}$$

Hence, by defining the mapping function as

$$\Phi(\underline{\mathbf{x}}) = \begin{bmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{bmatrix}, \tag{5}$$

we obtain

$$(\underline{\mathbf{x}} \cdot \underline{\mathbf{z}})^2 = \begin{bmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{bmatrix} \begin{bmatrix} z_1^2 \sqrt{2}z_1z_2 & z_2^2 \end{bmatrix} = \Phi(\underline{\mathbf{x}}) \cdot \Phi(\underline{\mathbf{z}}). \tag{6}$$

Therefore, we have shown that $k(\underline{\mathbf{x}}, \underline{\mathbf{z}}) = \Phi(\underline{\mathbf{x}}) \cdot \Phi(\underline{\mathbf{z}})$, where the dot product is actually evaluated in $\mathcal{H}_k = \mathbb{R}^3$.

For example, when the input patterns are confined in $[-1, 1]^2$, the image of $\Phi(\cdot)$ looks like the one shown in Fig. 5 (the figure is taken from [20, Figure 8]).

Now, we introduce some other mathematical background necessary to be able to deal with the concept of *product* of graphs.

Definition 16 (*Kronecker product*) Given two real matrices $\mathbf{A}^{n \times m}$ and $\mathbf{B}^{p \times q}$, the Kronecker product is denoted $\mathbf{A} \otimes \mathbf{B} \in \mathbb{R}^{np \times mq}$ and defined as

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} A_{1,1}\mathbf{B} & A_{1,2}\mathbf{B} & \cdots & A_{1,m}\mathbf{B} \\ \vdots & \vdots & \vdots & \vdots \\ A_{n,1}B & A_{n,2}\mathbf{B} & \cdots & A_{n,m}\mathbf{B} \end{bmatrix}.$$

Unlike matrix multiplication, the Kronecker product $\mathbf{A} \otimes \mathbf{B}$ does not entail a restriction on the size of the involved matrices [8]. Another interesting and useful property of the Kronecker product is

$$(\mathbf{A} \otimes \mathbf{B})(\mathbf{C} \otimes \mathbf{D}) = \mathbf{AC} \otimes \mathbf{BD}. \tag{7}$$

Definition 17 (*Schur product*) Given two real matrices $\mathbf{A}^{n \times m}$ and $\mathbf{B}^{n \times m}$, the Schur (or Hadamard) product is denoted $\mathbf{A} \odot \mathbf{B} \in \mathbb{R}^{n \times m}$ and is defined as the componentwise product $[\mathbf{A} \odot \mathbf{B}]_{ij} = A_{ij}B_{ij}$.

Kronecker and Schur products are linked with relation

$$(\mathbf{A} \otimes \mathbf{B}) \odot (\mathbf{C} \otimes \mathbf{D}) = (\mathbf{A} \odot \mathbf{B}) \otimes (\mathbf{C} \odot \mathbf{D}). \tag{8}$$

See [8, Chapter 7] for a more complete treatment of the Kronecker and Schur algebras. It is possible to extend the Kronecker algebra to graphs, introducing the *Tensor Product* operator for graphs (also called *Direct Product*) [56].

Definition 18 (*Tensor product of graphs*) The tensor productbetween two graphs G_1, G_2, denoted with $G_1 \otimes G_2$, produces a graph $G_\times = (\mathcal{V}_\times, \mathcal{E}_\times)$ defined as

$$\mathcal{V}_\times = \{(v_i, u_r) : v_i \in \mathcal{V}_1, u_r \in \mathcal{V}_2\},$$
$$\mathcal{E}_\times = \{((v_i, u_r), (v_j, u_s)) : (v_i, v_j) \in \mathcal{E}_1 \wedge (u_r, u_s) \in \mathcal{E}_2\}.$$

Note that we have used the same symbol $\otimes$ for both Kronecker and tensor products, because indeed the tensor product of two graphs corresponds to the computation of the Kronecker product of the two respective adjacency matrices of G_1 and G_2. Figure 6 shows an illustrative example of the tensor product between two simple graphs.

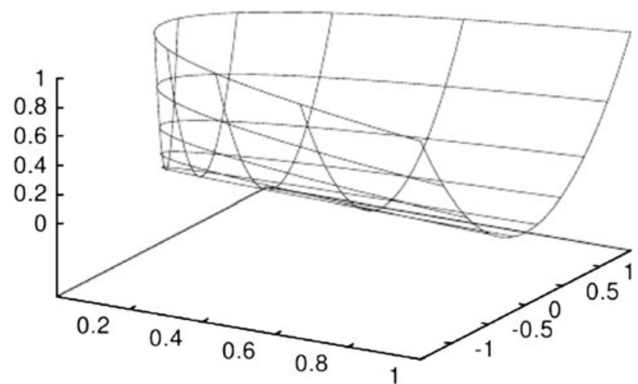

Fig. 5 Image of the mapping function $\Phi(\cdot)$ shown in Eq. 5

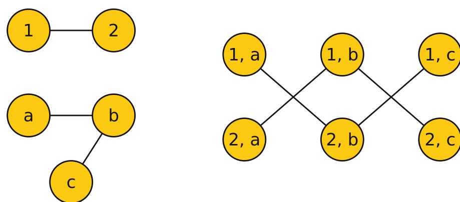

Fig. 6 Example of tensor product between graphs

The tensor product operator $\otimes$ is commutative, associative and has many other interesting and useful properties [56, 95, 108]. For example, it is possible to show that performing a (random) walk on the tensor product graph $G_{\times} = G_1 \otimes G_2$ is equivalent to performing two simultaneous (random) walks on G_1 and G_2. Another important property is that the neighborhood of a vertex $(v, u) \in \mathcal{V}(G_{\times})$,denoted with $\mathcal{N}((v, u))$, is given by the cartesian product $\mathcal{N}(v) \times \mathcal{N}(u)$, and consequently the degree of (v, u) is given by $\deg((v, u)) = \deg(v) \cdot \deg(u)$. The tensor product graph $G_{\times}$, in some sense, is able to encode the commonalities between the two input graphs G_1 and G_2.

2.2.1 Convolution kernels

Convolution kernels, first described in [53] as *R*-convolution kernels, infer the similarity of composite discrete objects from the similarity of their parts. It is intuitive to understand that a similarity function can be more easily defined for smaller parts rather than for the whole composite object. Assuming to be able to calculate the similarities between the simpler parts of the composite objects, a *convolution* operation is applied in order to turn them into a kernel function for the whole object. It is possible to construct a new valid kernel function starting from different distinct kernels, considering the closure property under addition and multiplication by a positive constant of the class of positive definite functions [6].

Definition 19 (*R-convolution kernel*) Let $\mathcal{X}$ be an input space of discrete objects. Let the decomposition in D parts of two elements $\mathbf{x}, \mathbf{x}' \in \mathcal{X}$ be defined as $\mathbf{x} = (x_1, \ldots, x_D)$ and $\mathbf{x}' = (x'_1, \ldots, x'_D)$, respectively. Assume that for each d-th part of the elements, we can calculate the similarity with the kernel $\kappa_{\mathrm{d}}(x_{\mathrm{d}}, x'_{\mathrm{d}})$. Then the similarity between $\mathbf{x}$ and $\mathbf{x}'$ as a whole is defined as the generalized convolution operation

$$k(\mathbf{x}, \mathbf{x}') = \sum_{\substack{(x_1,\ldots,x_D)\in R^{-1}(\mathbf{x}) \\ (x'_1,\ldots,x'_D)\in R^{-1}(\mathbf{x}')}} \prod_{i=1}^{D} \kappa_i(x_i, x'_i), \tag{9}$$

where $R^{-1}(\mathbf{x})$ stands for the set of all possible decompositions of element $\mathbf{x}$. The R-convolution of $\kappa_1, \ldots, \kappa_D$ is denoted with $\kappa_1 \star, \ldots, \star \kappa_D(\mathbf{x}, \mathbf{x}')$.

The ANOVA kernel [127], for instance, is a particular convolution kernel, which uses a subset of the components of a composite object for comparison. Another example is to adopt the RBF kernel [113, 114], i.e., a kernel function of the form

$$\kappa(x, y) = \mathrm{e}^{-\frac{(f(x)-f(y))^2}{2\sigma^2}}, \tag{10}$$

where $f(\cdot)$ is a function from the input space into $\mathbb{R}$. The R-convolution of these functions, assuming only one way to decompose these objects $|R^{-1}(x)| = |R^{-1}(y)| = 1$, is written as

$$\kappa_1(x, y)\star, \ldots, \star\kappa_{\mathrm{D}}(x, y) = \mathrm{e}^{-\sum_{d=1}^{\mathrm{D}} \frac{(f_d(x)-f_d(y))^2}{2\sigma_d^2}}. \tag{11}$$

The R-convolution kernel has laid the groundwork for many graph kernels. The decomposition of a graph G into d parts $(G_1, \ldots, G_d)$, is then mathematically denoted by $R^{-1}(G) = \{(G_1, \ldots, G_d) : R(G_1, \ldots, G_d, G)\}$. The most simple and immediate example of decomposition of a graph is the one that assume the set of all decompositions of a graph $G \in \mathcal{G}$ as the set of its vertices, $R^{-1}(G) = \mathcal{V}$. A general convolution kernel function for graphs $G, G' \in \mathcal{G}$ can then be written as

$$k(G, G') = \sum_{\substack{(G_1,\ldots,G_d)\in R^{-1}(G) \\ (G'_1,\ldots,G'_d)\in R^{-1}(G')}} \prod_{i=1}^{d} \kappa_i(G_i, G'_i), \tag{12}$$

using a specialized kernel function $\kappa_i(\cdot,\cdot)$ for each vertex v_i (i.e., its associated label). It is easy to understand that when dealing with graphs, the definition of $R^{-1}(G)$ becomes critical, because of the computational hardness associated with their combinatorial nature.

An interesting, and mathematically-grounded, way to apply the convolution property to graphs is through the tensor product operator explained in Definition 18. Indeed, this operator can be seen as a rigorous way to *merge* two labeled graphs, employing different specialized valid kernel functions for vertices and edges labels. In Sect. 3.2, we will see different examples of graph kernels using this approach.

2.2.2 Complete graph kernel computation

A complete graph kernel [43, Section 5.3] is based on the notion of isomorphism between two graphs and the relative equivalence classes induced on the set of all graphs $\mathcal{G}$. The quotient set $\mathcal{G}/\simeq$ implies the relation $G \simeq G' \Rightarrow \phi(G) = \phi(G')$, where $\phi(G)$ is the mapping of the graph in a high-dimensional feature space. Hence, a *complete* graph kernel is able to recognize all non-isomorphic graphs. This kernel is useful when one wants to distinguish between graphs that differ only in their vertices' identifiers. Now, we introduce the definition of complete graph kernel mainly to show an example of an NP-hard formulation of a graph kernel.

Definition 20 (*Complete graph kernel*) Let $\phi : \mathcal{G} \rightarrow \mathcal{H}$ be a map from $\mathcal{G}$ into a Hilbert space $\mathcal{H}$, and let $k : \mathcal{G} \times \mathcal{G} \rightarrow \mathbb{R}$ be a kernel such that $\langle \phi(G), \phi(G') \rangle = k(G, G')$. If $\phi(G) = \phi(G')$ implies that $G \simeq G'$ for all graphs in $\mathcal{G}$, then $k(\cdot, \cdot)$ is called a complete graph kernel.

It is easy to understand that with such definitions, $G \simeq G' \Leftrightarrow \phi(G) = \phi(G')$ holds for each pair of graphs in $\mathcal{G}$. It is possible to show that computing a complete kernel is as hard as deciding if two graphs are isomorphic, and consequently NP-hard. Similarly, if the graph kernel is based on (all) their common subgraphs, it is possible to show that there is no polynomial time algorithm for computing the graph kernel [43, section 5.3].

2.3 Graph embedding

In the field of *structural pattern recognition*, the input domain can be any generalized set $\mathcal{X}$ where the common mathematical structure of metric, normed or inner product spaces cannot be defined obviously. The notion of *neighborhood* of objects of a generic input space $\mathcal{X}$ is extremely linked to the notion of *commonality* between these. If two objects share many common descriptive attributes, then they must result *close* in the representation framework. Generalizing the definition of a *topological* space, one could be able to deal with this kind of very general representation [92]. The primitive and intuitive notion of *dissimilarity* between objects of a generic input space can be used to build various pattern recognition and learning systems. A dissimilarity is basically a generalization of a metric distance that requires fewer constraints about its definition. The dissimilarity is the dual concept of the similarity, of which kernel functions are only an example. The dissimilarity measures can be defined in various domains, and not only in metric spaces. For example, a measure of *divergence* between two distributions is the well-known Kullback–Leibler divergence [69]. This is not a metric because it is not symmetric, in general. The notion of dissimilarity can be employed to produce a generalized notion of topology over a generic set of objects. Another good reference containing (but not limited to) this aspect related to (dis)similarity functions on a generalized space $\mathcal{X}$ can be found in [117, Section 11.2].

The embedding between spaces plays a crucial role in this scenario, because it permits associating the generic input space $\mathcal{X}$ to a known space, where classical and well-tested recognition and learning algorithms can be applied directly. We can cite two main applications regarding this issue: dissimilarity space embedding [92] and metric space embedding [77]. The first one has been already applied in problems regarding graphs [100], i.e., where the input space is a set of graphs, $\mathcal{X} = \mathcal{G}$. We will proceed with some definitions regarding generalized metric spaces, Lipschitz functions, space embedding and related issues. For a more in-depth treatment see, for example, [82, 92, 128].

Definition 21 (*Generalized metric spaces*) Given a set $\mathcal{X}$ and a dissimilarity function $\rho : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}^+$, the pair $(\mathcal{X}, \rho)$ is said to be:

1. Hollow space: *if* $\rho(\cdot, \cdot)$ is reflexive.
2. Premetric space: a hollow space obeying the symmetry constraint.
3. Quasimetric space: a premetric space obeying the definiteness constraint.
4. Semimetric space: a premetric space that satisfies the triangle inequality.
5. Metric space: if $\rho(\cdot, \cdot)$ is reflexive, positive definite, symmetric and satisfies the triangle inequality.

As we have said before, the establishment of a dissimilarity $d : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}^+$ is able to naturally induce a *generalized topology* over the same generic set $\mathcal{X}$, via the notion of *open ball neighborhood basis*.

Definition 22 (*Open ball neighborhood basis*) Given a set $\mathcal{X}$ and a generalized distance $d : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}^+$, the open ball neighborhood for $x \in \mathcal{X}$ is defined as $B_\epsilon(x) = \{y \in \mathcal{X} : d(x, y) < \epsilon\}$, for $\epsilon > 0$. The open ball neighborhood basis is defined as $\mathcal{N}_B(x) = \{B_\epsilon(x) : \epsilon > 0\}$.

The neighborhood basis can be used to entirely describe the (pre-)topology of a set $\mathcal{X}$ [82, 92]; hence, also a generalized definition of the topology of this set can be given. This kind of definition of topology automatically induces a *generalized neighborhood function* $v : \mathcal{X} \rightarrow \mathcal{P}(\mathcal{P}(\mathcal{X}))$, that establishes at each $x \in \mathcal{X}$ its neighborhood $\mathcal{N}(x)$ (with $\mathcal{N}_B(x) \subseteq \mathcal{N}(x)$), i.e., a collection of subsets over $\mathcal{X}$ containing objects that are in some sense similar to each other. Given the neighborhoods, one can conceive a dissimilarity measure also between generalized sets using only topological information of the input data, employing, for instance, extensions of the Hausdorff distance [92, Section 5.5]. These generalizations are well described and widely contextualized in [92], where they are employed as some of the founding concepts of the theory of the *dissimilarity representations*.

When we basically deal with sets, it is possible to observe that the generalization can be conceived also in a more radical way, employing, for example, concepts from the fuzzy sets theory [133] and the more recent development of the field known as granular modeling of systems and data [5]. However, a deep treatment of these topics is out of the scope of this review.

2.3.1 Embeddings

The following results are taken mostly from [92, 128]. We start by introducing the concept of embedding between finite spaces, considering the properties derived from the dissimilarity measure adopted in the input space.

Definition 23 (*Isometric embedding*) Let $(\mathcal{X}, d)$ and $(\mathcal{Y}, \rho)$ be two metric spaces. $(\mathcal{X}, d)$ is isometrically embeddable into $(\mathcal{Y}, \rho)$ if there exists a mapping function, called isometry, $\phi : \mathcal{X} \rightarrow \mathcal{Y}$ such that $d(x, y) = \rho(\phi(x), \phi(y)),\ \forall x, y \in \mathcal{X}$.

Two spaces are isometrically isomorphic if there exists a bijective isometry between them. Every complete metric space is isometrically isomorphic to a closed subset of some *Banach* space. Every metric space is isometrically isomorphic to a subset of some normed vector space. Two Hilbert spaces are always isometrically isomorphic. A Euclidean space $(\mathbb{R}^m, d_2)$ is embeddable in a Hilbert space, and every finite subset of m elements in a Hilbert space can be embedded in $(\mathbb{R}^{m-1}, d_2)$. Not every metric space can be embedded in a Hilbert space:

Theorem 1 (Schoenberg) *Given* $p \in (0, 2]$, *and* $r \in (0, \frac{p}{2}]$ *the space* $(\mathbb{R}^m, l_p^r)$ *is isometrically embeddable in a Hilbert space*.

This theorem applies to both metric and non-metric spaces, using a suitable power r of the adopted dissimilarity function $l_p(\cdot, \cdot)$. However, not every metric space is isometrically embeddable to $l_1(\cdot, \cdot)$, $l_2(\cdot, \cdot)$ or $l_\infty(\cdot, \cdot)$ derived metric spaces. Note that when finite-dimensional spaces (e.g., Euclidean spaces) are considered, $l_2(\cdot, \cdot)$ induced metric corresponds to the Euclidean metric distance $d_2(\cdot, \cdot)$.

Now, we show an important concept of continuity of a mapping function between two metric spaces that finds interesting applications also in pattern recognition contexts employing explicit embedding strategies.

Definition 24 (*Lipschitz mapping function*) Let $(\mathcal{X}, d)$ and $(\mathcal{Y}, \rho)$ be two metric spaces. A mapping function $\phi : \mathcal{X} \rightarrow \mathcal{Y}$ is said to be Lipschitz continuous if there exists a constant $k \geq 0$ such that $\rho(\phi(x), \phi(y)) \leq k \cdot d(x, y),\ \forall x, y \in \mathcal{X}$.

The smallest k is called *Lipschitz number* or *Lipschitz constant* of $\phi(\cdot)$ and is usually denoted with $L(\phi)$ [128]. If $k < 1$ the mapping is a *contraction* of the original set. Conversely, if $k > 1$, it is called an *expansive map*. Finally, if $k = 1$ the mapping is called *non-expansive*. A Lipschitz mapping $\phi(\cdot)$ can be seen as an explicit embedding method able to preserve and bound, with a constant scaling factor k, mutual distances of the elements of the original input set into the embedding space. This fact is of practical importance for pattern recognition problems, since patterns distances in the representation space are the primary source of information of any inductive modeling system. A function $\phi(\cdot)$ is called *locally* Lipschitz continuous if for every $x \in \mathcal{X}$ there exists a neighborhood $\mathcal{N}(x)$ of x such that $\phi(\cdot)$, restricted to $\mathcal{N}(x)$, is Lipschitz continuous. Note that Lipschitz continuity always implies uniform continuity, and every uniform continuous function is continuous. Conversely, not every continuous function is uniformly continuous, and thus Lipschitz. For example, $f(x) = x^2$ is only locally Lipschitz.

Admitting *distortions* in the embedding procedure, it is possible to make another kind of embedding, called *bi-Lipschitz* embedding.

Definition 25 (*bi-Lipschitz embedding*) Given two metric spaces, $(\mathcal{X}, d)$ and $(\mathcal{Y}, \rho)$, and an embedding $\phi : \mathcal{X} \rightarrow \mathcal{Y}$, we say that the embedding function is a distorted embedding function, or bi-Lipschitz, if there exist $r > 0$ and $c \geq 1$ such that $r \cdot d(x, y) \leq \rho(\phi(x), \phi(y)) \leq c \cdot r \cdot d(x, y),\ \forall x, y \in \mathcal{X}$. The real number c is the distortion of the embedding.

A *scalar-valued* Lipschitz function is a function of the form $\phi : \mathcal{X} \rightarrow \mathbb{F}$, where $\mathbb{F}$ is a generic field (such as the reals $\mathbb{R}$).

Definition 26 (*Lipschitz space*) Let $(\mathcal{X}, d)$ be a metric space. Then $Lip(\mathcal{X})$ is the Lipschitz complete (Banach) vector space of all bounded scalar-valued Lipschitz functions $\phi(\cdot)$ on $\mathcal{X}$, with (Lipschitz) norm $\phi_L = \max(\phi_\infty, L(\phi))$.

A pseudo-Euclidean space [48] is a simple kind of *decomposable* inner product space where an *indefinite inner product* can be defined, forming an indefinite inner product space. It is basically an inner product space that satisfies only symmetry and linearity conditions (see Definition 14). We will see that this concept plays an important role in some topics concerning the *linear* embedding of spaces. Formally,

Definition 27 (*Pseudo-Euclidean space*) A pseudo-Euclidean space $\mathbb{E} = \mathbb{R}^{(n,m)}$ is a real vector space equipped with an indefinite inner product $\langle \cdot, \cdot \rangle_{\mathbb{E}}$ that admits a direct orthogonal decomposition $\mathbb{E} = \mathbb{E}_+ \oplus \mathbb{E}_-$, where $\mathbb{E}_+ = \mathbb{R}^n$ and $\mathbb{E}_- = \mathbb{R}^m$ are the set of positive and negative vectors, respectively.

Therefore, the indefinite inner product $\langle \cdot, \cdot \rangle_{\mathbb{E}}$ is definite positive on $\mathbb{R}^n$ and negative definite on $\mathbb{R}^m$. Note that a vector $\underline{\mathbf{x}}$ is said to be positive (negative) if $\langle \underline{\mathbf{x}}, \underline{\mathbf{x}} \rangle > 0$ ($\langle \underline{\mathbf{x}}, \underline{\mathbf{x}} \rangle < 0$) holds. Consequently, this definition extends directly to subspaces [92, Section 2.7].

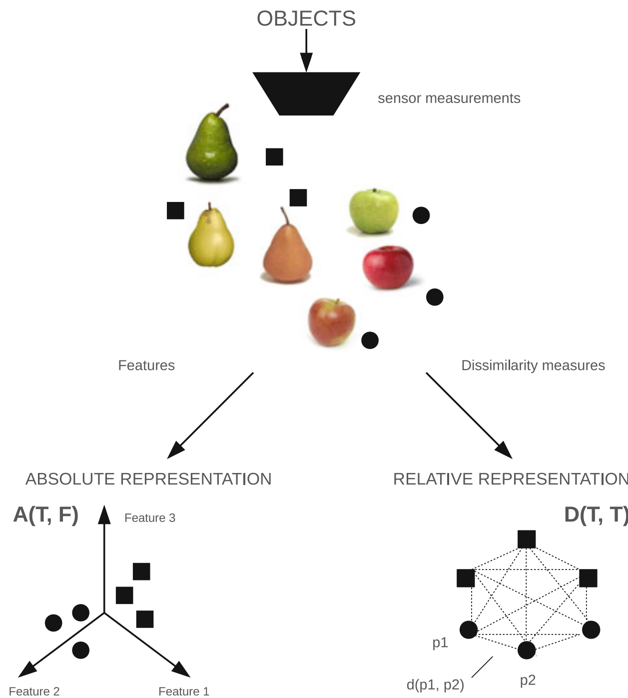

Fig. 7 Absolute and relative representations of data

2.3.2 Dissimilarity representations

Given a generalized finite metric space $(\mathcal{X}, d)$, where $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ and $d(\cdot, \cdot)$ is a general dissimilarity measure on the elements of $\mathcal{X}$, a *dissimilarity matrix* $\mathbf{D}$ is a (square) matrix with $D_{ij} = d(x_i, x_j), \forall x_i, x_j \in \mathcal{X}$. Thus $\mathbf{D}$ is nonnegative and has a zero diagonal. The dissimilarity representation [92] for an input space $\mathcal{X}$ is written as $D(\mathcal{X}, \mathcal{R}) \in \mathbb{R}^{n \times r}$. This notation means that all the objects in $\mathcal{X}$ are represented *relatively* to the objects in $\mathcal{R} \subseteq \mathcal{X}, |\mathcal{R}| = r \leq n$. This is the first and the most important characteristic of this type of representation, as the objects are described considering only *local* reference systems. Conversely, the positive definite kernel functions implicitly represent the objects (i.e., graphs, strings etc…) in a pre-defined feature space. Figure 7 (a re-elaboration of [92, Figure 3]) shows an intuitive explanation of the key aspects introduced by the relative representation scheme followed by the dissimilaritiescomputation, against the standard feature-based, or absolute, representations.

A dissimilarity matrix $\mathbf{D}$ is called *Euclidean*, also denoted with $\mathbf{D}^{*2}$, if $D_{ij} = d_2(x_i, x_j), \forall x_i, x_j \in \mathcal{X}$.

Definition 28 (*Metric for D*) A dissimilarity matrix $\mathbf{D}$ is metric if the triangle inequality $d_{ij} \leq d_{ik} + d_{kj}$ holds for all triplets (i, j, k).

If two objects, say the i-th and the j-th are equal, then $d_{ij} = 0$ and we have that $d_{ik} = d_{jk}, \forall k$. If the dissimilarity matrix $\mathbf{D}$ is not Euclidean, then it is possible to apply a correction of the form $\mathbf{D}' = \mathbf{D} + c(\underline{\mathbf{1}}\underline{\mathbf{1}}^{\mathrm{T}} - I)$, with $c > \max_{p,q,r} |d_{pq} + d_{pr} - d_{qr}|$. If $\mathbf{D}$ is Euclidean, then any concave and nondecreasing transformation $D_{ij} = f(d_{ij})$ of the dissimilarity values will preserve the Euclidean properties.

Definition 29 (*Euclidean behavior*) A dissimilarity matrix $\mathbf{D} \in \mathbb{R}^{n \times n}$ is Euclidean if it can be embedded in a Euclidean space $(\mathbb{R}^m, d_2)$, with $m \leq n$.

This means that an input space (feature vectors) $\mathcal{X} = \{\underline{\mathbf{x}}_i\}_{i=1}^n$ is embeddable in $\mathbb{R}^m$ with $d_2(\underline{\mathbf{x}}_i, \underline{\mathbf{x}}_j) = \underline{\mathbf{x}}_i - \underline{\mathbf{x}}_j 2 = D_{ij}, \ \forall i, j$. The Euclidean property is a favorable aspect of $\mathbf{D}$, that permits its embedding without loss of information regarding the input dissimilarities of data. Refer to [92, Section 3.4] for more details on how to test the Euclidean behavior of dissimilarity matrices and for various correcting techniques.

Generally speaking, considering a dissimilarity representation of a training set $\mathcal{T}$, it is convenient to find a subset of *prototypes* $\mathcal{P} \subseteq \mathcal{T}$ that are able to approximately describe the entire set considering some quality measure. In this case, the representation can be generalized using a relative representation $D(\mathcal{T}, \mathcal{P})$, that is, describing the entire set as a function of a relatively small set of its elements, mostly due to computational speed-up purposes. An embedding can be found directly for $D(\mathcal{P}, \mathcal{P})$, projecting the other $\mathcal{T} \backslash \mathcal{P}$ elements into the embedding space, or defining an embedding that resembles a Lipschitz mapping between the input space and the representation space. The latter was first exploited in [100] considering graphs, and is described in this paper in Sect. 3.3.1. In this case, prototypes selection techniques play a crucial role.

Given an input set of generic *non-represented* objects $\mathcal{R} = \{p_1, \ldots, p_n\}$ and the dissimilarity matrix $D(\mathcal{R}, \mathcal{R})$, three possibilities are outlined in [92] for the dissimilarity representation of $\mathcal{R}$: *direct employment* of the dissimilarities, *linear embedding* [92, Section 3.5] and finally *spatial representation* [92, Section 3.6] of $D(\mathcal{R}, \mathcal{R})$.

Direct employment of dissimilarities A direct use of the dissimilarity values D_{ij} in, for example, a classifier using the k-NN rule. Any monotone nondecreasing concave transformation can be applied directly to the dissimilarities D_{ij}. This is a simple, but absolutely not trivial, example of how to employ the dissimilarities directly in a recognition system in order to define a proper inductive logic inference.

Linear embedding A linear embedding in a (pseudo-)Euclidean space $\mathbb{R}^k, k \leq n$, consists in finding a vector *configuration* $\mathcal{X} = \{\underline{\mathbf{x}}_1, \ldots, \underline{\mathbf{x}}_n\}$ with $\underline{\mathbf{x}}_i \in \mathbb{R}^k$. The *representation matrix* $\mathbf{X} \in \mathbb{R}^{n \times k}$, built with the vectors $\underline{\mathbf{x}}_i, i = 1 \rightarrow n$, should be found preserving the dissimilarities of $D(\mathcal{R}, \mathcal{R})$. If the dissimilarity matrix $\mathbf{D}$ is Euclidean; a linear mapping preserving this kind of information can be

found using projection techniques of the family of *Classical Scaling* [12, 57]. The main problem here is how to find the right k such that the dissimilarities are preserved. The embedding dimension k is found with a PCA-like analysis [62], taking into account only the first $k \leq n$ dimensions corresponding to the first k eigenvectors with the largest eigenvalues (in absolute value). The representation $\mathbf{X}$ is found as

$$\mathbf{X} = \mathbf{Q}_k \Lambda_k^{1/2}, \tag{13}$$

where $\mathbf{Q}_k \in \mathbb{R}^{n \times k}$ is the matrix of the first k eigenvectors (as columns) and $\Lambda_k^{1/2}$ is the nonnegative diagonal matrix of the first k largest eigenvalues of the gram matrix $\mathbf{G} = \mathbf{Q}\Lambda^{1/2}\mathbf{Q}^{\mathrm{T}}$, obtained from the relation with the input (Euclidean) dissimilarity matrix $\mathbf{D}$. If the dissimilarity matrix $\mathbf{D}$ is Euclidean, then the Gram matrix $\mathbf{G} = -\frac{1}{2}\mathbf{JDJ}$ is positive (semi)definite, where $\mathbf{J}$ is referred as the *centering matrix*. If $\mathbf{D}$ is not *perfectly* Euclidean, some corrections should be applied. When these corrections are not applicable or not sufficient, a pseudo-Euclidean embedding of $\mathbf{D}$ can be applied in $\mathbb{R}^{(p,q)}$ [48], obtaining

$$\mathbf{X}\mathbf{J}_{pq}\mathbf{X}^{\mathrm{T}} = \mathbf{G} = \mathbf{Q}\Lambda\mathbf{Q}^{\mathrm{T}} = \mathbf{Q}|\Lambda|^{1/2}\begin{bmatrix} \mathbf{J}_{pq} & \\ & 0 \end{bmatrix}|\Lambda|^{1/2}\mathbf{Q}^{\mathrm{T}}, \tag{14}$$

with $k = p + q$ and Λ a diagonal matrix containing, in order, p positive and q negative eigenvalues both in decreasing order, followed by zeros. $\mathbf{J}_{pq}$ is called the *fundamental symmetry matrix*. This matrix allows the computation of the inner product (then the norm and distance) in a standard Euclidean space $\mathbb{R}^{p+q}$ related to the pseudo-Euclidean space $\mathbb{R}^{(p,q)}$ [92, Section 3.5.3]. The configuration $\mathbf{X} = \mathbf{Q}_k|\Lambda_k|^{1/2}$ can be found in $\mathbb{R}^{(p,q)}$ using only the first k non-zero eigenvalues. Other linear embedding techniques can be found in [92, Section 3.5].

Spatial representation Given an input set of generic objects $\mathcal{R}$ and the relative dissimilarity matrix $D(\mathcal{R}, \mathcal{R})$, a spatial representation of $\mathbf{D}$ is a configuration of vectors representing the objects in a space, directly derived considering the rows of $\mathbf{D}$ as vectors, usually in a (pseudo-)Euclidean space. Spatial representations are to be considered approximate embeddings into suitable low-dimensional vector spaces, mostly used for data visualization purposes. The cited methodologies include *FastMap* [37] and some nonlinear multidimensional scaling techniques [12], such as the *Sammon mapping* [107] with its variations, the *least squares scaling* (LSS) [26, 67, 68]. These methods are based on a *loss function* that estimates the *stress* $S(\cdot)$ after the projection in a vector configuration $\mathbf{X}$ (the row matrix with projected vectors). For example, the simplest loss function is the *raw stress*, defined as

$$S^{\mathrm{raw}}(\mathbf{X}) = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\left(f(D_{ij}) - d_{ij}(\mathbf{X})\right)^2, \tag{15}$$

where $f(\cdot)$ is a continuous monotonic transformation for the dissimilarities, and $d_{ij}(\mathbf{X})$ stands for the distance calculation between the i-th and j-th vector configuration. The family of Sammon mappings [92, Section 3.6] are nonlinear projection techniques from a high-dimensional Euclidean vector space to a low-dimensional space.

2.3.3 Embedding in structure spaces

In [59], a method to embed a graph into what is called the $\mathcal{T}$-space has been proposed. The basic idea is to look at the graphs as equivalence classes of vectors via their weight matrices $\mathbf{W}$, where the elements of the same equivalence class are different vector representations of the same graph. Direct edge-labeled graphs are considered in [59]. To apply this methodology, all input graphs are to be *aligned* to the same bounded order, say n. Considering a graph G, two weight matrices are said to be equivalent if one can be obtained from the other with a permutation of the vertices order. Formally,

$$\mathbf{W} \sim \mathbf{W}' \Leftrightarrow \exists \mathbf{P} \in \mathcal{T} : \mathbf{P}^{\mathrm{T}}\mathbf{W}\mathbf{P} = \mathbf{W}', \tag{16}$$

where $\mathcal{T}$ denotes the set of all $n \times n$ permutation matrices. It is possible to *embed* the matrix $\mathbf{W}$ into a Euclidean vector space $\mathcal{X} = \mathbb{R}^{n \times n}$; as a consequence, each graph can be represented with a vector $\underline{\mathbf{x}}$ just concatenating the columns of $\mathbf{W}$. So, in what follows we can consider equivalent the matrix and the respective vector representations, using, for simplicity, the latter.

Given a Euclidean vector space $\mathcal{X} = \mathbb{R}^m$, let $\mathcal{T}$ be the set of permutation matrices; the *orbit* of $\underline{\mathbf{x}} \in \mathcal{X}$ is defined as

$$[\underline{\mathbf{x}}]_{\mathcal{T}} = \{\mathbf{P}\underline{\mathbf{x}} : \mathbf{P} \in \mathcal{T}\}, \tag{17}$$

i.e., the set of all the equivalent vector representations of $\underline{\mathbf{x}}$. The quotient set over $\mathcal{X}$ is referred to as the $\mathcal{T}$-space and $\mathcal{X}$ is the *representation space* of $\mathcal{X}_{\mathcal{T}}$.

Definition 30 ($\mathcal{T}$*-space*) A $\mathcal{T}$-space over a Euclidean vector space $\mathcal{X}$ is the orbit space $\mathcal{X}_{\mathcal{T}} = \mathcal{X}/\mathcal{T}$ of all orbits of $\underline{\mathbf{x}} \in \mathcal{X}$ under the action of $\mathcal{T}$,

$$\mathcal{X}_{\mathcal{T}} = \mathcal{X}/\mathcal{T} = \bigcup_{\underline{\mathbf{x}} \in \mathcal{X}} [\underline{\mathbf{x}}]_{\mathcal{T}}. \tag{18}$$

The embedding space $\mathcal{X}_{\mathcal{T}}$ aims at getting rid of the different equivalent representations of a given graph. Figure 8, taken from [59, Figure 5], shows an illustrative example of this embedding method.

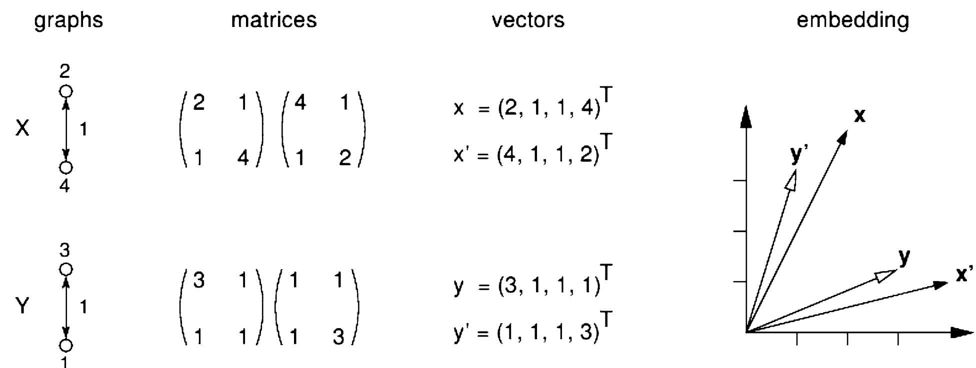

Fig. 8 The embedding of two sample graphs

2.3.4 Other embeddings

There are other kinds of embedding for graphs [34, 35, 96, 105]. For example we can cite the *Spectral Embedding* [94, 103, 104], that consists in finding a proper representation of the graph analyzing the set of its *eigenvectors*. For this purpose both adjacency, transition and *Laplacian* matrix can be used. Another possibility comes from the embedding of the graph into a *Riemannian manifold*, using metric properties derived from *differential geometry* operators [105].

3 Algorithms

The first macro-class of graphs taken in consideration can be defined as the ones with simple type of labels for both vertices and edges. A simple type is, for example, a scalar number or string considered as an element in a finite *nominal* set. For this kind of graph, a straightforward approach is the one that firstly transforms the graph into a sequence of its vertices, applying then a matching method for sequences over them. Two particular types of graphs are directed acyclic graph (DAG) and trees; in particular when dealing with trees, the tree edit distance (TED) is adopted [7, 9, 119, 120]. In this survey, we will describe only some of the seriation-based methods [94, 103, 104, 115, 115, 132]. For a good treatment of TED techniques, and other kind of matching between non-labeled graphs, see [41, Section 4.2.1].

Considering complexly labeled input graphs, i.e., labels in $\mathbb{R}^n$ or other kind of composite types, some issues arise. The most interesting in this case is the capability of inexact graph matching algorithms to manage the commonalities in terms of both topology and labels in a unified framework. The algorithms belonging to GED-based, graph kernel-based and graph embedding-based families are general enough to be applicable to a wide range of types of graphs.

3.1 GED based

Those methods are focused on the estimation of the amount of distortions needed to transform the data graph into the model graph. An important factor is how they define the edit costs. These costs can be known a priori or we will see that they can be defined in such a way that edit operations that are very likely to occur have in fact lower edit costs than the infrequent ones. Moreover, the edit costs can be estimated directly equipping each algorithm with specific low-level dissimilarity function tailored to the particular labels definition. Indeed, most of the GED-based algorithms are applicable to virtually any type of labeled graphs.

3.1.1 GED computation based on the A* algorithm

The A* algorithm [17] employs a search tree to model the edit paths, referred to as *OPEN* in Algorithm 1, which is constructed by considering each vertex of the first graph one after the other $(u_1, u_2, \ldots, u_{|\mathcal{V}_1|})$. $g(o)$ is the function evaluating the cost of the optimal path from the root node of the search tree to the current node o found by the A*. $h(o)$ denotes the estimated costs from o to a leaf node. Finally, $g(o) + h(o)$ gives the *heuristic* estimation of the current node (edit operation) o. In each step, the next unprocessed vertex of the data graph u_{k+1} is selected from *OPEN* and tentatively substituted by all unprocessed vertices of the model graph (line 11) as well as deleted (line 12). Edit operations on edges are implied by edit operations on their adjacent vertices and the costs of these are dynamically added to the corresponding paths. The currently most promising node o of the search tree is the one minimizing the A* search costs $g(o) + h(o)$ (line 5). When a complete edit path is obtained in this way, it is guaranteed to be an optimal one [52]. In the simplest scenario, the estimated lower bound $h(o)$ of the costs from o to a leaf node is set to zero for all o. This means that no heuristic information of the potentially best search direction is used at all for actually performing a breadth-first search. On the other hand, it

is possible to compute a complete edit distance for each node of the search tree. In this case, the function $h(o)$ is not a lower bound, but the exact value of the optimal costs. Whether or not heuristics $h(\cdot)$ are used to manage the search tree traversal process, the cost is exponential in the number of vertices of involved graphs [17].

Algorithm 1 GED computation with the A* Algorithm

Input: The data graph $G_1 = (\mathcal{V}_1, \mathcal{E}_1, \mu_1, \nu_1)$ and the model graph $G_2 = (\mathcal{V}_2, \mathcal{E}_2, \mu_2, \nu_2)$
Output: The minimum edit path o_{min} from G_1 to G_2

1: $OPEN = \emptyset$
2: $\forall w \in \mathcal{V}_2$, insert the vertex substitution $\{u_1 \rightarrow w\}$ into $OPEN$
3: insert vertex deletion $\{u_1 \rightarrow \varepsilon\}$ into $OPEN$
4: **loop**
5: Remove current $o_{min} = \arg\min_{o \in OPEN}\{g(o) + h(o)\}$ from $OPEN$
6: **if** o_{min} is a complete edit path **then**
7: **return** o_{min}
8: **else**
9: Let $o_{min} = \{u_1 \rightarrow v_{i_1}, ..., u_k \rightarrow v_{i_k}\}$
10: **if** $k < |\mathcal{V}_1|$ **then**
11: $\forall w \in \mathcal{V}_2 \setminus \{v_{i_1}, ..., v_{i_k}\}$, insert $o_{min} \cup \{u_{k+1} \rightarrow w\}$ into $OPEN$
12: Insert $o_{min} \cup \{u_{k+1} \rightarrow \varepsilon\}$ into $OPEN$
13: **else**
14: Insert $o_{min} \cup \bigcup_{w \in \mathcal{V}_2 \setminus \{v_{i_1}, ..., v_{i_k}\}} \{\varepsilon \rightarrow w\}$ into $OPEN$
15: **end if**
16: **end if**
17: **end loop**

A* *Beamsearch* The first approximation method of the original A* algorithm is based on *beam search* [90]. Instead of expanding all successor nodes in the search tree, only a fixed number s of nodes to be processed are kept in the *OPEN* set at all steps. Whenever a new partial edit path is added to the *OPEN* set, only the first s partial edit paths with the lowest costs, given by $g(o) + h(o)$, are kept, and the remaining partial edit paths in *OPEN* are simply removed. This means that only those vertices that belong to the most promising partial matches are expanded. If only the partial edit paths with the lowest costs are considered, a suboptimal edit path will be obtained which yields a suboptimal distance, almost close to the exact edit distance.

A* *Pathlength* The second variant [90] of the original A* algorithm follows the empirical observation that if we are considering graphs with a rather large number of vertices, it is likely that a considerable part of an optimal edit path $o_{\min}$ is constructed in the first few steps of the tree traversal, because most substitutions between similar graphs have small costs. Whenever the first significantly more expensive edit operation occurs (in the optimal edit path), this vertex will prevent the tree search algorithm from quickly reaching a leaf node and unnecessarily make it expand to a large part of the *OPEN* search tree. An additional weighting factor $t > 1$ is proposed, aiming to favor longer partial edit paths. Practically, the evaluation of the edit path (line 5 of Algorithm 1) is changed in $\frac{g(o)+h(o)}{t^{|o|}}$, where $|o|$ stands for the length of the current edit path o.

3.1.2 Neighborhood subgraph

In [89],a totally new and fast suboptimal algorithm based on the inexact matching of neighborhood subgraphs has been proposed . The main objective is to propose a polynomial-time algorithm that uses only local information. We can represent a neighborhood subgraph as a sequence of its vertices [89]. The matching is then performed as a *cyclic strings* matching algorithm [18, 93] based on standard string edit distance that is known to be resolvable in quadratic time. The proposed algorithm is a greedy iterative algorithm that adds a sequence of edit operations calculated on the neighborhood graph matching to the global edit path. If the maximum degree of the considered graphs is d, the alignment task has a complexity of $O(d^2)$ and the approximate algorithm terminates after $O(n)$ iterations, with $n = |\mathcal{V}_1| + |\mathcal{V}_2|$. So the total complexity of this algorithm is $O(nd^2)$. This simple method is suboptimal, but it was shown that it is substantially faster than the exact computation of GED [89]. The pseudo-code of the algorithm can be found in [89, Algorithm 3.2]. The type of graphs that can be analyzed with this method depends of the capability of the employed string edit distance algorithm of dealing with data from the set $\mathcal{L}_{\mathcal{V}}$. However, the edges labels are not considered in this scheme. This simple heuristic method does not depends on parameters different from the usual edit costs.

3.1.3 Quadratic programming approach

In [87], a strategy has been proposed that bypasses the standard A*-based algorithms by addressing the graph edit distance problem by means of quadratic programming [16, 91]. This approach is based on the definition of a *fuzzy edit path* between two labeled graphs that allows vertices and edges of one graph to be simultaneously assigned to several vertices and edges of another graph. A fuzzy edit path is defined by assigning a weight to each possible vertex substitution. Let G, G' be two labeled graphs with $|\mathcal{V}| = n$ and $|\mathcal{V}'| = n'$, then there exists $n \cdot n'$ distinct substitutions of a vertex $u \in \mathcal{V}$ with a vertex $v \in \mathcal{V}'$. Formally, a fuzzy edit path between G and G' is a function $w : \mathcal{V} \times \mathcal{V}' \rightarrow [0, 1]$ satisfying the conditions

$$\sum_{v \in \mathcal{V}'} w(u, v) = 1, \forall u \in \mathcal{V} \quad \text{and} \quad \sum_{u \in \mathcal{V}} w(u, v) = 1, \forall v \in \mathcal{V}'. \tag{19}$$

If two vertices, say u, v, have a large value of $w(u, v)$, then they are assumed to correspond to a good structural match. As we will see, this value is determined in the optimization process, assigning a high value to this substitution if the involved edit costs, considering also the edges costs, are relatively low.

The algorithm employs a cost matrix $\mathbf{Q} \in \mathbb{R}^{nn' \times nn'}$ to encode vertex and edge edit costs and minimizes the overall edit costs corresponding to a fuzzy edit path. The

matrix **Q** is constructed such that the rows and columns are indexed by the same substitutions $u \to v$, with $u \in \mathcal{V}$ and $v \in \mathcal{V}'$. Diagonal entries hold the costs of vertex substitutions, while off-diagonal entries correspond to edge edit costs. For instance, the entry of **Q** at position $(u \to v, u \to v)$ is set to the vertex substitution costs of $u \to v$ and the entry at position $(u \to v, p \to q)$ is set to the edge edit costs resulting from substituting $u \to v$ and $p \to q$, if the edges $e_i = (u, p)$ and $e_j = (v, q)$ exist. The $n \cdot n'$-dimensional vector of fuzzy weights $\underline{\mathbf{x}}$ (the solution), satisfying the conditions listed in Eq. 19, is determined as the one that minimizes the expression $\underline{\mathbf{x}}'\mathbf{Q}\underline{\mathbf{x}}$, considering every possible, and valid, vector of weights $\underline{\mathbf{x}}$. Once the optimal vector $\underline{\mathbf{x}}$ is determined, it follows a *defuzzification* stage, where a standard edit path is eventually obtained from $\underline{\mathbf{x}}$, and from which is derived the edit distance from G to G' [87].

The time complexity of this method depends on the complexity of the particular quadratic programming algorithm adopted in the optimization. This method is applicable to virtually any kind of labeled graphs. For what concerns the parameters, the edit costs must be defined or learned.

3.1.4 Assignment problem approach

A recent method, proposed in [98], is based on a polynomial time optimization procedure to solve the GED problem as an *assignment problem*, on the base of the *Munkres' algorithm* [81]. In practice, it is an optimization problem with the aim of finding the lowest cost assignment between objects from two different sets. The Munkres' algorithm is known to solve optimally the assignment problem in cubic time, but actually it provides a fast suboptimal solution to the exact GED computation shown in Eq. 2.

Given two labeled graphs G_1, G_2, with $|\mathcal{V}(G_1)| = n$ and $|\mathcal{V}(G_2)| = m$, a square cost matrix **C** of order $n + m$ is defined with the aim of encoding all the possible edit operations costs, considering all the vertices of the two graphs. The cost matrix **C** is defined as a square matrix of the form

$$\mathbf{C} = \left[\begin{array}{cccc|cccc} c_{1,1} & \cdots & \cdots & c_{1,m} & c_{1,\varepsilon} & \infty & \cdots & \infty \\ c_{1,2} & c_{2,2} & \cdots & c_{2,m} & \infty & c_{1,\varepsilon} & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \infty \\ c_{1,n} & \cdots & \cdots & c_{n,m} & \infty & \cdots & \infty & c_{n,\varepsilon} \\ \hline c_{\varepsilon,1} & \infty & \cdots & \infty & 0 & 0 & 0 & 0 \\ \infty & c_{\varepsilon,2} & \cdots & \cdots & 0 & 0 & 0 & 0 \\ \cdots & \cdots & \cdots & \cdots & 0 & 0 & 0 & 0 \\ \infty & \cdots & \infty & c_{\varepsilon,m} & 0 & 0 & 0 & 0 \end{array}\right],$$

where the symbols $c_{i,j}, c_{i,\varepsilon}, c_{\varepsilon,j}$ denote, respectively, substitution, deletion and insertion costs of vertices for G_1. The left upper corner of the cost matrix **C** represents the costs of all possible vertex substitutions, the diagonal of the right upper corner the costs of all possible vertex deletions, and the diagonal of the bottom left corner the costs of all possible vertex insertions. The edit operation costs of the involved edges can be added directly inside the vertices ones. That is, the information used by this algorithm is properly local. The optimization algorithm, with the cost matrix **C**, produces a permutation $p = p_1, \ldots, p_{n+m}$ of the integers $1, \ldots, n+m$ that minimizes $\sum_{i=1}^{n+m} [\mathbf{C}]_{ip_i}$, which is equivalent to the minimum cost assignment of vertices of G_1 to the vertices of G_2, represented, respectively, by the rows and columns of the matrix **C**.

The computational complexity of this method is $O(n^3)$, where $n = |\mathcal{V}(G_1)| + |\mathcal{V}(G_2)|$. This method is applicable to virtually any kind of labeled graphs. As usual, edit cost parameters must be determined.

3.1.5 Edit cost estimation strategies

One of the most important task in any GED-based algorithm is the definition of the edit costs $c(o_i)$. These costs can be known a priori in some application domain, but in general their definition remains a problem. The first and most intuitive way to define these costs is to directly estimate the distance between labels. The standard Euclidean distance could be employed for this purpose when $\mathcal{L}_\mathcal{V} \subseteq \mathbb{R}^n$ and $\mathcal{L}_\mathcal{E} \subseteq \mathbb{R}^m$. Given two graph G_1, G_2 and four nonnegative weighting parameters α_{vertex}, α_{edge}, β_{vertex}, β_{edge}, the edit costs for all vertices $u \in \mathcal{V}_1, v \in \mathcal{V}_2$ and edges $p \in \mathcal{E}_1, q \in \mathcal{E}_2$ could be defined as [89]:

$$\begin{aligned} c(u \to \varepsilon) &= \beta_{\text{vertex}} \\ c(\varepsilon \to v) &= \beta_{\text{vertex}} \\ c(u \to v) &= \alpha_{\text{vertex}} \cdot \parallel \mu_1(u) - \mu_2(v) \parallel_2 \\ c(p \to \varepsilon) &= \beta_{\text{edge}} \\ c(\varepsilon \to q) &= \beta_{\text{edge}} \\ c(p \to q) &= \alpha_{\text{edge}} \cdot \parallel \nu_1(p) - \nu_2(q) \parallel_2 \end{aligned} \quad . \tag{20}$$

Note that the Euclidean distance in Eq. 20 can be generalized to any dissimilarity measure defined over the vertex and edge labels sets. Edit costs depend on weighting parameters that can be defined as predefined values, based on the application at hand. If a set of graphs $\mathcal{G}$ is available, we can estimate the edit costs from the information contained in $\mathcal{G}$ in a supervised or unsupervised manner. In the first case, a *cost function* must be defined to guide the estimation procedure. It is worth stressing that this cost function is strictly application dependent. If, for example, the GED is used in a classification system, this cost function can be the *generalization capability* of the classification model. Two unsupervised edit cost estimation methods [83, 84] are briefly discussed in the following two paragraphs.

Probabilistic estimation In [83], it is proposed to automatically derive edit costs from an estimation of the (unknown) probability distribution of the edit operations in a sample set of labeled graphs $\mathcal{G}$. For each pair of graph, say G_1, G_2 in $\mathcal{G}$, two initially empty graphs are constructed in a stochastic fashion. That is, each vertex or edge insertion is interpreted as a random event, transforming the data graph into the model graph through edit operations. For instance, a simultaneous vertex or edge insertion in both graphs is equivalent to a substitution operation, and an insertion in one single graph is equivalent to an insertion or deletion operation. Weighted Gaussian mixture densities are used for the approximation of every type of edit event-operation cost (normally three types for the vertices and three for the edges). Given a multivariate Gaussian density (MGD) $f(\cdot|\mu, \Sigma)$ with mean μ and covariance matrix Σ, the probability of an edit path $(o_1, \ldots, o_l)$ is given by

$$p(o_1, \ldots, o_k) = \prod_{j=1}^{k} \beta_{t_j} \sum_{i=1}^{m_{t_j}} \alpha^i_{t_j} f(o_j|\mu^i_{t_j}, \Sigma^i_{t_j}), \tag{21}$$

where β_{t_j} is a model weight for the type of edit operation t_j, and, for each component $i \in \{1, \ldots, m_{t_j}\}$, a mixture weight $\alpha^i_{t_j}$, a mean vector $\mu^i_{t_j}$, and a covariance matrix $\Sigma^i_{t_j}$. The estimation of the parameters Φ of the MGD is then performed using the well-known EM algorithm [30]. Assuming $\Psi(G_1, G_2)$ as the set of all edit paths between G_1 and G_2, the joint probability $p(G_1, G_2)$ of two graphs is defined as

$$p(G_1, G_2) = \max_{(o_1,\ldots,o_l)\in\Psi(G_1,G_2)} P(o_1, \ldots, o_l|\Phi). \tag{22}$$

A dissimilarity measure $d(\cdot, \cdot)$ is obtained as $d(G_1, G_2) = \log(p(G_1, G_2))$.

SOMs-based estimation In [84], a procedure based on self-organizing maps (SOM) [65] is proposed to infer the edit costs of graphs with labels in $\mathbb{R}^n$. The idea is to represent the distribution of the vertex and edge labels occurring in a set of fully labeled graphs through an SOM model, where distances of mapped labels in the *sampling grid* of neurons of the SOM correspond to the inferred edit operation costs. The initial sampling grid is built with equidistantly labeled neurons, with weights of the same size as the ones of the labels. In practice, a mapping between the labels space of the graphs and the grid of the SOM is performed. Given two graphs, say G_1 and G_2, the spatial distance in the sampling grid of two mapped elements (vertex or edge labels), say $\underline{\mathbf{x}} \in \mathbb{R}^n$ and $\underline{\mathbf{y}} \in \mathbb{R}^n$, whose edit operations $\underline{\mathbf{x}} \rightarrow \underline{\mathbf{y}}$ occur frequently in the GED computation, is iteratively reduced during the learning process of the SOM. Thus, the sampling grid is *deformed* according to the labels distribution frequency in the input set of graphs, with the aim of minimizing the resulting edit cost of the GED. Actually, a set of SOMs, one for each edit operation, is employed in this learning model, assigning to each edit operation o_i the respective SOM.

Once an SOM, one for each type of edit operation, has been trained, the cost of a substitution of two labels is defined to be proportional to the deformed distance between the respective neurons in the trained grid. Similarly, the cost of an insertion (deletion) operation is proportional to the average deformed distance from the winner grid neuron to its connected neighbors.

3.2 Graph kernels based

In this section, we will describe some polynomial time algorithms concerning graph kernels. They are based on the well known kernel tricks, that is, given a valid graph kernel function $k(\cdot, \cdot)$, an implicitly defined high-dimensional features space $\mathcal{H}_k$ exists, where graphs are represented and the recognition task is (indirectly) performed.

3.2.1 Exact matching direct product kernel

The idea, described in [43, Section 5.4.2], is to define a graph kernel on the basis of the number of *exact* matching walks in the two involved graphs, taking into account the label sequences of both vertices and edges.

This method employs a tensor product graph $G_\times$ that considers also the labels, that is, two vertices are considered adjacent in $G_\times$ if the related labels are exactly the same in the two input graphs. This is a seminal work concerning this approach, and we will see that, regardless the clear limitations implied by the exactness of the constraints imposed on the labels, it has spawn many other similar approaches. Considering that the number of walks in the product graph is equal to the product of the number of walks in the two original input graphs [56], it is possible to define the direct product kernel as follows:

Definition 31 (*Direct product kernel*) Let G_1, G_2 be two graphs. For a given sequence of weighting parameters $\lambda_1, \lambda_2, \ldots, (\lambda_n \in \mathbb{R}; \lambda_n \geq 0\ \forall n \in \mathbb{N})$, the direct product kernel is defined as

$$k(G_1, G_2) = \sum_{i,j=1}^{|V_\times|} \left[\sum_{n=0}^{\infty} \lambda_n \mathbf{A}^n_\times \right]_{ij}, \tag{23}$$

if the limit exists.

By definition, the power of the adjacency matrix $\mathbf{A}_\times$ of $G_\times$ encodes the information of the number of walks between the two input graphs. The computation of the matrix power series shown in Eq. 23 is carried out via the *eigendecomposition* of $\mathbf{A}_\times$, that also binds the overall computational complexity in the polynomial class [43,

Section 5.2.4]. However, the precise time complexity cost is dependent on the particular type of eigendecomposition algorithm (that can be thought as roughly cubic in the order of the matrix $\mathbf{A}_\times$).

3.2.2 Similarity-based random walk kernel

The original random walk kernel is defined by means of the transition matrix $\mathbf{T}_\times$ of the direct product graph [44]. The kernel can be interpreted as a measure of the probability of *exact-matching* labeled random walks in both graphs. Note that $\mathbf{A}_\times$ and $\mathbf{T}_\times$ can be used independently in Eq. 23, but inducing a different, yet valid, meaning for the matching. The main limitation of this kind of kernel is certainly that it is only applicable in contexts where the strict exact match of the labels is meaningful.

To overcome this limitation, a new random walk kernel has been proposed in [14], with good results to the problem of protein function prediction. The idea is not to evaluate if two walks are identical, but rather if they are similar. To this aim, the adjacency matrix of the direct product graph $G_\times$ is modified as follows:

$$[\mathbf{A}_\times]_{(u,u'),(v,v')} = \begin{cases} k((u,u')(v,v')) & (u,u'),(v,v') \in \mathcal{E}_\times, \\ o & \text{otherwise.} \end{cases} \tag{24}$$

where the kernel $k(\cdot,\cdot)$ measures the similarity of two pair of vertices of $G_\times$, considering both vertex and edge labels of the two input graphs. To this aim, it is possible to convolute different specialized valid kernels $k_\mathcal{V}(\cdot,\cdot), k_\mathcal{E}(\cdot,\cdot)$ as follows:

$$k((u,u'),(v,v')) = k_\mathcal{V}(u,u') \cdot k_{\in}\mathcal{E}_\times((u,v),(u',v')) \cdot k_\mathcal{V}(v,v'). \tag{25}$$

Note that different convolution schemes can be adopted, such as taking the minimum or the average. For example, $k_\mathcal{V}(\cdot,\cdot)$ and $k_\mathcal{E}(\cdot,\cdot)$ can be evaluated as Gaussian RBF kernels of the form:

$$k_\mathcal{V}(u,u') = \exp\left(-\frac{d(\mu(u)-\mu(u'))^2}{2\sigma_\mathcal{V}^2}\right), \tag{26}$$

where $d(\cdot,\cdot)$ is a suitable dissimilarity function for the specific labels set (i.e., $\mathcal{L}_\mathcal{V}$ or $\mathcal{L}_\mathcal{E}$). Note that the matrix $\mathbf{A}_\times$ is by definition a weighted adjacency matrix. As usual, to obtain a valid random walk kernel, we just need to use the transition matrix $\mathbf{T}_\times$ instead.

When using the Gaussian RBF kernels in the product scheme, as shown in Equations 26 and 25, respectively, the matching algorithms become dependent on three parameters, namely the two $\sigma_\mathcal{V}$ and the $\sigma_\mathcal{E}$. These parameters must be adapted for each specific dataset. With regard to the computational complexity, it is easy to understand that, asymptotically, it is equal to the one of the methods studied in Sect. 3.2.1, with additional constant-time costs associated to the labels dissimilarity computations. However, these costs associated with the labels dissimilarities are not in general cost free. Indeed, this method can be applied virtually to any kind of labeled graphs. Hence, the sets $\mathcal{L}_\mathcal{V}$ and $\mathcal{L}_\mathcal{E}$ can be defined as sets of any complex composite types, such as text excerpts and chemical formulas.

3.2.3 Random walk edit kernel

The aim of the method proposed in [86] is to use together the GED and a graph kernel. The basic idea is to enhance the random walk kernel shown in Sect. 3.2.2 with an edit distance matching at the global level, reducing the size of resulting direct product graph $G_\times$. Assume that an optimal edit path from G to G' has been computed with a GED-based method (see for example Sect. 3.1), and let $\mathcal{S} = (v_1 \to v'_1, v_2 \to v'_2, \ldots)$ denote the set of vertex substitutions in the optimal edit path. The adjacency matrix of the direct product graph $G_\times$ is then defined as

$$[\mathbf{A}_\times]_{(u,u'),(v,v')} = \begin{cases} k((u,u'),(v,v')) & \text{if holds } ((u,u'),(v,v')) \in \varepsilon_x, \\ & u \to u' \in S, \\ & \text{and } v \to v' \in S, \\ 0 & \text{otherwise.} \end{cases} \tag{27}$$

The walks are restricted to vertices that satisfy the optimal vertex-to-vertex correspondences identified by the edit distance computation by the GED. This adjacency matrix is then used with the direct product kernel described in Definition 31. It is possible to observe that this method is actually a hybridized algorithm employing a GED-based algorithm with the convolution scheme based on tensor product operator. Consequently, the method as a whole becomes dependent on the particular GED design and in the convolution scheme adopted in the tensor product (i.e., the particular convolution scheme of kernel functions shown in Eqs. 25 and 26, respectively). Moreover, also its complexity is dependent on the particular GED algorithm. As a whole, this algorithm is applicable to any kind of labeled graphs, considering the same observations given in Sect. 3.2.2.

3.2.4 Marginalized graph kernel

Another error-tolerant random walk graph kernel for labeled graphs, based on the convolution operation briefly introduced in Sect. 2.2.1, is described in [63]. Let $\underline{\mathbf{h}} = (h_1,\ldots,h_k), h_i \in \mathcal{L}_\mathcal{E}$, be the labels sequence of a random walk of length k over a given graph G. In a scenario where both vertex and edge labels must be considered, the

sequence of labels $\underline{\mathbf{h}}$ of a random walk of length k is defined as $\underline{\mathbf{h}} = (h_1, \ldots, h_{2k-1})$, i.e., as a sequence of alternating vertex and edge labels. Let the starting and stopping probability vectors be $\underline{\mathbf{p}}$ and $\underline{\mathbf{q}}$, respectively. Assuming dealing only with the labels of the edges, it is possible to compute the probability of the random walk $i_1, \ldots, i_{t+1}$ (sequences of vertex indices) of length t, with the associated labels sequence $\underline{\mathbf{h}} = (h_1, \ldots, h_t)$ as

$$p(\underline{\mathbf{h}}|G) = q_{i_{t+1}} \prod_{j=1}^{t} T_{i_j, i_{j+1}} p_{i_1}. \tag{28}$$

A valid kernel for the computation of the similarity of two sequences of labels, belonging to two different graphs G and G', can be defined as

$$k_z(\underline{\mathbf{h}}, \underline{\mathbf{h}}') = \prod_{i=1}^{t} \kappa(h_i, h'_i) = \prod_{i=1}^{t} \langle \phi(h_i), \phi(h'_i) \rangle, \tag{29}$$

if $\underline{\mathbf{h}}$ and $\underline{\mathbf{h}}'$ have the same length, and otherwise zero. In Eq. 29, $\kappa(\cdot,\cdot)$ is a valid kernel defined over the labels of the edges (e.g., an RBF kernel). The *marginalized kernel* [63] between graphs is then obtained as the *expectation* of $k_z(\cdot,\cdot)$ over every possible labels sequences,

$$k(G, G') = \sum_{\underline{\mathbf{h}}} \sum_{\underline{\mathbf{h}}'} k_z(\underline{\mathbf{h}}, \underline{\mathbf{h}}') p(\underline{\mathbf{h}}|G) p(\underline{\mathbf{h}}'|G'). \tag{30}$$

The worst case time complexity for computing $k(G, G')$, when both G and G' are DAG, is $O(c \cdot c' \cdot |\mathcal{V}(G)| \cdot |\mathcal{V}(G')|)$ where c and c' are the maximum out degree of G and G', respectively. This computational complexity can be achieved because, in this case, it is possible to perform a *topological sort* of the graphs, employing the *one-pass dynamic programming* algorithm for DAGs. When G and G' are general direct graphs, the time complexity is given by the inversion of a matrix of order $|\mathcal{V}(G)| \cdot |\mathcal{V}(G')|$ [112, Chapter 7].

3.2.5 Generalized random walk graph kernel

In [123], a generalized method for random walk graph kernels computation is shown. For simplicity, only edge-labeled graphs are considered. With this assumption, let $\mathbf{L}$ and $\mathcal{L}$ be the edge-labels matrix and the set of edge labels, respectively. Let $\mathcal{H}_\kappa$ be an RKHS induced by the kernel $\kappa : \mathcal{L} \times \mathcal{L} \rightarrow \mathbb{R}$, and let the feature map $\phi : \mathcal{L} \rightarrow \mathcal{H}_\kappa$ that also maps the element ζ to the zero element of the induced Hilbert space $\mathcal{H}_\kappa$. An extension of tensor algebra to RKHS is defined in [123, Appendix A] to deal with the definition of features map for matrices as $\Phi : \mathcal{L}^{n \times m} \rightarrow \mathcal{H}^{n \times m}$, with $[\Phi(\mathbf{L})]_{ij} = \phi(L_{ij})$.

Let G, G' be two graphs. Let the initial distribution of the product graph be $\underline{\mathbf{p}}_\times = \underline{\mathbf{p}} \otimes \underline{\mathbf{p}}'$ and the relative stopping probability vector $\underline{\mathbf{q}}_\times = \underline{\mathbf{q}} \otimes \underline{\mathbf{q}}'$. One of the most important concepts introduced in [123] is the weight matrix of the direct product graph $\mathbf{W}_\times^{nn' \times nn'}$, that encodes edge-labels similarity, and is defined, in the most general case, as:

$$\mathbf{W}_\times = \Phi(\mathbf{L}) \otimes \Phi(\mathbf{L}'). \tag{31}$$

Each entry of $\mathbf{W}_\times$ is non-zero iff the corresponding edge exists in $G_\times$. If $\mathcal{H} = \mathbb{R}$, then $\Phi(\mathbf{L}) = \mathbf{T}$ (or $\mathbf{A}$) and consequently

$$\mathbf{W}_\times = \mathbf{T}_\times. \tag{32}$$

If $\mathcal{L} = \{1, \ldots, d\}$ is a finite set, then $\mathcal{H} = \mathbb{R}^d$ and we have that

$$\phi = (L_{ij}) = \begin{cases} \frac{\mathbf{e}_l}{\deg(\mathrm{v}_i)} & \text{if the edge } e_{ij} \in \mathcal{E} \wedge label(e_{ij}) = l, \\ 0 & \text{otherwise.} \end{cases} \tag{33}$$

where $\underline{\mathbf{e}}_l$ is a d dimensional vector with only a one in the position l. Practically, the weight matrix $\mathbf{W}_\times$ has a nonzero entry iff there is an edge in $G_\times$ and the labels of the two original graphs are the same.

In this case, we can redefine Eq. 31 as:

$$\mathbf{W}_\times = \sum_{l=1}^{d} \mathbf{T}(l) \otimes \mathbf{T}'(l), \tag{34}$$

where $\mathbf{T}(l)$ is the *filtered matrix* defined as:

$$T(l_{ij}) = \begin{cases} T_{ij} & \text{if } L_{ij} = l, \\ 0 & \text{otherwise.} \end{cases} \tag{35}$$

Generally speaking, each entry of $\mathbf{W}_\times^k$ represents the similarity between simultaneous random walks of length k on G and G', using the kernel $\kappa(\cdot,\cdot)$ on the edge labels, considering different possible scenarios about the graphs definitions. Given the starting and stopping probability vectors $\underline{\mathbf{p}}_\times$ and $\underline{\mathbf{q}}_\times$, the *generalized graph kernel* is defined as:

$$k(G, G') = \sum_{k=1}^{\infty} \mu(k) \underline{\mathbf{q}}_\times^{\mathrm{T}} \mathbf{W}_\times^k \underline{\mathbf{p}}_\times, \tag{36}$$

where $\mu(k)$ is a nonnegative function of k needed for convergence assurance and for generalize the behavior of the kernel.

The authors of [123] show different optimized methods for the computation of graph kernel shown in Eq. 36, stating that this formulation can be seen as a *bridge* between different seminal formulations, such as the ones shown in Sects. 3.2.1, 3.2.2 and 3.2.4. However, these optimizations are graph specific, that is they exploit particular characterizations of the input graphs, such as the type of labels and if they are fully vertex or edge labeled graphs [123].

3.2.6 Convolution edit kernel

This algorithm is known as *convolution edit kernel* [85] and can be seen as a hybrid method defined by a decomposition of pairs of graphs into edit paths, based on the convolution property discussed in Sect. 2.2.1. Given the (arbitrarily ordered) sequence of all vertices and edges of a graph, any non-empty subsequence can be considered as a decomposition of the graph. If two sequences, representing two different graph G and G', have the same length and at each position there are either vertices or edges in both sequences, then the two sequences can be interpreted as a partial valid edit path. It is possible to establish the validity of each derived edit path checking the validity of the substitutions (vertex–vertex, edge–edge) with a positive definite function. Given two vertices $u \in \mathcal{V}(G)$ and $u' \in \mathcal{V}(G')$, the similarity of the (label) substitution $u \rightarrow u'$ is given by the Gaussian RBF kernel shown in Eq. 26. The same function, with a different parameter $\sigma_{\mathcal{E}}$, is also used to evaluate the similarity of edge labels. If $R^{-1}(G)$ is the set of edit decompositions of G and $k_{val}(\cdot,\cdot)$ is the function evaluating whether or not two edit path decompositions are equivalent to a valid edit path, it is possible to check the structural similarity employing the convolution property. For $x \in R^{-1}(G)$ and $x' \in R^{-1}(G')$ we obtain

$$k_{val}(x, x') = \begin{cases} 1 & \text{if they are valid edit paths,} \\ 0 & \text{otherwise.} \end{cases} \qquad (37)$$

The convolution edit kernel as a whole is defined as

$$k(G, G') = \sum_{\substack{x \in R^{-1}(G) \\ x' \in R^{-1}(G')}} k_{val}(x, x') \prod_i k_{sim}\big((x)_i, (x')_i\big). \qquad (38)$$

The kernel function assigns high values to similar graphs and low values to dissimilar graphs in terms of the similarity of the involved elements in the edit substitutions. Unfortunately, the number of valid decomposition, $|R^{-1}|$, is exponential in the number of edges. Consequently, this number must be carefully controlled limiting the number of decompositions taken into account. For what concerns the applicability, this method can deal with any kind of labeled graph, provided, as usual, that suitable dissimilarity functions (i.e., each specific $d(\cdot,\cdot)$ in Eq. 26), tailored for the specific nature of $\mathcal{L}_{\mathcal{V}}$ and $\mathcal{L}_{\mathcal{E}}$, are given.

3.3 Graph embedding-based

An embedding for a graph consists in explicitly define a mapping function $\phi : \mathcal{G} \rightarrow \mathcal{D}$, such that a graph G is represented into a n-dimensional space, say $\mathcal{D} \subseteq \mathbb{R}^n$, bringing back the problem to a space rich of established recognition and learning methods. These methods are very interesting, because the embedding space can be directly modeled and explicitly analyzed. Indeed, once the embedding $\mathcal{D}$ is computed, different post-processing techniques can be applied to transform and analyze data, such as PCA-like analysis.

3.3.1 GED embedding

The approach, broadly described in [100], consists in producing a dissimilarity representation (see Sect. 2.3.2) for the input graphs $\mathcal{G}$ using a GED as a basic dissimilarity scheme. Consequently, the dissimilarity between two given input graphs is defined as the (minimum) cost needed to transform the data graph into the model graph. The dissimilarity function $d(\cdot,\cdot)$ is then employed as a key element in the following embedding procedure.

Definition 32 (*Graph embedding with GED*) Given a set of labeled graphs $\mathcal{G} = \{G_1, \ldots, G_t\}$, a GED-based dissimilarity function $d : \mathcal{G} \times \mathcal{G} \rightarrow \mathbb{R}^+$, a prototypes set $\mathcal{P} = \{P_1, \ldots, P_n\}, \mathcal{P} \subseteq \mathcal{G}$, the embedding vector is defined as

$$\phi^{\mathcal{P}}(G) = [d(G, P_1), \ldots, d(G, P_n)]^{\mathrm{T}}, \quad \forall G \in \mathcal{G}, \qquad (39)$$

where the superscript $\mathcal{P}$ remarks that the embedding is relative to the chosen set of prototypes $\mathcal{P}$.

The first importantly derived property is that any two graphs that have a relatively low GED in the input space are mapped into close points in the embedding space. The maximum distance between any pair of graphs is bounded by a positive constant $\sqrt{n}$, where $n = |\mathcal{P}|$ [100]. That is, this embedding can be seen as a Lipschitz continuous mapping since $\phi^{\mathcal{P}}(G) - \phi^{\mathcal{P}}(G') \leq \sqrt{n} \cdot d(G, G')$ holds $\forall G, G' \in \mathcal{G}$. Note that the equality is easily obtained if the components of the vectors in the embedding space are scaled by a factor equal to $\sqrt{n}$, that depends only on the number of prototypes. However, in order to assert that $\phi^{\mathcal{P}}(\cdot)$ is a Lipschitz mapping function, the pair $(\mathcal{G}, d)$ must form a metric space.

The most critical issue is the selection of $\mathcal{P}$. Assuming to deal with classification and clustering problems over $\mathcal{G}$, the authors of [100] distinguish between two types of selection: *class-wise* and *class-independent*. In the first case we need to select l_i prototypes for k classes with $\sum_{i=1}^{k} l_i = n$. In the latter case, we simply need to select n prototypes from the set $\mathcal{G}$. Different strategies are proposed to deal with this issue. For example, the *Random Prototype Selector* (RPS), that just randomly selects n prototypes. The *Spanning Prototype Selector* (SPS) approach is aimed to *cover* the set $\mathcal{G}$ with equally distanced prototypes, starting from the *set median graph* [4, 61].

Being basically a hybrid algorithm, its behavior depends also on the particular design of the GED algorithm. That is, the optimization effort should be devoted to

the determination of the optimal $\mathcal{P}$ and at the same time to the learning of the GED-related parameters (i.e., the edit costs). The applicability of this method depends, in turn, on the kind of labeled graphs that the particular GED can deal with. Usually, any kind of labeled graphs can be processed.

3.3.2 Symbolic histograms embedding

The methodology of the *symbolic histograms*, firstly proposed in [28, 29, 101], consists in producing an embedding through the identification of the *frequent subgraphs* (FS) of the input dataset $\mathcal{G}$ [70]. Given a set of input graphs $\mathcal{G} = \{G_1, \ldots, G_n\}$, the first (non-obvious) problem to solve is the determination of the set of frequent subgraphs $\mathcal{A} = \{S_1, \ldots, S_m\}$ on $\mathcal{G}$. Once obtained $\mathcal{A}$, the embedding consists in a mapping function $\phi^{\mathcal{A}} : \mathcal{G} \rightarrow \mathbb{R}^m$, that assigns an integer valued vector $\underline{\mathbf{h}}_i$ to each graph G_i, defined as follows,

$$\phi^{\mathcal{A}}(G_i) = [\text{occ}(S_1), \ldots, \text{occ}(S_m)]^{\mathrm{T}}, \quad \forall G_i \in \mathcal{G}, \tag{40}$$

where the function $\text{occ} : \mathcal{A} \rightarrow \mathbb{N}$ counts the *occurrences* of each subgraphs $S_j \in \mathcal{A}$ in each given input graph G_i. The subgraphs $S_j \in \mathcal{A}$ are called *symbols*, and the set $\mathcal{A}$ is called the *symbols alphabet*. Each vector representation $\underline{\mathbf{h}}_i$ is then called the *symbolic histogram* of the graph G_i. The occurrence of a subgraph S_j into a graph G_i is evaluated with a weighted GED-based inexact graph matching procedure $d(\cdot, \cdot)$ [28]. If the matching score reaches a symbol-dependent threshold τ_j, the occurrence is considered. Hence, the developed embedding space can be thought as a dissimilarity representation of $\mathcal{G}$, using $\mathcal{A}$ as the representation set. The values of the components of $\underline{\mathbf{h}}_i$ can be normalized or modified with some transformation function.

The first hard and crucial problem is certainly how to determine the symbols alphabet $\mathcal{A}$, assuming that $\mathcal{G}$ is a generalized set of arbitrarily labeled graphs. Note that $\mathcal{A}$, is actually automatically determined by the following procedure. The proposed algorithm is based on an iterative incremental strategy that builds the set $\mathcal{A}$ of all symbols of order $1 \leq k \leq r$, with $r \ll q$, where q is the minimum order in $\mathcal{G}$. A clustering ensemble procedure is employed to identify the recurrent substructures relying on $d(\cdot, \cdot)$ as the inexact graph matching evaluation between subgraphs. This is an important difference between various FS algorithms [13, 70, 118, 125, 130, 131], because they are usually based on some exact matching scheme (e.g., isomorphism based approaches). Practically, a set of partitions $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_l\}$, with $\mathcal{P}_u = \{\mathcal{C}_1, \ldots, \mathcal{C}_{q_u}\}$, $u = 1 \rightarrow l$, is constructed over the current set $\mathcal{A}^{(i)}, i = 1 \rightarrow r$, for a given instance of the weighting parameters of $d(\cdot, \cdot)$. The representative subgraph S_h of a cluster $\mathcal{C}_h$ is defined as a symbol if it satisfies a cluster-dependent quality measure $\tau_{\mathcal{C}_h}$. Such a threshold relies on size and *compactness* descriptors of $\mathcal{C}_h$. Note that r is actually a parameter establishing a compromise between the computational cost and the accuracy of graph embedding procedure as a whole.

Also, this embedding method depends on a direct dissimilarity algorithm from the GED family, and consequently the behavior as a whole is influenced by its definition (i.e., the edit costs). Moreover, as well as stated for the method described in Sect. 3.3.1, also the applicability is strongly induced by the peculiarities of the employed GED algorithm.

3.3.3 Structure space embedding

The theory of the structure spaces [59] has been briefly described in Sect. 2.3.3. This is an embedding-based methodology that represents the graphs into the so-called $\mathcal{T}$-space, considering their equivalence in terms of weighted adjacency matrices.

The embedding into this vector space permits to extend concepts like inner product, norm and metric, called respectively $\mathcal{T}$-inner product ($\langle \cdot, \cdot \rangle^*$), $\mathcal{T}$-norm ($\cdot_*$) and $\mathcal{T}$-metric (D_*) between vectors in the set $\mathcal{X}_{\mathcal{T}}$. The first two functions are not strictly an inner product and a norm, as they satisfy weaker properties (symmetry is not necessarily satisfied). For example, $\langle \cdot, \cdot \rangle^*$ is a *maximizer* of the value assumed by the standard inner product on $\mathcal{X}$. The operators $\| \cdot \|_*$ and D_* are *minimizers* of the norm and distance on $\mathcal{X}$, respectively. The dissimilarity measure between two given graphs G_1 and G_2 is obtained as the minimum one considering each possible vector representation of them:

$$d(G_1, G_2) = \min_{\underline{\mathbf{x}} \in G_1, \underline{\mathbf{y}} \in G_2} \|\underline{\mathbf{x}} - \underline{\mathbf{y}}\|_2, \tag{41}$$

where the notation $\underline{\mathbf{x}} \in G_1$ means a possible vector representation $\underline{\mathbf{x}}$ of the graph G_1 into $\mathcal{X}_{\mathcal{T}}$, as defined in Sect. 2.3.3. Unfortunately, the solution to Eq. 41 is an NP-hard problem, since there are a factorial number of possible representations of each involved graph.

For instance, to better understand this embedding method, consider the two sample graphs X and Y shown in Fig. 8. The $\mathcal{T}$-inner product between the vector representations of these graphs is equal to $\langle X, Y \rangle^* = \langle \underline{\mathbf{x}}, \underline{\mathbf{y}}' \rangle = \langle \underline{\mathbf{x}}', \underline{\mathbf{y}} \rangle = 16$. With regard to the norm, we have $\sqrt{\langle X, X \rangle^*} = X_* = \underline{\mathbf{x}} = \underline{\mathbf{x}}' = \sqrt{22}$. The distance $D_*(\cdot, \cdot)$ is a metric and we have $D_*(X, Y) = d(\underline{\mathbf{x}}, \underline{\mathbf{y}}') = d(\underline{\mathbf{x}}', \underline{\mathbf{y}}) = \sqrt{2}$.

Once the graphs are embedded in $\mathcal{X}_{\mathcal{T}}$, different graph-based problems have been considered in [59], such as the computation of the *sample mean* and the *central clustering* of k-structures [60]. These tasks are conceived as *non-smooth* continuous optimization problems on the metric space $(\mathcal{X}_{\mathcal{T}}, D)$, where $D(\cdot, \cdot)$ is the metric distance $D_*(\cdot, \cdot)$ previously defined, or any other more appropriate distance

function. For example, the sample mean of $X_1, X_2, \ldots, X_k \in \mathcal{X}_\mathcal{T}$ is the the element $\hat{X}$ of the set $\mathcal{X}_\mathcal{T}$ that minimizes the objective function $F(\hat{X}) = \sum_{i=1}^{k} D(\hat{X}, X_i)^2$. In [59, Prop. C1], it is shown that $D(\cdot,\cdot)$ being a locally Lipschitz function, the objective function $F(\cdot)$ is also locally Lipschitz. For instance, sub-gradient methods can be employed to solve the optimization task. The determination of the sample mean of a set of graphs is a well-known problem in the context of graph-based pattern recognition, where it is referred as the *set median graph* computation [4, 61].

However, from the applicability viewpoint, this method can be used for a restricted type of labeled graphs. Indeed, only graphs labeled with real-valued feature vectors (or scalars) can be adopted to construct the structure space $\mathcal{X}_\mathcal{T}$.

3.4 Seriation based

The seriation of a graph consists in finding an ordering of the vertices, to obtain a sequence-like representation. This family of currently available algorithms is suited for graphs where edge labels are real numbers and the vertex labels do not affect the seriation. Considering our taxonomy given in Sect. 2, we should see these as both embedding and GED-based methods, where the embedding space is (mostly) defined by the *spectrum* of the graph. Given two seriated graphs, the matching is then performed using known methods such as string edit distance [72] or string kernels [71, 116, 124, 132]. In the first case, various edit cost systems can be used to learn the cost of each edit operation. Online methods derive these costs during the algorithm execution, adapting the underlying edit scheme. The *template* of a typical seriation-based algorithm follows a three-stage process, where firstly the graphs are transformed into sequences of vertices, then the edit operation costs are learned, and eventually a matching strategy is applied to the sequences using the learned edit costs.

3.4.1 Shortest edit path

The method explained in [104] seriates a graph G using the leading eigenvector of its matrix representation (e.g., ajacency matrix or transition matrix). The leading eigenvector of its adjacency matrix contains the information about the structural connectivity of each vertex of the graph. Similarly, analyzing the (symmetric) transition matrix $\mathbf{T}^{n\times n}$, it is possible to obtain information about the a priori probability of a given vertex in a random walk scenario on G. Following this fact, the vertices are seriated performing a random walk-like traverse of the graph, using the leading eigenvector of $\mathbf{T}$. The edit costs are learned in an online fashion estimating an elaborated combination of the transition probabilities between involved vertices. The minimum edit cost to transform the data graph into the model graph is calculated using the Dijkstra algorithm [32] over the *edit lattice* built considering the edit matrix. Calculating the minimum edit cost using the Dijkstra's shortest path algorithm on the edit lattice or using the Levenshtein distance [72] directly on the seriated graph representations is equivalent in practice.

Firstly, a symmetric version of $\mathbf{T}$ is obtained as $\mathbf{T} = \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$. Denoting with ϕ the leading eigenvector of $\mathbf{T}$, the sequence of vertices is computed with the procedure outlined in Algorithm 2.

Algorithm 2 Graph Seriation

Input: A graph $G = (\mathcal{V}, \mathcal{E})$
Output: A sequence of its vertices $L = (v_{j_1}, \ldots, v_{j_n})$
1: $j_1 = \arg\max_j \phi(j)$
2: $add(L_1, j_1)$
3: $k = 2$
4: **repeat**
5: $\quad j_k = \arg\max_{j \in \mathcal{N}_{j_{k-1}} \wedge j \notin L} \phi(j)$
6: $\quad add(L_k, j_k)$
7: **until** All vertices $\mathcal{V}(G)$ are in the list L
8: **return** L

In Algorithm 2, $\mathcal{N}_{j_{k-1}} = \{m | (j_{k-1}, m) \in \mathcal{E} \wedge m \notin L_k\}$ stands for the neighbors of the last selected vertex, indexed by j_{k-1} that are not already in the list L (line 6). The algorithm terminates when the size of the list L_k is equal to $|\mathcal{V}|$. At the end of the procedure, we have two sequences of vertex labels, $X = (x_1, \ldots, x_n)$ for the data graph and $Y = (y_1, \ldots, y_m)$ for the model graph.

Given a generic edit path of h edit operations $\Gamma = (\gamma_1, \ldots, \gamma_h)$, the edit cost of this path is obtained as

$$c(\Gamma) = \sum_k \eta(\gamma_k \rightarrow \gamma_{k+1}). \quad (42)$$

The edit costs of each $\eta(\gamma_k \rightarrow \gamma_{k+1})$ are calculated evaluating the probability of a state transition from γ_k to γ_{k+1}, considering that each $\gamma_k = (\mathcal{V}(G_1) \cup \epsilon, \mathcal{V}(G_2) \cup \epsilon)$ is a valid edit operation. If a state transition is highly probable, the relative edit cost should be low, i.e., $\eta(\gamma_k \rightarrow \gamma_{k+1}) = -\log(P(\gamma_k \rightarrow \gamma_{k+1}))$. The probability $P(\gamma_k \rightarrow \gamma_{k+1})$ is defined as

$$P(\gamma_k \rightarrow \gamma_{k+1}) = \beta_{a,b}\beta_{c,d} R_{G_1}(a,c) R_{G_2}(b,d), \quad (43)$$

where for simplicity, (a, c) and (b, d) are basic edit operations of the form $(\mathcal{V}(G_1) \cup \epsilon, \mathcal{V}(G_2) \cup \epsilon)$. In Eq. 43, $\beta_{a,b}$ and $\beta_{c,d}$ are referred to as *morphological affinity* parameters of the vertices and

$$R_{G1}(a,c) = \begin{cases} P_{G1}(a,c) & \text{if } (a,c) \in \mathcal{E}(G_1), \\ \frac{2 \cdot abs(|v(G_1)| - |v(G_2)|)}{|v(G_1)| + |v(G_2)|} & \text{if } a = \epsilon \vee c = \epsilon \\ 0 & \text{otherwise.} \end{cases} \quad (44)$$

The same holds for $R_{G_2}(b,d)$. So, the problem of computing GED is posed as finding the shortest path

through the lattice by Dijkstra's algorithm and the GED $c(\Gamma^*)$ between these two graphs is given by $\Gamma^* = \arg\min_{\Gamma} c(\Gamma)$.

3.4.2 Maximum a posteriori

The Maximum a Posteriori (MaP) approach [103] is very similar to the one shown in Sect. 3.4.1, but the graphs are seriated using the leading eigenvector of the adjacency matrix $\mathbf{A}$ and the least expensive edit path Γ^* is obtained using the Levenshtein algorithm [72] on the edit matrix. In this method, edge density of the two graphs is used to estimate the edit costs. The algorithm is very similar to the one shown in Algorithm 2. The edit costs of a state transition $\eta(\gamma_k \rightarrow \gamma_{k+1})$ is obtained as

$$\begin{aligned}\eta(\gamma_k \rightarrow \gamma_{k+1}) = & -\log(P(\gamma_k|\phi_{G_1}(x_i), \phi_{G_2}(y_j)) \\ & -\log(P(\gamma_{k+1}|\phi_{G_1}(x_{i+1}), \phi_{G_2}(y_{j+1})) \\ & -\log R_{k,k+1},\end{aligned} \tag{45}$$

where $R_{k,k+1} = P(\gamma_k, \gamma_{k+1})/(P(\gamma_k)P(\gamma_{k+1}))$ is referred to as the *edge compatibility* value and is obtained as

$$\begin{cases} \rho G_2 \rho G_1 & \text{if } \gamma_k \rightarrow \gamma_{k+1} \text{ is a diagonal transition,} \\ & (x_i, x_{i+1}) \in \mathcal{E}(G_1) \wedge (y_i, y_{i+1}) \in \mathcal{E}(G_2) \\ \rho G_2 & \textit{if } (\gamma_k \rightarrow \gamma_{k+1}) \text{ is a vertical transition,} \\ & (x_i, x_{i+1}) \in \mathcal{E}(G_1) \wedge (y_j = \vee y_{j+1} =) \\ \rho G_1 & \text{if } (\gamma_k \rightarrow \gamma_{k+1}) \text{ is a horizontal transition,} \\ & (y_j, y_{j+1}) \in \mathcal{E}(G_2) \wedge (x_i = \vee x_{i+1} =) \\ 1 & \text{if } (x_i = \vee x_{i+1} =) \wedge (y_j = \vee y_{j+1} =) \end{cases} \tag{46}$$

where ρ_{G_1} and ρ_{G_2} are the *edge densities*.

In Eq. 45, the conditional probabilities $P(\gamma_k|\phi_{G_1}(x_i), \phi_{G_2}(y_j))$ are calculated as

$$\begin{cases} \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{1}{2\sigma^2}\left(\phi G_1(x_i) - \phi G_2(y_j)\right)^2\right) & \text{if } x_i \neq \epsilon \wedge y_j \neq \epsilon, \\ \alpha & \text{if } x_i = \epsilon \vee y_j = \epsilon. \end{cases} \tag{47}$$

where α and σ are two parameters which need to be set a priori.

3.4.3 String kernels

Given two seriated graphs $X = (x_1, \ldots, x_n)$ and $Y = (y_1, \ldots, y_m)$, a kernel function for strings could be employed to determine the similarity between these string representations of the two graphs. For the sake of conciseness, we can cite only a method based on *semidefinite programming* [132] and a fast kernel techniques for strings that employ *suffix trees* [116, 124].

3.5 Algorithms analysis

In this survey, three main families of algorithms have been considered: graph edit distance based, graph kernels based, and graph embedding based. In addition, we have considered also hybridized formulations based on graph seriation techniques. In what follows, we summarize some of the peculiarities associated with the algorithms we discussed in this paper, together with a focused analysis on algorithms comparison and parameters related issues.

Graph edit distance approaches Modern GED-based algorithms search for a suboptimal edit path, usually solving another task, such as the assignment (see Sect. 3.1.4) and quadratic optimization (see Sect. 3.1.3) problems. That is, they do not solve exactly the optimization problem defined in Eq. 2. However, the loss, in terms of recognition rate, is more than acceptable considering the huge average achieved speed-up [90, 98]. The dissimilarity is conceived as the amount of edit operations costs needed to transform the data graph into the model graph. As a result, these inexact graph matching functions are usually not symmetric. Furthermore, usually algorithms of this family focus on the vertices-related edit operations, deriving automatically the ones on the edges. An interesting characteristic of this family of algorithms is that edit operations are intuitive and easily comprehensible, that is, it is possible to visually understand the operations needed to edit the data graph into the model graph. This feature can be very helpful when domain-dependent knowledge is required to understand the recognition problem at hand.

GED-based algorithms are usually polynomial. Indeed, the bipartite graph matching algorithm [98], described in this survey in Sect. 3.1.4, has a computational complexity of $O(n^3)$. The computational complexity associated with the quadratic optimization problem depends on the particular adopted algorithmic scheme [87], but it is always bounded on the polynomial class. GED-based algorithms are able to deal with virtually any kind of labels. Indeed, usually the edit operation that corresponds to the substitution is carried out just providing a suitable basic dissimilarity function for the specific nature of the labels set.

Graph kernels approaches Graph kernels algorithms are, in one way or another, founded on the convolution property of valid kernel functions. These methods are able to unify structural and semantic related analysis of the input patterns in a single, and usually mathematically well-founded, operation, such as the tensor product operator (see Sect. 3.2.1). Standard computation of the tensor product has a computational cost of $O(n^4)$, i.e., it is an expensive operation. Moreover, once the tensor product is computed, further operations are required to extract the similarity value from the output adjacency matrix $\mathbf{A}_\times$. For example, computing its eigendecomposition requires an additional

cost of $O(n^3)$ (in the general case). In order to face this issue, in [123] (see also Sect. 3.2.5) a generalized approach to the computation of the random walk graph kernel is depicted, together with different optimized strategies for faster computations, although dependent on the type of labeled graphs. Tensor product-based graph kernels suffer a (not so penalizing) limitation. Being based on the adjacencies (or, in a similar way, on the transition probabilities) of the two input graphs, these methods cannot be applied to graphs defined with only one vertex, or graphs without edges. Another drawback is that, usually, the resulting matrix $\mathbf{A}_\times$ (or $\mathbf{T}_\times$) becomes very big extremely fast, limiting the application from the memory usage viewpoint.

The use of graph kernels can be thought as an implicit embedding method, as discussed in Sect. 2.2. Unfortunately, usually the specific embedding space $\mathcal{H}_k$, induced by the valid graph kernel function $k(\cdot,\cdot)$, is unknown and further post-processing analysis cannot be conducted on it. Methods of this family are usually applicable to any kind of labeled graphs. For example, considering the Gaussian RBF (see Eq. 26), this kernel function can be applied to any type of label, using a suitable dissimilarity function $d(\cdot,\cdot)$, tailored to the specific labels set.

Graph embedding approaches Graphs embedding methods are very interesting, mainly because of the controllable embedding space. Indeed, it is possible to perform further post-processing analysis on the explicit embedding, and, in the same time, inspect with PCA-like techniques the peculiarities of the data. Moreover, they can be thought as a two-stage algorithms, where firstly the set of prototypes, used for embedding purpose, is determined. This stage can be defined as the synthesis of the embedding space $\mathcal{D}$. Subsequently, the matching of graphs is performed explicitly on $\mathcal{D}$. Therefore, the set of embedding prototypes (both $\mathcal{P}$ and $\mathcal{A}$ defined in Sects. 3.3.1 and 3.3.2, respectively), can be considered pre-computed in some pattern recognition applications. These methods are, usually, hybridized, since their overall behavior depends on a *core* inexact graph matching algorithm, such as the GED-based one. This fact introduces more parameter dependencies and computational complexity in the method as a whole. It is worth stressing that also a graph kernel algorithm can be thought as a dissimilarity measure. Indeed, it is always possible to define a dissimilarity value starting from a similarity measure, even when this similarity is unbounded [92]. Consequently, in both algorithms described in Sects. 3.3.1 and 3.3.2, also a suitable graph kernel, used as a dissimilarity, can be adopted. Note that when a graph kernel is used as a dissimilarity, *any* kernel function can be employed in the various convolution schemes, not limiting the choice to valid kernel functions.

The symbolic histograms embedding, described in Sect. 3.3.2, is founded on a clustering-based approach, searching for a set $\mathcal{A}$ of FS (called the alphabet of symbols), using an inexact graph matching algorithm as dissimilarity measure, introducing a tolerance parameter in the FS identification scheme. For what concerns the embedding synthesis stage, assuming employing a simple sequential clustering scheme, such as the one provided by the Basic Sequential Algorithmic Scheme (BSAS) [117], this method has a complexity of $O(|\mathcal{A}| \cdot GC \cdot (Q + |\mathcal{C}|^2))$, where GC is the cost associated with each graph matching computation, Q is the maximum number of allowed clusters and $\mathcal{C}$ is the set of clusters representatives, which is kept fixed in [28, 29, 101] using a constant-size *cache replacement policy* [27]. Hence, it consists of a linear, in the cardinality of the set of symbols $\mathcal{A}$, number of inexact graph matching computations. It is easy to understand that this approach, considering the matching stage, has a cost that is greater than, for example, a direct GED-based method. Indeed, once $\mathcal{A}$ is determined, the embedding procedure of an input graph G_i is performed issuing $|\mathcal{A}| \cdot |expand(G_i)|$ inexact graph matching computations, where the function $expand(\cdot)$ expands a graph into a (not complete) set of its subgraphs. Given $\mathcal{A}$, for what concerns the time complexity of the matching stage, the number of inexact graph matching evaluations for computing, say, $d(G_1, G_2)$, is given by $|\mathcal{A}| \cdot (|expand(G_1)| + |expand(G_2)|)$, which is again linear in the number of derived symbols.

Obviously, also the embedding method described in Sect. 3.3.1 is dependent on the cost associated with the direct graph matching algorithm (i.e., to the particular GED-based algorithm). The number of inexact graph matching evaluations depends strictly on the size of $\mathcal{P}$. For what concerns the matching stage, the matching of $d(G_1, G_2)$ is carried out executing exactly $2|\mathcal{P}|$ inexact graph matching computations needed to produce the embedding vectors. Prototypes selection strategies permits also to limit the number of computations to $|\mathcal{P}| \ll |\mathcal{G}|$. However, the determination of this set of prototypes $\mathcal{P}$ has an important cost, subjected to the specific selection strategy adopted [100].

The algorithm based on the embedding into the structure space, described in Sect. 3.3.3, is not a two-stages algorithm and does not require any additional core graph matching procedure, being based on a particular interpretation of the (weighted) adjacency matrix of each graph.

Graph seriation approaches The algorithms based on seriation, discussed in this survey in Sect. 3.4, are actually hybrid methods that rely on both edit distance performed on sequences and embedding approaches, and also for this reason they have been discussed separately. The reviewed algorithms use the information provided by the edges to produce a sequence of the vertices identifiers, according to their *importance* in the graph. As used in the literature, these algorithms are applicable only to edge-labeled graphs, with

$\mathcal{L}_{\mathcal{E}} \subseteq \mathbb{R}^n$. However, this characterization of the edges labels is not mandatory, because the transition and adjacency information on a graph can be always extracted, regardless the specific definition of $\mathcal{L}_{\mathcal{E}}$. However, we stress that the information of the edges labels should be employed in the seriation stage, especially considering that this information is lost once the graph has been represented by a sequence of its vertices labels. Indeed, string edit distance algorithms are then employed to the seriated graphs, computing the similarity of the sequence of vertices identifiers. With regard to the computational complexity of the presented algorithms (see Sect. 3.4), the seriation stage is dominated by the eigendecomposition cost, that is $O(n^3)$. If the alignment of the sequences of vertices identifiers is carried out with the Levenshtein algorithm, we need to add a cost of $O(n^2)$. Conversely, if the Dijkstra algorithm is used to find the shortest edit path on the edit lattice, the additional cost becomes $O(|\mathcal{E}| + |\mathcal{V}| \cdot \log(|\mathcal{V}|))$, where $\mathcal{E}$ and $\mathcal{V}$ are the edges and vertices sets of the edit lattice built for two input graphs.

To be able to apply the same inexact graph matching scheme based on seriation to fully labeled graphs, it could be interesting to employ a general sequence matching procedure, such as the one provided by the Dynamic Time Warping (DTW) [106] algorithm. In fact, the DTW can be tailored to the specific nature of the sequence, virtually opening the possibility to match any kind of complex labels for the vertices. The DTW algorithm has the same scheme of the Levenshtein distance, but its time complexity is determined considering also the specific definition of $\mathcal{L}_{\mathcal{V}}$.

3.5.1 Considerations on algorithms comparison

At least for the pattern recognition and soft computing viewpoints, it is impossible to state a priori what matching method is better than others. This kind of information can be inferred only through a systematic experimentation over a benchmarking dataset that is far beyond the scope of this paper. For this purpose, the scientific community should agree on more shared datasets as the briefly described IAM (see Sect. 1.2), where different research groups are already publishing their results [38, 46, 58, 75, 98]. In addition, although the computational complexity of the algorithms gives an important information concerning the limit costs, also the *real* computing time performances should be verified using shared ad hoc benchmarks, since each algorithm is characterized by many factors that influence the time performance outcome in a benchmarking analysis on finite-size data. For this purpose, when using fully labeled graphs, there are constant costs associated with each single labels matching computation, that can become a very important bottleneck if the type is non-trivial, such as text excerpts, digital images or any complex composite objects.

3.5.2 Graph matching parameters tuning

Each described algorithm for the inexact graph matching computation depends on some parameters that are in some sense critical and deeply specific to the problem instance. For example, in the GED-based methods, at least the edit costs of each edit operation play a crucial role, and should be inferred from the specific context of application. Considering the graph kernel, many of the given formulations are founded on the convolution property of different valid kernel functions, defined over smaller parts of the whole object (i.e., the graph). The number of parts taken into consideration is certainly an important parameter of this approach, as well as the parameter σ in the widely adopted RBF kernel. Note that different valid kernel functions, such as the polynomial one, depend also on parameters (see Table 1). As concerns the embedding of graphs considering a local reference framework, such as the one described in Sect. 3.3.1, a critical task is the selection of the prototypes set $\mathcal{P}$, extremely relevant and specific to the domain of application. Moreover, its cardinality is relevant to the whole computational complexity, as indeed it determines the number of inexact graph matching computations needed to perform the computation of the dissimilarity between two given graphs. Not exempt from this dependence is also the symbolic histogram method, described in Sect. 3.3.2. Indeed, both the maximum subgraphs order and the specific clustering procedure parameters (such as the symbols alphabet set, and thus the maximum number of allowed cluster Q in the BSAS case) influence the recognition procedure as a whole.

This means that any inexact graph matching procedure, regardless of the family it belongs, can be seen as a parametric (dis)similarity measure. Therefore, any inductive modeling system, adopting such a graph matching procedure as the basic dissimilarity scheme, should include a suited *meta-heuristic* optimization procedure in order to automatically determine these parameters on the basis of the dataset at hand. Moreover, it is important to underline that in general, besides the parameters set characterizing a given graph matching procedure, it is needed also to consider the possible parameters sets of the dissimilarity measures adopted in the vertices and edges spaces. Each parameter in this overall set should be optimized tailoring their values on the application at hand, enabling adaptation to the specific problem semantic.

Simple experiment on parameters tuning To show how important is the learning of the inexact graph matching algorithms parameters, we report a simple example in which we describe a dataset of labeled graphs, together with a relatively simple recognition system based on the k-NN rule. We will see that the generalization capability, with and without a parameters learning stage, drastically

changes. Although the learning stage is fundamental in these systems based on inexact graph matching, we will see that there is a computational price to pay. We stress again that the same claim holds for any inexact graph matching algorithm, because this parameter dependence, in one way or other, enables the wide-range applicability to different contexts.

We have considered the Letter dataset, with the highest level of distortions, taken from the IAM graphs database [97]. The dataset is composed of a triplet of training, validation and test sets, each of 750 patterns, equally distributed into 15 different classes. The recognition system is a k-NN based on the inexact graph matching algorithm known as the *Graph Coverage* [75]. It is a graph kernel function based on the well-known format provided by the tensor product computation, largely described in both the methodological and algorithmic-related sections (see Sects. 2.2, 3.2.1, 3.2.2, 3.2.3 and 3.2.5). The graph coverage algorithm is dependent on three parameters, i.e., $\sigma_{\mathcal{V}_1}, \sigma_{\mathcal{V}_2}, \sigma_{\mathcal{E}}$, of the respective RBF kernels (see Eq. 26). Hence, in our simple experiments, we will show how the determination of these three parameters influence considerably, and reasonably, the recognition rate performance on the test set. For simplicity, we have tested only the case for $k = 1$ in the k-NN classifier. Using the default parameters, that is setting $\sigma_{\mathcal{V}_1} = 1, \sigma_{\mathcal{V}_2} = 1$ and $\sigma_{\mathcal{E}} = 1$, we obtain a recognition rate of 69.86 %, computed in 12 s. Repeating the experiment with a parameters learning stage, based on a genetic algorithm optimization scheme with (only) ten evolutions and considering as the fitness function the classification accuracy on the validation set, we achieve a recognition rate on the test set of 74.66 %, but with a computing time of 40 min. The learned parameters setup is $\sigma_{\mathcal{V}_1} = 0.7908, \sigma_{\mathcal{V}_2} = 0.4548$ and $\sigma_{\mathcal{E}} = 1.8684$.

To stress that the same fact holds for any inexact graph matching scheme, we repeated the same experiment substituting only the core matching procedure with a GED-based algorithm, called weighted *Best-Matching Vertex First* (weighted BMF) [29]. It is a very fast algorithm (its computational complexity is $O(n^2)$, where n is the order of the data graph) founded on the well-known approach provided by the graph edit distance (see Sect. 2.1), performing a greedy assignment of pairs of vertices (i.e., the ones with lowest labels-dissimilarity value are assigned in each iteration, without the possibility of modifying this decision). As usual in this scenario, the edit operations of the edges are induced by the ones performed on the vertices. The weighted BMF algorithm parameters set is defined by three weights, the substitution, insertion and deletion, for both vertices and edges, for a total of six parameters, each assuming value in [0, 1]. These weighting parameters define the importance of each kind of computed edit operation cost in the whole edit path. For simplicity, we will denote these parameters as $s_{\mathcal{V}}, i_{\mathcal{V}}, d_{\mathcal{V}}$ (for the vertices) and $s_{\mathcal{E}}, i_{\mathcal{E}}, d_{\mathcal{E}}$ (for the edges). The achieved recognition rate on the test set, using the default setting $s_{\mathcal{V}} = s_{\mathcal{E}} = i_{\mathcal{V}} = i_{\mathcal{E}} = d_{\mathcal{V}} = d_{\mathcal{E}} = 1$, is 72.4 %, with a computing time of about 2 s. Performing a stage of parameters optimization, using the same genetic algorithm-based scheme with again ten evolutions, we improve the recognition rate at 89.60%. The learned parameters setup is $s_{\mathcal{V}} = 0.4678, i_{\mathcal{V}} = 0.7175, d_{\mathcal{V}} = 0.9309, s_{\mathcal{E}} = 0.1306, i_{\mathcal{E}} = 0.2189, d_{\mathcal{E}} = 0.7380$. The computing time becomes about 8 min.

As it is easy to understand from this very simple experiment that the learning stage of inexact graph matching algorithms parameters is crucial from the pattern recognition viewpoint. However, a computational price must be paid, because the learning of the parameters must be performed using some optimization scheme, such as the one previously described. As a consequence, the matching method employed in the recognition system becomes adaptable, at least in theory, to many contexts.

4 Conclusions

Given a problem at hand, defining a suitable and flexible graph matching procedure is a challenging problem, mainly for two reasons: its intrinsic complexity and the heterogeneity of the graphs definitions. Graphs can be arbitrarily labeled, i.e., these measures, to be fully meaningful, must take into account both topological and labels related information. In the fields of pattern recognition and machine learning, everything is a matter of a compromise between the computational cost and the quality of the results of a given method. For this purpose, different algorithms are conceived to focus on a particular aspect of the data. We have classified the algorithms into three main categories, namely graph edit distance based, graph kernels based and graph embedding based. The first family of algorithms relies on searching for a (suboptimal) edit path between the data and the model graphs. As a whole, this approach is highly adaptable to many labeled graphs types. Graph kernels approaches are founded on the kernel trick (see Sect. 2.2): to be able to use these kernel functions on various kernel machines. Usually, these algorithms are conceived using mathematically well-grounded operators, such as the tensor product of graphs. Modern approaches of this family are applicable to virtually any type of graph. Finally, algorithms from the embedding-based family are usually hybrid methods, based on some core inexact graph matching algorithm, such as the ones from the other two families. Consequently, they result in being more complex,

yet providing, in general, the possibility of *lookong into* the data, analyzing the produced embedding.

As discussed in Sect. 3.5, each described algorithm depends on a set of parameters that are at the same time the key to the adaptability to a specific dataset and a critical problem from the learning viewpoint. Indeed, as briefly shown in Sect. 3.5.2, the adaptation of these parameters for the specific dataset results in a straightforward performance improvement. The claim extends clearly to any parametrized inexact graph matching algorithm.

4.1 Future directions

Many approaches for the graph matching problem have been formulated in the scientific literature. Although there are yet a lot of possible developments in this context, we think that a very important issue is the computational speed-up of these methods, regardless of the specific algorithmic scheme (i.e., GED, graph kernels or graph embedding). As a basic building block of more complex pattern recognition and soft computing systems, these methods should be very efficient, to be able to effectively preserve the algorithm adaptability to a specific dataset via parameters optimization (see Sect. 3.5.2) in a reasonable computing time. Moreover, the efficiency is required if we deal with very large datasets or, even more, with very big labeled graphs. Note that big graphs are encountered often in many fields of high interest, such as social networks [126], biochemical compounds and different kinds of interaction networks [47, 134], brain networks [51], smart grids [33] and so on. Consequently, the extendability of these techniques to larger graphs is subjected critically to the capability of conceiving faster matching algorithms.

One interesting possibility comes from the formulation of *parallel algorithms* [76], conceived for some parallel abstract model, such as the PRAM [39, 49] or the bulk synchronous parallel (BSP) [122] model. The speed-up, in this case, can be achieved also theoretically. Moreover, there is the possibility of developing (or employ) specialized *devices* such as graphic processing units (GPUs) [76] and field programmable gate array (FPGA) [22], accelerating the algorithms using modern parallel hardware capability.

Acknowledgments The authors would like to thank the anonymous reviewers for their appreciated effort for making this work better.

References

1. Aggarwal C, Wang H (2010) Managing and mining graph data. Advances in Database Systems. Springer. http://books.google.com/books?id=Ox39uLyYh-wC
2. Aizerman A, Braverman EM, Rozoner LI (1964) Theoretical foundations of the potential function method in pattern recognition learning. Automat Remote Control 25:821–837
3. Ambauen R, Fischer S, Bunke H (2003) Graph edit distance with node splitting and merging, and its application to diatom identification. In: Proceedings of the 4th IAPR international conference on Graph based representations in pattern recognition, GbRPR'03. Springer-Verlag, Berlin, Heidelberg, pp 95–106. http://portal.acm.org/citation.cfm?id=1757868.1757880
4. Bardaji I, Ferrer M, Sanfeliu A (2010) A comparison between two representatives of a set of graphs: median vs barycenter graph. In: Proceedings of the 2010 joint IAPR international conference on structural, syntactic, and statistical pattern recognition, SSPR& SPR'10. Springer-Verlag, Berlin, Heidelberg, pp 149–158. http://portal.acm.org/citation.cfm?id=1887003.1887022
5. Bargiela A, Pedrycz W (2003) Granular computing: an introduction. No. v. 2002 in Kluwer international series in engineering and computer science. Kluwer Academic Publishers. http://books.google.com/books?id=F_3t7XTMhBkC
6. Berg C, Christensen J, Ressel P (1984) Harmonic analysis on semigroups: theory of positive definite and related functions. Graduate texts in mathematics. Springer-Verlag http://books.google.com/books?id=zz2DQgAACAAJ
7. Bernard M, Boyer L, Habrard A, Sebban M (2008)Learning probabilistic models of tree edit distance. Pattern Recognit. 41:2611–2629. doi:10.1016/j.patcog.2008.01.011. http://portal.acm.org/citation.cfm?id=1367147.1367314
8. Bernstein D (2009) Matrix mathematics: theory, facts, and formulas. Princeton University Press. http://books.google.com/books?id=jgEiuHlTCYcC
9. Bille P (2005) A survey on tree edit distance and related problems. Theor Comput Sci 337:217–239. doi:10.1016/j.tcs.2004.12.030. http://dx.doi.org/10.1016/j.tcs.2004.12.030
10. Boccaletti S, Latora V, Moreno Y, Chavez M, Hwang D (2006) Complex networks: structure and dynamics. Phys. Rep. 424(4–5):175–308. doi:10.1016/j.physrep.2005.10.009. http://dx.doi.org/10.1016/j.physrep.2005.10.009
11. Bollobás B (1998) Modern graph theory. Graduate texts in mathematics. Springer. http://books.google.ca/books?id=SbZKSZ-1qrwC
12. Borg I, Groenen P (2005) Modern multidimensional scaling: theory and applications. Springer series in statistics. Springer. http://books.google.com/books?id=duTODldZzRcC
13. Borgelt C (2002) Mining molecular fragments: finding relevant substructures of molecules. In: Proceedings of 2002 IEEE international conference on data mining (ICDM). IEEE Press, pp 51–58
14. Borgwardt KM, Ong CS, Schönauer S, Vishwanathan SVN, Smola AJ, Kriegel HP (2005) Protein function prediction via graph kernels. Bioinformatics 21:47–56. http://dx.doi.org/10.1093/bioinformatics/bti1007
15. Boser BE, Guyon IM, Vapnik VN (1992) A training algorithm for optimal margin classifiers. In: Proceedings of the fifth annual workshop on computational learning theory, COLT '92. ACM, New York, NY, USA, pp 144–152. doi:10.1145/130385.130401. http://doi.acm.org/10.1145/130385.130401
16. Boyd S, Vandenberghe L (2004) Convex optimization. Cambridge University Press. http://books.google.com/books?id=mYm0bLd3fcoC
17. Bunke H, Allermann G (1983) Inexact graph matching for structural pattern recognition. Pattern Recognit Lett 1(4):245–253 (1983). doi:10.1016/0167-8655(83)90033-8. http://www.sciencedirect.com/science/article/B6V15-48MPV00-1K/2/6f816d072c71e50b1a80858a8b488463
18. Bunke H, Bühler U (1993) Applications of approximate string matching to 2D shape recognition. Pattern Recognit 26(12):1797–1812. doi:10.1016/0031-3203(93)90177-X. http://www.sciencedirect.com/science/article/B6V14-48MPPK4-1V6/2/c7f7a4bd6aae48534f11137815852e32

19. Bunke H, Shearer K (1998) A graph distance metric based on the maximal common subgraph. Pattern Recognit. Lett. 19:255–259. doi:10.1016/S0167-8655(97)00179-7.
20. Burges CJC (1998) A tutorial on support vector machines for pattern recognition. Data Min Knowl Disc 2:121–167
21. Buriol LS, Castillo C, Donato D, Leonardi S, Millozzi S (2006) Temporal analysis of the wikigraph. In: Web intelligence conference. IEEE CS Press, pp 45–51
22. Cinti A, Rizzi A (2011) Neurofuzzy min–max networks implementation on FPGA. In: International joint conference on computational intalligence (IJCCI), neural computation theories and analysis (NCTA)
23. Conte D, Foggia P, Sansone C, Vento M (2004) Thirty years of graph matching In pattern recognition. Int J Pattern Recognit Artif Intell 18:265–298. doi:10.1142/S0218001404003228
24. Cook D, Holder L (2007) Mining graph data. Wiley-Interscience. http://books.google.com/books?id=jp8ZIpMVB54C
25. Cortes C, Vapnik V (1995) Support-vector networks. Mach Learn 20:273–297. http://dx.doi.org/10.1023/A:1022627411411. 10.1023/A:1022627411411
26. Cox T, Cox M (2001) Multidimensional scaling. No. v. 1 in Monographs on statistics and applied probability. Chapman & Hall/CRC. http://books.google.com/books?id=SKZzmEZqvqkC
27. Del Vescovo G, Livi L, Rizzi A, Frattale Mascioli FM (2011) Clustering structured data with the SPARE library. In: Proceeding of 2011 4th IEEE international conference on computer science and information technology, vol 9, pp 413–417
28. Del Vescovo G, Rizzi A (2007) Automatic classification of graphs by symbolic histograms. In: Proceedings of the 2007 IEEE international conference on granular computing, GRC '07. IEEE Computer Society, pp 410–416. doi:10.1109/GRC.2007.46. http://dx.doi.org/10.1109/GRC.2007.46
29. Del Vescovo G, Rizzi A (2007) Online handwriting recognition by the symbolic histograms approach. In: Proceedings of the 2007 IEEE international conference on granular computing, GRC '07. IEEE Computer Society, Washington, DC, USA, p 686. doi:10.1109/GRC.2007.116. http://dx.doi.org/10.1109/GRC.2007.116
30. Dempster AP, Laird NM, Rubin DB (1977) Maximum likelihood from incomplete data via the EM algorithm. J R Stat Soc Ser B 39(1):1–38
31. Diestel R (2006) Graph theory. Graduate texts in mathematics. Springer. http://books.google.com/books?id=aR2TMYQr2CMC
32. Dijkstra EW (1959) A note on two problems in connexion with graphs. Numerische Mathematik 1:269–271 http://dx.doi.org/10.1007/BF01386390.10.1007/BF01386390
33. Dorfler F, Bullo F (2011) Kron reduction of graphs with applications to electrical networks. ArXiv e-prints
34. ElGhawalby H, Hancock ER (2008) Graph characteristic from the Gauss–Bonnet Theorem. In: Lobo NdV, Kasparis T, Roli F, Kwok JTY, Georgiopoulos M, Anagnostopoulos GC, Loog M (eds) SSPR/SPR, lecture notes in computer science, vol 5342. Springer, pp 207–216
35. Emms, D., Wilson, R.C., Hancock, E. (2007) Graph embedding using quantum commute times. In: Proceedings of the 6th IAPR-TC-15 international conference on graph-based representations in pattern recognition, GbRPR'07. Springer-Verlag, Berlin, Heidelberg, pp 371–382. http://portal.acm.org/citation.cfm?id=1769371.1769412
36. Eshera MA, Fu KS (1984) A graph distance measure for image analysis. IEEE Trans Syst Man Cybern 14(3):398–408
37. Faloutsos C, Lin KI (1995) FastMap: a fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. SIGMOD Rec 24:163–174. doi:10.1145/568271.223812. http://doi.acm.org/10.1145/568271.223812
38. Fankhauser S, Riesen K, Bunke H (2011) Speeding up graph edit distance computation through fast bipartite matching. In: Jiang X, Ferrer M, Torsello A (eds) Graph-based representations in pattern recognition. Lecture notes in computer science, vol 6658. Springer, Berlin, pp 102–111. http://dx.doi.org/10.1007/978-3-642-20844-7_11.10.1007/978-3-642-20844-7_11
39. Fortune S, Wyllie J (1978) Parallelism in random access machines. In: Proceedings of the tenth annual ACM symposium on Theory of computing, STOC '78. ACM, New York, NY, USA, pp 114–118. doi:10.1145/800133.804339. http://doi.acm.org/10.1145/800133.804339
40. Gao X, Xiao B, Tao D, Li X (2008) Image categorization: graph edit direction histogram. Pattern Recognit 41(10):3179–3191. doi:10.1016/j.patcog.2008.03.025 http://www.sciencedirect.com/science/article/pii/S0031320308001246
41. Gao X, Xiao B, Tao D, Li X (2010) A survey of graph edit distance. Pattern Anal Appl 13(1):113–129
42. Garey MR, Johnson DS (1990) Computers and Intractability: a guide to the theory of NP-completeness. W. H. Freeman & Co., New York, NY, USA
43. Gärtner T (2008) Kernels for structured data. No v 72 in kernels for structured data. World Scientific. http://books.google.com/books?id=ykomKZ5rD1gC
44. Gartner T, Flach P, Wrobel S (2003) On graph kernels: hardness results and efficient alternatives. Lecture notes in computer science, pp 129–143
45. Ghias A, Logan J, Chamberlin D, Smith BC (1995) Query by humming: musical information retrieval in an audio database. In: ACM Multimedia, pp 231–236
46. Gibert J, Valveny E, Bunke H (2011) Dimensionality reduction for graph of words embedding. In: Jiang X, Ferrer M, Torsello A (eds) Graph-based representations in pattern recognition. Lecture notes in computer science, vol 6658. Springer Berlin, pp 22–31. http://dx.doi.org/10.1007/978-3-642-20844-7_3.10.1007/978-3-642-20844-7_3
47. Giuliani A, Benigni R, Zbilut JP, Webber Jr CL, Sirabella P, Colosimo A (2002) Nonlinear signal analysis methods in the Elucidation of protein sequence—structure relationships. ChemInform 33(28). doi:10.1002/chin.200228300. http://dx.doi.org/10.1002/chin.200228300
48. Goldfarb L (1984) A unified approach to pattern recognition. Pattern Recognit 17(5):575–582. doi:10.1016/0031-3203(84)90056-6. http://www.sciencedirect.com/science/article/B6V14-48MPJHK-J1/2/b156c1fd23bfed84bd0db8f8ec523c88
49. Goldschlager LM (1982) A universal interconnection pattern for parallel computers. J ACM 29:1073–1086 doi:10.1145/322344.322353. http://doi.acm.org/10.1145/322344.322353
50. Gori M, Maggini M, Sarti L (2004) Graph matching using random walks. In: Proceedings of the pattern recognition, 17th international conference on (ICPR'04) volume 3, vol 03, ICPR '04. IEEE Computer Society, Washington, DC, USA , pp 394–397. doi:10.1109/ICPR.2004.422. http://dx.doi.org/10.1109/ICPR.2004.422
51. Hagmann P, Cammoun L, Gigandet X, Meuli R, Honey CJ, Sporns O (2008) Mapping the structural core of human cerebral cortex. PLoS Biol 6(7):e159. doi:10.1371/journal.pbio.0060159. http://dx.doi.org/10.1371/journal.pbio.0060159
52. Hart P, Nilsson N, Raphael B (1968) A formal basis for the heuristic determination of minimum cost paths. IEEE Trans Syst Sci Cybern 4(2):100–107. doi:10.1109/TSSC.1968.300136. http://dx.doi.org/10.1109/TSSC.1968.300136
53. Haussler D (1999) Convolution kernels on discrete structures. Technical report
54. Hell P, Nesšetřil J (2004) Graphs and homomorphisms. Oxford lecture series in mathematics and its applications. Oxford University Press. http://books.google.it/books?id=bJXWV-qK7kYC

55. Hopcroft JE, Wong JK (1974) Linear time algorithm for isomorphism of planar graphs (Preliminary Report). In: Proceedings of the sixth annual ACM symposium on Theory of computing, STOC '74. ACM, New York, NY, USA, pp 172–184. doi:10.1145/800119.803896. http://doi.acm.org/10.1145/800119.803896
56. Imrich W, Klavžar S (2000) Product graphs, structure and recognition. Wiley-Interscience series in discrete mathematics and optimization. Wiley. http://books.google.com/books?id=EOnuAAAAMAAJ
57. Izenman A (2008) Modern multivariate statistical techniques: regression, classification, and manifold learning. Springer texts in statistics. Springer. http://books.google.com/books?id=1CuznRORa3EC
58. Jain B, Obermayer K (2011) Maximum likelihood for Gaussians on graphs. In: Jiang X, Ferrer M, Torsello A (eds) Graph-based representations in pattern recognition. Lecture notes in computer science, vol 6658. Springer, Berlin, pp 62–71. http://dx.doi.org/10.1007/978-3-642-20844-7_7.10.1007/978-3-642-20844-7_7
59. Jain BJ, Obermayer K (2009) Structure spaces. J Mach Learn Res 10:2667–2714. http://portal.acm.org/citation.cfm?id=1577069.1755876
60. Jain BJ, Wysotzki F (2004) Central clustering of attributed graphs. Mach Learn 56:169–207. doi:10.1023/B:MACH.0000033119.52532.ce. http://portal.acm.org/citation.cfm?id=1007760.1007768
61. Jiang X, Müunger A, Bunke H (2001) On median graphs: properties, algorithms, and applications. IEEE Trans Pattern Anal Mach Intell 23:1144–1151. doi:10.1109/34.954604. http://dx.doi.org/10.1109/34.954604
62. Jolliffe I (2002) Principal component analysis. Springer series in statistics. Springer. http://books.google.com/books?id=_olByCrhjwIC
63. Kashima H, Tsuda K, Inokuchi A (2003) Marginalized kernels between labeled graphs. In: Proceedings of the twentieth international conference on machine learning. AAAI Press, pp 321–328
64. Kazius J, McGuire R, Bursi R (2005) Derivation and validation of toxicophores for mutagenicity prediction. J Med Chem 48(1):312–320. doi:10.1021/jm040835a. http://pubs.acs.org/doi/abs/10.1021/jm040835a
65. Kohonen T (2001) Self-organizing maps. Springer series in information sciences. Springer. http://books.google.com/books?id=e4igHzyf078C
66. Kondor RI, Lafferty J (2002) Diffusion kernels on graphs and other discrete structures. In: Proceedings of the ICML, pp 315–322
67. Kruskal J (1962) Nonmetric multidimensional scaling: a numerical method. Psychometrika 29(2):115–129. http://ideas.repec.org/a/spr/psycho/v29y1964i2p115-129.html
68. Kruskal J, Wish M (1978) Multidimensional scaling. Quantitative applications in the social sciences. Sage Publications. http://books.google.com/books?id=ZzmIPcEXPf0C
69. Kullback S, Leibler RA (1951) On information and sufficiency. Ann Math Stat 22(1):79–86
70. Kuramochi M, Karypis G (2002) An efficient algorithm for discovering frequent subgraphs. Technical report, IEEE Transactions on Knowledge and Data Engineering
71. Leslie C, Kuang R (2004) Fast string kernels using inexact matching for protein sequences. J Mach Learn Res 5: 1435–1455. http://dl.acm.org/citation.cfm?id=1005332.1044708
72. Levenshtein VI (1966) Binary codes capable of correcting deletions, insertions, and reversals. Technical Report 8
73. Levi G (1973) A note on the derivation of maximal common subgraphs of two directed or undirected graphs. Calcolo 9:341–352. http://dx.doi.org/10.1007/BF02575586.10.1007/BF02575586
74. Livi L, Del Vescovo G, Rizzi A (2012) Graph recognition by seriation and frequent substructures mining. In: Proceeding of the first international conference on pattern recognition applications and methods 1:186–191. doi:10.5220/0003733201860191
75. Livi L, Del Vescovo G, Rizzi A (2012) Inexact graph matching through graph coverage. In: Proceeding of the first international conference on pattern recognition applications and methods 1:269–272. doi:10.5220/0003732802690272
76. Livi L, Rizzi A (2012) Parallel algorithms for tensor product-based inexact graph matching. In: Proceeding of the 2012 IEEE International Joint Conference on Neural Networks. IEEE, Brisbane, Australia, pp 2276–2283. doi:10.1109/IJCNN.2012.6252681. ISBN 978-1-4673-1489-3
77. Luxburg UV, Bousquet O (2003) Distance-based classification with Lipschitz functions. J Mach Learn Res 5:669–695
78. Mascioli FMF, Rizzi A, Panella M, Martinelli G (2000) Scale-based approach to hierarchical fuzzy clustering. Signal Process 80(6):1001–1016
79. Menchetti S, Costa F, Frasconi P (2005) Weighted decomposition kernels. In: Proceedings of the 22nd international conference on Machine learning, ICML '05. ACM, New York, NY, USA, pp 585–592. doi:10.1145/1102351.1102425. http://doi.acm.org/10.1145/1102351.1102425
80. Mercer J (1909) Functions of positive and negative type, and their connection with the theory of integral equations. In: Philosophical transactions of the royal society of London. Series A, containing papers of a mathematical or physical character, vol 209, pp 415–446. http://www.jstor.org/stable/91043
81. Munkres J (1957) Algorithms for the assignment and transportation problems. J Soc Ind Appl Math 5(1):32–38. http://scitation.aip.org/getabs/servlet/GetabsServlet?prog=normal&id=SMJMAP000005000001000032000001&idtype=cvips&gifs=yes
82. Munkres J (2000) Topology. Prentice Hall. http://books.google.com/books?id=XjoZAQAAIAAJ
83. Neuhaus M, Bunke H (2004) A probabilistic approach to learning costs for graph edit distance. In: Proceedings of the 17th international conference on pattern recognition, pp 389–393
84. Neuhaus M, Bunke H (2005) Self-organizing maps for learning the edit costs in graph matching. IEEE Trans Syst Man Cybern B 35:503–514
85. Neuhaus M, Bunke H (2006) A convolution edit kernel for error-tolerant graph matching. In: ICPR (4). IEEE Computer Society, pp 220–223
86. Neuhaus M, Bunke H (2006) A random walk kernel derived from graph edit distance. In: Yeung DY, Kwok J, Fred A, Roli F, de Ridder D (eds) Structural, syntactic, and statistical pattern recognition. Lecture notes in computer science, vol 4109. Springer, Berlin, pp 191–199. http://dx.doi.org/10.1007/11815921_20.10.1007/11815921_20
87. Neuhaus M, Bunke H (2007) A quadratic programming approach to the graph edit distance problem. In: Proceedings of the 6th IAPR-TC-15 international conference on Graph-based representations in pattern recognition, GbRPR'07. Springer-Verlag, Berlin, pp 92–102. http://portal.acm.org/citation.cfm?id=1769371.1769382
88. Neuhaus M, Bunke H (2007) Automatic learning of cost functions for graph edit distance. Inf Sci 177(1):239–247
89. Neuhaus M, Bunke H (2007) Bridging the gap between graph edit distance and kernel machines. Series in machine perception and artificial intelligence. World Scientific. http://books.google.com/books?id=xM_5hvL1AlkC
90. Neuhaus M, Riesen K, Bunke H (2006) Fast suboptimal algorithms for the computation of graph edit distance. In: Structural, syntactic, and statistical pattern recognition. LNCS. Springer, pp 163–172

91. Nocedal J, Wright S (2006) Numerical optimization. Springer series in operations research. Springer. http://books.google.com/books?id=eNlPAAAAMAAJ
92. Pekalska E, Duin R (2005) The dissimilarity representation for pattern recognition: foundations and applications. Series in machine perception and artificial intelligence. World Scientific. http://books.google.com/books?id=YPPr6eypHFwC
93. Peris G (2002) Fast cyclic edit distance computation with weighted edit costs in classification. In: Proceedings of the 16th international conference on pattern recognition (ICPR'02) volume 4, vol 4. ICPR '02. IEEE Computer Society, Washington, DC, USA, pp 40,184. http://portal.acm.org/citation.cfm?id=846227.848570
94. Qiu H, Hancock ER (2006) Graph matching and clustering using spectral partitions. Pattern Recognit 39:22–34. doi:10.1016/j.patcog.2005.06.014.. http://portal.acm.org/citation.cfm?id=1220964.1221155
95. Rao I, Sarma K (2010) On tensor product of standard graphs. Int J Comput Cognit 8(3):99
96. Ren P, Wilson RC, Hancock ER (2009) Characteristic polynomial analysis on matrix representations of graphs. In: Torsello A, Escolano F, Brun L (eds) GbRPR. Lecture notes in computer science, vol 5534. Springer, pp 243–252
97. Riesen K, Bunke H (2008) IAM graph database repository for graph based pattern recognition and machine learning. In: Proceedings of the 2008 joint IAPR international workshop on structural, syntactic, and statistical pattern recognition, SSPR & SPR '08. Springer-Verlag, Berlin, pp 287–297. doi:10.1007/978-3-540-89689-0_33.. http://dx.doi.org/10.1007/978-3-540-89689-0_33
98. Riesen K, Bunke H (2009) Approximate graph edit distance computation by means of bipartite graph matching. Image Vis Comput 27:950–959. doi:10.1016/j.imavis.2008.04.004. http://portal.acm.org/citation.cfm?id=1534927.1534959
99. Riesen K, Bunke H (2009) Reducing the dimensionality of dissimilarity space embedding graph kernels. Eng Appl Artif Intell 22:48–56. doi:10.1016/j.engappai.2008.04.006.. http://portal.acm.org/citation.cfm?id=1497654.1498530
100. Riesen K, Bunke H (2010) Graph classification and clustering based on vector space embedding. Series in Machine Perception and Artificial Intelligence. World Scientific Pub Co Inc. http://books.google.com/books?id=hKr9QwAACAAJ
101. Rizzi A, Del Vescovo G (2006) Automatic image classification by a granular computing approach. In: Machine learning for signal processing, 2006. Proceedings of the 2006 16th IEEE signal processing society workshop, pp 33–38. doi:10.1109/MLSP.2006.275517
102. Rizzi A, Panella M, Frattale Mascioli FM (2002) Adaptive resolution min-max classifiers. IEEE Trans Neural Netw 13:402–414
103. Robles-Kelly A, Hancock E (2009) String edit distance, random walks and graph matching. In: Caelli T, Amin A, Duin RPW, Ridder D, Kamel M (eds) Structural, syntactic, and statistical pattern recognition. Lecture notes in computer science 2396, chap 10. Springer, Berlin, Berlin, pp. 107–129. doi:10.1007/3-540-70659-3_10.. http://dx.doi.org/10.1007/3-540-70659-3_10
104. Robles-Kelly A, Hancock ER (2005) Graph edit distance from spectral seriation. IEEE Trans Pattern Anal Mach Intell 27:365–378. doi:10.1109/TPAMI.2005.56. http://dx.doi.org/10.1109/TPAMI.2005.56
105. Robles-Kelly A, Hancock ER (2007) A Riemannian approach to graph embedding. Pattern Recognit 40(3):1042–1056
106. Sakoe H (1978) Dynamic programming algorithm optimization for spoken word recognition. IEEE Trans Acoust Speech Signal Process 26:43–49
107. Sammon JW (1969) A nonlinear mapping for data structure analysis. IEEE Trans Comput 18:401–409. doi:10.1109/T-C.1969.222678. http://dx.doi.org/10.1109/T-C.1969.222678
108. Sampathkumar E (1975) On tensor product graphs. J Aust Math Soc Ser A 20(03):268–273
109. Sanfeliu A, Fu KS (1983) A distance measure between attributed relational graphs for pattern recognition. IEEE Trans Syst Man Cybern 13(3):353–362
110. Schenker A, Bunke H, Last M, Kandel A (2005) Graph-theoretic techniques for web content mining 62. World Scientific Pub. http://books.google.com/books?hl=en&lr=&id=hNJozkPJAEwC&oi=fnd&pg=PP1&dq=Graph-Theoretic+Techniques+for+Web+Content+Mining&ots=PPVMc-VCA1&sig=d6Fok33vLb-WBFstYIpr7ijn4jM
111. Schölkopf B, Smola A (2002) Learning with kernels: support vector machines, regularization, optimization, and beyond. Adapt Comput Mach Learn. MIT Press. http://books.google.com/books?id=y8ORL3DWt4sC
112. Schölkopf B, Tsuda K, Vert J (2004) Kernel methods in computational biology. Comput Mol Biol. MIT Press. http://books.google.it/books?id=SwAooknaMXgC
113. Shawe-Taylor J, Cristianini N (2004) Kernel methods for pattern analysis. Cambridge University Press. http://books.google.com/books?id=9i0vg12lti4C
114. Smola AJ, Kondor RI (2003) Kernels and regularization on graphs. In: Scholkopf B, Warmuth MK (eds) Computational learning theory and kernel machines, 16th annual conference on computational learning theory and 7th Kernel workshop, COLT/Kernel 2003. Lecture notes in computer science, vol 2777. Springer, Washington, pp 144–158. ISBN 3-540-40720-0
115. Tang J, Zhang C, Luo B (2006) A new approach to graph seriation. In: Proceedings of the first international conference on innovative computing, information and control, vol 3, ICICIC '06. IEEE Computer Society, Washington, DC, USA, pp 625–628. doi:10.1109/ICICIC.2006.385. http://dx.doi.org/10.1109/ICICIC.2006.385
116. Teo CH, Vishwanathan SVN (2006) Fast and space efficient string kernels using suffix arrays. In: Proceedings of the 23rd international conference on Machine learning, ICML 2006. ACM, New York, NY, USA , pp 929–936. doi:10.1145/1143844.1143961.. http://doi.acm.org/10.1145/1143844.1143961
117. Theodoridis S, Koutroumbas K (2006) Pattern recognition. Elsevier, Academic Press. http://books.google.com/books?id=gAGRCmp8Sp8C
118. Thomas LT, Valluri SR, Karlapalem K (2006) Margin: maximal frequent subgraph mining. In: Proceedings of the sixth international conference on data mining, ICDM'06. IEEE Computer Society, Washington, pp 1097–1101. doi:10.1109/ICDM.2006.102. http://dx.doi.org/10.1109/ICDM.2006.102. ISBN 0-7695-2701-9
119. Torsello A, Hancock ER (2007) Graph embedding using tree edit-union. Pattern Recognit 40:1393–1405 doi:10.1016/j.patcog.2006.09.006. http://portal.acm.org/citation.cfm?id=1224549.1224568
120. Torsello A, Robles-Kelly A, Hancock ER (2007) Discovering shape classes using tree edit-distance and pairwise clustering. Int J Comput Vis 72:259–285. doi:10.1007/s11263-006-8929-y.. http://portal.acm.org/citation.cfm?id=1210315.1210321
121. Tun K, Dhar P, Palumbo M, Giuliani A (2006) Metabolic pathways variability and sequence/networks comparisons. BMC Bioinf 7(1):24. doi:10.1186/1471-2105-7-24. http://www.biomedcentral.com/1471-2105/7/24
122. Valiant LG (1990) A bridging model for parallel computation. Commun. ACM 33:103–111. doi:10.1145/79173.79181. http://doi.acm.org/10.1145/79173.79181

123. Vishwanathan SVN, Borgwardt KM, Kondor RI, Schraudolph NN (2010) Graph kernels. J Mach Learn Res 11:1201–1242
124. Vishwanathan SVN, Smola AJ (2002) Fast kernels for string and tree matching. In: Neural information processing systems, pp 569–576
125. Washio T, Motoda H (2003) State of the art of graph-based data mining. SIGKDD Explor. Newsl 5:59–68. doi:10.1145/959242.959249. http://doi.acm.org/10.1145/959242.959249
126. Wasserman S, Faust K (1994) Social network analysis: methods and applications. Cambridge University Press, Cambridge
127. Watkins C (1999) Kernels from matching operations. Technical report, CSD-TR 98-07, University of London, Computer Science Department, Royal Holloway
128. Weaver N (1999) Lipschitz algebras. World Scientific. http://books.google.com/books?id=45rnwyVjg_QC
129. Xiao B, Gao X, Tao D, Li X (2008) HMM-based graph edit distance for image indexing. Int J Imaging Syst Technol 18(2–3):209–218. doi:10.1002/ima.20146.. http://dx.doi.org/10.1002/ima.20146
130. Yan X, Han J (2002) gSpan: graph-based substructure pattern mining. In: Proceedings of the 2002 IEEE international conference on data mining, ICDM '02. IEEE Computer Society, Washington, pp. 721–724. http://dl.acm.org/citation.cfm?id=844380.844811. ISBN 0-7695-1754-4
131. Yan X, Han J (2003) CloseGraph: mining closed frequent graph patterns. In: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining, KDD '03. ACM, New York, NY, USA, pp 286–295. doi:10.1145/956750.956784. http://doi.acm.org/10.1145/956750.956784
132. Yu H, Hancock ER (2006) String kernels for matching seriated graphs. In: Proceedings of the 18th international conference on pattern recognition, vol 04, ICPR '06. IEEE Computer Society, Washington, DC, USA, pp 224–228. doi:10.1109/ICPR.2006.1081. http://dx.doi.org/10.1109/ICPR.2006.1081
133. Zadeh LA (1965) Fuzzy sets. Inf Control 8(3):338–353. doi:10.1016/S0019-9958(65)90241-X. http://dx.doi.org/10.1016/S0019-9958(65)90241-X
134. Zinman GE, Zhong S, Bar-Joseph Z (2011) Biological interaction networks are conserved at the module level. BMC Syst Biol 5(1):134+. doi:10.1186/1752-0509-5-134.. http://dx.doi.org/10.1186/1752-0509-5-134