Chapter 2
Graph Edit Distance

Abstract Graph edit distance measures distances between two graphs g_1 and g_2 by the amount of distortion that is needed to transform g_1 into g_2. The basic distortion operations of graph edit distance can cope with arbitrary labels on both nodes and edges as well as with directed or undirected edges. Therefore, graph edit distance is one of the most flexible dissimilarity models available for graphs. The present chapter gives a formal definition of graph edit distance as well as some basic properties of this distance model. In particular, it presents an overview of how the cost model can be chosen in a certain graph edit distance application. Moreover, the exact computation of graph edit distance based on a tree search algorithm is outlined. In the last section of this chapter, three general approaches for graph edit distance-based pattern recognition are briefly reviewed.

2.1 Basic Definition and Properties

The basic idea of *edit distance* is to derive a dissimilarity measure from the number as well as the strength of the distortions that have to be applied to transform a source pattern into a target pattern. Originally, the concept of edit distance has been proposed for string representations [1, 2]. Eventually, the edit distance has been extended from strings to more general data structures such as trees [3] and graphs [4–8] (see [9] for a recent survey on the development of graph edit distance).

Given two graphs, the source graph $g_1 = (V_1, E_1, \mu_1, \nu_1)$ and the target graph $g_2 = (V_2, E_2, \mu_2, \nu_2)$, the basic idea of graph edit distance is to transform g_1 into g_2 using some edit operations. A standard set of edit operations is given by *insertions*, *deletions*, and *substitutions* of both nodes and edges. Note that other edit operations such as *merging* or *splitting* of both nodes and edges might be useful in some applications but not considered in the present book (we refer to [10] for an application of additional edit operations). We denote the substitution of two nodes $u \in V_1$ and $v \in V_2$ by $(u \rightarrow v)$, the deletion of node $u \in V_1$ by $(u \rightarrow \varepsilon)$, and the insertion of node $v \in V_2$ by $(\varepsilon \rightarrow v)$, where ε refers to the empty node. For edge edit operations we use a similar notation.

K. Riesen, *Structural Pattern Recognition with Graph Edit Distance*,
Advances in Computer Vision and Pattern Recognition,
DOI 10.1007/978-3-319-27252-8_2

Definition 2.1 (*Edit Path*) A set $\{e_1, \ldots, e_k\}$ of k edit operations e_i that transform g_1 completely into g_2 is called a *(complete) edit path* $\lambda(g_1, g_2)$ between g_1 and g_2. A *partial edit path*, i.e., a subset of $\{e_1, \ldots, e_k\}$, edits proper subsets of nodes and/or edges of the underlying graphs.

Note that the definition of an edit path perfectly corresponds to the definition of an error tolerant graph matching stated in Chap. 1 (see Definition 1.9). Remember that the matching of the edge structure is uniquely defined via operations which are actually carried out on the nodes (see the discussion about implicit edge mappings derived from node mappings in Sect. 1.3.2). The same applies for edit operations. That is, it is sufficient that an edit path $\lambda(g_1, g_2)$ covers the nodes from V_1 and V_2 only. Thus, from now on we assume that an edit path $\lambda(g_1, g_2)$ explicitly describes the correspondences found between the graphs' nodes V_1 and V_2, while the edge edit operations are implicitly given by these node correspondences.

Example 7 In Fig. 2.1 an edit path $\lambda(g_1, g_2)$ between two undirected and unlabeled graphs g_1 and g_2 is illustrated. Obviously, this edit path is defined by

$$\lambda = \{(u_1 \rightarrow \varepsilon), (u_2 \rightarrow v_3), (u_3 \rightarrow v_2), (u_4 \rightarrow v_1)\}.$$

This particular edit path implies the following edge edit operations:

$$\{((u_1, u_2) \rightarrow \varepsilon), ((u_2, u_3) \rightarrow (v_3, v_2)), ((u_3, u_4) \rightarrow (v_2, v_1)), ((u_2, u_4) \rightarrow \varepsilon)\}.$$

Let $\Upsilon(g_1, g_2)$ denote the set of all complete edit paths between two graphs g_1 and g_2. To find the most suitable edit path out of $\Upsilon(g_1, g_2)$, one introduces a cost $c(e)$ for every edit operation e, measuring the strength of the corresponding operation. The idea of such a cost is to define whether or not an edit operation e represents a strong modification of the graph. Clearly, between two similar graphs, there should exist an inexpensive edit path, representing low-cost operations, while for dissimilar graphs an edit path with high cost is needed. Consequently, the edit distance of two graphs is defined as follows.

Definition 2.2 (*Graph Edit Distance*) Let $g_1 = (V_1, E_1, \mu_1, \nu_1)$ be the source and $g_2 = (V_2, E_2, \mu_2, \nu_2)$ the target graph. The *graph edit distance* $d_{\lambda_{\min}}(g_1, g_2)$, or $d_{\lambda_{\min}}$ for short, between g_1 and g_2 is defined by

$$d_{\lambda_{\min}}(g_1, g_2) = \min_{\lambda \in \Upsilon(g_1, g_2)} \sum_{e_i \in \lambda} c(e_i), \tag{2.1}$$

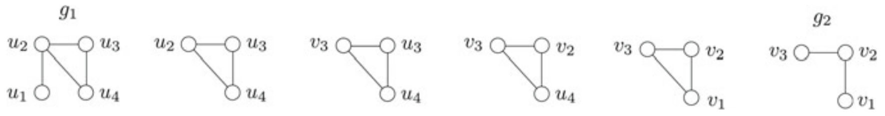

Fig. 2.1 An edit path λ between two graphs g_1 and g_2

where $\Upsilon(g_1, g_2)$ denotes the set of all complete edit paths transforming g_1 into g_2, c denotes the cost function measuring the strength $c(e_i)$ of node edit operation e_i (including the cost of all edge edit operations implied by the operations applied on the adjacent nodes of the edges), and $\lambda_{\min}$ refers to the minimal cost edit path found in $\Upsilon(g_1, g_2)$.

Clearly, there might be two (or more) edit paths with equal minimal cost in $\Upsilon(g_1, g_2)$. That is, the minimal cost edit path $\lambda_{\min} \in \Upsilon(g_1, g_2)$ is not necessarily unique.

2.1.1 Conditions on Edit Cost Functions

From the theoretical point of view, it is possible to extend a complete edit path $\{e_1, \ldots, e_k\} \in \Upsilon(g_1, g_2)$ with an arbitrary number of additional insertions $(\varepsilon \rightarrow v_1), \ldots, (\varepsilon \rightarrow v_n)$ followed by their corresponding deletions $(v_1 \rightarrow \varepsilon), \ldots, (v_n \rightarrow \varepsilon)$ (where $\{v_i\}_{i=1,\ldots,n}$ are arbitrary nodes). Hence, the size of the set of possible edit paths $\Upsilon(g_1, g_2)$ is infinite. In practice, however, three weak conditions on the cost function c are sufficient such that only a finite number of edit paths have to be evaluated to find the minimum cost edit path among all valid paths between two given graphs. First, we define the cost function to be nonnegative, i.e.,

$$c(e) \geq 0, \quad \text{for all node and edge edit operations } e. \tag{2.2}$$

We refer to this condition as *non-negativity*. Next we aim at assuring that only substitution operations of both nodes and edges have a zero cost, i.e.,

$$c(e) > 0, \quad \text{for all node and edge deletions and insertions } e. \tag{2.3}$$

Given this condition, edit paths containing an insertion of a node or edge followed by its subsequent deletion can be safely omitted. Finally, we want to prevent unnecessary substitutions to be added to an edit path. This is achieved by asserting that the elimination of such unnecessary substitutions from edit paths will not increase the corresponding sum of edit operation cost [11]. Formally,

$$\begin{aligned} c(u \rightarrow w) &\leq c(u \rightarrow v) + c(v \rightarrow w) \\ c(u \rightarrow \varepsilon) &\leq c(u \rightarrow v) + c(v \rightarrow \varepsilon) \\ c(\varepsilon \rightarrow v) &\leq c(\varepsilon \rightarrow u) + c(u \rightarrow v) \end{aligned} \tag{2.4}$$

for all nodes u, v, w and corresponding node substitutions, deletions, and insertions. We refer to this condition as *triangle inequality*. For instance, instead of substituting u with v and then substituting v with w (line 1), one can safely replace the two

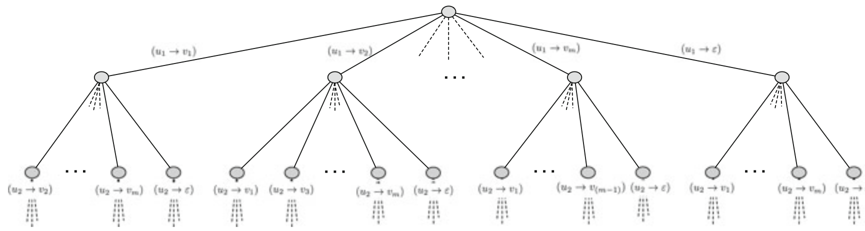

Fig. 2.2 The combinatorial explosion of edit paths between two graphs g_1 and g_2

right-hand side operations by the edit operation $(u \to w)$ on the left and will never miss out a minimum cost edit path. The same accounts for unnecessary substitutions of edges, of course. Therefore, each edit path $\{e_1, \dots, e_k\} \in \Upsilon(g_1, g_2)$, containing superfluous substitutions of both nodes and edges, can be replaced by a shorter edit path with a total cost that is equal to, or lower than, the sum of cost of the former edit path.

Given the above stated conditions 2.2, 2.3, and 2.4 on the edit cost function, it is guaranteed that adding edit operations to an edit path $\{e_1, \dots, e_k\} \in \Upsilon(g_1, g_2)$ containing operations on nodes or edges, which are neither involved in g_1 nor in g_2, will never decrease the overall edit cost of the edit path. Consequently, in order to find the minimum cost edit path $\lambda_{\min}$ among all possible edit paths $\Upsilon(g_1, g_2)$, we have to consider the $|V_1|$ node deletions, the $|V_2|$ node insertions, and the $|V_1| \times |V_2|$ possible node substitutions only.[1] In other words, the size of $\Upsilon(g_1, g_2)$ is bounded by a finite number of edit paths.

However, the upper bound on the number of edit paths in $\Upsilon(g_1, g_2)$ is exponential in the number of nodes of the involved graphs. Let us consider n nodes in g_1 ($V_1 = \{u_1, \dots, u_n\}$) and m nodes in g_2 ($V_2 = \{v_1, \dots, v_m\}$). By starting with an arbitrary node u_1 from V_1, $(m + 1)$ different edit operations have to be considered to build the following initial set of partial edit paths of size 1:

$$\{(u_1 \to v_1)\}, \{(u_1 \to v_2)\}, \dots, \{(u_1 \to v_m)\}, \{(u_1 \to \varepsilon)\} \qquad (2.5)$$

The next (arbitrarily chosen) node $u_2 \in V_1$ can now be substituted with one of the remaining nodes of V_2 or deleted. These edit operations applied on node $u_2 \in V_1$ can be appropriately combined with the $(m + 1)$ partial edit paths from list 2.5 resulting in $O(m^2)$ partial edit paths in total. This combinatorial process has to be continued until all nodes of both graphs are processed and thus, the set of possible edit paths $\Upsilon(g_1, g_2)$ contains $O(m^n)$ edit paths. In Fig. 2.2 the combinatorial explosion of possible edit paths between two graphs is illustrated.

Note that graph edit distance is not necessarily a metric. However, by adding the following two conditions to the above stated conditions of *non-negativity* and

[1]Remember that the source graph g_1 is edited such that it is transformed into the target graph g_2. Hence, the edit direction is essential and only nodes in g_1 can be deleted and only nodes in g_2 can be inserted.

triangle inequality (Conditions 2.2 and 2.4, respectively), the graph edit distance becomes metric [7]. First, we define identical substitutions to have zero cost, i.e.,

$$c(e) = 0, \tag{2.6}$$

if, and only if, edit operation e is an identical node or edge substitution (*identity of indiscernibles*). Second, we define the cost function c to be symmetric, i.e.,

$$c(e) = c(e^{-1}), \tag{2.7}$$

holds for any edit operation e on nodes and edges, where e^{-1} denotes the inverse edit operation to e (*Symmetry*).

2.1.2 Example Definitions of Cost Functions

The effectiveness of edit distance-based pattern recognition relies on the adequate definition of cost functions for the basic edit operations. In [12] an extensive review on different cost functions for graph edit distance can be found. In the present section, some important classes of cost functions for common label alphabets are defined.

In case of unlabeled graphs, the cost is usually defined via unit cost for all deletions and insertions of both nodes and edges, while substitutions are free of cost. Formally,

$$c(u \to \varepsilon) = c(\varepsilon \to u') = c((u, v) \to \varepsilon) = c(\varepsilon \to (u', v')) = 1$$
$$c(u \to u') = c((u, v) \to (u', v')) = 0$$

for all nodes $u, v \in V_1$ and $u', v' \in V_2$ as well as all edges $(u, v) \in E_1$ and $(u', v') \in E_2$.

In general, however, the cost $c(e)$ of a particular edit operation e is defined with respect to the underlying label alphabets L_V and L_E. For instance, for numerical node and edge labels, i.e., for label alphabets $L_V, L_E = \mathbb{R}^n$, a Minkowski distance can be used to model the cost of a substitution operation on the graphs (referred to as *Minkowski cost function* from now on). The Minkowski cost function defines the substitution cost proportional to the Minkowski distance of the two corresponding labels. The basic intuition behind this approach is that the more dissimilar the two labels are, the stronger is the distortion associated with the corresponding substitution.

Formally, given two graphs $g_1 = (V_1, E_1, \mu_1, \nu_1)$ and $g_2 = (V_2, E_2, \mu_2, \nu_2)$, where $\mu_1, \mu_2 : V_1, V_2 \to \mathbb{R}^n$, the cost for the three node edit operations can be defined by

$$
\begin{aligned}
c(u \rightarrow \varepsilon) &= \tau \\
c(\varepsilon \rightarrow v) &= \tau \\
c(u \rightarrow v) &= ||\mu_1(u) - \mu_2(v)||_p
\end{aligned}
$$

where $u \in V_1$, $v \in V_2$, and $\tau \in \mathbb{R}^+$ is a positive constant representing the cost of a node deletion/insertion.[2] Note that $||\mu_1(u) - \mu_2(v)||_p$ refers to the Minkowski distance of order p between two vectors $\mu_1(u), \mu_2(v) \in \mathbb{R}^n$. A similar cost model can be defined for edges, of course.

Note that any node substitution having a higher cost than 2τ can be safely replaced by a composition of a deletion and an insertion of the involved nodes (the same accounts for the edges). This behavior reflects the basic intuition that substitutions should be favored over deletions and insertions to a certain degree. A substitution cost for numerically labeled nodes (or edges) that is guaranteed to be in the interval $[0, 2\tau]$ can be defined, for instance, by

$$
c(u \rightarrow v) = \frac{1}{\frac{1}{2\tau} + \exp(-\alpha ||\mu_1(u) - \mu_2(v)||_p + \sigma)}.
$$

That is, the substitution cost for two nodes $u \in V_1$ and $v \in V_2$ is defined via a *Sigmoid function* of the (weighted) Minkowski distance between the corresponding labels $\mu_1(u)$ and $\mu_2(v)$. Note that we have two meta parameters in this cost function, viz., α and σ, which control the gradient and the left–right shift of the Sigmoid curve, respectively.

In some applications, it might be that the edges are attributed by an angle that specifies an undirected orientation of a line. That is, the angle value $\nu(e)$ of every edge e might be in the interval $(-\pi/2, +\pi/2]$. Because of the cyclic nature of angular measurements, a Minkowski-based distance would not be appropriate for the definition of a substitution cost. The following cost model for edges $e \in E_1$ and $e' \in E_2$ could be used in this case

$$
c(e \rightarrow e') = \min(\pi - |\nu(e) - \nu(e')|, |\nu(e) - \nu(e')|).
$$

In other applications, the node and/or edge labels might be not numerical and thus nonnumerical distance functions have to be employed to measure the cost of a particular substitution operation. For instance, the label alphabet can be given by the set of all strings of arbitrary size over a finite set of symbols. In this case a distance model for strings, as for instance the *string edit distance* [1, 2], could be used for measuring the cost of a substitution. In other problem domains, the label alphabet might be given by a finite set of n symbolic labels $L_{V/E} = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$. In such case a substitution cost model using a *Dirac function*, which returns zero when the involved labels are identical and a nonnegative constant otherwise, could be the method of choice.

[2]For the sake of symmetry, an identical cost τ for deletions and insertions is defined here.

Note that also combinations of various cost functions are possible. This might be particularly interesting when the nodes (or edges) are labeled with more than one attribute, for instance with a type (i.e., a symbolic label) together with a numerical measurement. For identically typed nodes, a Minkowski cost function could then be employed, for instance. In case of nonidentical types on the nodes, however, the substitution cost could be set to 2τ, which reflects the intuition that nodes with different types of labels cannot be substituted but have to be deleted and inserted, respectively.

The definition of application-specific cost functions, which can be adopted to the peculiarity of the underlying label alphabet, accounts for the flexibility of graph edit distance. Yet, prior knowledge about the labels and their meaning has to be available. If in a particular case this prior knowledge is not available, automatic procedures for learning the cost model from a set of sample graphs are available as well [13–17].

In [13], for instance, a cost inference method that is based on a distribution estimation of edit operations has been proposed (this particular approach is based on an idea originally presented in [18]). An *Expectation Maximization* algorithm is then employed in order to learn mixture densities from a labeled sample of graphs and derive edit costs. In [14] a system of *self-organizing maps* (SOMs) is proposed. This system represents the distance measuring spaces of node and edge labels and the learning process is based on the concept of self-organization. That is, it adapts the edit costs in such a way that the similarity of graphs from the same class is increased, while the similarity of graphs from different classes decreases. In [15] the graph edit process is formulated in a stochastic context and a maximum likelihood parameter estimation of the distribution of edit operations is performed. The underlying distortion model is also learned using an expectation maximization algorithm. From this model the desired cost functions can be finally derived. The authors of [16] present an optimization method to learn the cost model such that the *Hamming distance* between an oracle's node assignment and the automatically derived correspondence is minimized. Finally, in [17] another method for the automatic definition of edit costs has been proposed. This approach is based on an assignment defined by a specialist and an interactive and adaptive graph recognition method in conjunction with human interaction.

2.2 Computation of Exact Graph Edit Distance

In order to compute the graph edit distance $d_{\lambda_{\min}}(g_1, g_2)$ often A*-based search techniques using some heuristics are employed [19–23]. A* is a best-first search algorithm [24] which is *complete* and *admissible*, i.e. it always finds a solution if there is one and it never overestimates the cost of reaching the goal.

The basic idea of A*-based search methods is to organize the underlying search space as an ordered tree. The root node of the search tree represents the starting point of the search procedure, inner nodes of the search tree correspond to partial solutions, and leaf nodes represent complete—not necessarily optimal—solutions. In case of graph edit distance computation, inner nodes and leaf nodes correspond to partial and complete edit paths, respectively. Such a search tree is dynamically constructed at runtime by iteratively creating successor nodes linked by edges to the currently considered node in the search tree.

Algorithm 1 Exact Graph Edit Distance Algorithm

```
Input:   Non-empty graphs $g_1 = (V_1, E_1, \mu, \nu)$ and $g_2 = (V_2, E_2, \mu, \nu)$,
         where $V_1 = \{u_1, \ldots, u_n\}$ and $V_2 = \{v_1, \ldots, v_m\}$
Output:  A minimum cost edit path from $g_1$ to $g_2$
         e.g. $\lambda_{\min} = \{u_1 \rightarrow v_3, u_2 \rightarrow \varepsilon, \ldots, \varepsilon \rightarrow v_2\}$

1: initialize $OPEN$ to the empty set {}
2: For each node $w \in V_2$, insert the substitution $\{u_1 \rightarrow w\}$ into $OPEN$
3: Insert the deletion $\{u_1 \rightarrow \varepsilon\}$ into $OPEN$
4: loop
5:   Remove $\lambda_{\min} = \arg\min_{\lambda \in OPEN}\{g(\lambda) + h(\lambda)\}$ from $OPEN$
6:   if $\lambda_{\min}$ is a complete edit path then
7:     Return $\lambda_{\min}$ as the solution
8:   else
9:     Let $\lambda_{\min} = \{u_1 \rightarrow v_{\varphi_1}, \cdots, u_k \rightarrow v_{\varphi_k}\}$
10:      if $k < n$ then
11:        For each $w \in V_2 \setminus \{v_{\varphi_1}, \cdots, v_{\varphi_k}\}$, insert $\lambda_{\min} \cup \{u_{k+1} \rightarrow w\}$ into $OPEN$
12:        Insert $\lambda_{\min} \cup \{u_{k+1} \rightarrow \varepsilon\}$ into $OPEN$
13:      else
14:        Insert $\lambda_{\min} \cup \bigcup_{w \in V_2 \setminus \{v_{\varphi_1}, \cdots, v_{\varphi_k}\}} \{\varepsilon \rightarrow w\}$ into $OPEN$
15:      end if
16:    end if
17: end loop
```

In Algorithm 1, the A*-based method for optimal graph edit distance computation is given. The nodes of the source graph g_1 are processed in fixed, yet arbitrary, order $u_1, u_2, \ldots, u_n$. The substitution (line 11) and the deletion of a node (line 12) are considered simultaneously, which produces a number of successor nodes in the search tree. If all nodes of the first graph have been processed, the remaining nodes of the second graph are inserted in a single step (line 14). The set *OPEN* contains the search tree nodes, i.e., (partial or complete) edit paths, to be processed in the next steps.

In order to determine the most promising (partial) edit path $\lambda \in OPEN$, i.e., the edit path to be used for further expansion in the next iteration, a heuristic function is usually used (line 5). Formally, for a (partial) edit path λ in the search tree, we use $g(\lambda)$ to denote the accumulated cost of the edit operations $e_i \in \lambda$, and we use $h(\lambda) \geq 0$ for denoting the estimated cost to complete the edit path λ (several heuristics for the computation of $h(\lambda)$ exist [19–23]). The sum $g(\lambda) + h(\lambda)$ gives the total cost assigned to an open node in the search tree. Given that the estimation of the future cost $h(\lambda)$ is lower than, or equal to, the real cost, the algorithm is admissible. Hence, this procedure guarantees that if the current edit path $\lambda_{\min}$ removed from *OPEN* is complete (line 6), then $\lambda_{\min}$ is always optimal in the sense of providing minimal cost among all possible competing paths (line 7).

Note that the edge operations implied by the node edit operations can be derived from every partial or complete edit path λ during the search procedure given in Algorithm 1. The cost of these implied edge operations are dynamically added to the corresponding path $\lambda \in \mathit{OPEN}$ and are thus considered in the edit path assessment on line 5. Formally, for every node edit operation $(u \rightarrow v)$, which is included in λ, it is verified whether there are adjacent nodes to u and/or v which have been already edited in λ. If this is the case, the corresponding edge edit operations can be instantly triggered and the resulting edge edit cost added to the overall cost $g(\lambda)$ (the same accounts for the estimation $h(\lambda)$).

Example 8 In Fig. 2.3 a part of a search tree for graph edit distance computation between two undirected graphs g_1 and g_2 is shown. The nodes are labeled with integers, while the edges are unlabeled. We use unit cost for deletions and insertions of both nodes and edges. Edge substitutions are free of cost, while the cost for substituting a node $u \in V_1$ with a node $v \in V_2$ is defined via $c(u \rightarrow v) = |\mu_1(u) - \mu_2(v)|$.

The total cost $g(\lambda)$ of (partial) edit paths λ is computed by the node edit operation cost plus the cost of all edge operations that can be triggered according to the node operations carried out so far (we set $h(\lambda) = 0$ for all edit paths, i.e., no heuristic information is employed in this example). The accumulated edit costs are indicated in the tree search nodes.

Regard, for instance, the edit path displayed with dotted arrows. When $(u_1 \rightarrow v_2)$ and $(u_2 \rightarrow v_1)$ have been added to λ, the cost amounts to 2 (both node substitutions have a cost of 1 and the implied edge edit operation $((u_1, u_2) \rightarrow (v_1, v_2))$ has zero

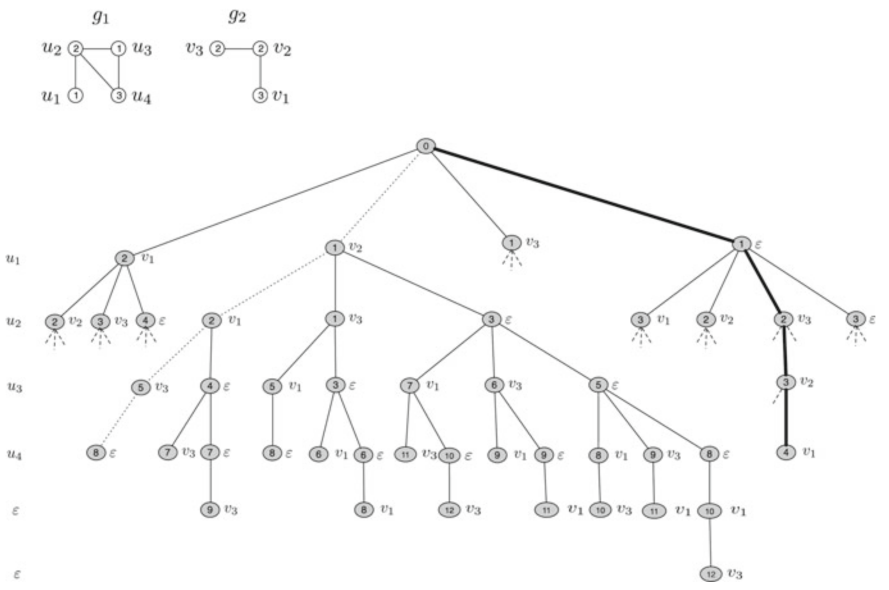

Fig. 2.3 Part of the search tree for graph edit distance computation between two graphs g_1 and g_2

cost). The next node substitution added to λ is $(u_3 \rightarrow v_3)$ with a cost of 1. This operation implies two edge edit operations, viz., the deletion of $(u_2, u_3) \in E_1$ and the insertion of $(v_2, v_3) \in E_2$. Hence, we have a total cost of 5 for this partial edit path. Finally, by adding $(u_4 \rightarrow \varepsilon)$ to the edit path, we add a cost of 3 to the overall cost (one node deletion and two implied edge deletions). That is, this edit path offers a total cost of 8.

The edit path displayed with bold arrows, i.e.,

$$\lambda_{\min} = \{(u_1 \rightarrow \varepsilon), (u_2 \rightarrow v_3), (u_3 \rightarrow v_2), (u_4 \rightarrow v_1)\}$$

corresponds to a minimal cost edit path with a total cost of 4.

2.3 Graph Edit Distance-Based Pattern Recognition

2.3.1 Nearest-Neighbor Classification

The traditional approach to graph edit distance-based pattern recognition is given by the *k-nearest-neighbor classification* (k-NN). In contrast with other classifiers such as artificial neural networks, Bayes classifiers, or decision trees [25], the underlying pattern space need not be rich in mathematical operations for nearest-neighbor classifiers to be applicable. More formally, in order to use the nearest-neighbor classifier, only a pattern dissimilarity measure must be available. Therefore, the k-NN classifier is perfectly suited for the graph domain, where several graph dissimilarity models, but only little mathematical structure, are available.

The k-NN classifier proceeds as follows. Let us assume that a graph domain $\mathcal{G}$, an appropriate definition of a graph edit distance $d : \mathcal{G} \times \mathcal{G} \rightarrow \mathbb{R}$, a set of labels Ω, and a labeled set of N training graphs $\{(g_i, \omega_i)\}_{1 \le i \le N} \subseteq \mathcal{G} \times \Omega$ is given. The 1-nearest-neighbor classifier (1-NN) is defined by assigning an input graph $g \in \mathcal{G}$ to the class of its most similar training graph. That is, the 1-NN classifier $f : \mathcal{G} \rightarrow \Omega$ is defined by

$$f(g) = \omega_j, \quad \text{where } j = \underset{1 \le i \le N}{\operatorname{argmin}}\ d(g, g_i).$$

If $k = 1$, the k-NN classifier's decision is based on just one graph from the training set, no matter if this graph is an outlier or a true class representative. That is, the decision boundary is largely based on empirical arguments. To render nearest-neighbor classification less prone to outlier graphs, it is common to consider not only the single most similar graph from the training set, but evaluate several of the most similar graphs. Formally, if $\{(g_{(1)}, \omega_{(1)}), \ldots, (g_{(k)}, \omega_{(k)})\} \subseteq \{(g_i, \omega_i)\}_{1 \le i \le N}$ are those k graphs in the training set that have the smallest distance $d(g, g_{(i)})$ to an input graph $g \in \mathcal{G}$, the k-NN classifier $f : \mathcal{G} \rightarrow \Omega$ is defined by

$$f(g) = \underset{\omega \in \Omega}{\operatorname{argmax}} |\{(g_{(i)}, \omega_{(i)}) : \omega_{(i)} = \omega\}|.$$

Nearest-neighbor classifiers provide us with a natural way to classify graphs by means of graph edit distance. However, the major restriction of nearest-neighbor classifiers is that a sufficiently large number of training graphs covering a substantial part of the graph domain must be available.

2.3.2 *Kernel-Based Classification*

Kernel methods have become one of the most rapidly emerging subfields in pattern recognition and related areas (see [26, 27] for a thorough introduction to kernel theory). The reason for this is twofold. First, kernel theory makes standard algorithms for pattern recognition (originally developed for vectorial data) applicable to more complex data structures such as strings, trees, or graphs. That is, kernel methods can be seen as a fundamental theory for bridging the gap between statistical and structural pattern recognition. Second, kernel methods allow one to extend basic linear algorithms to complex nonlinear ones in a unified and elegant manner.

The key idea of kernel methods is based on an essentially different way how the underlying data is represented [28]. In the kernel approach, an explicit data representation is of secondary interest. That is, rather than defining individual representations for each pattern, the data is represented by pairwise comparisons via *kernel functions* [27, 29].

Definition 2.3 (*Positive Definite Kernel*) Given a pattern domain $\mathcal{X}$, a *kernel function* $\kappa : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is a symmetric function, i.e., $\kappa(x_i, x_j) = \kappa(x_j, x_i)$, mapping pairs of patterns $x_i, x_j \in \mathcal{X}$ to real numbers. A kernel function κ is called positive definite[3] if, and only if, for all $N \in \mathbb{N}$,

$$\sum_{i,j=1}^{N} c_i c_j \kappa(x_i, x_j) \geq 0$$

for all $\{c_1, \ldots, c_N\} \subseteq \mathbb{R}$, and any choice of N objects $\{x_1, \ldots, x_N\} \subseteq \mathcal{X}$.

Kernel functions that are positive definite are often called *valid kernels*, *admissible kernels*, or *Mercer kernels*.

Kernels can be seen as pattern similarity measures satisfying the condition of symmetry and positive definiteness. Hence, graph edit distance becomes particularly interesting as it provides us with a symmetric graph dissimilarity measure, which can be readily turned into a similarity measure. In [30] monotonically decreasing transformations have been proposed which map low distance values to high similarity

[3]Note that positive definite functions according to the definition given in this section are sometimes called positive semi-definite since $\sum_{i,j=1}^{n} c_i c_j \kappa(x_i, x_j)$ can be zero and need not be strictly positive.

values and vice versa. Formally, given the edit distance $d(g, g')$ of two graphs g and g', the following similarity kernels can be defined, for instance (the list makes no claim to be complete).

- $\kappa(g, g') = -d(g, g')^2$
- $\kappa(g, g') = -d(g, g')$
- $\kappa(g, g') = tanh(-d(g, g'))$
- $\kappa(g, g') = \exp(-\gamma d(g, g')^2)$, where $\gamma > 0$

In [31] the fourth similarity kernel from the above-listed similarity functions is explicitly suggested for classifying distance-based data. Note that these kernel functions are not positive definite in general. However, there is theoretical and empirical evidence that using indefinite kernels may be reasonable if some conditions are fulfilled [11, 31].

Note that other, in particular more sophisticated, graph kernels have been proposed in conjunction with graph edit distance [11]. For instance, kernel functions that measure the similarity of two graphs by considering the node and edge substitutions from an optimal edit path $\lambda_{\min}$ only (i.e., omitting the deletions/insertions of both nodes and edges). The similarity of the substituted nodes and edges is then individually quantified and appropriately combined (by means of a multiplication). Moreover, also standard graph kernels such as *convolution kernels*, *random walk kernels*, or *diffusion kernels* have been substantially extended by means of graph edit distance in [11].

The following theorem gives a good intuition what kernel functions actually are (for proofs we refer to [26, 27]).

Theorem 2.1 [26, 27] *Let $\kappa : \mathscr{X} \times \mathscr{X} \rightarrow \mathbb{R}$ be a valid kernel on a pattern space $\mathscr{X}$, then there exists a possibly infinite-dimensional Hilbert space $\mathscr{F}$ and a mapping $\phi : \mathscr{X} \rightarrow \mathscr{F}$ such that*

$$\kappa(x, x') = \langle \phi(x), \phi(x') \rangle,$$

for all $x, x' \in \mathscr{X}$ where $\langle \cdot, \cdot \rangle$ denotes the dot product in $\mathscr{F}$.

In other words, kernels κ can be thought of as a dot product $\langle \cdot, \cdot \rangle$ in some (implicitly existing) feature space $\mathscr{F}$, and thus, instead of mapping patterns from the original pattern space $\mathscr{X}$ to the feature space $\mathscr{F}$ and computing their dot product there, one can simply evaluate the value of the kernel function in $\mathscr{X}$ [11].

In recent years, a huge amount of important algorithms has been *kernelized*, i.e., entirely reformulated in terms of dot products. These algorithms include support vector machine, nearest-neighbor classifier, perceptron algorithm, principal component analysis, Fisher discriminant analysis, canonical correlation analysis, k-means clustering, self-organizing map, partial least squares regression, and many others [26, 27]. Kernelized algorithms are commonly referred to as *kernel machines*.

Clearly, any kernel machine can be turned into an alternative algorithm by merely replacing the dot product $\langle \cdot, \cdot \rangle$ by a valid kernel $\kappa(\cdot, \cdot)$. This procedure is commonly referred to as *kernel trick* [26, 27]. The kernel trick is especially interesting for graph-based pattern representation since a graph kernel value (for instance the transformed graph edit distances defined above) can be fed into any kernel machine (e.g., a support vector machine). In other words, the graph kernel approach makes many powerful pattern recognition algorithms instantly applicable to graphs.

2.3.3 Classification of Vector Space Embedded Graphs

The motivation of *graph embedding* is similar to that of the kernel approach, viz., making the arsenal of algorithmic tools originally developed for vectorial data applicable to graphs. Yet, in contrast with kernel methods, which provide an implicit graph embedding only, graph embedding techniques result in an explicit vectorial description of the graphs.

The idea of a recent graph embedding framework [32] is based on the seminal work done by Pekalska and Duin [33]. The key idea of this graph embedding approach is to use the distances of an input graph to a number of training graphs, termed *prototype graphs*, as a vectorial description of the graph. That is, one makes use of the *dissimilarity representation* for pattern recognition rather than the original graph-based representation.

Definition 2.4 (*Graph Embedding*) Let us assume that a graph domain $\mathscr{G}$ is given. If $\mathscr{T} = \{g_1, \dots, g_N\} \subseteq \mathscr{G}$ is a set with N graphs and $\mathscr{P} = \{p_1, \dots, p_n\} \subseteq \mathscr{T}$ is a prototype set with $n \leq N$ graphs, the mapping

$$\phi_n^{\mathscr{P}} : \mathscr{G} \to \mathbb{R}^n$$

is defined as the function

$$\phi_n^{\mathscr{P}}(g) = (d(g, p_1), \dots, d(g, p_n)),$$

where $d : \mathscr{G} \times \mathscr{G} \to \mathbb{R}$ is an appropriately defined graph edit distance.

Obviously, by means of this definition we obtain a vector space where each axis is associated with a prototype graph $p_i \in \mathscr{P}$ and the coordinate values of an embedded graph g are the distances of g to the elements in $\mathscr{P}$. In this way, we can transform any graph g from the set $\mathscr{T}$, as well as any other graph from $\mathscr{G}$, into a vector of real numbers. Note that graphs, which have been selected as prototypes before, have a zero entry in their corresponding graph map.

The selection of the n prototypes $\mathscr{P} = \{p_1, \dots, p_n\}$ is a critical issue in the embedding framework. That is, not only the prototypes p_i themselves but also their number n affect the resulting graph embedding $\varphi_n^{\mathscr{P}}(\cdot)$, and thus the performance of the pattern recognition algorithm in the resulting embedding space. In [32] the selection of prototypes $\mathscr{P} = \{p_1, \dots, p_n\}$ is addressed by various procedures. Three of them are briefly outlined in the next three paragraphs.

First, a number of *prototype selection methods* have been presented [34–36]. These prototype selection strategies use some heuristics based on the underlying dissimilarities in the original graph domain. Basically, these approaches select prototypes from $\mathscr{T}$ that best possibly reflect the distribution of the graph set $\mathscr{T}$ or that cover a predefined region of $\mathscr{T}$. The rationale of this procedure is that capturing distances to significant prototypes from $\mathscr{T}$ lead to meaningful dissimilarity vectors.

A severe shortcoming of prototype selection strategies is that the dimensionality of the embedding space has to be determined by the user. Thus, a prototype selection method that automatically infers the dimensionality of the resulting embedding space has been proposed in [37]. This scheme is adopted from well-known concepts of *prototype reduction* [38] originally used for the task of condensing training sets in nearest-neighbor classification systems.

Finally, in [39, 40] the problem of prototype selection has been reduced to a feature subset selection problem. That is, for graph embedding, all available elements from the complete set $\mathscr{T}$ are used as prototypes, i.e., $\mathscr{P} = \mathscr{T}$. Next, various *feature selection strategies* [41–44] are applied to the resulting large-scale vectors eliminating redundancies and noise, finding good features, and simultaneously reducing the dimensionality of the graph maps.

References

1. V. Levenshtein, Binary codes capable of correcting deletions, insertions and reversals. Sov. Phys. Dokl. **10**(8), 707–710 (1966)
2. R.A. Wagner, M.J. Fischer, The string-to-string correction problem. J. Assoc. Comput. Mach. **21**(1), 168–173 (1974)
3. S.M. Selkow, The tree-to-tree editing problem. Inf. Process. Lett. **6**(6), 184–186 (1977)
4. M.A. Eshera, K.S. Fu, A graph distance measure for image analysis. IEEE Trans. Syst. Man Cybern. (Part B) **14**(3), 398–408 (1984)
5. W.H. Tsai, K.S. Fu, Error-correcting isomorphism of attributed relational graphs for pattern analysis. IEEE Trans. Syst. Man Cybern. (Part B) **9**(12), 757–768 (1979)
6. W.H. Tsai, K.S. Fu, Subgraph error-correcting isomorphisms for syntactic pattern recognition. IEEE Trans. Syst. Man Cybern. (Part B) **13**, 48–61 (1983)
7. H. Bunke, G. Allermann, Inexact graph matching for structural pattern recognition. Pattern Recognit. Lett. **1**, 245–253 (1983)
8. A. Sanfeliu, K.S. Fu, A distance measure between attributed relational graphs for pattern recognition. IEEE Trans. Syst. Man Cybern. (Part B) **13**(3), 353–363 (1983)
9. X. Gao, B. Xiao, D. Tao, X. Li, A survey of graph edit distance. Pattern Anal. Appl. **13**(1), 113–129 (2010)
10. R. Ambauen, S. Fischer, H. Bunke, Graph edit distance with node splitting and merging and its application to diatom identification, in *Proceedings of the 4th International Workshop on Graph Based Representations in Pattern Recognition*, ed. by E. Hancock, M. Vento. LNCS, vol. 2726 (Springer, New York, 2003), pp. 95–106
11. M. Neuhaus, H. Bunke, *Bridging the Gap Between Graph Edit Distance and Kernel Machines* (World Scientific, Singapore, 2007)
12. F. Serratosa, X. Cortés, A. Solé-Ribalta, On the graph edit distance cost: Properties and applications. Int. J. Pattern Recognit. Artif. Intell. **26**(5) (2012)

13. M. Neuhaus, H. Bunke, A probabilistic approach to learning costs for graph edit distance, in *Proceedings of the 17th International Conference on Pattern Recognition*, ed. by J. Kittler, M. Petrou, M. Nixon, vol. 3 (2004), pp. 389–393
14. M. Neuhaus, H. Bunke, Self-organizing maps for learning the edit costs in graph matching. IEEE Trans. Syst. Man Cybern. (Part B) **35**(3), 503–514 (2005)
15. M. Neuhaus, H. Bunke, Automatic learning of cost functions for graph edit distance. Inf. Sci. **177**(1), 239–247 (2007)
16. X. Cortes, F. Serratosa, Learning graph-matching edit-costs based on the optimality of the oracle's node correspondences. Pattern Recognit. Lett. **56**, 22–29 (2015)
17. F. Serratosa, A. Solé-Ribalta, X. Cortes, Automatic learning of edit costs based on interactive and adaptive graph recognition, in *Proceedings of the 8th International Workshop on Graph Based Representations in Pattern Recognition*, ed. by X. Jiang, M. Ferrer, A. Torsello. LNCS, vol. 6658 (2011), pp. 152–163
18. E. Ristad, P. Yianilos, Learning string edit distance. IEEE Trans. Pattern Anal. Mach. Intell. **20**(5), 522–532 (1998)
19. A.C.M. Dumay, R.J. van der Geest, J.J. Gerbrands, E. Jansen, J.H.C. Reiber, Consistent inexact graph matching applied to labelling coronary segments in arteriograms, in *Proceedings of the 11th IAPR International Conference on Pattern Recognition*, Conference C: Image, Speech and Signal Analysis, vol. III (1992), pp. 439–442
20. L. Gregory, J. Kittler, Using graph search techniques for contextual colour retrieval, in *Proceedings of the Joint IAPR International Workshop on Structural, Syntactic, and Statistical Pattern Recognition*, ed. by T. Caelli, A. Amin, R.P.W. Duin, M. Kamel, D. de Ridder. LNCS, vol. 2396 (2002), pp. 186–194
21. S. Berretti, A. Del Bimbo, E. Vicario, Efficient matching and indexing of graph models in content-based retrieval. IEEE Trans. Pattern Anal. Mach. Intell. **23**(10), 1089–1105 (2001)
22. K. Riesen, S. Fankhauser, H. Bunke, Speeding up graph edit distance computation with a bipartite heuristic, in *Proceedings of the 5th International Workshop on Mining and Learning with Graphs*, ed. by P. Frasconi, K. Kersting, K. Tsuda (2007), pp. 21–24
23. A. Fischer, R. Plamandon, Y. Savaria, K. Riesen, H. Bunke, A hausdorff heuristic for efficient computation of graph edit distance, in *Proceedings of the International Workshop on Structural and Syntactic Pattern Recognition*, ed. by P. Fränti, G. Brown, M. Loog, F. Escolano, M. Pelillo. LNCS, vol. 8621 (2014), pp. 83–92
24. P.E. Hart, N.J. Nilsson, B. Raphael, A formal basis for the heuristic determination of minimum cost paths. IEEE Trans. Syst. Sci. Cybern. **4**(2), 100–107 (1968)
25. R. Duda, P. Hart, D. Stork, *Pattern Classification*, 2nd edn. (Wiley-Interscience, New York, 2000)
26. J. Shawe-Taylor, N. Cristianini, *Kernel Methods for Pattern Analysis* (Cambridge University Press, Cambridge, 2004)
27. B. Schölkopf, A. Smola, *Learning with Kernels* (MIT Press, Cambridge, 2002)
28. B. Schölkopf, K. Tsuda, J.-P. Vert (eds.), *Kernel Methods in Computational Biology* (MIT Press, Cambridge, 2004)
29. C.H. Berg, J. Christensen, P. Ressel, *Harmonic Analysis on Semigroups* (Springer, New York, 1984)
30. M. Neuhaus, H. Bunke, *Bridging the Gap Between Graph Edit Distance and Kernel Machines* (World Scientific, Singapore, 2007)
31. B. Haasdonk, Feature space interpretation of SVMs with indefinite kernels. IEEE Trans. Pattern Anal. Mach. Intell. **27**(4), 482–492 (2005)
32. K. Riesen, H. Bunke, *Graph Classification and Clustering Based on Vector Space Embedding* (World Scientific, Singapore, 2010)
33. E. Pekalska, R. Duin, *The Dissimilarity Representation for Pattern Recognition: Foundations and Applications* (World Scientific, Singapore, 2005)
34. E. Pekalska, R. Duin, P. Paclik, Prototype selection for dissimilarity-based classifiers. Pattern Recognit. **39**(2), 189–208 (2006)

35. B. Spillmann, M. Neuhaus, H. Bunke, E. Pekalska, R. Duin, Transforming strings to vector spaces using prototype selection, in *Proceedings of the 11th International Workshop on Strucural and Syntactic Pattern Recognition*, ed. by D.-Y. Yeung, J.T. Kwok, A. Fred, F. Roli, D. de Ridder. LNCS, vol. 4109 (2006), pp. 287–296
36. K. Riesen, H. Bunke, Graph classification based on vector space embedding. Int. J. Pattern Recognit. Artif. Intell. **23**(6), 1053–1081 (2008)
37. K. Riesen, H. Bunke, Dissimilarity based vector space embedding of graphs using prototype reduction schemes, in *Proceedings of the 6th International Conference Machine Learning and Data Mining in Pattern Recognition*, ed. by P. Perner (2009), pp. 617–631. (Accepted for publication in)
38. J.C. Bezdek, L. Kuncheva, Nearest prototype classifier designs: an experimental study. Int. J. Intell. Syst. **16**(12), 1445–1473 (2001)
39. K. Riesen, V. Kilchherr, H. Bunke, Reducing the dimensionality of vector space embeddings of graphs, in *Proceedings of the 5th International Conference on Machine Learning and Data Mining*, ed. by P. Perner. LNAI, vol. 4571 (Springer, Berlin, 2007), pp. 563–573
40. K. Riesen, H. Bunke, Reducing the dimensionality of dissimilarity space embedding graph kernels. Eng. Appl. Artif. Intell. **22**(1), 48–56 (2008)
41. K. Kira, L.A. Rendell, A practical approach to feature selection, *Ninth International Workshop on Machine Learning* (Morgan Kaufmann, Burlington, 1992), pp. 249–256
42. P. Pudil, J. Novovicova, J. Kittler, Floating search methods in feature-selection. Pattern Recognit. Lett. **15**(11), 1119–1125 (1994)
43. A. Jain, D. Zongker, Feature selection: evaluation, application, and small sample performance. IEEE Trans. Pattern Anal. Mach. Intell. **19**(2), 153–158 (1997)
44. R.A. Fisher, The statistical utilization of multiple measurements. Ann. Eugen. **8**, 376–386 (1938)