# BRIDGING THE GAP BETWEEN GRAPH EDIT DISTANCE AND KERNEL MACHINES

Michel Neuhaus • Horst Bunke

SERIES IN MACHINE PERCEPTION ARTIFICIAL INTELLIGENCE
Volume 68

World Scientific

# BRIDGING THE GAP BETWEEN GRAPH EDIT DISTANCE AND KERNEL MACHINES

**SERIES IN MACHINE PERCEPTION AND ARTIFICIAL INTELLIGENCE***

*Editors:* **H. Bunke** (Univ. Bern, Switzerland)
**P. S. P. Wang** (Northeastern Univ., USA)

---

Vol. 54: Fundamentals of Robotics — Linking Perception to Action
(*M. Xie*)

Vol. 55: Web Document Analysis: Challenges and Opportunities
(Eds. *A. Antonacopoulos and J. Hu*)

Vol. 56: Artificial Intelligence Methods in Software Testing
(Eds. *M. Last, A. Kandel and H. Bunke*)

Vol. 57: Data Mining in Time Series Databases y
(Eds. *M. Last, A. Kandel and H. Bunke*)

Vol. 58: Computational Web Intelligence: Intelligent Technology for Web Applications
(Eds. *Y. Zhang, A. Kandel, T. Y. Lin and Y. Yao*)

Vol. 59: Fuzzy Neural Network Theory and Application
(*P. Liu and H. Li*)

Vol. 60: Robust Range Image Registration Using Genetic Algorithms and the Surface Interpenetration Measure
(*L. Silva, O. R. P. Bellon and K. L. Boyer*)

Vol. 61: Decomposition Methodology for Knowledge Discovery and Data Mining: Theory and Applications
(*O. Maimon and L. Rokach*)

Vol. 62: Graph-Theoretic Techniques for Web Content Mining
(*A. Schenker, H. Bunke, M. Last and A. Kandel*)

Vol. 63: Computational Intelligence in Software Quality Assurance
(*S. Dick and A. Kandel*)

Vol. 64: The Dissimilarity Representation for Pattern Recognition: Foundations and Applications
(*Elżbieta Pękalska and Robert P. W. Duin*)

Vol. 65: Fighting Terror in Cyberspace
(*Eds. M. Last and A. Kandel*)

Vol. 66: Formal Models, Languages and Applications
(Eds. *K. G. Subramanian, K. Rangarajan and M. Mukund*)

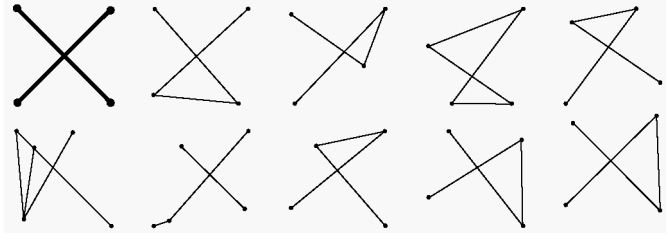
Vol. 67: Image Pattern Recognition: Synthesis and Analysis in Biometrics
(Eds. *S. N. Yanushkevich, P. S. P. Wang, M. L. Gavrilova and S. N. Srihari* )

Vol. 68 Bridging the Gap Between Graph Edit Distance and Kernel Machines
(*M. Neuhaus and H. Bunke*)

*For the complete list of titles in this series, please write to the Publisher.

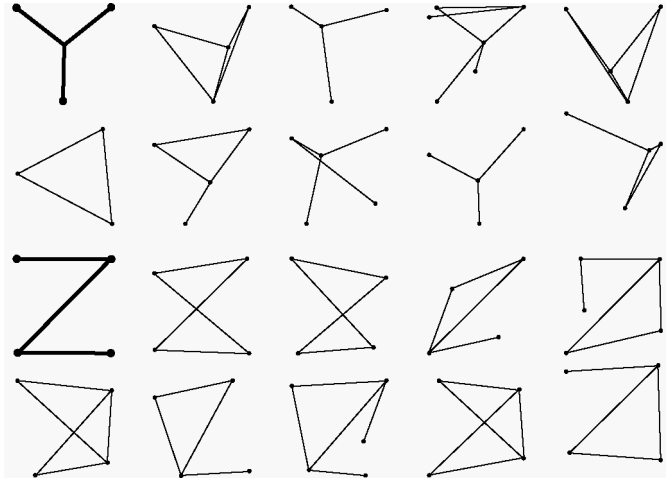
Series in Machine Perception and Artificial Intelligence – Vol. 68

# BRIDGING THE GAP BETWEEN GRAPH EDIT DISTANCE AND KERNEL MACHINES



**Michel Neuhaus**
**Horst Bunke**

University of Bern, Switzerland



World Scientific

NEW JERSEY • LONDON • SINGAPORE • BEIJING • SHANGHAI • HONG KONG • TAIPEI • CHENNAI

*Published by*

World Scientific Publishing Co. Pte. Ltd.

5 Toh Tuck Link, Singapore 596224

*USA office:* 27 Warren Street, Suite 401-402, Hackensack, NJ 07601

*UK office:* 57 Shelton Street, Covent Garden, London WC2H 9HE

**British Library Cataloguing-in-Publication Data**
A catalogue record for this book is available from the British Library.

**BRIDGING THE GAP BETWEEN GRAPH EDIT DISTANCE AND KERNEL MACHINES**
**Series in Machine Perception and Artificial Intelligence — Vol. 68**



ISBN-13 978-981-270-817-5
ISBN-10 981-270-817-0

Printed in Singapore.

Michel Neuhaus dedicates this book to Angelika and Fred
Horst Bunke dedicates this book to Helga

This page intentionally left blank

# Preface

When we first started working on the problem of making the kernel machine approach applicable to the classification of graphs a couple of years ago, our efforts were mainly driven by the fact that kernel methods had led to impressive performance results on many data sets. It didn't take us long to appreciate the sheer elegance of how difficult pattern recognition problems can be addressed by means of kernel machines, which is particularly the case for complex data structures such as graphs. To witness researchers from a large variety of domains bring together new perspectives on kernel machines and their applications has always been exciting. We are very delighted that we have been given the opportunity to address a few extremely interesting open issues in such a rapidly evolving research field. In this book, which is an extended and revised version of the first author's PhD thesis, we present the major results of our work related to graph kernels. We give an introduction to the general problem, discuss theoretical issues of graph matching and kernel machines, present a number of error-tolerant graph kernels applicable to a wide variety of different graphs, and give an experimental evaluation on real-world data.

The first author owes many thanks to all the people who have accompanied him in the last few years. First of all, he is grateful to the second author for directing him through this work with excellent advise and an open attitude towards new ideas. Many colleagues have contributed invaluable parts to this work. Roman Bertolami, Dr. Christophe Irniger, Vivian Kilchherr, Marcus Liwicki, Kaspar Riesen, Andreas Schlapbach, Barbara Spillmann, and Dr. Tamás Varga—thank you very much. The first author is also very grateful to his family and friends for their support and encouragement during all these years. Furthermore, we would like to acknowledge partial funding by the Swiss National Science Foundation NCCR program *Inter-*

*active Multimodal Information Management IM2* in the individual project *Multimedia Information Access and Content Protection*.

*Michel Neuhaus and Horst Bunke*
*May 2007*

# Contents

This page intentionally left blank

# Chapter 1

# Introduction

Graphs are a popular concept for the representation of structured information. On a very basic level, a graph can be defined by a set of entities and a set of connections between these entitites. Due to their universal definition, graphs have found widespread application for many years as tools for the representation of complex relations. For instance, in software engineering, graphs are used for the visualization of dependencies and interactions of components in large software systems. In relational databases, information is stored by defining relations between individual entities by means of unique identifiers, which is essentially equivalent to a graph structure. Flow-chart diagrams can be used to decompose complex processes into consise systems of states, rules, and transitions, which can be interpreted in a straightforward manner as nodes and edges in a graph. The world-wide web is another example of a graph, where webpages are regarded as documents and hyperlinks as connections between related documents. The general concept of representing and processing information in terms of binary relations, that is, links between two objects at a time, seems to be applicable to a wide range of problems. In the context of this book, graphs are used for the matching of structured data.

The key task in pattern recognition is the analysis and classification of objects [Friedman and Kandel (1999); Duda *et al.* (2000)]. In principle these objects, or patterns, can be of any kind, including images taken with a digital camera, spoken words captured by a microphone, or handwritten texts obtained by means of a digital pen. The detection and extraction of individual objects in images, for instance, can be referred to as a pattern analysis, whereas the recognition of cursive handwriting corresponds to pattern classification. The first step in any pattern recognition system consists of the representation of patterns by data structures that can be interpreted

by the recognition algorithm to be employed. For example, an image captured with a digital camera can be regarded as a pixel matrix of grayscale values, and recorded speech can be represented by a time-amplitude signal. In statistical pattern recognition [Duda *et al.* (2000)], patterns are represented by feature vectors of a fixed dimension. The basic idea is to define a set of features describing properties of the underlying patterns that are relevant for the recognition task under consideration. Such feature extraction procedures can be developed for patterns from completely different domains, which makes the statistical pattern recognition approach widely applicable. The main advantage of statistical pattern recognition is that feature vectors belong to a mathematically rich vector space. If a nearest-neighbor method is used for classification, the Euclidean distance between vectors can easily be computed; if principal component analysis is used for dimensionality reduction, vectors can be projected onto the maximal-variance subspace; if the K-means algorithm is used for clustering, the mean of a set of patterns can easily be computed; and if Gaussian mixture modelling is used for novelty detection, the mean and covariance matrix can be estimated empirically from a sample set of vectors.

Yet, in some cases, representing patterns uniformly by feature vectors of a fixed dimension is not appropriate. Especially if structure plays an important role, graphs offer a versatile alternative to feature vectors. If a feature vector is regarded as an unary attributed relation, graphs can analogously be interpreted as a set of attributed unary and binary relations of arbitrary size. The key advantage of graphs is that their representational power is clearly higher than that of feature vectors, since nodes and edges can be labeled with feature vectors themselves, and the number of nodes and edges can be adapted to the complexity of the pattern under consideration. For instance, if objects are to be detected and extracted from images, graphs are a natural choice of representation, since they allow us to represent extracted objects by nodes and object neighborhood relations by edges. A major difficulty of graph based pattern recognition, however, is the definition and application of algorithms for pattern analysis and classification in the space of graphs. While the representational power of graphs is higher than that of feature vectors, the sparse space of graphs contains almost no mathematical structure, in contrast to feature vector spaces. Basic mathematical concepts that are well-defined in vector spaces, such as the Euclidean distance or linear combinations of feature vectors, are often required for standard algorithms to be applicable. For graphs, these basic operations cannot be defined in a standardized way, and the

application-specific definition of graph operations often involves a tedious and time-consuming development process.

In recent years, a large number of graph matching methods based on various matching paradigms have been proposed [Conte *et al.* (2004)], ranging from the spectral decomposition of graph matrices to the training of artificial neural networks and from continuous optimization algorithms to optimal tree search procedures. Most graph matching algorithms are either restricted to a certain class of graphs or only applicable to weakly distorted data. For instance, a number of graph matching problems can be solved more efficiently for planar graphs than for general graphs. On the other hand, spectral graph matching methods are in principle applicable to arbitrary unlabeled graphs, but appear to be rather sensitive to noise. In view of these limitations, graph edit distance is considered one of the most powerful methods for graph matching [Bunke and Allermann (1983); Sanfeliu and Fu (1983)]. Graph edit distance is an error-tolerant dissimilarity measure for arbitrarily structured and arbitrarily labeled graphs. The basic idea is to define the dissimilarity of two graphs by the minimum amount of distortion that is needed to transform one graph into the other. The edit distance provides us with a general dissimilarity measure in the space of graphs, but this is not sufficient for most standard pattern recognition algorithms. In fact, edit distance based graph matching is basically limited to nearest-neighbor classification and K-median clustering. Unfortunately, for nearest-neighbor classifiers to be successful, a large number of training patterns covering a substantial part of the pattern space are required. In vector spaces, where the Euclidean distance to a large number of patterns can easily be computed, such a procedure is usually feasible. The edit distance algorithm, on the other hand, is computationally inefficient, and edit distance based nearest-neighbor classification is therefore restricted to rather small training sets.

The basic limitation of graph matching is due to the lack of mathematical structure in the space of graphs. Kernel machines, a novel class of algorithms for pattern analysis and classification, offer an elegant solution to this problem [Schölkopf and Smola (2002); Shawe-Taylor and Cristianini (2004)]. The basic idea of kernel machines is to address a pattern recognition problem in a related vector space instead of the original pattern space. That is, rather than defining mathematical operations in the space of graphs, all graphs are mapped into a vector space where these operations are readily available. Obviously, the difficulty is to find a mapping that preserves the structural similarity of graphs, at least to a certain ex-

tent. In other words, if two graphs are structurally similar, the two vectors representing these graphs should be similar as well, since the objective is to obtain a vector space embedding that preserves the characteristics of the original space of graphs. A key result from the theory underlying kernel machines states that an explicit mapping from the pattern space into a vector space is not required. Instead, from the definition of a pattern similarity measure, or kernel function, it follows that there exists such a vector space embedding and that the kernel function can be used to extract the information from vectors that is relevant for recognition. In fact, the family of kernel machines consists of all algorithms that can be formulated in terms of such a kernel function, including standard methods for pattern analysis and classification such as principal component analysis and nearest-neighbor classification. Hence, from the definition of a graph similarity measure, we obtain an implicit embedding of the entire space of graphs into a vector space.

In statistical pattern recognition, where patterns are not represented by graphs, but feature vectors, standard algorithms can directly be applied to the original pattern space without mapping and without kernel function. Principal component analysis, to give an example, extracts dominant linear directions in vectorial data sets. If principal component analysis is applied in conjunction with a kernel function and an implicit mapping of patterns into another vector space, one obtains an extension of standard principal component analysis that is able to capture non-linear characteristics. Hence, the kernel machine framework can be used to extend, in a single step, linear algorithms to non-linear ones. Moreover, there is theoretical evidence that, under some weak conditions, mapping patterns into vector spaces of higher dimension may be of advantage for pattern analysis and classification. The most prominent kernel machine is undoubtedly the support vector machine (SVM). The training of SVMs is based on insights from statistical learning theory and is considered one of the most promising methods for estimating hidden class boundaries from a sample set of patterns. The basic idea of the SVM training is to strive for a balanced condition between overfitting and underfitting of the training data at hand, so that the resulting classifier is expected to perform well on unseen data from the same population. Such theoretical considerations and their integration in the kernel machine framework render SVMs extremely powerful for pattern recognition. For graph matching, in addition to these interesting properties, the most important advantage of kernel machines is that to make the large class of kernel machines applicable to graphs, the underly-

ing pattern space need not be endowed with any mathematical structure beyond the existence of a kernel function.

The objective of this book is to define graph kernel functions that are to a certain extent related to graph edit distance, either by deriving kernel functions from edit distance or by incorporating the edit operation distortion model into the definition of kernel functions. The rationale behind this idea is that the application of edit distance based graph kernels to kernel machines is expected to outperform traditional edit distance based nearest-neighbor classifiers.

This book is structured as follows. In Chap. 2, basic notations and graph definitions are introduced and pointers to graph matching applications are given. Exact graph matching algorithms, such as subgraph isomorphism, are briefly covered, and error-tolerant approaches to graph matching proposed in recent years are discussed. Chapter 3 is devoted to graph edit distance. The edit distance measure is introduced and exact and approximate edit distance algorithms are given. Edit distance based nearest-neighbor classification is briefly addressed, and an application of edit distance to the fusion of graphs is presented. Chapter 4 is concerned with theoretical considerations and properties of kernel machines. Kernel functions are formally introduced, and support vector machines, kernel principal component analysis, and Fisher discriminant analysis are discussed. In Chap. 5, the main contribution of this book, novel graph kernels related to edit distance are presented. An experimental evaluation of these kernels, in comparison to a traditional nearest-neighbor classifier, follows in Chap. 6. The performance of the graph kernels in terms of recognition accuracy and running time is evaluated on five graph data sets representing line drawings, pictures, microscopic images, fingerprints, and molecules. Finally, a summary and concluding remarks are given in Chap. 7.

This page intentionally left blank

## Chapter 2

# Graph Matching

The objective in pattern recognition is to develop systems for the analysis or classification of input patterns. The first, and very important, issue to be addressed by any pattern recognition system is the representation of input patterns by data structures that can be interpreted by the recognition algorithm under consideration. If the description of patterns is not sufficiently accurate, even the most powerful classifier will fail. One of the most common data representation techniques is to compute a number of descriptive properties, or features, for each pattern, and represent patterns by their corresponding set of properties, or feature vector. Such a feature extraction procedure has the advantage that patterns are represented by vectors of real numbers, which makes it easy to perform vector space operations required by many algorithms for pattern analysis and classification.

For complex patterns, however, the homogeneous nature of feature vectors — each vector has the same dimension, and the $k$-th component of each vector describes the same feature — restricts their representational power substantially. For example, it is rather difficult to come up with a feature vector representation for patterns that are composed of various individual parts. In such cases it is often more suitable to represent patterns by graphs. The main advantage of graphs over feature vectors is that graphs allow for the representation of complex structures with arbitrary node and edge labels. Whereas feature vectors are only able to describe patterns by a set of properties, graphs can be used to explicitly model attributed relations between different parts of an object. Also, the dimensionality of graphs, that is, the number of nodes and edges, need not be constant for all objects. This means that a complex object can be represented by a larger and more complex graph than a simple object. In the case of feature vectors, all objects are represented by a vector of fixed dimension without

recourse to the complexity of the actual object. Conversely, working with graphs is unequally more challenging than working with feature vectors, as even basic operations, such as the mean of a set of graphs or a graph distance measure, cannot be defined in a standard way, but must be provided depending on the specific application. Almost none of the common methods for pattern analysis can be applied to graphs without significant modifications.

Generally, graph representations are more appropriate than feature vectors if structure plays an important role in the characterization of patterns. One class of patterns for which graphs offer a suitable representation are those patterns that are already given in terms of structured networks. Examples of structured networks include systems dealing with the analysis of physical networks such as the internet [Dickinson *et al.* (2004)], logical networks such as web documents [Schenker *et al.* (2004b)], conceptual networks [Montes-y-Gómez *et al.* (2000)], or semantic networks [Ehrig (1992)]. Interpreting the term *network* in a broader sense, one could also include the classification of fingerprints [Maio and Maltoni (1996)], the analysis of chemical structures [Rouvray and Balaban (1979)], applications from bioinformatics [Baxter and Glasgow (2000)], the recognition of graphical symbols [Lladós and Sánchez (2004); Cordella *et al.* (2000)], character recognition [Rocha and Pavlidis (1994); Suganthan and Yan (1998)], and shape analysis [Shokoufandeh and Dickinson (2001)], to name a few. Another important class of patterns often represented by graphs are patterns that are composed of several smaller parts, such as images segmented into regions [Ambauen *et al.* (2003); Le Saux and Bunke (2005)].

Using graph pattern representations for classification, the key task is to determine whether two graphs represent the same object or two different objects, or more formally whether two graphs belong to the same class or different classes. The process of evaluating the structural similarity of two graphs is commonly referred to as graph matching. The main problem in graph matching is to define the amount and type of structural noise that is admissible for a matching to be successful, that is, for two graphs to be deemed similar. This issue has been addressed by a large number of methods modeling structural variation. For an extensive review of graph matching methods and applications, the reader is referred to [Conte *et al.* (2004)]. In this chapter, basic notations and definitions are introduced and an overview of standard techniques is given for both exact and error-tolerant graph matching.

## 2.1 Graph and Subgraph

Graphs can be defined in various ways. The following well-established definition is sufficiently general to include a number of important special cases of graphs.

**Definition 2.1 (Graph).** *Assume that a label alphabet $L$ is given. A graph $g$ is defined by the four-tuple $g = (V, E, \mu, \nu)$, where*

- *$V$ is a finite set of nodes,*
- *$E \subseteq V \times V$ is the set of edges,*
- *$\mu : V \to L$ is the node labeling function, and*
- *$\nu : E \to L$ is the edge labeling function.*

*The number of nodes of a graph $g$ is denoted by $|g|$.*

This definition includes arbitrarily structured graphs with any kind of node and edge labels. The set of nodes, or vertices, $V$ can be regarded as a set of node identifiers. Edges are defined in a directed manner by a pair of nodes $(u, v)$, where $u \in V$ is the source node and $v \in V$ the target node. In graph based pattern recognition, edges $(u, u)$ connecting a node $u \in V$ with itself are often ignored. The label alphabet $L$ is not constrained any further. In most cases, the alphabet is defined as a vector space $L = \mathbb{R}^n$ or a set of discrete labels $L = \{\alpha, \beta, \gamma, \ldots\}$.

The class of *undirected graphs* is obtained if for each edge $(u, v) \in E$ there has to be an edge $(v, u) \in E$ such that $\nu(u, v) = \nu(v, u)$. In this case, the direction of an edge can be ignored, since there are always edges in both directions. If the label alphabet is defined as $L = \{\varepsilon\}$, all nodes and edges are labeled with the same label, and the graph can therefore be considered *unlabeled*. If all nodes are labeled with $\varepsilon$ and all edges with a number between 0 and 1, the resulting graph is called a *weighted graph*. Other special classes of graphs satisfying the definition above as well are *bounded-valence graphs* [Luks (1982)], *planar graphs* [Hopcroft and Wong (1974)], *rooted trees* [Pelillo *et al.* (1999)], and *hierarchical graphs* [Lu *et al.* (1991)].

The notion of a part of a graph is defined in equivalence to the subset relation in set theory.

**Definition 2.2 (Subgraph).** *Let $g_1 = (V_1, E_1, \mu_1, \nu_1)$ and $g_2 = (V_2, E_2, \mu_2, \nu_2)$ be graphs. Graph $g_1$ is a subgraph of $g_2$, written $g_1 \subseteq g_2$, if*

*(1)* $V_1 \subseteq V_2$ ,

*(2)* $E_1 = E_2 \cap V_1 \times V_1$ ,
*(3)* $\mu_1(u) = \mu_2(u)$ *for all* $u \in V_1$, *and*
*(4)* $\nu_1(e) = \nu_2(e)$ *for all* $e \in E_1$ .

A subgraph is obtained from a graph by removing some nodes (and their adjacent edges). The second condition in this definition is sometimes replaced by $E_1 \subseteq E_2$, and subgraphs satisfying the more stringent condition given above are then called *induced subgraphs*. An example is shown in Fig. 2.1. The segmented image in Fig. 2.1a is first transformed into the region adjacency graph in Fig. 2.1b by representing regions by nodes and the adjacency of regions by edges. Note that the numbers from 1 to 5 constitute the set of nodes, and node 1 represents the background region in the image. The graphs in Fig. 2.1c and 2.1d are induced and non-induced subgraphs, respectively, of the graph in Fig. 2.1b.

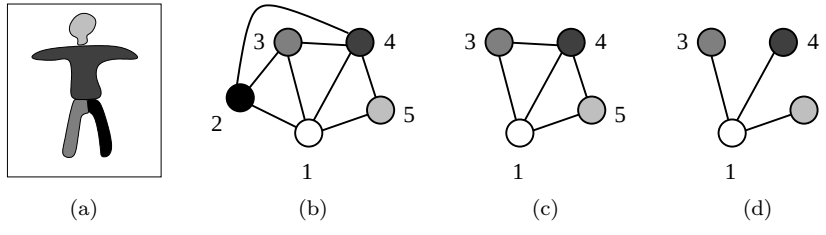



Fig. 2.1 (a) Segmented image, (b) region adjacency graph, (c) subgraph of (b), and (d) non-induced subgraph of (b)

## 2.2 Exact Graph Matching

Graph matching methods can roughly be split into two categories: methods for exact graph matching and methods for error-tolerant graph matching. In exact graph matching, the aim is to determine whether two graphs, or parts of two graphs, are identical in terms of structure and labels. Whereas it is trivial to decide if two vectors are identical, the same task for graphs is much more complex, since one cannot simply compare the sets of nodes and edges and the labeling functions. Generally, the nodes and edges of a graph cannot be ordered, unlike the components of a feature vector or the symbols of a string, which makes the computation of graph equality so demanding. The identity of two graphs is commonly established by defining a bijective function, termed graph isomorphism, mapping one graph to the

other.

**Definition 2.3 (Graph Isomorphism).** *Let* $g_1 = (V_1, E_1, \mu_1, \nu_1)$ *and* $g_2 = (V_2, E_2, \mu_2, \nu_2)$ *be graphs. A graph isomorphism is a bijective function* $f : V_1 \to V_2$ *satisfying*

*(1)* $\mu_1(u) = \mu_2(f(u))$ *for all nodes* $u \in V_1$ ,
*(2) for each edge* $e_1 = (u, v) \in E_1$*, there exists an edge* $e_2 = (f(u), f(v)) \in E_2$ *such that* $\nu_1(e_1) = \nu_2(e_2)$ ,
*(3) for each edge* $e_2 = (u, v) \in E_2$*, there exists an edge* $e_1 = (f^{-1}(u), f^{-1}(v)) \in E_1$ *such that* $\nu_1(e_1) = \nu_2(e_2)$ .

*Two graphs are called isomorphic if there exists an isomorphism between them.*

Isomorphic graphs are identical in terms of structure and labels. To establish a graph isomorphism, one has to find a function mapping each node of the first graph to a node of the second graph such that the edge structure is preserved and node and edge labels are consistent. The graph isomorphism relation satisfies the conditions of reflexivity, symmetry, and transitivity and can therefore be regarded as an equivalence relation on graphs.

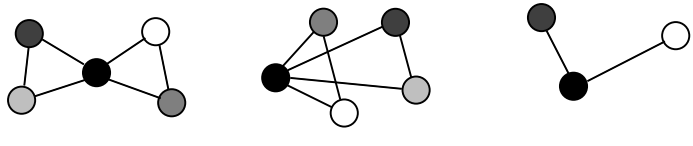


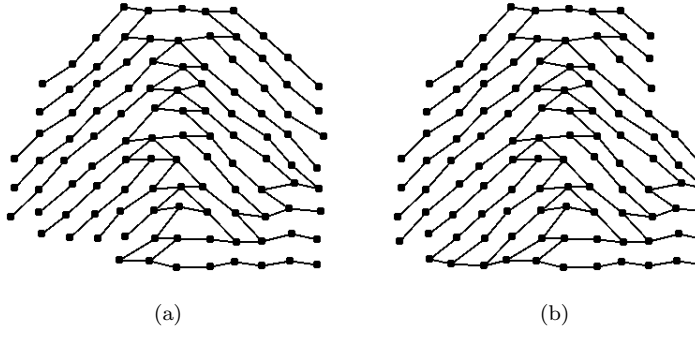

Fig. 2.2 Graph b) is isomorphic to (a), and graph (c) is isomorphic to a subgraph of (a)

A straightforward approach to determine whether two graphs are isomorphic or not is to consider all possible node-to-node correspondences in a tree search algorithm [Ullman (1976)]. The idea is to proceed by mapping nodes from the first graph to nodes from the second graph until either the edge structure of the two graphs does not match or node or edge labels are inconsistent with the mapping. If all nodes of one graph can be mapped to nodes of the other graph without violating structure and label constraints, the two graphs are isomorphic. In general, the computational complexity of this procedure is exponential in the number of nodes of the two graphs. Two isomorphic graphs are shown in Fig. 2.2.

Closely related to graph isomorphism is the concept of subgraph isomorphism. If graph isomorphism is regarded as a formal notion of graph equality, subgraph isomorphism can be seen as subgraph equality.

**Definition 2.4 (Subgraph Isomorphism).** *Let* $g_1 = (V_1, E_1, \mu_1, \nu_1)$ *and* $g_2 = (V_2, E_2, \mu_2, \nu_2)$ *be graphs. An injective function* $f : V_1 \to V_2$ *is called subgraph isomorphism from* $g_1$ *to* $g_2$ *if there exists a subgraph* $g \subseteq g_2$ *such that* $f$ *is a graph isomorphism between* $g_1$ *and* $g$*.*

A subgraph isomorphism exists between two graphs $g_1$ and $g_2$ if the larger graph $g_2$ can be turned into a graph that is isomorphic to $g_1$ by removing some nodes and edges. In other words, a subgraph isomorphism indicates that a smaller graph is contained in a larger graph. For an example, see the graphs in Fig. 2.2a and Fig. 2.2c. The procedure outlined above for detecting graph isomorphism can also be applied to subgraph isomorphism [Ullman (1976)].



(a) (b)

Fig. 2.3 Two similar graphs not related by a (sub)graph isomorphism

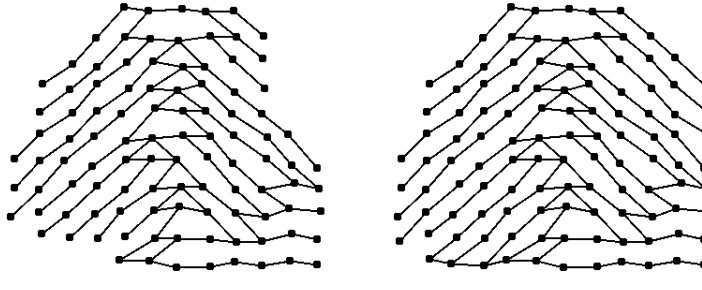
Matching graphs by means of graph isomorphism and subgraph isomorphism is limited in the sense that structure and labels of two graphs must be completely identical for a positive match, that is, for two graphs to be deemed similar. For example, consider the graphs illustrated in Fig. 2.3. Note that in this example each node is labeled with a two-dimensional attribute giving its position in the plane. It is clear that none of the two graphs is related to the other one by a subgraph isomorphism, since each of the two graphs contains nodes missing in the other graph. Therefore, if we use a graph matching paradigm based on (sub)graph isomorphism, the two graphs will be considered different. The observation that most nodes

and edges are identical in both graphs, which means that the two graphs are in fact very similar, is not accounted for at all in the computation of (sub)graph isomorphism. This obvious shortcoming of graph isomorphism and subgraph isomorphism leads to the formal definition of the largest common part of two graphs.

**Definition 2.5 (Maximum Common Subgraph, mcs).** *Let $g_1$ and $g_2$ be graphs. A graph $g$ is called common subgraph of $g_1$ and $g_2$ if there exist subgraph isomorphisms from $g$ to $g_1$ and from $g$ to $g_2$.*

*A common subgraph of $g_1$ and $g_2$ is called maximum common subgraph (mcs) if there exists no other common subgraph of $g_1$ and $g_2$ with more nodes than $g$.*

The maximum common subgraph of two graphs can be regarded as the intersection of two graphs, that is, the largest part of two graphs that is identical in terms of structure and labels. In general, the maximum common subgraph is not uniquely defined, that is, there may be more than one common subgraph with a maximum number of nodes. A standard approach to computing the maximum common subgraph of two graphs is related to the maximum clique problem [Levi (1972)]. The basic idea is to construct an association graph representing all node-to-node correspondences between the two graphs that do not violate the structure and labels of both graphs. A maximum clique in the association graph, that is, a maximum fully connected subgraph, is equivalent to a maximum common subgraph of the two original graphs. The only maximum common subgraph of the two graphs in Fig. 2.3 is illustrated in Fig. 2.4a.



(a) (b)

Fig. 2.4 (a) Maximum common subgraph and (b) minimum common supergraph of the two graphs in Fig. 2.3

Interpreting the maximum common subgraph as the intersection of two graphs, we can define a similar concept for the union of two graphs.

**Definition 2.6 (Minimum Common Supergraph, MCS).** *Let $g_1$ and $g_2$ be graphs. A graph $g$ is called common supergraph of $g_1$ and $g_2$ if there exist subgraph isomorphisms from $g_1$ to $g$ and from $g_2$ to $g$.*

*A common supergraph of $g_1$ and $g_2$ is called minimum common supergraph (MCS) if there exists no other common supergraph of $g_1$ and $g_2$ with fewer nodes than $g$.*

The minimum common supergraph of two graphs is a graph without unnecessary nodes containing both graphs as subgraphs. The computation of a minimum common supergraph can be reduced to a maximum common subgraph computation [Bunke *et al.* (2000)]. For an example, the minimum common supergraph of the two graphs in Fig. 2.3 is shown in Fig. 2.4b.

The notions of maximum common subgraph and minimum common supergraph can be used to define the similarity, or dissimilarity, of graphs: The larger the common part of two graphs, the higher their similarity. Formally, relating the size of the maximum common subgraph to the size of the larger one of the two graphs [Bunke and Shearer (1998)] leads to a distance metric on graphs, according to

$$d_1(g_1, g_2) = 1 - \frac{|mcs(g_1, g_2)|}{\max(|g_1|, |g_2|)} \quad . \tag{2.1}$$

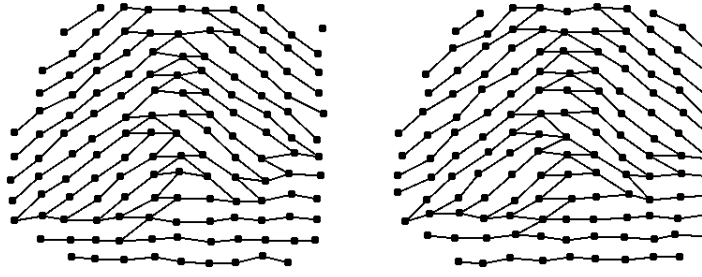
If two graphs are very similar, their maximum common subgraph will obviously be almost as large as the two graphs, and the ratio will therefore be close to 1. For two dissimilar graphs, the maximum common subgraph will be small, and the ratio will be close to 0. Clearly, the distance metric $d_1$ is confined to values between 0 and 1, and the more similar two graphs are, the lower is the corresponding distance value. In a related work, the size of the union of two graphs is used for normalization rather than the size of the larger one of the two graphs [Wallis *et al.* (2001)].

In another approach, the difference of the size of the minimum common supergraph and maximum common subgraph of two graphs is evaluated [Fernandez and Valiente (2001)], that is,

$$d_2(g_1, g_2) = |MCS(g_1, g_2)| - |mcs(g_1, g_2)| \quad . \tag{2.2}$$

For similar graphs, the maximum common subgraph will be almost as large as the minimum common supergraph, and the resulting distance value will therefore be small, whereas for dissimilar graphs, the two terms will be significantly different, resulting in larger distance values. It should be noted

that the two graph distance measures defined above constitute valid distance metrics [Bunke and Shearer (1998); Fernandez and Valiente (2001)], that is, they satisfy the conditions of positive definitness, symmetry, and the triangle inequality.



(a) (b)

Fig. 2.5 Two similar graphs without a single matching label

Such dissimilarity measures defined with respect to the maximum common subgraph of two graphs offer a certain amount of error-tolerance. In other words, two graphs need not be directly related in terms of a (sub)graph isomorphism to be sucessfully matched. Yet, for two graphs to be deemed similar, parts of the two graphs still need to be isomorphic. This means that for these distance measures to be small, two graphs must be identical to a large extent in terms of structure and labels. In practice, node and edge labels are used to describe properties of the underlying object represented by the graph, similar to feature vectors in statistical pattern recognition. These labels often describe non-discrete properties and can only be represented by continuous values. It is therefore advisable not to simply evaluate if two labels are equal or not, but rather measure how similar two labels are. For an illustration, consider the two graphs in Fig. 2.5, where each node is labeled with a position attribute. The node and edge structure of the two graphs is clearly identical, but none of the nodes have exactly the same position. From an objective point of view, the two graphs are very similar and obviously represent the same object. The maximum common subgraph of the two graphs, however, is the empty graph, so that the two graphs will be considered completely different by all graph matching methods presented so far.

### 2.3 Error-Tolerant Graph Matching

The main advantage of exact graph matching methods is their stringent definition and solid mathematical foundation. However, the severe shortcomings outlined above demonstrate that graph matching based on (sub)graph isomorphism is only applicable to a very small range of real-world problems. For this reason, a large number of error-tolerant, or inexact, graph matching methods have been proposed, dealing with a more general graph matching problem than the one formulated in terms of (sub)graph isomorphism.

The idea is to evaluate the similarity of two graphs in a broader sense that better reflects the intuitive understanding of graph similarity. In fact, error-tolerant graph matching methods need not be defined by means of a graph similarity measure at all, but can be formulated in entirely different terms, as will be clear in the examples given below. In the ideal case, two graphs representing the same class of objects are identical. That is, a graph extraction process turning objects into graphs should always result in exactly the same graph for objects of the same class. In practice, of course, the graph extraction process suffers from noise. In graph matching, error-tolerance means that the matching algorithm is to a certain extent able to correct errors, that is, the difference between the actually extracted graph and the ideal graph. For instance, an error-tolerant method is expected to discover that the two graphs in Fig. 2.5 are highly similar. In the remainder of this section, a number of methods proposed for error-tolerant graph matching are briefly reviewed.

One class of error-tolerant graph matching methods employs *relaxation labeling techniques* for structural matching. In [Christmas *et al.* (1995); Wilson and Hancock (1997)], the idea is to formulate the graph matching problem as a labeling problem — nodes of an input graph are to be assigned symbols specifying a matching node of a model graph. Gaussian probability distributions are used to model compatilibity coefficients measuring how suitable a given node labeling is. The labeling is then refined in an iterative procedure until a sufficiently accurate labeling, that is, matching of two graphs, is obtained. This method allows for an error-tolerant matching of graphs, since one can explicitly model spurios input graph nodes that have no corresponding node in the model graph, missing nodes of a model graph due to an incomplete model, and of course the difference of non-identical labels. On the other hand, edge information is presumed to be independent, and edge labels are required to convey metric measurements.

The use of *artificial neural networks* for graph matching has also been

proposed. In [Suganthan *et al.* (1995b)] the authors perform graph matching by means of Hopfield neural networks. Hopfield networks consist of a set of neurons connected by synapses such that, upon activation of the network, the neuron output is fed back into the network. The objective of Hopfield network training is to minimize a given energy criterion in the process of an iterative learning procedure. The trained network can then be used to recognize input objects. In the graph matching context, the idea is to model the matching of two graphs by an energy optimization formulation. Again, compatibility coefficients are used to evaluate whether two nodes or edges constitute a successful match. In [Suganthan *et al.* (1995a)], the same authors note that the optimization behavior of the approach outlined above is often insufficiently stable and propose an approach based on the Potts MFT network instead. Another kind of a neural network for graph matching is proposed in [Barsi (2003)]. The idea is to use a Kohonen map, or self-organizing map, to adapt a given model graph according to an input scene. The learning is effected by means of an unsupervised self-organization procedure based on estimating the distribution of input patterns in a layer of competitive neurons. This matching method admits a large amount of error-tolerance, but its applicability is clearly restricted to a few particular applications and cannot be used for the general graph matching problem considered in this book. Recursive neural networks have been used for error-tolerant graph matching as well [Frasconi *et al.* (1998); Yao *et al.* (2003)]. Here, each node is represented by a feature vector representing all nodes that can be reached from this node. The characterization of the training graphs and the classification of input graphs is then performed by means of recursive neural networks. This method is applicable to arbitrarily labeled graphs, but requires graphs to belong to the class of directed acyclic graphs containing a root node from which every other node can be reached. Further examples of graph matching methods based on artificial neural networks can be found in [Xu and Oja (1990); Feng *et al.* (1994); Schädler and Wysotzki (1999); Jain and Wysotzki (2003, 2004)].

Another class of graph matching methods is based on *genetic algorithms* [Cross *et al.* (1997); Wang *et al.* (1997); Singh *et al.* (1997); Suganthan (2002)]. The main advantage of genetic algorithms is that they offer an efficient way to cope with huge search spaces. Genetic algorithms require the graph matching problem to be formalized in terms of states (chromosomes) and their performance (fitness function). The search space is explored in a random fashion, but the algorithm tends to favor promising, well-performing candidates. Two serious shortcomings of genetic algorithms are

that the choice of initial state strongly affects the resulting performance, and that the algorithm is generally nondeterministic. On the other hand, the well-defined formalism of genetic algorithms often makes it easy to address difficult optimization problems that cannot be solved efficiently in an exact manner. For instance, in [Singh *et al.* (1997)], the matching of two graphs is explicitly encoded by a set of node-to-node correspondences of two graphs. The quality of such a matching can then readily be computed by means of an underlying distortion model (in a way similar to the edit distance concept presented in the next chapter) by simply accumulating individual costs. Exact algorithms solving the same optimization problem belong to the class of NP-complete problems [Garey and Johnson (1979)].

Quite a number of papers are concerned with the matching of graphs based on their *spectral decomposition* [Umeyama (1988); Luo *et al.* (2003); Caelli and Kosinov (2004); Wilson and Hancock (2004); Robles-Kelly and Hancock (2005); Shokoufandeh *et al.* (2005)]. The basic idea is to represent graphs by the eigendecomposition of their adjacency matrix or their Laplacian matrix. It turns out that the resulting representation exhibits interesting properties and can further be exploited for graph matching. In [Luo *et al.* (2003)], for instance, features derived from the eigendecomposition of graphs are studied. This feature extraction defines an embedding of graphs into vector spaces, which makes it possible to apply statistical methods for pattern analysis to graphs. In [Wilson and Hancock (2004)], the authors suggest to apply an error-tolerant string matching algorithm to the eigensystem of graphs to infer distances of graphs. These distances are then used to embed the graphs into a vector space by means of multidimensional scaling. In a different approach [Robles-Kelly and Hancock (2005)], graphs are turned into strings according to their leading eigenvectors. The matching of graphs can then be effected by means of string matching, which is faster to compute and offers a different interpretation than graph matching. However, in comparison to other methods, it seems that spectral methods are not fully able to cope with larger amounts of noise. The main problem is that the eigendecomposition is very sensitive towards structural errors, such as missing nodes. Also, most spectral methods are only applicable to unlabeled graphs or labeled graphs with severely constrained label alphabets. Therefore, spectral methods are only to a limited extent applicable to graphs extracted from real-world data.

A method that is recognized as one of the most flexible and universal error-tolerant matching paradigms is *graph edit distance*. Graph edit distance is not constrained to particular classes of graphs and can be tailored

to specific applications. The main restriction is that the computation of edit distance is inefficient, particularly for large graphs. The next chapter is devoted to a detailed description of graph edit distance, as graph edit distance is the basic graph matching method employed in this book.

*Kernel machines* are a novel class of algorithms for pattern analysis and recognition. Several well-known algorithms can be formulated as special cases of kernel machines. Kernel machines are not specifically aimed at graph matching, but the kernel framework can be applied to graphs in a very natural way. The development of graph kernels based on the edit distance concept is the main topic in this book. For a brief discussion of kernel functions applicable to graph matching, see Sec. 5.2.

A large number of other models have been proposed for graph matching, among them methods based on the Expectation Maximization algorithm [Luo and Hancock (2001)], graduated assignment [Gold and Rangarajan (1996)], approximate least-squares and interpolation theory algorithms [van Wyk and Clark (2000); van Wyk *et al.* (2003)], random walks in graphs [Robles-Kelly and Hancock (2004); Gori *et al.* (2005)], and random graphs [Wong and You (1985); Sanfeliu *et al.* (2004)], to name a few. For an extensive review of graph matching methods and applications, refer to [Conte *et al.* (2004)].

This page intentionally left blank

## Chapter 3

# Graph Edit Distance

In the previous chapter, a number of graph matching methods have been mentioned. Models for exact graph matching, such as methods based on subgraph isomorphism, are clearly too constrained to provide for a general purpose graph matching system. The requirement that a significant number of node and edge labels in two graphs must be identical for a positive match is not realistic in applications on graphs extracted from real-world data. Error-tolerant, or inexact, graph matching methods offer a wider range of models for structural matching. If the area of application is confined to a certain class of graphs, it is often possible to come up with sophisticated methods exploiting particular properties of the underlying graphs. For instance, if all objects of interest can be modeled with sufficient accuracy by unlabeled graphs consisting of a constant number of nodes, methods based on the spectral decomposition of graphs may be suitable and might provide an efficient solution to the graph matching problem (see the previous chapter). Yet, for unconstrained graph matching, it is difficult to define models that reflect the intuitive understanding of the similarity of graphs.

One of the most widely used methods for error-tolerant graph matching is graph edit distance. The concept of edit distance has originally been proposed in the context of the string-to-string correction problem [Levenshtein (1966); Wagner and Fischer (1974)]. The basic idea is to define a distortion model describing how two patterns differ from each other. The aim is to derive a pattern dissimilarity measure from the number of distortions one has to apply to transform one pattern into the other. Because of the flexibility of the underlying distortion model and its associated penalty costs, the edit distance approach is considered very powerful and is applicable to various problems. Accordingly, the edit distance concept has been transferred from strings to trees [Selkow (1977)] and eventually to graphs [Bunke

and Allermann (1983); Sanfeliu and Fu (1983); Tsai and Fu (1979); Eshera and Fu (1984)]. The main advantage over other graph matching methods is that graph edit distance is defined for arbitrary graphs and is particularly flexible due to its ability to cope with any kind of structural error and any type of node and edge labels. On the other hand, its exponential time and space complexity make the computation of exact edit distance feasible for small graphs only.

This book addresses the issue of matching arbitrarily structured and arbitrarily labeled graphs. The proposed novel graph matching methods have in common that they are all related to graph edit distance. In this chapter, we give a formal definition of graph edit distance and present a standard algorithm for its computation. We then proceed by proposing an efficient approximate algorithm for edit distance applicable to an important class of graphs, and finally present an application of edit distance to the data-level fusion of graphs.

## 3.1 Definition

The basic idea of graph edit distance is to define the dissimilarity of two graphs by the minimum amount of distortion that is required to transform one graph into the other [Bunke and Allermann (1983); Sanfeliu and Fu (1983); Tsai and Fu (1979); Eshera and Fu (1984)]. In a first step, the underlying distortion model needs to be defined. A standard set of distortion operations, or edit operations, consists of an insertion, a deletion, and a substitution operation for both nodes and edges. Applying an edge deletion operation to a graph, for instance, is equivalent to removing an edge from the graph, and substituting a node with another node is equivalent to changing the label of the node. Note that other edit operations are useful for certain applications as well, such as node splitting and merging [Ambauen *et al.* (2003); Gomila and Meyer (2001); Cesar *et al.* (2002)], but will not be considered in the remainder of this book. The deletion of a node $u$ is denoted by $u \to \varepsilon$, the insertion of a node $u$ by $\varepsilon \to u$, and the substitution of a node $u$ by a node $v$ is denoted by $u \to v$ (and analogously for edges).

For every pair of graphs, there exists a sequence of edit operations, or edit path, transforming one graph into the other. An example of an edit path between two graphs (consisting of three edge deletions, a node deletion, a node insertion, two edge insertions, and a node substitution) is illustrated in Fig. 3.1. For arbitrary graphs, a valid edit path can always

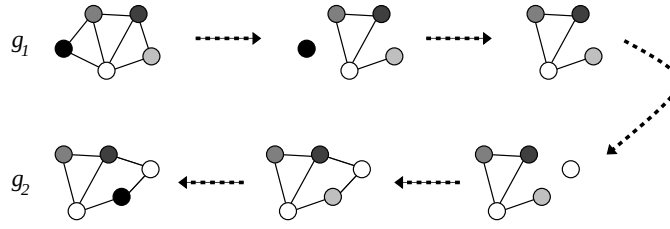



Fig. 3.1 Example edit path from $g_1$ to $g_2$

be constructed by first removing all nodes and edges from the first graph and then inserting all nodes and edges of the second graph. However, this trivial edit path does not tell us anything about whether or not the two graphs are structurally similar. More interesting are those edit paths that map a number of nodes and edges from one graph onto nodes and edges of the other graph. Substitutions of nodes and edges can be regarded as positive matches in the sense that substitutions identify the common part of two graphs. Insertions and deletions, on the other hand, are used to model nodes and edges that cannot be matched to any node or edge of the other graph. Hence, every edit path between two graphs can be understood as a model describing which nodes and edges of a graph can successfully be matched to nodes and edges of another graph.

Accordingly, the edit path that best represents the matching of two graphs is used to define their similarity. The optimal edit path is the one that maps, in a reasonable way, a large part of one graph onto the other graph. To evaluate in quantitative terms which edit path is best, edit cost functions are introduced. The idea is to assign a penalty cost to each edit operation, reflecting how strong the associated distortion is. For instance, changing the value of a node label from 0.8 to 0.9 should usually have lower costs than changing the label from 0.8 to 0.1. At the same time, insertions and deletions should have higher costs than moderate substitutions. The overall objective is to use edit costs that tend to favor weak substitutions over strong substitutions and over insertions and deletions. For graphs intuitively considered similar, there should consequently exist an edit path between them with cheap edit operations only. Conversely, for dissimilar graphs, all edit paths will have high costs, since each edit path will contain at least some edit operations associated with strong distortions. Although it is easy to outline such general rules for edit costs, the definition of actual cost functions for practical applications turns out to be quite difficult and

requires careful examination of the underlying graph representations and the meaning of node and edge labels. Section 3.2 is concerned in greater detail with the definition of edit costs.

Given a set of edit operations and an edit cost function, the dissimilarity of two graphs is defined by the minimum cost edit path that transforms one graph into the other.

**Definition 3.1 (Graph Edit Distance).** *Let* $g_1 = (V_1, E_1, \mu_1, \nu_1)$ *and* $g_2 = (V_2, E_2, \mu_2, \nu_2)$ *be graphs. The graph edit distance of* $g_1$ *and* $g_2$ *is defined by*

$$d(g_1, g_2) = \min_{\substack{(e_1,\ldots,e_k) \\ \in \mathcal{P}(g_1,g_2)}} \sum_{i=1}^{k} c(e_i) \ , \tag{3.1}$$

*where* $\mathcal{P}(g_1, g_2)$ *denotes the set of edit paths transforming* $g_1$ *into* $g_2$*, and* $c$ *denotes the edit cost function.*

The edit distance is defined by the edit path with minimum accumulated edit operation costs. That is, in the context of the distortion and penalty cost model introduced above, the edit distance determines among all sets of distortions transforming one graph into the other the one with the weakest distortions. For similar graphs, it will be sufficient to apply edit operations with low costs to turn one graph into the other, whereas for dissimilar graphs, even the minimum cost edit path will contain some strong distortions.

An illustration of this idea is provided in Fig. 3.2 and Fig. 3.3. According to the argumentation in the previous chapter, exact graph matching methods are considered insufficiently flexible to successfully match the two graphs in Fig. 2.5a,b. But if edit distance is used to match the two graphs, the similar structure of the two graphs is readily recognized. In Fig. 3.2, the node substitutions of the optimal edit path between the two graphs are visualized. If two graphs are even less similar, such as those in Fig. 3.3, the edit distance is still capable of identifying how one graph is mapped onto the other graph in an optimal way. Note that in these illustrations thin lines connecting two gray nodes represent node substitutions, and white nodes represent nodes to be deleted or inserted.

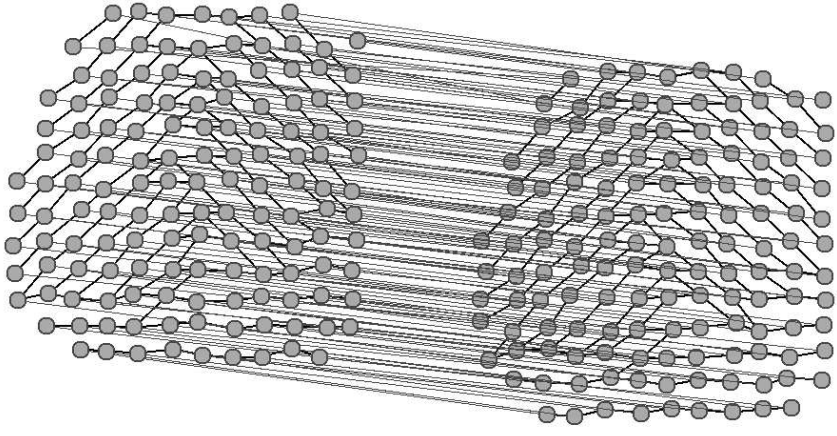


Fig. 3.2 Matching the graph in Fig. 2.5a with the graph in Fig. 2.5b

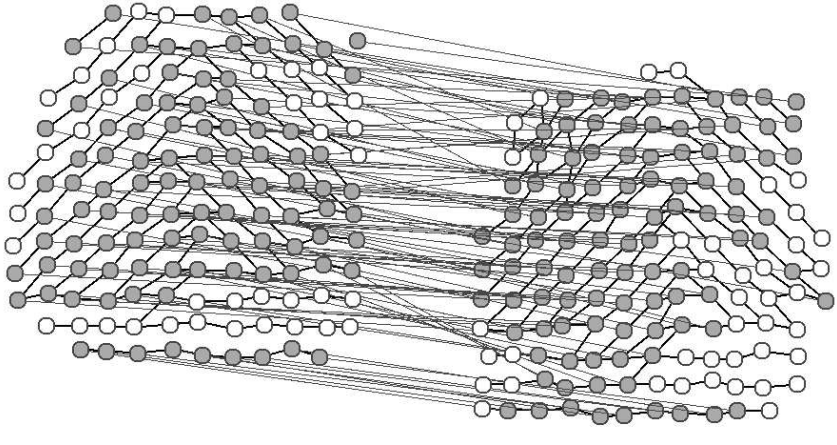


Fig. 3.3 Matching the graph in Fig. 2.5a with a different graph

## 3.2 Edit Cost Functions

Edit operation costs specify if it is less expensive to apply a node deletion and a node insertion instead of a node substitution, and if the principal emphasis is on node operations or edge operations. That is, in order to define edit costs in an appropriate way, the underlying graph representation must be taken into account. For some representations it may be crucial whether or not two nodes are linked by an edge, for instance in a network monitoring application, where a missing edge represents a broken physical link. For other representations, nodes and their labels may be more important than edges. To obtain a suitable graph edit distance measure, it is therefore of key importance to define edit costs such that the structural variation of graphs is modeled in an application-specific manner.

### 3.2.1 *Conditions on Edit Costs*

In Def. 3.1, the edit distance of two graphs $g_1$ and $g_2$ is defined over the set of edit paths $\mathcal{P}(g_1, g_2)$ from $g_1$ to $g_2$. Theoretically, this set of edit paths is infinite, since from a valid edit path one can construct an infinite number of edit paths by additionally inserting and deleting a single node arbitrarily often. If the cost function satisfies a few weak conditions, however, it is sufficient to evaluate a finite set of edit paths to find one with minimum cost among all edit paths. First, we require edit costs to be non-negative,

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

Since edit costs represent penalty costs of distortions associated with edit operations, this condition is certainly reasonable. The other condition states that eliminating unnecessary substitutions from edit paths does not increase the corresponding sum of edit operation costs,

$$\begin{aligned} c(u \to w) &\leq c(u \to v) + c(v \to w), \\ c(u \to \varepsilon) &\leq c(u \to v) + c(v \to \varepsilon), \\ c(\varepsilon \to v) &\leq c(\varepsilon \to u) + c(u \to v), \quad \text{for all substitutions } u \to v \\ &\qquad\qquad\qquad\qquad\quad \text{and all nodes } w. \end{aligned}$$

For instance, instead of substituting $u$ with $v$ and then substituting $v$ with $w$ (line 1), one can safely replace the two right-hand side operations by the operation $u \to w$ on the left and will never miss out a minimum cost edit path. Note that the analogous triangle inequalities must also be satisfied for edge operations. Hence, each edit path in $\mathcal{P}(g_1, g_2)$ containing

unnecessary substitutions $u \rightarrow v$ according to the conditions above, can be replaced by a shorter edit path that is possibly less, and definitely not more, expensive. After removing irrelevant edit paths by refined ones such that no unnecessary substitutions are present in the set of edit paths $\mathcal{P}(g_1, g_2)$, one finally has to eliminate from the remaining ones those edit paths that contain unnecessary insertions of nodes followed by their deletion, which is reasonable due to the non-negativity of edit costs.

Provided that these conditions are satisfied, it is clear that only the deletions, insertions, and substitutions of nodes and edges of the two involved graphs have to be considered. Since the objective is to edit the first graph $g_1 = (V_1, E_1, \mu_1, \nu_1)$ into the second graph $g_2 = (V_2, E_2, \mu_2, \nu_2)$, we are only interested in how to map nodes and edges from $g_1$ to $g_2$. The conditions stated above guarantee that including any other edit operation in an edit path will never lead to lower overall edit costs. We therefore proceed by taking into account only the $|V_1|$ deletions of nodes from $g_1$, the $|V_2|$ insertions of nodes from $g_2$, the $|V_1| \cdot |V_2|$ substitutions of nodes from $g_1$ by nodes from $g_2$, and the $|E_1| + |E_2| + |E_1| \cdot |E_2|$ analogous edge operations. Hence, in Def. 3.1, the infinite set of edit paths $\mathcal{P}(g_1, g_2)$ can be reduced to the finite set of edit paths containing edit operations of this kind only. In the remainder of this book, only edit cost functions satisfying the conditions stated above will be considered.

Note that the resulting graph edit distance measure need not be metric. For instance, the edit distance is not symmetric in the general case, that is $d(g_1, g_2) \neq d(g_2, g_1)$ may hold true. However, the graph edit distance can be turned into a metric by requiring the underlying cost functions on edit operations to satisfy the metric conditions of positive definiteness, symmetry, and the triangle inequality [Bunke and Allermann (1983)].

### 3.2.2 *Examples of Edit Costs*

For numerical node and edge labels, it is common to measure the dissimilarity of labels by the Euclidean distance and assign constant costs to insertions and deletions. For two graphs $g_1 = (V_1, E_1, \mu_1, \nu_1)$ and $g_2 = (V_2, E_2, \mu_2, \nu_2)$ and non-negative parameters $\alpha_{node}, \alpha_{edge}, \gamma_{node}, \gamma_{edge} \in \mathbb{R}^+ \cup \{0\}$, this cost function is defined for all nodes $u \in V_1, v \in V_2$ and

for all edges $p \in E_1, q \in E_2$ by

$$\textit{Euclidean cost function} \qquad \begin{aligned} c(u \to \varepsilon) &= \gamma_{node} \\ c(\varepsilon \to v) &= \gamma_{node} \\ c(u \to v) &= \alpha_{node} \cdot \|\mu_1(u) - \mu_2(v)\| \\ c(p \to \varepsilon) &= \gamma_{edge} \\ c(\varepsilon \to q) &= \gamma_{edge} \\ c(p \to q) &= \alpha_{edge} \cdot \|\nu_1(p) - \nu_2(q)\| \ . \end{aligned} \tag{3.2}$$

Note that edit costs are defined with respect to labels of nodes and edges, rather than the node and edge identifiers themselves. The cost function above defines substitution costs proportional to the Euclidean distance of the respective labels. The basic idea is that the further away two labels are, the stronger is the distortion associated with the corresponding substitution. Insertions and deletions are assigned constant costs, so that any node substitution having higher costs than a fixed threshold $\gamma_{node} + \gamma_{node}$ will be replaced by a node deletion and insertion operation. This behavior reflects the intuitive understanding that the substitution of nodes and edges should be favored to a certain degree over insertions and deletions. The edit cost conditions from the previous section are obviously fulfilled, and if each of the four parameters is greater than zero, we obtain a metric graph edit distance. The main advantage of the cost function above is that its definition is very straightforward.

Even though the Euclidean cost function defined above is flexible enough for many problems, more complex edit cost models have been developed. The method proposed in [Neuhaus and Bunke (2004, 2007)] is based on a probabilistic estimation of edit operation densities. This method is interesting because it allows us to learn edit costs from a labeled sample set of graphs, such that graphs from the same class tend to be closer to one another than graphs from different classes. The cost function is also more powerful than the simple model based on Euclidean distance in that edit operation costs depend on the actual label values. That is, for a substitution, not only the distance of two labels is taken into account, but also where the source label and the target label are actually located in the label space. Similarly, edit operation costs of insertions and deletions are not constant, but depend on the label of the node or edge to be inserted or deleted. In another approach [Neuhaus and Bunke (2005)] based on a similar idea, the label space is deformed by means of self-organizing maps to provide for a more complex label dissimilarity measure than the Euclidean distance.

### 3.3 Exact Algorithm

Using Def. 3.1, the problem of evaluating the structural similarity of graphs is turned into the problem of finding a minimum-cost edit path between graphs. The computation of an optimal edit path is usually performed by means of a search tree algorithm [Bunke and Allermann (1983); Tsai and Fu (1979)]. The idea is to represent the optimal edit path problem in a search tree, such that an optimal path through the search tree is equivalent to an edit path solution. Formally, inner nodes of the search tree correspond to partial edit paths and leaf nodes to complete edit paths. The search tree is constructed by processing one node from the first graph after the other. For each node, all possible edit operations are considered, and newly created partial solutions are inserted into the tree. In practice, the A* best-first search algorithm [Hart *et al.* (1968)] is often used for tree traversal. That is, from the current set of edit path candidates, the most promising one, the one with minimum cost, is chosen at each step of the tree traversal. To speed up the search, a heuristic function estimating the expected cost of the best route from the root node through the current node of the search tree to a leaf node can be used. If the estimated costs are always lower than the real costs, the search procedure is known to be admissible, that is, an optimal path from the root node to a leaf node is guaranteed to be found [Hart *et al.* (1968)]. For the graph edit distance problem, a number of heuristic functions estimating the expected costs of the unprocessed parts of two graphs have been proposed [Bunke and Allermann (1983); Tsai and Fu (1979)].

A best-first search algorithm for the computation of graph edit distance is given in Alg. 3.1. The algorithm is formulated in terms of node edit operations only, but the mapping of edges can easily be inferred from the mapping of their adjacent nodes. Note that the nodes of the first graph are processed in the order $(u_1, u_2, \ldots)$. The set OPEN contains all partial edit paths constructed so far. In each step, the next unprocessed node $u_{k+1}$ of the first graph is selected and tentatively substituted by all unprocessed nodes of the second graph (line 11) as well as deleted (line 12), thus producing a number of sucessor nodes in the search tree. If all nodes of the first graph have been processed, the remaining nodes of the second graph are inserted into the graph in a single step (line 14). The procedure always selects the most promising partial edit path in OPEN (line 5) and terminates as soon as a complete edit path is encountered (line 7).

Unlike exact graph matching methods, the edit distance allows every

**Algorithm 3.1** Edit distance algorithm

Input: Non-empty graphs $g_1 = (V_1, E_1, \mu_1, \nu_1)$ and $g_2 = (V_2, E_2, \mu_2, \nu_2)$, where $V_1 = \{u_1, \ldots, u_{|V_1|}\}$ and $V_2 = \{v_1, \ldots, v_{|V_2|}\}$

Output: A minimum-cost edit path from $g_1$ to $g_2$
e.g. $p = \{u_1 \rightarrow v_3, u_2 \rightarrow \varepsilon, \ldots, \varepsilon \rightarrow v_2\}$

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: Retrieve minimum-cost partial edit path $p_{min}$ from OPEN
6: **if** $p_{min}$ is a complete edit path **then**
7: Return $p_{min}$ as solution
8: **else**
9: Let $p_{min} = \{u_1 \rightarrow v_{i_1}, \ldots, u_k \rightarrow v_{i_k}\}$
10: **if** $k < |V_1|$ **then**
11: For each $w \in V_2 \setminus \{v_{i_1}, \ldots, v_{i_k}\}$, insert $p_{min} \cup \{u_{k+1} \rightarrow w\}$ into OPEN
12: Insert $p_{min} \cup \{u_{k+1} \rightarrow \varepsilon\}$ into OPEN
13: **else**
14: Insert $p_{min} \cup \bigcup_{w \in V_2 \setminus \{v_{i_1}, \ldots, v_{i_k}\}} \{\varepsilon \rightarrow w\}$ into OPEN
15: **end if**
16: **end if**
17: **end loop**

node of a graph to be matched to every node of another graph. This flexibility makes graph edit distance particularly appropriate for noisy data, but on the other hand increases the computational complexity in contrast to simpler graph matching models. The time and space complexity of graph edit distance is exponential in the number of nodes of the two involved graphs. This means that for large graphs the computation of edit distance is intractable. In practice, it turns out that edit distance can typically be computed for graphs with up to 12 nodes. Addressing this problem, the next section describes an efficient approximate edit distance algorithm for a common class of graphs.

## 3.4 Efficient Approximate Algorithm

The edit distance algorithm in Alg. 3.1 is computationally inefficient mainly because the number of edit paths to be evaluated grows exponentially for linearly growing graphs. Therefore, an approximate method is proposed in the following limiting considerations to a small number of edit paths based on a fast matching algorithm for small subgraphs. For the approximate algorithm to be applicable, the only requirement the graphs under consideration have to satisfy is that nodes must be endowed with a label giving its position in the plane.

### 3.4.1 *Algorithm*

First, the neighborhood of a node in a graph is defined, and then algorithms for the efficient local neighborhood matching and the approximate edit distance computation are presented.

**Definition 3.2 (Neighborhood Subgraph).** *Given a graph* $g = (V, E, \mu, \nu)$ *and a node* $u \in V$*, the neighborhood subgraph of* $u$ *in* $g$*, denoted by* $g_{|u}$*, is defined as the subgraph* $g_{|u} = (V_u, E_u, \mu_u, \nu_u) \subseteq g$*, where*

$$\begin{aligned} V_u &= \{u\} \cup \{v \in V : (u, v) \in E \text{ or } (v, u) \in E\} \\ E_u &= E \cap (V_u \times V_u) \\ \mu_u &= \mu|_{V_u} \\ \nu_u &= \nu|_{E_u} \quad . \end{aligned}$$

The neighborhood subgraph of a node $u$ in a graph is the subgraph consisting of the node $u$, all nodes connected to $u$, and all edges between these nodes. The matching of neighborhoods will be used for the approximate edit distance algorithm. An illustration of a neighborhood subgraph is provided in Fig. 3.4. The key idea of the approximate algorithm is to avoid optimizing a global condition, that is, finding a globally optimal edit path, but rather, in an iterative process, match more and more parts of two graphs by locally matching neighborhood subgraphs. For this reason, an efficient matching algorithm for neighborhood subgraphs is needed.

A neighborhood subgraph consists of a center node, a set of adjacent nodes, and edges between these nodes. If we assume that each node is assigned a two-dimensional label representing its position, the set of adjacent nodes can be considered an ordered sequence of nodes. To obtain such a sequence from a neighborhood graph, we start at a randomly chosen adjacent

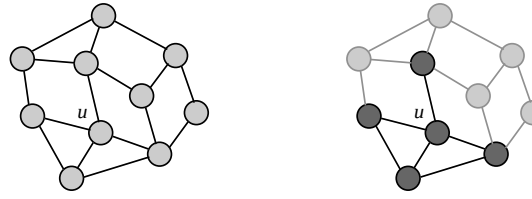



Fig. 3.4 (a) Graph and (b) marked neighborhood subgraph of $u$

node and traverse all nodes around the center node in a clockwise manner. In Fig. 3.5, the extraction of node sequences from neighborhood graphs is illustrated. Instead of regarding a neighborhood as a graph to be matched, we represent a neighborhood as an ordered sequence of nodes and match two neighborhood subgraphs by finding an optimal node alignment of the two sequences. In contrast to a full-fledged matching of two graphs, we only consider those edit paths that preserve the ordering of the nodes around the center node. Finding an optimal node alignment is effected by means of a cyclic string matching algorithm [Bunke and Bühler (1993); Lladós *et al.* (2001); Peris and Marzal (2002); Mollineda *et al.* (2002)] based on string edit distance. The edit distance of two strings can be computed by constructing an edit cost matrix and searching a minimum cost path through this matrix [Wagner and Fischer (1974)]. Dynamic programming is usually used for this purpose. The standard string edit distance algorithm can be extended to cyclic strings, that is, ordered sequences of symbols without explicit starting and ending symbol [Bunke and Bühler (1993)]. The cyclic string edit distance problem is less complex than the graph edit distance problem and can be solved in quadratic time and space (in terms of the length of strings). For our graph matching task — the alignment of two sequences of nodes — the choice of first symbol of the node sequences does not matter, since a cyclic string edit distance algorithm is employed. The string edit costs for insertions, deletions, and substitutions are simply derived from the corresponding node and edge edit operation costs. Hence, although a string matching algorithm is used for the alignment of ordered sequences of nodes, the resulting minimum-cost edit path reflects the optimal way to transform one neighborhood graph into the other one, taking all their nodes and edges into account. An illustration of the neighborhood matching is provided in Fig. 3.5, where nodes are marked in different colors for reasons of readability. Note that the cyclic string edit distance

algorithm does not actually perform the stepwise string matching shown in the illustration, but computes all edit distances between cyclically shifted instances of the two strings in a single run.

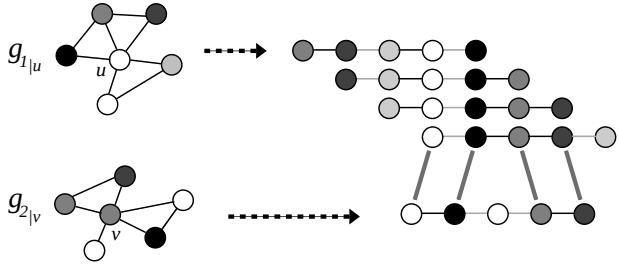



Fig. 3.5 Example matching of two neighborhoods, resulting in four substituted nodes, one deleted node, and one inserted node

The approximate graph edit distance algorithm is outlined in Alg. 3.2. The solution of the approximate algorithm, a suboptimal edit path $p$ between the two input graphs, is constructed in a stepwise manner. At the beginning, a seed substitution is selected (line 3). In some cases it may be possible to infer a node from the first graph and a node from the second graph that are known to correspond to each other based on the graph representation. Otherwise, a tentative graph matching process can be used to generate a number of promising candidates. In the next step, the neighborhood subgraphs defined by the two nodes of the seed substitution are matched (lines 6–8). The result of the neighborhood matching, an edit path between the two neighborhood subgraphs, is then added to the solution edit path $p$ (line 10). Note that the approximate algorithm is a greedy algorithm, that is, edit operations are only added, but never removed from the solution $p$. Finally, all node substitutions in the edit path resulting from the neighborhood matching are added to the first-in-first-out queue OPEN (line 11). The same matching steps are then performed for the next substitution in OPEN instead of the seed substitution. The neighborhood matching is carried out with respect to substituted nodes in $p$. That is, if a node already substituted in a previous step is to be processed in a neighborhood matching, the substitution from the previous step is preserved. Eventually, when all node substitutions in OPEN have been processed, the remaining nodes from the first graph are deleted and the remaining nodes from the second graph are inserted (lines 13–18). The resulting edit path is the approximate solution. This algorithm in its basic form only reaches

the connected part of a graph, but can be extended to deal with unconnected graphs as well. A possible solution to this problem is to run the approximate algorithm on the isolated parts of a graph and evaluate the best combination of matchings of isolated parts.

**Algorithm 3.2** Approximate edit distance algorithm

Input: Non-empty graphs $g_1 = (V_1, E_1, \mu_1, \nu_1)$ and $g_2 = (V_2, E_2, \mu_2, \nu_2)$, where $V_1 = \{u_1, \ldots, u_{|V_1|}\}$ and $V_2 = \{v_1, \ldots, v_{|V_2|}\}$

Output: A suboptimal edit path from $g_1$ to $g_2$ e.g. $p = \{u_1 \to v_3, u_2 \to \varepsilon, \ldots, \varepsilon \to v_2\}$

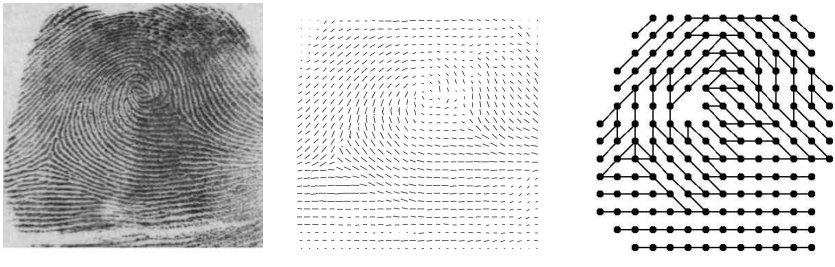
1: Initialize $p$ to the empty set
2: Initialize OPEN to the empty first-in-first-out queue
3: Insert the seed substitution $u_{i_0} \to v_{i_0}$ into OPEN and into $p$
4: **while** OPEN is not empty **do**
5: Fetch the next substitution $u \to v$ from OPEN
6: Extract neighborhood subgraphs $g_{1|u}$ from $g_1$ and $g_{2|v}$ from $g_2$
7: Represent $g_{1|u}$ and $g_{2|v}$ by ordered sequences of nodes
8: Align the two neighborhood node sequences by cyclic string matching
9: Let $q$ be the resulting optimal edit path between $g_{1|u}$ and $g_{2|v}$
10: Insert all edit operations from $q$ into $p$
11: Insert all node substitutions occurring in $q$ into OPEN
12: **end while**
13: **for** each unprocessed node $u_j$ of $g_1$ **do**
14: Insert $u_j \to \varepsilon$ into $p$
15: **end for**
16: **for** each unprocessed node $v_j$ of $g_2$ **do**
17: Insert $\varepsilon \to v_j$ into $p$
18: **end for**
19: Return the suboptimal edit path $p$ as solution

If we consider graphs with a bounded valence of $v$ only, the alignment of nodes takes $O(v^2)$. The approximate edit distance algorithm terminates after $O(n)$ iterations, where $n$ denotes the number of nodes in the graphs, resulting in an overall theoretical computational complexity of $O(nv^2)$. The computational complexity of the cyclic string matching algorithm can further be reduced by preserving node substitutions that have been established in previous iterations. To this end, the edit cost of a node substitution $u \to u'$ are set to zero if $u \to u'$ has occurred previously, to infinity if a sub-

stitution $u \rightarrow v'$ or $v \rightarrow u'$ with $u \neq v$ and $u' \neq v'$ has occurred previously, and to graph edit operation costs $c(u \rightarrow u')$ in all other cases. These conditions guarantee that newly added edit operations will never violate the current edit path. In view of these considerations, the approximate edit distance algorithm is found to be clearly less computationally expensive than the standard edit distance algorithm.

### 3.4.2 *Experimental Results*

The result of the approximate edit distance algorithm outlined above is a valid edit path between two graphs. Hence, it is clear that the approximate edit distance constitutes an upper bound of the exact edit distance, since the exact edit distance is derived from the edit path with minimum cost. The key question that needs to be addressed is how accurate the approximation of the exact edit distance is.



(a) (b) (c)

Fig. 3.6 (a) Fingerprint image, (b) orientation field, and (c) extracted graph

First, the accuracy of the approximation is examined on graphs extracted from fingerprint images of the standard NIST-4 database of fingerprints [Watson and Wilson (1992)]. To obtain an estimation of the orientation of ridge lines in fingerprints, a Sobel operator [Maltoni *et al.* (2003)] is applied to a grayscale fingerprint image. The estimated orientation field is then split into windows of equal size, and each window is represented by a node in the graph. For each node, the dominant ridge direction in the corresponding window is computed, and edges are generated in the two, out of eight, possible directions that best match the dominant direction. A single discrete attribute is attached to each edge representing its direction. For an illustration of a fingerprint image, its orientation field, and the cor-

Table 3.1 Running time and resulting distances of exact edit distance algorithm (1 run) and approximate edit distance algorithm (50 runs)

| | Running time | | Edit distance | |
|---|---|---|---|---|
| Nodes | Exact | Approx. | Exact | Approx. |
| 5 | <1s | <1s | 7.5 | 9.5 |
| 7 | <1s | <1s | 8 | 8 |
| 9 | 9s | 1s | 10 | 15 |
| 12 | — [a] | 1s | — [a] | 24.5 |
| 20 | — [a] | 1s | — [a] | 68 |
| 30 | — [a] | 2s | — [a] | 118.5 |
| 42 | — [a] | 5s | — [a] | 218 |
| 169 | — [a] | 15s | — [a] | 457.5 |

[a] Empty entries indicate failure due to lack of memory.

responding graph, see Fig. 3.6. The problem of fingerprint classification is treated in greater detail in Sec. 6.2.

As mentioned previously, graph edit distance is usually only applicable to graphs containing up to around 12 nodes, but the extracted graphs typically consist of about 170 nodes and 190 edges. Hence, to make the exact computation of edit distance feasible, we proceed by turning two large fingerprint graphs into smaller ones by successively removing those nodes (and their edges) that are further away from the center of the fingerprint than any other node. At various steps of this process, the exact and approximate edit distance between the two graphs is computed and their running times are measured. The results obtained on a workstation equipped with 1,024MB RAM are given in Table 3.1. While the exact edit distance can only be computed for small graphs, the approximate algorithm is also applicable to large graphs. The superior performance of the approximate algorithm in terms of running time is obvious, even though the exact algorithm is only performed once and the approximate algorithm 50 times for different pairs of seed substitutions. Note that the results are similar if different numbers of seed substitutions are used. In this experiment, we assume that nodes located in the center area of fingerprint graphs should usually not be mapped to nodes located in peripheral areas. Seed substitutions are therefore constructed from nodes of the center area of the two graphs under consideration.

To investigate how severely the approximate algorithm overestimates the exact edit distance, we compute the approximate edit distance of pairs of fingerprint graphs for which the exact edit distance is known. Because

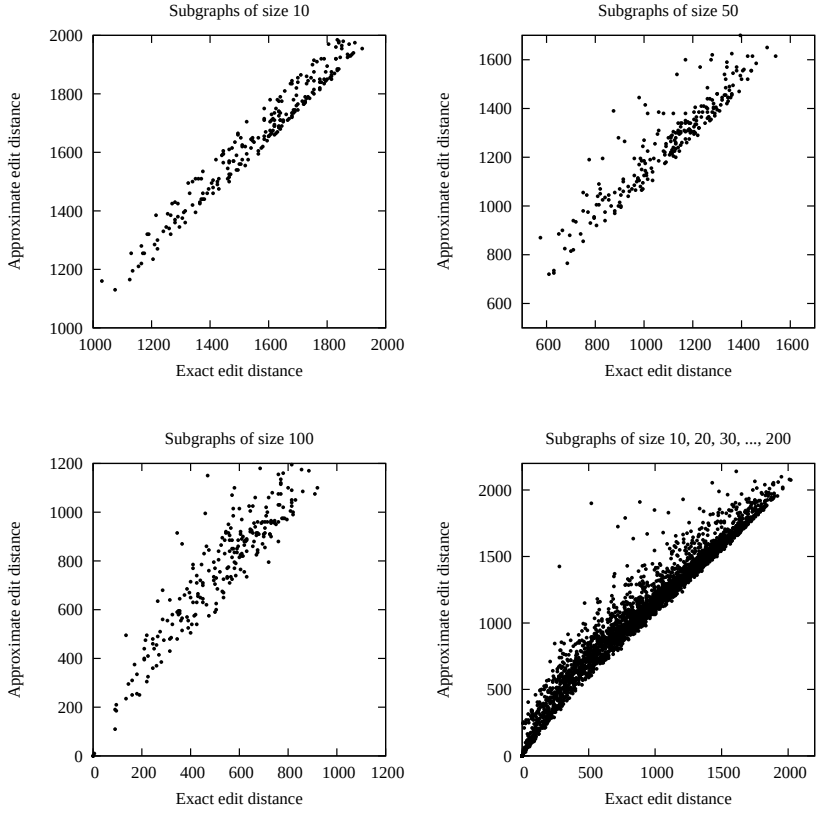
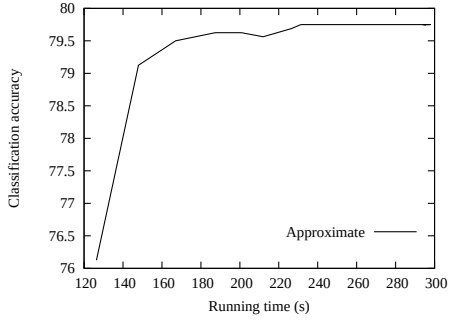



Fig. 3.7 Exact and approximate edit distance for subgraphs of various size

of the large size of these graphs, the exact edit distance cannot be computed by means of the standard algorithm. However, if we proceed by removing all but the $t$ nodes closest to the center of a fingerprint graph, we obtain a subgraph of size $t$ of the larger graph and know that the optimal edit path from the large graph to its subgraph is equivalent to deleting all but the $t$ nodes (and their edges) that are also present in the subgraph. The edit distance between a fingerprint graph and one of its subgraphs computed by the approximate algorithm can then be compared to the exact edit distance inferred from the optimal edit path. In this experiment, for 250 fingerprint graphs from the database, subgraphs of size $t = 10, 20, 30, \ldots, 90, 100, 125, 150, 175, 200$ are generated. Illustrations of

the exact and approximate distance between the original graphs and their subgraphs of size $t = 10$, subgraphs of size $t = 50$, subgraphs of size $t = 100$, and to all generated subgraphs are shown in Fig. 3.7. Comparing the first three illustrations, the approximate edit distance seems to be more accurate for smaller subgraphs ($t = 10$) than for larger ones ($t = 100$). However, it should be noted that the more nodes we remove from a fingerprint graph, the larger will be the resulting edit distance. That is, the approximate and exact edit distance of the original graphs to the smaller subgraphs ($t = 10$) will tend to be larger than those to the larger subgraphs ($t = 100$). For a comparison, the different scales of the diagrams should therefore be taken into account. In the fourth (uniformly scaled) diagram containing the distances to all generated subgraphs, the approximate edit distance seems to be relatively close to the exact edit distance, even though the approximate distance is slightly less accurate for smaller distances than for large distances.




| | Exact | Approx. |
|---|---|---|
| Accuracy | 80.6% | 79.8% |
| Time | 68,415s | 231s |

Fig. 3.8 Classification accuracy and running time

For the purpose of graph classification by means of edit distance, it is common to use nearest-neighbor classifiers. The basic idea is to classify input graphs according to their most similar graph from a labeled set of prototypes. For more details, see Sec. 3.6. The key question in the present context is whether graphs from the same class are assigned low distances by the approximate method, which is crucial for correct classification. For graphs from different classes, if the approximate distance is substantially higher than the exact distance, the classification accuracy will not be negatively affected. For an evaluation of the classification accuracy, an edit distance based nearest-neighbor classifier is applied to a fingerprint graph

representation different from the one used above (see Sec. 6.2 for details). To make the computation of the exact edit distance feasible, we select from the full test set of 2,000 graphs the 1,600 graphs containing less than 10 nodes. For these graphs, the edit distance is computed once by the exact algorithm and once by the approximate algorithm. A comparison of the resulting running times and classification rates is provided in Fig. 3.8. The illustration demonstrates how the running time of the approximate algorithm is related to its accuracy for different numbers of seed substitutions (incrementing the number of seed substitutions from 1 to 10 and then in larger steps to 200). That is, if the number of seed substitutions is increased, the classification accuracy is improved, but the computation takes longer. The optimal area of the diagram is obivously in the upper left corner, which corresponds to fast and simultaneously very accurate systems. Note that the running time of the exact distance is much too high to be included in the illustration. It takes only 4 minutes for the fastest approximate classifier among those with maximum accuracy to classify the 1,600 fingerprint graphs, whereas the running time of the exact algorithm amounts to over 19 hours for the same task. Yet, in terms of classification accuracy, the exact method outperforms all approximate systems. The improvement of the classification rate from 79.8% to 80.6% is small, but statistically significant (significance level $\alpha = 0.01$), as the exact method classifies all those graphs (except one) correctly that are correctly classified by the approximate method, and 14 graphs more.

We conclude that the approximate algorithm provides us with a valuable alternative to the exact algorithm for computing the edit distance of large graphs, but the exact algorithm is more appropriate where applicable. In the following, the approximate edit distance algorithm will therefore only be applied in cases where the exact edit distance computation is unfeasible.

## 3.5 Quadratic Programming Algorithm

The efficient approximate edit distance algorithm described in the previous section exploits the two-dimensional positional node information that is available in many graph representations in pattern recognition. In this section, we introduce another algorithm for the computation of graph edit distance. Unlike previously, the idea here is not to refine the standard exact edit distance algorithm, but rather to provide a novel perspective on the graph matching problem. In the following, the aim is to formulate

the minimum-cost optimization problem of edit distance in the well-known mathematical framework of quadratic programming [Nocedal and Wright (2000)]. Such an approach allows us to tackle the complex graph matching problem using standard optimization methods, instead of relying on heuristic search strategies.

The proposed quadratic programming approach requires the definition of fuzzy edit paths. The result of the proposed method is either a minimum-cost fuzzy edit path or, after defuzzification, a standard edit path between two graphs. The idea is to assign to each possible node substitution a fuzzy weight value indicating how well this substitution matches the structure and labels of involved nodes and edges. The algorithm aims at constructing a fuzzy edit path that minimizes the structural error in a manner similar to the minimization of edit costs in graph edit distance. In this respect, the proposed quadratic programming method is loosely related to relaxation labeling techniques for graph matching [Christmas *et al.* (1995); Wilson and Hancock (1997)], where the idea is to define the matching problem as a node labeling problem and apply iterative procedures refining the labeling until a sufficiently accurate matching is obtained. Unlike these relaxation labeling techniques, the method proposed in this section is applicable to arbitrarily labeled graphs and is closely related to the standard edit distance measure. A related method based on linear programming has recently been developed [Justice and Hero (2006)]. This method can be used to derive lower and upper bounds on the edit distance in polynomial time.

### 3.5.1 *Algorithm*

Quadratic programming is a particular type of mathematical optimization problem [Nocedal and Wright (2000)]. It turns out that the graph edit distance problem needs only a few slight adaptations to fit into the quadratic programming framework, which allows us to employ a whole new class of algorithms for the computation of graph edit distance.

#### 3.5.1.1 *Quadratic Programming*

Quadratic programming refers to a range of optimization problems satisfying a general mathematical form. In the following, the quadratic programming problem will be described and briefly discussed. First, let the set of real matrices of dimension $a \times b$ be denoted by $\mathbb{R}^{a \times b}$. For a given dimension $n \geq 1$, let us assume that a symmetric matrix $Q \in \mathbb{R}^{n \times n}$ and a

vector $c \in \mathbb{R}^n$ are given. Furthermore for $l, m \geq 1$, let matrices $R \in \mathbb{R}^{l \times n}$ and $S \in \mathbb{R}^{m \times n}$ as well as vectors $u \in \mathbb{R}^l$ and $v \in \mathbb{R}^m$ be given. The general quadratic programming problem can then be formulated as [Nocedal and Wright (2000)]

$$\text{Minimize} \quad f(x) = \frac{1}{2} x' Q x + c' x \qquad \text{for } x \in \mathbb{R}^{\mathrm{n}} \tag{3.3}$$

such that

$$\begin{aligned} Rx &= u \\ Sx &\geq v \ . \end{aligned}$$

Note that the vector inequality constraint in the last line means that all components of the two vectors must satisfy the inequality. Solving the quadratic programming problem consists in finding an $x \in \mathbb{R}^n$ that minimizes $f(x)$ such that the given equality and inequality conditions are satisfied. The expression *quadratic programming* is due to the fact that the target function $f(x)$ is a quadratic function of the argument $x$. The equality constraint can be seen as a compact representation of $l$ independent equality conditions (one per line of matrix $R$), and similarly the inequality constraint is equivalent to $m$ inequality conditions.

Quadratic programming problems can always be solved, or shown to be unfeasible, in a finite amount of time. However, the actual complexity of the computation depends strongly on the characteristics of the problem, in particular on the matrix $Q$ and the number of relevant inequality constraints [Nocedal and Wright (2000)]. If $Q$ is positive definite, for instance, the quadratic programming problem can typically be solved as efficiently as linear programming problems. Furthermore, it is also known in this case that there exists a globally optimal solution, provided that the equality and inequality constraints are satisfied for at least one vector. If $Q$ is an indefinite matrix, the optimization process may have to deal with stationary points and local minima. The methods commonly used to solve quadratic programming problems can roughly be divided into interior point methods, active set methods, and conjugate gradient methods [Nocedal and Wright (2000)]. In our experiments, we use the interior point algorithm from the Computational Optimization Program Library [Zhang and Ye (1998)].

A classic example of a quadratic programming problem is the management of investment portfolios [Nocedal and Wright (2000)]. The idea is to model the tradeoff between risk and expected return for a collection of investments. In the classic Markowitz model, the expected return of an investment is modeled as a random variable following a normal distribution

[Markowitz (1952)]. While this model is fairly simple from the theoretical point of view, its real-world application is challenging due to the lack of reliable estimators of mean value, variance, and correlation of investments. The aim is to primarily spend money on investments with high mean value and low variance, representing high returns and low risks. In the following, let us assume that the mean vector $\mu = (\mu_1, \ldots, \mu_n)$ of $n$ investments and the corresponding covariance matrix $Q \in \mathbb{R}^{n \times n}$ are given. An optimal investment strategy can then be found by solving the optimization problem

$$\begin{aligned} &\text{Maximize} \quad f(x) = x'\mu - \kappa x'Qx \qquad \text{for } x \in \mathbb{R}^n \\ &\text{such that} \\ &\qquad \sum_{i=1}^{n} x_i = 1 \\ &\qquad x_i \geq 0 \text{ for } i = 1, \ldots, n \ , \end{aligned} \tag{3.4}$$

where the parameter vector $x = (x_1, \ldots, x_n)$ encodes the percentage of a full amount to be invested into investment $1, \ldots, n$. Obviously, this optimization problem satisfies the general form of quadratic programming problems in Eq. (3.3). The target function $f(x)$ aims at maximizing the expected value $x'\mu$ of investments while minimizing the associated risk $x'Qx$. In the Markowitz model, these two contradicting objectives are combined by means of a risk tolerance parameter $\kappa$. Obviously, conservative investors would choose a large value for $\kappa$ to put a strong emphasis on the minimization of the associated risk, while investors who are prepared to take higher risks would set $\kappa$ to smaller values. Hence, in this simple example, quadratic programming optimization is used to derive an investment strategy that provides a trade-off between high returns and low variance.

The popular support vector machine method for classification and regression is another quadratic programming application. The maximum-margin hyperplane of support vector machines separating two classes can be found by solving a quadratic programming problem, namely by minimizing the squared norm of the hyperplane weight vector given a number of linear constraints [Schölkopf and Smola (2002)]. Support vector machines will be discussed in greater detail in Sec. 4.3.1. In the following, quadratic programming will be applied to the computation of graph edit distance.

#### 3.5.1.2 *Fuzzy Edit Path*

The standard graph edit distance is defined by the minimum-cost edit path between two graphs. A common interpretation of substitutions in an opti-

mal edit path is that they indicate which parts of one graph can be identified in the other graph. That is, a set of node substitutions can be seen as a mapping of nodes from one graph to nodes of another graph. Analogously, deleted (or inserted) nodes and edges can be interpreted as those nodes and edges of the first graph (second graph) that cannot be matched, with sufficient accuracy, to nodes and edges of the second graph (first graph). Hence, given an edit path between two graphs, each node and edge is either substituted with another node and edge, or deleted or inserted.

The basic idea of fuzzy edit paths is to allow nodes and edges of one graph to be simultaneously assigned to several nodes and edges of another graph. In the following, let us assume that two graphs $g = (V, E, \mu, \nu)$ and $g' = (V', E', \mu', \nu')$ with $|V| = n$ and $|V'| = n'$ are given. Clearly, there exist $n \cdot n'$ distinct substitutions $u \to v$ of nodes $u \in V$ with nodes $v \in V'$. A fuzzy edit path is defined by assigning a weight to each possible node substitution such that weights associated with substitutions involving a given node sum up to 1. Formally, a fuzzy edit path between $g$ and $g'$ is a function $w : V \times V' \to [0, 1]$ satisfying the conditions

$$\sum_{v \in V'} w(u, v) = 1 \text{ for each } u \in V \quad \text{and} \quad \sum_{u \in V} w(u, v) = 1 \text{ for each } v \in V' \ . \tag{3.5}$$

This weighting function $w$ can be understood as a kind of membership function reflecting how well a node substitution conforms to, or conversely how strongly it violates, the structure and labels of the two graphs. The interpretation of a fuzzy edit path that is optimal with respect to some matching criterion is that two nodes $u, v$ with a large value of $w(u, v)$ are likely to correspond to a good structural match, while nodes with small values of $w(u, v)$ should be considered unmatchable. The advantage of fuzzy edit paths over standard edit paths is that they allow us to integrate ambiguity directly in the definition of edit paths, instead of being forced to settle for a single edit transformation for each node and edge.

In order to construct a standard edit path from a fuzzy edit path, a defuzzification procedure can be carried out. A straight-forward defuzzification method consists in selecting from all fuzzy node substitutions those with large fuzzy weights. The standard edit path to be derived from the fuzzy edit path is first initialized as an empty set. The first node substitution to be added to the standard edit path is then obtained by selecting from all fuzzy node substitutions the one with largest fuzzy weight, say the substitution $u \to v$. In the following steps, all fuzzy node substitutions involving either $u$ or $v$ will no longer be considered, thus essentially removing

substitutions of the form $u \to \ldots$ and $\ldots \to v$ from the fuzzy edit path. The second node substitution of the standard edit path is obtained by selecting from the remaining fuzzy node substitutions the one with largest fuzzy weight. Again, all fuzzy node substitutions containing either one of the two nodes of the selected substitution are ignored in successive steps. This iterative procedure is continued until no more node substitutions can be extracted. The remaining nodes are considered equivalent to node deletions and insertions. Finally, edge operations are inferred from node operations. Note that in the computation of fuzzy edit paths, edge edit operation costs will be included in the definition of fuzzy weights attached to node substitutions. That is, not only the substitution of nodes, but also the edge structure plays a role in the defuzzification procedure outlined above. In experimental results, it turns out that applying a defuzzification procedure to fuzzy edit paths and using standard edit paths for classification is superior to using fuzzy edit paths directly. It should be noted, however, that there does not exist a unique way to turn fuzzy edit paths into standard edit paths. For instance, applying error-minimization techniques such as Munkres' algorithm [Munkres (1957)] for defuzzification might be a viable alternative to the iterative procedure proposed above.

#### 3.5.1.3 *Quadratic Programming Edit Path Optimization*

In the preceding section, fuzzy edit paths have been introduced as an extension to standard edit paths. The remaining question is how to compute a fuzzy edit path between two graphs that is optimal with respect to some node and edge matching criterion. The method we propose is based on a quadratic programming formulation of the graph matching problem. The basic idea is to encode node and edge edit costs in a cost matrix and minimize the overall node and edge matching costs corresponding to fuzzy edit paths.

Again, let the two graphs under consideration be denoted by $g = (V, E, \mu, \nu)$ and $g' = (V', E', \mu', \nu')$, and let $|V| = n$ and $|V'| = n'$. It is clear that there exist $n \cdot n'$ substitutions between $g$ and $g'$. In view of this, we construct a real matrix $Q \in \mathbb{R}^{nn' \times nn'}$ where rows and columns are indexed by substitutions $u \to v$ ($u \in V, v \in V'$). That is, each row, and the corresponding column, of the matrix is associated with one distinct node substitution. The matrix $Q$ is then constructed in such a way that diagonal entries hold the costs of node substitutions, while off-diagonal entries correspond to edge edit costs. The entry at position $(u \to v, u \to v)$ is set to

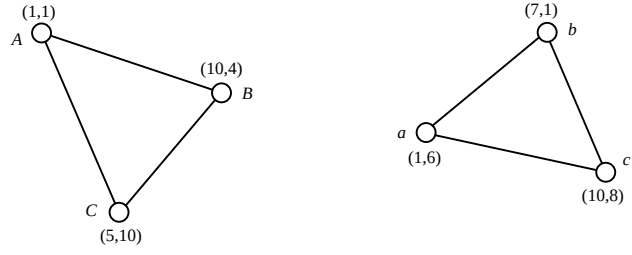



Fig. 3.9 Two example graphs $g$ (left) and $g'$ (right)

the node substitution cost of $u \to v$; the entry at position $(u \to v, p \to q)$ is set to the edge edit costs resulting from substituting $u \to v$ and $p \to q$, depending on the existence of edges between $u$ and $p$ as well as between $v$ and $q$.

An example of two graphs with $n = n' = 3$ is provided in Fig. 3.9, where nodes are labeled with a two-dimensional position attribute and edges are unlabeled. It is clear that in this example the nine possible distinct node substitutions are $A \to a, A \to b, A \to c, B \to a, B \to b, B \to c, C \to a, C \to b$, and $C \to c$. When constructing the $9 \times 9$ matrix $Q$, each row and column is associated with one of these substitutions. In Fig. 3.10, an example cost matrix $Q$ is shown for the two graphs in Fig. 3.9. Note that for the sake of clarity, only rows of the matrix are indexed with their associated node substitution; the corresponding column indices are omitted. In this example, node substitution costs are set equal to the squared Euclidean distance of the two node labels, and node and edge insertion and deletion costs are set to a constant value of 10. The substitution of unlabeled edges can be carried out for free. For example, since node $A$ is labeled with $(1, 1)$ and node $a$ with $(1, 6)$, the substitution $A \to a$ results in costs $Q_{A\to a, A\to a} = 25$. Since there exists an edge between $A$ and $B$ as well as an edge between $a$ and $b$, the substitutions $A \to a$ and $B \to b$ involve no edge operation costs, hence $Q_{A\to a, B\to b} = 0$. As node $A$ is not connected to itself by an edge, the substitutions $A \to a$ and $A \to b$ involve the insertion of an edge, which leads to $Q_{A\to a, A\to b} = 10$.

Recall that fuzzy edit paths are defined in the preceding section as functions assigning weights to all possible node substitutions between two graphs. Also, the matrix $Q$ consists of one row and one column per node substitution. In view of this, we define a fuzzy cost function assigning each row of $Q$ a weight according to the rules stated in Eq. (3.5). That is, each

$$Q = \begin{pmatrix} 25 & 10 & 10 & 10 & 0 & 0 & 10 & 0 & 0 \\ 10 & 36 & 10 & 0 & 10 & 0 & 0 & 10 & 0 \\ 10 & 10 & 130 & 0 & 0 & 10 & 0 & 0 & 10 \\ 10 & 0 & 0 & 85 & 10 & 10 & 10 & 0 & 0 \\ 0 & 10 & 0 & 10 & 18 & 10 & 0 & 10 & 0 \\ 0 & 0 & 10 & 10 & 10 & 16 & 0 & 0 & 10 \\ 10 & 0 & 0 & 10 & 0 & 0 & 32 & 10 & 10 \\ 0 & 10 & 0 & 0 & 10 & 0 & 10 & 85 & 10 \\ 0 & 0 & 10 & 0 & 0 & 10 & 10 & 10 & 29 \end{pmatrix}$$

| | | |
|---|---|---|
| $A \to a$ | 0.662 | Solution |
| $A \to b$ | 0.297 | |
| $A \to c$ | 0.041 | |
| $B \to a$ | 0.000 | |
| $B \to b$ | 0.619 | |
| $B \to c$ | 0.381 | |
| $C \to a$ | 0.338 | |
| $C \to b$ | 0.084 | |
| $C \to c$ | 0.578 | |

| | | |
|---|---|---|
| $A \to \ldots$ : | $0.662 + 0.297 + 0.041 = 1$ | Constraints satisfied |
| $B \to \ldots$ : | $0.000 + 0.619 + 0.381 = 1$ | |
| $C \to \ldots$ : | $0.338 + 0.084 + 0.578 = 1$ | |
| $\ldots \to a$ : | $0.662 + 0.000 + 0.338 = 1$ | |
| $\ldots \to b$ : | $0.297 + 0.619 + 0.084 = 1$ | |
| $\ldots \to c$ : | $0.041 + 0.381 + 0.578 = 1$ | |

Fig. 3.10 Example quadratic programming problem matrix $Q$ (corresponding to the graphs in Fig. 3.9) and solution weight vector satisfying fuzzy edit path constraints

row, and the corresponding column, is associated with a node substitution and a fuzzy weight. It should be noted that these fuzzy weights are not predefined, but to be determined in the optimization process. Hence, the idea of the reformulated graph matching problem is to find fuzzy weights that satisfy the conditions of a fuzzy edit path and minimize the structural error. To this end, we propose to minimize the expression $x'Qx$, where $x$ denotes the $n \cdot n'$-dimensional vector of fuzzy weights, one for each row of $Q$. The minimization is carried out over all fuzzy weights $x$ satisfying the conditions defined in Eq. (3.5). In this optimization formulation, the weight associated with a node substitution $u \to v$ will influence not only the weighting of the node substitution costs of $u \to v$ (in the diagonal entry $Q_{u \to v, u \to v}$), but also all edge edit costs involving the substitution $u \to v$ (in off-diagonal entries $Q_{u \to v, p \to q}$ and $Q_{p \to q, u \to v}$). Clearly, this optimization process aims at assigning large weights to node substitutions that involve low node and edge costs, and assigning small weights to node substitutions that result in high costs. Note that this optimization principle is not identical to the minimum cost edit path concept in standard graph edit distance, but the intuitive interpretation of minimizing penalty costs for structural errors is comparable.

The optimization problem described above can be formulated in the standard quadratic programming framework. To this end, the matrix $Q$

Table 3.2 Recognition accuracy of the QP edit distance algorithm compared to the exact algorithm (Sec. 3.3) and the efficient approximate algorithm (Sec. 3.4)

| Data set | Method | Validation | Test set | Running time |
|---|---|---|---|---|
| Letters | Approximate algorithm | 67.3 | 69.3 | 12.4' |
| | QP algorithm | 76.7 | 74.9 • | 19.5' |
| Images | Exact algorithm | 64.8 | 48.1 | 9s |
| | QP algorithm | 72.2 | 59.3 • | 18s |
| Diatoms | Approximate algorithm | 86.5 | 66.7 • | 8s |
| | QP algorithm | 54.1 | 47.2 | 15s |

• Improvement over other method statistically significant ($\alpha = 0.05$)

in Eq. (3.3) is defined as the cost matrix $Q$ described above, the solution vector $x$ in Eq. (3.3) is the weight vector $x$ mentioned above, and vector $c$ in Eq. (3.3) is the zero vector. Furthermore, it is easy to see that the conditions of consistent fuzzy weights can be formulated in terms of equality and inequality conditions of the form $Rx = u$ stating that fuzzy weights sum up to 1 as shown in Eq. (3.5), and $Sx \geq v$ restricting considerations to fuzzy weights between 0 and 1. In Fig. 3.10, the result of the quadratic programming approach to matching the two graphs in Fig. 3.9 is shown. The solution vector clearly satisfies the fuzzy edit path constraints. After defuzzification, we obtain the same optimal edit path as the standard edit distance algorithm, $\{A \to a, B \to b, C \to c\}$. Note that from the solution vector, it is not only possible to extract the most likely edit path, but also other edit paths that seem to be rather likely, such as the one with edit operations $\{A \to b, B \to c, C \to a\}$ in our case.

### 3.5.2 *Experimental Results*

In the following, we evaluate how the quadratic programming method performs in comparison to the standard edit distance method on three graph data sets representing letters, images, and diatoms. The letter data set consists of line drawings of 15 capital letters; the image data set consists of 5 classes of region adjacency graphs representing images after processing and filtering; and the diatom data set is based on microscopic images of diatoms from 22 classes (for details, see Sec. 6.1).

In Table 3.2, the recognition accuracy of a $k$-nearest-neighbor classifier based on the standard edit distance method from Sec. 3.3 and the approximate edit distance method from Sec. 3.4 are compared to the accuracy of the quadratic programming method described above. Note that the rele-

vant classification accuracy is the one on the test set. For details about the classification procedure and the experimental setup, refer to Chap. 6. In two out of three cases, the quadratic programming method outperforms the standard methods significantly, while on the third data set, the proposed method is clearly inferior. These classification results show that the graph matching method based on quadratic programming constitutes a viable alternative to the standard edit distance method. As far as the running time is concerned, the proposed method seems to require typically twice the running time of the standard algorithm. However, it should be noted that the efficiency of the proposed method heavily depends on the optimization implementation of the quadratic programming algorithm at hand.

The application of the proposed quadratic programming method to large graphs is more challenging. The number of possible node substitutions, and thus the size of the fundamental cost matrix $Q$, grows quadratically with an increasing graph size. For the two graph data sets consisting of rather large graphs representing fingerprints and molecules (the reader is referred to Chap. 6 for details), the graph matching method based on quadratic programming is not feasible. A possible solution to this problem might be to restrict considerations to a subset of all node substitutions, taking into account only a few candidate substitutions deemed promising by some local node and edge matching criterion. Further investigations of this issue are left to future research.

For these reasons, the quadratic programming method will not be considered in the remainder of this book. Instead, we will propose a number of kernel functions for graphs to be applied in conjunction with support vector machines. Hence, the quadratic programming principle will not directly be applied to the graph matching problem, but rather to the graph classification task in the underlying training of support vector machines.

## 3.6 Nearest-Neighbor Classification

The traditional approach to graph matching using edit distance is based on nearest-neighbor classification. In the nearest-neighbor paradigm, test patterns are classified according to their most similar patterns from the training set. The main advantage of nearest-neighbor classifiers over other graph classifiers is that a graph similarity measure, or dissimilarity measure, is the only operation that must be defined in the pattern space. Furthermore, nearest-neighbor classifiers are able to model arbitrarily complex

class distributions with a convincingly simple approach [Cover and Hart (1967)].

Nearest-neighbor classifiers are based on a labeled training set, or set of prototypes. For an unknown test pattern, the basic idea is to compute the similarity of the test pattern to all patterns from the training set and assign the test pattern to the class of the most similar training pattern, or nearest neighbor. The class-separating functions resulting from nearest-neighbor classification are composed of piecewise linear functions, which are a very flexible class of decision boundaries compared to those of Bayes classifiers and others, but are largely based on empirical arguments. To render nearest-neighbor classifier less prone to outlier patterns, it is common to consider not only the single most similar pattern from the training set, but evaluate several of the most similar patterns, so that the sensitivity of nearest-neighbor classification towards outliers can be controlled to a certain degree.

Unlike for classifiers such as artificial neural networks, Bayes classifiers, decision trees, and others [Duda *et al.* (2000)], the underlying pattern space need not be rich in mathematical operations for nearest-neighbor classifiers to be applicable. The fact that only a pattern dissimilarity measure must be available makes nearest-neighbor classifiers very appealing to graph matching. In conjunction with the powerful edit distance measure, nearest-neighbor classifiers appear to be naturally suited to graph classification. For a more formal treatment of nearest-neighbor classification, refer to Sec. 4.4.

## 3.7 An Application: Data-Level Fusion of Graphs

Pattern classification is the most common application of graph edit distance, where the edit distance of graphs is used to classify input graphs according to their similarity to other graphs. In this scenario, the only information that is extracted from an optimal edit path is the sum of individual edit operation costs. Hence, the optimal correspondence of nodes and edges in terms of substitutions, which is the actual result of edit distance based graph matching, is not taken into account at all in this scenario. To demonstrate that edit distance is not confined to graph classification, another application explicitly making use of this additional matching information is described in this section.

The idea of multiple classifier systems, sometimes referred to as classi-

fier fusion, is to combine several classifiers such that the resulting combined system performs better than each classifier individually [Kuncheva (2004)]. For statistical patterns represented by feature vectors, a large variety of methods for multiple classifier systems have been proposed over the past few years. For structural patterns, and graph patterns in particular, the fusion of classifiers has mainly been constrained to the decision level [Yao *et al.* (2003); Marcialis *et al.* (2003); Schenker *et al.* (2004a)], that is, to the combination of the classification results of several classifiers. Such a fusion can be effected, for instance, by having classifiers vote for their predicted classes. The fusion of graph matching systems at levels other than the decision level has not yet been considered. In this section, an algorithm for the data-level fusion of graphs is proposed. Related methods for the representation of several graphs by a single representative include the median graph concept [Jiang *et al.* (2001)] and the notion of weighted minimum common supergraph [Bunke *et al.* (2003)].

### 3.7.1 *Fusion of Graphs*

In the following, edit distance is applied to the problem of data-level fusion of graphs. The idea is to merge several graphs representing the same underlying object into a graph that is more robust against noise than each graph individually. In practice, it is often difficult to develop one graph extraction procedure that performs well on all patterns. In such a case it may be advantageous to develop several slightly different graph extraction procedures, extract several graphs per object, and merge these into a single graph. For the proposed method, these graph extraction procedures only need to meet the requirement that graphs extracted from a given object have similar characteristics. This is satisfied, for instance, if the same procedure is applied several times with different parameters. The proposed method for the data-level fusion of graphs is based on the idea of detecting common substructures occurring in several graphs representing a single object. The common substructures typically represent those parts of the graphs that are robust against noise and simultaneously characteristic of the underlying pattern and are therefore considered more reliable for classification.

The merging of several graphs is based on an iterative merging of two graphs at a time. First, the edit distance between all graphs to be merged is computed. The two best matching graphs, that is, the two graphs with smallest distance, are selected for merging. For these graphs, the nodes and

edges involved in a substitution in the optimal edit path are regarded as the matching part of the two graphs. From the optimal edit path, a new graph is constructed consisting of one node for each node substitution and one edge for each edge substitution, so that each node (and edge) of the new graph corresponds to a node (edge) included in both of the two graphs to be merged. The new nodes (edges) are then assigned a label constructed from the labels of the two corresponding original nodes (edges). If the nodes are labeled with a two-dimensional position attribute, for instance, the merged node could be assigned the average of the two original node positions. In cases where the merging of labels is not feasible, for instance, if symbolic labels are used, the merged node has to be assigned one of the two original labels. Hence, it is clear that the label merging has to be defined in an application-specific way. The procedure outlined above can be used to construct from two graphs a graph representing their common substructures.

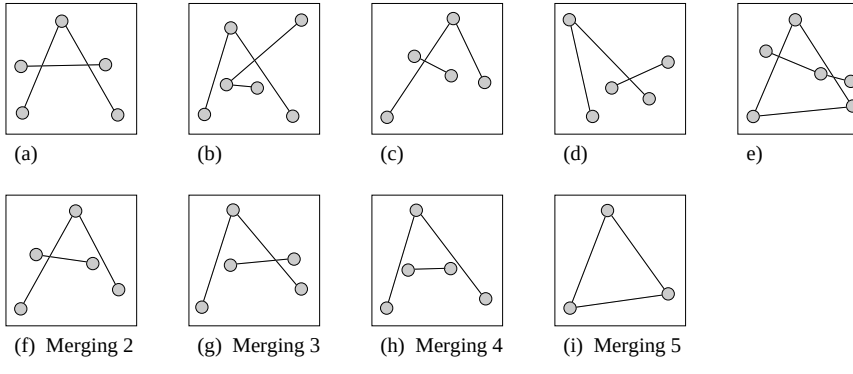



Fig. 3.11 (a)–(e) Five graphs representing a letter $A$ and (f)–(i) graphs resulting from their fusion

The two graphs selected from the set of graphs to be merged are then replaced by the newly constructed graph, and the merging continues with the two most similar graphs from the resulting set of graphs. Note that the graph similarity is recomputed after each merging step. In this manner, the set of graphs is reduced until a single graph representing all original graphs is left. For two graphs that are very similar, their merged graph will be large, since there will be a considerable number of substitutions in the optimal edit path between the two graphs. For two dissimilar graphs, on the other hand, their merged graph will be small. This means that even

Table 3.3 Classification accuracy on five graph data sets ($T_1$–$T_5$) and on one data set consisting of merged graphs ($T_{merged}$)

| Test set | Prototype set | Accuracy |
|---|---|---|
| $T_1$ | $P_1$ | 54.8 |
| $T_2$ | $P_2$ | 54.9 |
| $T_3$ | $P_3$ | 51.7 |
| $T_4$ | $P_4$ | 70.0 |
| $T_5$ | $P_5$ | 76.0 |
| $T_{merged}$ | $P_{merged}$ | 85.1 |

if only a single graph is significantly different from all other graphs to be merged, the merging will eventually lead to a small or even empty graph. To eliminate the influence of outlier graphs in the fusion process, it is therefore recommended to terminate the merging before a single graph is obtained at the end, so that only the most similar graphs are taken into account for merging. In Fig. 3.11a–e, five graphs representing a letter $A$ are illustrated. Note that the nodes contain a two-dimensional position attribute. Merging the two most similar ones of these five graphs (the graphs in Fig. 3.11a and c) results in the graph shown in Fig. 3.11f, and the graphs obtained in successive merging steps are illustrated in Fig. 3.11g–i. The merged graphs, except for the last one in Fig. 3.11i, represent the characteristics of a letter $A$ better than the individual graphs in Fig. 3.11b–e. In this example, it would be advantageous to terminate the process after merging three or four graphs.

### 3.7.2 *Experimental Results*

In the following, we investigate whether the data-level fusion method for graphs described above can be used to improve the classification performance. To this end, five data sets are constructed, each containing 1,500 graphs representing randomly distorted line drawings of 15 capital letters. For a detailed description of these letter data sets, see Sec. 6.1.1. Each of the five data sets is split into a test set $(T_1, \ldots, T_5)$ and a training set $(P_1, \ldots, P_5)$ of equal size. To obtain more robust graphs from these test sets, we then construct a test set $T_{merged}$ that consists of graphs resulting from the merging of five graphs, one from each test set $T_1, \ldots, T_5$. That is, the $n$-th graph of the merged test set $T_{merged}$ corresponds to merging the

Table 3.4 Five individual fingerprint classifiers and their combinations using (a) majority voting, (b) maximum confidence, and (c) confidence voting

| Test set | Prototype set | Accuracy | |
|---|---|---|---|
| $T_1$ | $P_1$ | 80.9 | |
| $T_2$ | $P_2$ | 74.6 | |
| $T_3$ | $P_3$ | 73.4 | |
| $T_4$ | $P_4$ | 75.5 | |
| $T_5$ | $P_5$ | 74.6 | |
| $T_1, \ldots, T_5$ | $P_1, \ldots, P_5$ | 77.3 | (a) |
| $T_1, \ldots, T_5$ | $P_1, \ldots, P_5$ | 86.8 | (b) |
| $T_1, \ldots, T_5$ | $P_1, \ldots, P_5$ | 83.2 | (c) |

$n$-th graphs of the data sets $T_1, \ldots, T_5$. A merged set of training graphs $P_{merged}$ is similarly created from $P_1, \ldots, P_5$. The performance of the classification of graphs from $T_i$ according to the most similar graphs from $P_i$ are shown in Table 3.3. Apparently, the merged graphs constitute a more robust representation of the underlying patterns than the original graphs, which can be concluded from the superior classification performance on the merged test set.

For a more thorough analysis, the graph fusion method is applied to the difficult problem of fingerprint classification. Here, one graph extraction method is used with five different parameter configurations to extract five different graphs from a single fingerprint image (see Sec. 6.2 for details), which results in five sets of fingerprint graphs $T_1, \ldots, T_5$. For each data set $T_i$, a specific set of prototype graphs $P_i$ is constructed in a heuristic fashion. The idea is to select the first 50 samples (out of 4,000 fingerprints in total) and manually derive from them a number of prototypes representing apparent characteristics of the underlying classes. The graphs from test set $T_i$ are then classified according to the set of prototypes $P_i$. Input graphs are assigned the class of the most similar graph from the set of prototypes (1-nearest-neighbor classifier).

Since the $n$-th graph of each test set $T_i$ in fact represents the same fingerprint, it is straightforward to combine the results of the five individual classifiers for the $n$-th graph of $T_i$ ($i = 1, \ldots, 5$) so as to increase the overall classification accuracy. The simplest method for this purpose is *majority voting*. That is, each of the five classifiers casts a vote for the class it predicts, and the input graph is finally assigned the class with the maximum

Table 3.5 Fingerprint classification using data-level fusion and (a) majority voting, (b) maximum confidence, and (c) confidence voting

| Test set | Prototype set | Merge 5 | Merge 4 | Merge 3 | Merge 2 |
|---|---|---|---|---|---|
| $T_{merged}$ | $P_1$ | 79.6 | 79.9 | 80.8 | 80.5 |
| $T_{merged}$ | $P_2$ | 79.3 | 79.6 | 80.4 | 79.9 |
| $T_{merged}$ | $P_3$ | 78.2 | 78.6 | 79.2 | 78.8 |
| $T_{merged}$ | $P_4$ | 63.2 | 63.3 | 63.8 | 63.6 |
| $T_{merged}$ | $P_5$ | 64.7 | 65.2 | 65.6 | 65.9 |
| $T_{merged}, \ldots, T_{merged}$ | $P_1, \ldots, P_5$ | 73.3 | 74.7 | 74.9 | 75.2 (a) |
| $T_{merged}, \ldots, T_{merged}$ | $P_1, \ldots, P_5$ | 81.5 | 87.3 | 88.3 | 87.8 (b) |
| $T_{merged}, \ldots, T_{merged}$ | $P_1, \ldots, P_5$ | 81.4 | 82.4 | 82.6 | 82.8 (c) |

number of votes. A more refined method is based on a confidence measure. In addition to predicting a single class, each classifier also computes a measure of reliability for its decision. The combination can then be effected by following the classifier with *maximum confidence*. Alternatively, these two methods can also be combined by having each classifier cast a vote for a class weighted with its confidence (*confidence voting*). The confidence measure used in this experiment is defined by the ratio of the distance of the input graph to its nearest-neighbor in relation to the distance of the input graph to the second-nearest-neighbor (belonging to another class). Note that this confidence measure is confined to the interval $[0, 1]$. A small value of the confidence measure, which is equivalent to the nearest-neighbor being significantly closer than the second-nearest-neighbor, represents a high confidence in the resulting classification. The classification accuracy of the five individual classifiers and their decision-level combinations using majority voting, maximum confidence, and confidence voting are shown in Table 3.4. The advantage of combined classifiers over individual classifiers is very clear.

In the following, the objective is to improve the performance given in Table 3.4. We proceed by merging the test sets $T_1, \ldots, T_5$ into a single set $T_{merged}$, which is accomplished by selecting the $n$-th graph of each $T_i$ and merging these five graphs. Hence, each graph in $T_{merged}$ corresponds to five graphs, one from each $T_1, \ldots, T_5$. Instead of classifying graphs from $T_i$ according to their nearest neighbors in $P_i$, we classify graphs from $T_{merged}$ according to $P_i$ (for $i = 1, \ldots, 5$). The resulting five classifiers can then be combined, in analogy to the combination in Table 3.4, by majority voting, maximum confidence, and confidence voting. The corresponding results are

Table 3.6 Summary of fingerprint classification accuracy

| Fingerprint classifier | Full data set | Second half of data set |
|---|---|---|
| Best individual classifier | 80.9 | 80.3 |
| Best decision-level fusion | 86.8 | 87.0 |
| Best data-level fusion | 80.8 | 80.6 |
| Best data- and decision-level fusion | 88.3 • | 88.8 • |

• Significant improvement over other classifiers (statistical sign. level $\alpha = 0.01$)

shown in Table 3.5. Note that all classification rates are given for the fusion of the two best matching graphs only (Merge 2), the three best matching graphs (Merge 3), the four best matching graphs (Merge 4), and all five graphs (Merge 5). Merging all five graphs is not recommended, since the influence of outlier graphs may be too strong in this case. Accordingly, the best results are obtained for the fusion of two, three, or four graphs.

Table 3.4 and Table 3.5 demonstrate clearly that the data-level fusion of graphs improves the matching of graphs. If the merged graphs in $T_{merged}$ are classified according to the set of prototypes $P_i$, the results are inconsistent. In some cases, the accuracy can be improved (for $P_2$ from 74.6% to 80.4%), while in other cases, the accuracy drops significantly (for $P_4$ from 75.5% to 63.8%). This effect can be explained by the definition of the sets of prototypes $P_i$, which have been developed specifically for the characteristics of the graphs in $T_i$, and not $T_{merged}$. Applying data-level fusion in conjunction with decision-level classifier combination is especially advantageous. The combination method based on the maximum confidence of classifiers performs particularly well, since here the most appropriate set of prototypes $P_i$ for a merged input graph, that is, the one with maximum confidence, will be chosen for classification.

A summary of the results discussed above on the 4,000 graphs extracted from the NIST-4 database [Watson and Wilson (1992)] and the second half of the data set, which is often used as test set in fingerprint classification, is provided in Table 3.6. For a comparison of these recognition rates to those obtained by other systems, see Sec. 6.2. It should be noted that the accuracy of the combined data-level and decision-level fusion is better than that of any other classifier mentioned in this book. Hence, as expected, the positive effect of merging several graphs into a single more robust representative can also be observed in practice. Also, the significant improvement of the classification accuracy is achieved without taking into account which graphs belong to which class. Instead, it is perfectly sufficient to construct

graphs representing the common substructures of several graphs to make the graph matching and successive classification more accurate. That is, the characteristics of graph classes can more easily be identified in merged graphs than in the original ones. This particular graph fusion application demonstrates how the information resulting from an edit distance computation between graphs can be used for tasks other than classification.

## Chapter 4

# Kernel Machines

Kernel machines are a powerful class of algorithms for pattern analysis and classification. Although some of the mathematical foundations of kernel machines have been known for a long time [Vapnik and Chervonenkis (1971); Vapnik (1982); Berg *et al.* (1984)], the practical application of results from kernel theory and statistical learning theory to pattern recognition problems has only been realized in recent years [Schölkopf and Smola (2002); Shawe-Taylor and Cristianini (2004); Vapnik (1998)]. Since then, kernel machines have attracted a huge amount of interest in the area of pattern recognition and machine learning, which is mainly due to the elegance of the kernel approach, the widespread applicability of kernels to standard and novel algorithms, and the elaborate underlying theory of statistical learning. Statistical learning theory casts the problem of learning a classifier from a training set of patterns into well-defined statistical terms and offers a way to derive classifiers expected to perform well on unseen data. By means of kernel functions, the extension of basic linear algorithms to complex non-linear methods can be carried out in a unified manner. Furthermore, kernel functions make standard algorithms applicable to complex data structures for which a vector space representation does not exist. There is theoretical evidence that, under some conditions, kernel machines are more appropriate for difficult pattern recognition problems than traditional methods, such as nearest-neighbor classifiers or artifical neural networks. In recent years, the advantage of kernel machines over other methods has empirically been confirmed in many pattern recognition areas.

This chapter gives a brief introduction to kernel machines. First, the problem of learning classifiers is addressed in a very general setting. Then kernel functions are introduced and some important kernel machines are

described. The concepts discussed in a very general setting in this chapter will be applied to error-tolerant graph matching in the next chapter.

## 4.1 Learning Theory

Supervised learning is the task of learning the regularities of an underlying pattern distribution from a labeled training set of examples. The general idea is to observe a certain number of training patterns and try to capture relevant criteria for the distinction of different classes. For example, learning a Bayes classifier means in fact estimating the parameters of a postulated probability density from the training set. Or, the training of multilayer artificial neural networks consists of propagating error terms from the output layer back to internal layers.

The formal definition of the supervised learning problem requires the definition of a space of patterns $\mathcal{X}$ and a space of class labels $\mathcal{Y}$. The pattern space $\mathcal{X}$ consists of all instances of objects potentially to be analyzed or recognized, that is, $\mathcal{X}$ denotes the entire population of patterns. Example pattern spaces are the space of graphs, vector spaces of real numbers, or finite sets of elements. The space of class labels $\mathcal{Y} = \{y_1, y_2, \ldots, y_k\}$ consists of a finite number of discrete symbols, each representing a class of the classification problem to be addressed. For the common two-class classification problem, $\mathcal{Y} = \{-1, +1\}$ is usually chosen. Note that in this book considerations are restricted to pattern classification problems, but the supervised learning problem can be formulated in a more general way to include other recognition tasks such as regression. The key problem of supervised learning is to derive from a finite training set of patterns and their labels $\{(x_i, y_i)\}_{i=1\ldots m} \subseteq \mathcal{X} \times \mathcal{Y}$ a prediction function $f : \mathcal{X} \to \mathcal{Y}$ assigning patterns $x \in \mathcal{X}$ to classes $y \in \mathcal{Y}$, that is, a function classifying patterns. Given such a prediction function, or *classifier*, unknown patterns $x' \in \mathcal{X}$ can then be assigned to the class $f(x') \in \mathcal{Y}$. Obviously, the overall objective is to derive classifiers that are able to correctly classify not only the patterns from the training set $\{x_i\}_{i=1\ldots m}$, but also unseen patterns from the underlying population $\mathcal{X}$. This property is generally referred to as *generalization* ability of classifiers.

To quantify the amount of generalization achieved by different classifiers, the definition of supervised learning is commonly formulated in the context of an underlying distribution of patterns and labels [Schölkopf and Smola (2002)]. In such a model, patterns are related to their class labels

by a joint probability measure $P(x, y)$ on $\mathcal{X} \times \mathcal{Y}$. This probability measure is generally unknown, but is assumed to be consistent, that is, it does not change over time. Hence, the only assumption in this learning model is that patterns and their labels are randomly drawn according to an underlying probability distribution over the whole population of patterns and class labels, and the randomly drawn training set of patterns is the only information the learning algorithm has access to. The classifier learning task can then be regarded as deriving from a randomly drawn set of (independent and identically distributed) training patterns a prediction function that will perform well on the entire population of patterns.

### 4.1.1 *Empirical Risk Minimization*

In the following, assume that a set of potential classifiers $\mathcal{F} = \{f \mid f : \mathcal{X} \to \mathcal{Y}\}$ is given. The classifier learning task can then be formalized as the selection of a single classifier from $\mathcal{F}$. For instance, if Bayes classifiers based on Gaussian distributions are used, the set $\mathcal{F}$ can be regarded as the set of all Bayes classifiers with arbitrary mean vectors and covariance matrices. The supervised learning task is then equivalent to selecting from $\mathcal{F}$ the one classifier whose parameters correspond best to the training set of patterns at hand.

Since the overall objective is to derive classifiers that perform well on independent test sets, the learning task can be formalized as the search for a classifier in $\mathcal{F}$ with minimum error on the test set. Formally, the aim is to minimize the *expected risk* [Schölkopf and Smola (2002)],

$$R[f] = \int_{\mathcal{X} \times \mathcal{Y}} V(y, f(x))\, dP(x, y) \ , \tag{4.1}$$

where $V : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$ is a loss function evaluating how accurately $f(x)$ estimates the class $y$. A standard loss function for binary classification problems is the zero-one-loss function $V_{01}$ returning 0 if $f(x) = y$ and 1 if $f(x) \neq y$. The expected risk $R[f]$ measures the degree of error that is to be expected for classifier $f$ on the whole population. Hence, the classifier that could be expected to perform best would be the one from $\mathcal{F}$ that minimizes the expected risk $R[f]$. Unfortunately, the computation of $R[f]$ is intractable, since the distribution $P(x, y)$ of the population is unknown. However, if we restrict ourselves to the known set of training patterns $\{(x_i, y_i)\}_{i=1\ldots m}$, the analogous term, called the *empirical risk*, can

easily be computed,

$$R_{emp}[f] = \frac{1}{m} \sum_{i=1}^{m} V(y_i, f(x_i)) \ . \tag{4.2}$$

The empirical risk $R_{emp}[f]$ quantifies how well classifier $f$ performs on the training set. If the zero-one-loss function is used, the empirical risk is equal to the error rate on the training set. Unlike the unknown expected risk, the empirical risk can readily be computed, and it may therefore be tempting to approximate the expected risk with the empirical risk. The resulting supervised learning procedure hence selects from the set of classifiers $\mathcal{F}$ the one $f^*$ that performs best on the training set,

$$f^* = \arg\min_{f \in \mathcal{F}} R_{emp}[f] \ , \tag{4.3}$$

although the actual objective is in fact to minimize the expected risk $R[f]$. This induction principle is called *empiricial risk minimization* [Vapnik and Chervonenkis (1971)].

The empirical risk minimization method seems to be appealing because of its conceptual simplicity. However, it turns out that using empirical risk minimization for classifier selection may lead to classifiers that are seriously constrained in their ability to generalize well on unseen data. That is, the empirical risk is often a weak approximation of the expected risk. One caveat is that in cases where only a limited number of patterns is available, empirical risk minimization will not be appropriate. It can be shown that the probability that empirical risk and expected risk differ increases exponentially for a decreasing number of training patterns [Schölkopf and Smola (2002)]. Even more important is the fact that empirical risk minimization is focussed exclusively on the set of training patterns. The overall objective of supervised learning is not to capture the training data as accurately as possible, but rather to capture the characteristics of the underlying classes. The problem of classifiers that are too strongly adapted to the training set to be able to generalize well on unseen data from the same distribution is called *overfitting*. For instance, in the development of learning algorithms for artificial neural networks, a huge effort has been put into defining ways to avoid the problem of overfitting, for instance by reducing the number of neurons in hidden layers or terminating the training before convergence [Bishop (1996)]. Conversely, *underfitting* occurs in cases where the classifier is unable to model class boundaries with sufficient complexity, that is, where the prediction function is overly simple.

### 4.1.2 *Structural Risk Minimization*

What is needed is a prediction function that constitutes a trade-off between underfitting and overfitting, that is, a classifier that is only as complex as necessary so as to be able to capture the boundaries of the underlying classes. The key idea of the *structural risk minimization* induction principle discussed in this section is to quantify the degree of underfitting and overfitting and determine which classifier can be expected to achieve the best performance on unseen data.

For the formal definition of the complexity of classifiers, the term *capacity* is used. The capacity describes the ability of a classifier to learn any training set without error. Hence, the capacity is a measure of flexibility or complexity of classifiers. For instance, if three patterns are given, there are eight ways to assign these patterns to two classes. A classifier that is able to learn and correctly classify the three patterns for each of these eight labelings has a higher capacity than one that is unable to. Hence high-capacity classifiers are able to learn and flawlessly classify almost any set of training patterns, but they may perform poorly on new data. One of the most common capacity measures is the Vapnik-Chervonenkis dimension (VC dimension) [Vapnik (1998)]. If two class labels $\{-1, +1\}$ are given and a classifier is able to learn all $2^m$ possible ways to label $m$ patterns with $-1$ or $+1$, the classifier is said to *shatter* these $m$ points. The largest $m$ for which there exists a set of $m$ patterns which the classifier can shatter is called the classifier's VC dimension.

It should be noted that there are many other formal concepts for measuring the capacity of classifiers [Schölkopf and Smola (2002); Shawe-Taylor and Cristianini (2004)]. Statistical learning theory shows that it is crucial to restrict the set of functions from which a classifier is selected. The suggested procedure is to consider only classes of functions whose capacity is suitable for the amount of training data available. Hence, it may still be important that a classifier is able to correctly classify most training patterns, but not all of them. Limiting the classifier capacity reduces the influence of outliers in the training data, which is expected to improve the classification accuracy on unseen data drawn from the same distribution.

In the following example, it is demonstrated how the size of the training set and the classifier capacity affect the recognition performance on the training set and on an independent test set. To this end, a two-class problem ($\mathcal{Y} = \{-1, +1\}$) of one-dimensional data ($\mathcal{X} = \mathbb{R}$) is considered. The number of training patterns is varied from 10 to 500 patterns per class.

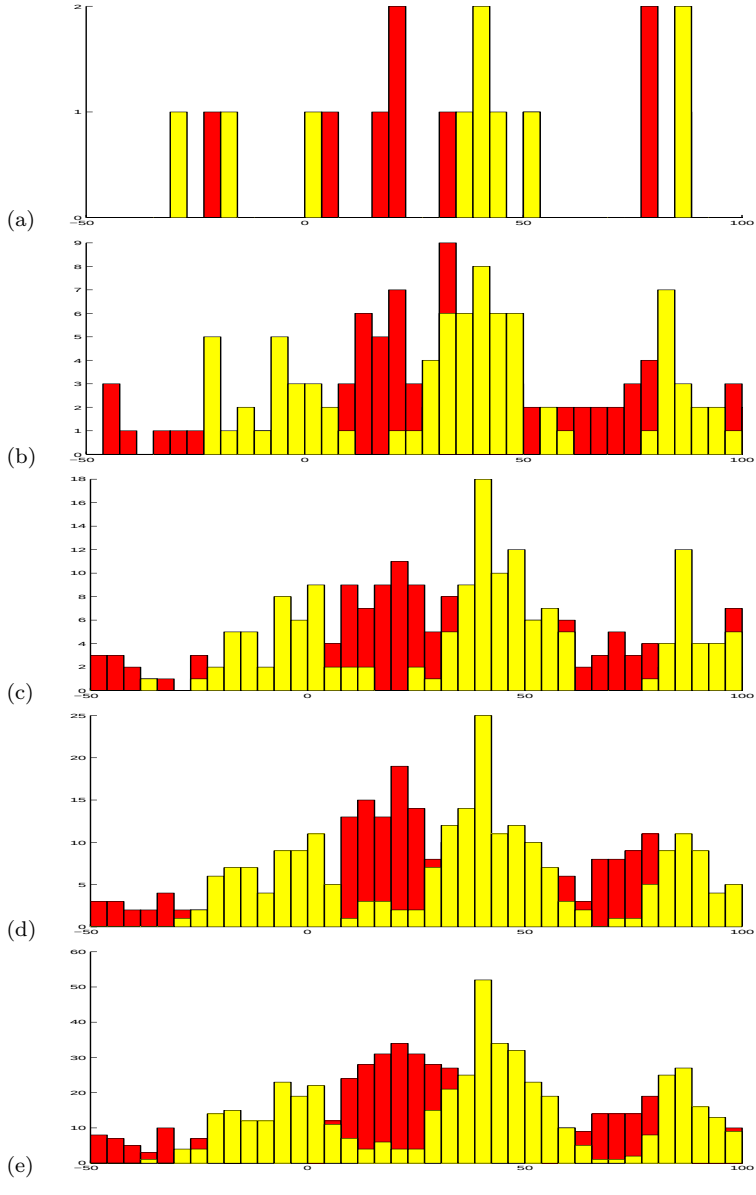



Fig. 4.1 Distribution of two classes. Sample sets of size (a) 10, (b) 80, (c) 150, (d) 220, and (e) 500

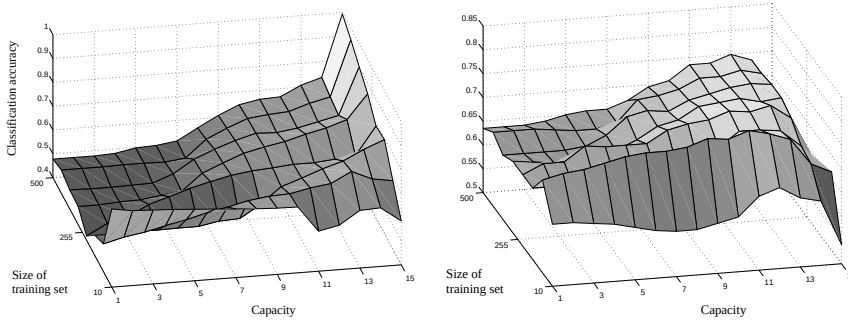
Illustrations of the distribution of training sets of various size are provided in Fig. 4.1. Obviously, the larger the training sets are, the more apparent becomes the underlying distribution of the two classes. In this experiment, a simple approach using Parzen windows [Duda *et al.* (2000); Parzen (1962)] is used for classification. Given a test pattern, the idea of classification based on Parzen windows is to compute how many training patterns of either class are close to the test pattern. Using RBF functions, the Parzen windows classification rule $f : \mathcal{X} \to \mathcal{Y}$ can be written

$$f(x) = \operatorname{sign}\left(\sum_{i=1}^{m} y_i \cdot \exp\left(-\frac{\|x - x_i\|^2}{\sigma^2}\right)\right) \tag{4.4}$$

for a training set $\{(x_i, y_i)\}_{i=1\ldots m} \subseteq \mathcal{X} \times \mathcal{Y}$ and a test pattern $x \in \mathcal{X}$. The Parzen windows classifier assigns a pattern to the first class if its accumulated similarity to training patterns from the first class is higher than the one to training patterns from the second class.

For very small values of the Parzen windows parameter $\sigma$, only training patterns that are located very close to the test pattern will be taken into account, since for more distance patterns the term in the exponential function will be strongly negative and the resulting contribution to the sum only marginal. Conversely, for very large values of $\sigma$, the whole set of training patterns will be considered, and the actual distances to the test pattern will become less important. In view of this, parameter $\sigma$ can be understood as a term controlling the capacity of the Parzen windows classifier. Informally, it is clear that large values of $\sigma$ correspond to lower capacities of the classifier, since the number of class labelings that can be learned, as well as the influence of outliers, is small in these cases. Similarly, low values of $\sigma$ correspond to high capacities, because for very low values the classifier will only take the closest training pattern of the test pattern into account, which is equivalent to a maximal number of class labelings that can be learned. In the following illustrations, the Parzen windows parameter $\sigma$ will not explicitly be mentioned, but a capacity parameter $h$ related to $\sigma$ (such that a small $\sigma$ corresponds to a large $h$, and vice versa) will be used instead.

For an illustration of the influence of the size of the training set and the capacity on the recognition performance, the Parzen windows classifier is applied to the full training set of size 500 (as if it were a test set) and to an independent test set of size 100. The resulting classification accuracy on the training set and test set is illustrated in Fig. 4.2a and 4.2b, respectively. First, a certain number of training patterns ($m = 10, 80, 150, \ldots, 500$) are




(a) (b)

Fig. 4.2 Classification accuracy on (a) training set and (b) test set

randomly selected from the training set of size 500. Based on these training sets, the Parzen windows classifier is then applied to the full training set of 500 patterns for Fig. 4.2a, and to the full test set of 100 patterns for Fig. 4.2b. This procedure is carried out for values of the capacity parameter $h = 1, 2, \ldots, 15$ (being in fact equivalent to Parzen windows parameters $\sigma = 10, 9.3, 8.6, 7.9, \ldots, 0.8, 0.1$). The classification accuracy for all configuration pairs $(m, h)$ is shown in Fig. 4.2. On the training set, it can be observed that the performance tends to be better for larger training sets. This behavior is clearly reasonable, since subsets of size $m = 10, 80, 150, \ldots, 500$ of the training set are used to classify the patterns from the full training set of size 500. Obviously, if the Parzen windows classifier knows all the patterns to be classified ($m = 500$), the classification accuracy will be higher than in cases where the classifier knows only 2% of the patterns ($m = 10$). On the test set, it turns out that a certain minimum number of training patterns is required in order to obtain satisfactory classification rates. This can be explained by the observation that in Fig. 4.1a (with $m = 10$ training patterns per class) it is difficult to intuitively derive class boundaries, whereas in Fig. 4.1c (with $m = 150$) the class regions are already clearly visible.

The part that is more relevant for the discussion in this chapter is the influence of the capacity on the performance. On the training set, it turns out that the higher the capacity is, the better the classifier performs (Fig. 4.2a). On the other hand, if the independent test set is considered, classifiers with medium capacity clearly outperform those with high capacity (Fig. 4.2b).

As expected, for the classification of the training patterns, it is best to increase the classifier capacity as much as possible, so that each individual pattern of the training set has a strong influence on the resulting classification. For the classification of the test patterns, a capacity around $h = 13$ (corresponding to $\sigma = 1.5$) seems to be optimal. For smaller capacities, the Parzen windows classifier is unable to identify class boundaries because the decision function is too simple, which is equivalent to underfitting. For larger capacities, the classifier is too strongly adapted to the actual training set and does not lead to generally valid boundaries of the underlying classes, which means that the classifier overfits the training data.

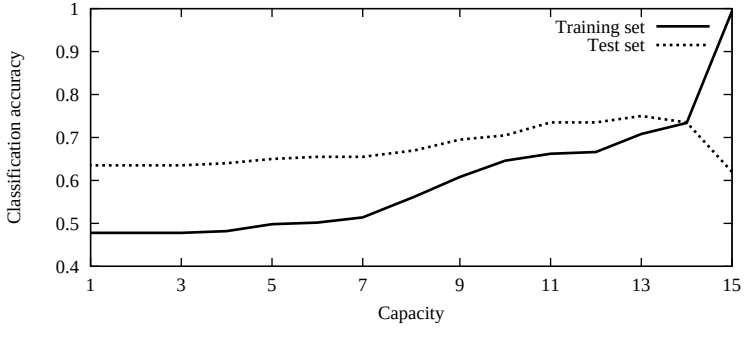



Fig. 4.3 Classification accuracy on training set and test set

An illustration of the performance results for $m = 500$ is shown in Fig. 4.3. The important observation is that the performance on the training set is best for maximal capacity, whereas the performance on the test set is best for medium capacity. If we regard the empirical risk as being equivalent to the training error, which is satisfied in case of the zero-one-loss function, we find that optimizing the classifier on the training set will not lead to a satisfactory recognition of independent test patterns from the same distribution. That is, the resulting classifier will not able to generalize well on unseen data. In this respect, empiricial risk minimization seems to be too restricted to be universally applicable. Using the empirical risk minimization principle, the classifier with maximal capacity ($h = 15$) is chosen, which corresponds to minimum empirical risk. If structural risk minimization is employed instead, the capacity of classifiers to be considered is limited to a certain interval, depending on the number of training patterns available. In this case, chances are that the structural risk mini-

mization principle would select a classifier from the optimal range around $h = 13$, such that overfitting and underfitting could largely be avoided.

The main objective in statistical learning theory is to be able to define in quantitative terms which class of functions will be appropriate for specific pattern recognition problems. For this purpose, one can apply results from statistical learning theory. To give one example, if the VC dimension of a classifier is known, it is possible to compute, for a given probability, an upper bound of the difference between empirical risk and expected risk. If $h$ denotes the VC dimension of the classifier under consideration and $m$ is the number of training patterns, then the bound [Schölkopf and Smola (2002)]

$$R[f] \leq R_{emp}[f] + \sqrt{\frac{1}{m}\left(h\left(\log\frac{2m}{h}+1\right)+\log\frac{4}{\delta}\right)} \tag{4.5}$$

holds with probability of at least $1-\delta$ over the distribution of the training sample. The second addend in the formula above, sometimes referred to as *capacity term*, increases monotonically with the VC dimension $h$. For a graphical illustration of the capacity term, see Fig. 4.4. It should be noted that in this example small values of the bound are the optimal ones, in contrast to the classification accuracy in Fig. 4.2. Recall that the expected risk $R[f]$ can be regarded as test error and the empirical risk $R_{emp}[f]$ as training error, which means that the aim is to make the expected risk $R[f]$, and thus the test error, small. For classifiers with a large capacity, it will not be possible to obtain a tight bound on the test error, because the capacity term will be large. That is, if $h$ is large, a small training error does not guarantee the test error to be small as well. Clearly, the result is that the bound on the test error will be tighter for low capacities.

The key idea of structural risk minimization is to use bounds on the test error, such as the one given in Eq. (4.5), to determine which classifiers provide a good trade-off between complexity and generality, so as to classify not only patterns from the known training set correctly, but also unknown test patterns. The bound clearly shows that in order to obtain a small bound on the test error, classifiers must perform well on the training data and exhibit a low capacity. That is, structural risk minimization favors low-capacity classifiers that are able to learn the decision boundaries of the training set sufficiently well. For a schematic example, consider the illustration in Fig. 4.5. The training error, or empirical risk $R_{emp}$, is assumed to decrease for increasing capacity. The capacity term can then be computed for various values of the VC dimension $h$. The smallest train-

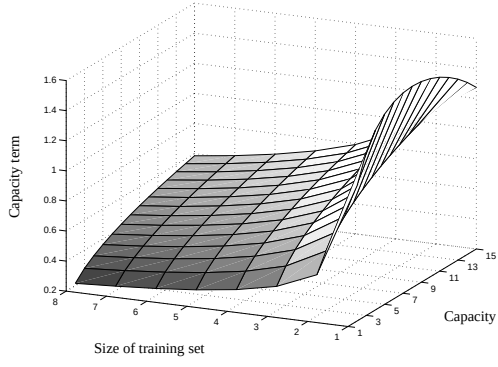



Fig. 4.4 Capacity term with respect to size of training set and capacity

ing error is obtained for large capacities, whereas the lowest value of the capacity term is obtained for small capacities. These two arguments are then combined in Eq. (4.5) to give the bound on the test error. Structural risk minimization then selects the classifier that minimizes this bound on the test error. Hence, structural risk minimization reflects the trade-off between small training errors and the ability to generalize well on unseen data.

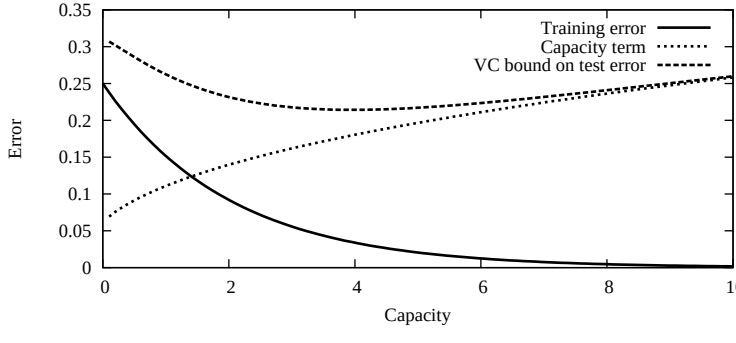



Fig. 4.5 Bound on test error based on training error

In practice, capacity measures are usually difficult to compute. However, one of the most popular kernel machines, the support vector machine (SVM), smoothly integrates principles of structural risk minimization into its training algorithm. The ideas behind structural risk minimization are therefore crucial for understanding how SVM classification works and why

SVMs often generalize well on unseen data. Hence, statistical learning theory provides strong arguments for the use of SVMs and may serve as an explanation for the fact that in many cases SVMs are able to outperform traditional classifiers in completely different application areas. An introduction to SVMs and other kernel machines is provided in Sec. 4.3.

## 4.2 Kernel Functions

In this section, kernel functions for pattern recognition are introduced. Kernel functions offer a very elegant way to construct non-linear extensions of linear algorithms for pattern analysis and recognition by mapping patterns into a kernel feature space. In the following, properties of valid kernels are discussed and the feature space embedding by means of kernel functions is described.

### 4.2.1 *Valid Kernels*

Kernel functions are used to extract information from patterns that is relevant for classification. The basic idea of kernel functions is to define pattern similarity measures satisfying certain conditions that eventually lead to a powerful interpretation of the kernel function in terms of geometrical properties. For kernel functions to be valid, the conditions of symmetry and positive definiteness [Schölkopf and Smola (2002); Berg *et al.* (1984)] must be satisfied. For the sake of completeness, note that positive definite functions according to the definition below are sometimes called positive semi-definite.

**Definition 4.1 (Positive Definite Kernel).** *A kernel function is a symmetric function mapping pairs of patterns to real numbers, $k : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$. A kernel function $k$ is positive definite if, for all $m \in \mathbb{N}$,*

$$\sum_{i,j=1}^{m} c_i c_j k(x_i, x_j) \geq 0 \quad \begin{array}{l} \textit{for all } \{c_1, \ldots, c_m\} \subseteq \mathbb{R}, \\ \textit{and } \{x_1, \ldots, x_m\} \subseteq \mathcal{X} \end{array} .$$

*An $m \times m$-matrix $K$ of real numbers constructed from a positive definite kernel function $k$ and elements $\{x_1, \ldots, x_m\} \subseteq \mathcal{X}$ according to $K_{ij} = k(x_i, x_j)$ is called kernel matrix, or Gram matrix. Kernel functions that are symmetric and positive definite are often called valid kernels, admissible kernels, or Mercer kernels.*

The property of positive definiteness is crucial for the definition of kernel machines and turns out to be sufficiently strong to implicate a considerable number of theoretical properties associated with kernel functions. To verify whether a function is positive definite, one can check, of course, if the condition stated above is satisfied. Alternatively, an equivalent condition is that all eigenvalues of the kernel matrix associated with the kernel function are non-negative. Also, if there exists an $n \times m$-matrix $B$ (where $n \leq m$) such that $K = B^T B$, the kernel matrix $K$ is known to be positive definite as well. Finally, a kernel function is positive definite if all principal minors of the associated kernel matrix are non-negative. The principal minors of a matrix are the determinants of its submatrices obtained by removing a certain number of columns and the same rows.

A larger class of kernels closely related to positive definite functions are conditionally positive definite kernels.

**Definition 4.2 (Conditionally Positive Definite Kernel).** *A kernel function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is conditionally positive definite if, for all $m \in \mathbb{N}, m \geq 2$,*

$$\sum_{i,j=1}^{m} c_i c_j k(x_i, x_j) \geq 0 \quad \begin{array}{l} \textit{for all } \{c_1, \ldots, c_m\} \subseteq \mathbb{R} \\ \textit{satisfying } \sum_{i=1}^{m} c_i = 0, \\ \textit{and } \{x_1, \ldots, x_m\} \subseteq \mathcal{X} \ . \end{array}$$

Obviously, every positive definite kernel is conditionally positive definite as well. Conditionally positive definite kernels constitute a larger class of kernels that can sometimes be used as substitutes for positive definite kernels in cases where these are not available. From given kernels one can easily construct derived kernels, as pointwise addition and multiplication preserves positive definiteness (and conditional positive definiteness).

**Lemma 4.1 (Closure Properties).** *If $k_1, k_2$ are positive definite kernels, $c_1, c_2 \geq 0$, and $b \in \mathbb{R}$, then the kernel functions defined by*

$$\begin{aligned} k_a(x, x') &= c_1 k_1(x, x') + c_2 k_2(x, x') \\ k_b(x, x') &= k_1(x, x') \cdot k_2(x, x') \\ k_c(x, x') &= k_1(x, x') + b \end{aligned}$$

*are positive definite ($k_a, k_b$) and conditionally positive definite ($k_c$). For more closure properties, such as those related to pointwise convergence, tensor products, direct sums, and convolution, see [Berg* et al. *(1984)].*

Table 4.1 Sample (a) positive definite and (b) conditionally positive definite kernel functions on real vectors

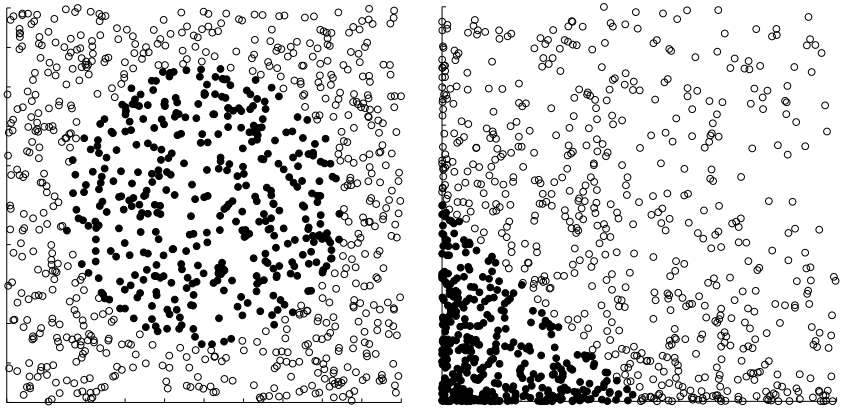
| | Kernel | Definition | Parameter |
|---|---|---|---|
| a) | RBF kernel | $k(x,x') = \exp\left(-\frac{\|x-x'\|^2}{\sigma^2}\right)$ | $\sigma > 0$ |
| | Polynomial kernel | $k(x,x') = (\langle x,x'\rangle + c)^d$ | $d \in \mathbb{N}, c \geq 0$ |
| | Sigmoid kernel | $k(x,x') = \tanh(\alpha\langle x,x'\rangle + \beta)$ | $\alpha > 0, \beta < 0$ |
| b) | Norm kernel | $k(x,x') = -\|x-x'\|^\beta$ | $0 \leq \beta \leq 2$ |
| | Log kernel | $k(x,x') = -\log(1+\|x-x'\|)$ | |

For a given pattern recognition problem to be addressed by kernel machines, one of the most difficult parts is the definition of a kernel function that extracts the appropriate amount of information from the underlying patterns. For the common case of patterns being represented by feature vectors, a number of standard valid kernels are widely used [Schölkopf and Smola (2002)]. Some of them are given in Table 4.1.

### 4.2.2 *Feature Space Embedding and Kernel Trick*

A large number of linear methods for pattern analysis and classification have been developed addressing various basic problems such as the linear separation of data or the detection of dominant linear relations in data sets. These methods are quite useful for understanding and processing the data at hand, but they are inherently limited because of their linear nature. Whenever non-linear regularities occur, these linear methods will fail.

A universal solution to this basic problem is to modify the problem domain such that non-linear regularities are turned into linear ones. For an example, consider the two-class problem illustrated in Fig. 4.6a. Obviously, the white dots can only be separated from the black dots by a second-order function with respect to the two features. Hence, applying linear classifiers to this data set will always be insufficient. In this particular example, however, the problem of non-linearity can easily be solved [Schölkopf and Smola (2002)] by mapping the two-dimensional pattern space $\mathcal{X} = \mathbb{R}^2$ into a three-dimensional vector space $\mathcal{H} = \mathbb{R}^3$, that is $\Phi : \mathcal{X} \to \mathcal{H}$, according to $\Phi(x_1, x_2) = (x_1^2, \sqrt{2}x_1x_2, x_2^2)$. The resulting pattern distribution is illustrated in Fig. 4.6b. Note that the horizontal axis represents the component $x_1^2$ and the vertical axis the component $x_2^2$ (the remaining component $\sqrt{2}x_1x_2$ will be needed in an example following below). Clearly, by applying a non-linear transformation to the patterns, the classes are turned into



(a) (b)

Fig. 4.6 Mapping (a) the problem domain to (b) a feature space

linearly separable classes. Whereas in the original pattern space a second-order function is needed for separating the classes, in the transformed space a linear function (a plane) is sufficient.

The general procedure of mapping data into a space that is better suited for the recognition task under consideration seems to be quite reasonable. The difficult problem to be solved in this context is the definition of the high-dimensional space the patterns are mapped into and the mapping function itself. In fact, Cover's theorem [Cover (1965)] states that a set of patterns, associated with a complex pattern recognition problem, cast non-linearly into a high-dimensional space is more likely to be linearly separable in this space than in the original low-dimensional space. This means that even if the properties a high-dimensional feature space should exhibit to render the recognition problem easier are unknown, the mere construction of a such a feature space in conjunction with a non-linear mapping is on the average advantageous.

Many algorithms for pattern analysis and classification are defined in geometrical terms, such as distances to prototype patterns, projections of the pattern space into lower-dimensional subspaces, linear combinations of patterns, convex hulls of sets of patterns, etc. Consequently, the ability to compute geometrical terms in the feature space where the recognition problem is to be addressed is of key importance. In view of this, an important observation is that some functions measuring how patterns are related

to each other in geometrical terms can be expressed by means of inner products, provided that the feature space is in fact a vector space.

**Definition 4.3 (Inner Product, Dot Product, Scalar Product).** *An inner product in a vector space $\mathcal{H}$ is a function $\langle .,. \rangle : \mathcal{H} \times \mathcal{H} \to \mathbb{R}$ satisfying*

$$\begin{aligned} &\langle x, x' \rangle = \langle x', x \rangle && \textit{(Symmetry)} \\ &\langle \alpha x + \beta x', x'' \rangle = \alpha \langle x, x'' \rangle + \beta \langle x', x'' \rangle && \textit{(Bilinearity)} \\ &\langle x, x \rangle = 0 \quad \textit{for } x = 0 \\ &\langle x, x \rangle > 0 \quad \textit{for } x \neq 0 \ , \end{aligned}$$

*for vectors $x, x', x'' \in \mathcal{H}$ and scalars $\alpha, \beta \in \mathbb{R}$. In case of $\mathcal{H} = \mathbb{R}^k$, a standard inner product is defined by $\langle x, x' \rangle = \sum_{i=1}^{k} x_i x'_i$. If a vector space is provided with an inner product, one can derive*

$$\begin{aligned} \|x\| &= \sqrt{\langle x, x \rangle} && \textit{(Norm)} \\ \|x - x'\| &= \sqrt{\langle x, x \rangle + \langle x', x' \rangle - 2\langle x, x' \rangle} && \textit{(Distance)} \\ \angle(x, x') &= \arccos \tfrac{\langle x, x' \rangle}{\|x\| \cdot \|x'\|} && \textit{(Angle)} \ . \end{aligned}$$

Inner products can be regarded as pattern similarity measures. For instance, the standard inner product of two vectors is maximal if the two vectors point in the same direction, and minimal if they point in opposite directions. Continuing with the example presented above, the standard inner product of two patterns mapped from $\mathcal{X} = \mathbb{R}^2$ to $\mathcal{H} = \mathbb{R}^3$ according to

$$\Phi(x_1, x_2) = (x_1^2, \sqrt{2} x_1 x_2, x_2^2)$$

amounts to

$$\begin{aligned} \langle \Phi(x), \Phi(x') \rangle &= \langle \Phi(x_1, x_2), \Phi(x'_1, x'_2) \rangle \\ &= x_1^2 x_1'^2 + 2 x_1 x_2 x'_1 x'_2 + x_2^2 x_2'^2 \\ &= (x_1 x'_1 + x_2 x'_2)^2 \\ &= \langle x, x' \rangle^2 \ . \end{aligned}$$

This means that from the inner product in the original space $\mathcal{X}$ one can infer the inner product in the feature space $\mathcal{H}$ without actually mapping the patterns into the feature space. While it may not seem very spectacular to derive from the inner product of a two-dimensional space the inner product of a three-dimensional space, a similar method can be formulated for a general class of functions and for high-dimensional, and even infinitely-dimensional, vector spaces.

**Theorem 4.1.** *[Schölkopf and Smola (2002); Shawe-Taylor and Cristianini (2004)] Let $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ be a valid kernel. There exists a vector space $\mathcal{H}$ endowed with an inner product $\langle .,. \rangle : \mathcal{H} \times \mathcal{H} \to \mathbb{R}$ and a mapping $\Phi : \mathcal{X} \to \mathcal{H}$ such that*

$$k(x, x') = \langle \Phi(x), \Phi(x') \rangle \quad \textit{for all } x, x' \in \mathcal{X} \ .$$

Although this theorem appears to be quite simple, its impact on various pattern recognition methods is immense. Instead of mapping patterns from $\mathcal{X}$ to the feature space $\mathcal{H}$ and computing their inner product in $\mathcal{H}$, one can simply evaluate the value of the kernel function in $\mathcal{X}$. Conversely, every kernel function can be understood as an inner product in an explicitly existing feature vector space.

From the condition of positive definiteness (and symmetry), it is guaranteed that any valid kernel function represents in fact a richer structure of vectors and geometrical entities, even if the original pattern space $\mathcal{X}$ does not contain any mathematical structure beyond the kernel function. In such cases, the kernel approach allows us to apply well-known algorithms defined in vector spaces to non-vectorial data in a unified way. The average of a set of graphs, for instance, cannot be computed in the space of graphs, but computing the average of a set of vectors representing graphs is trivial. Moreover, the structure of the feature vector space is defined entirely in terms of the kernel function, or kernel similarity measure. That is, in order to obtain an embedding of the pattern space $\mathcal{X}$ into a vector space $\mathcal{H}$, one simply needs to define a pattern similarity measure in $\mathcal{X}$ and verify that this function is symmetric and positive definite. The geometry of $\mathcal{H}$ is then based on this underlying pattern similarity measure.

By way of this theorem, every algorithm formulated entirely in terms of an inner product, or generally a symmetric positive definite function, can be turned into a different algorithm by replacing the inner product by any other valid kernel function. This procedure is commonly referred to as *kernel trick* [Schölkopf and Smola (2002); Shawe-Taylor and Cristianini (2004)]. The kernel trick is admissible, since any valid kernel function can be understood as an inner product in some feature vector space. By means of this approach, it is possible to run algorithms for pattern analysis or classification in an implicitly existing vector space without explicitly computing vectors or inner products. For instance, the vectors in the feature vector space may consist of infinitely many dimensions, but the inner product of such infinitely large vectors reduces to the kernel function defined beforehand. In view of this, kernel functions can be seen as shortcuts

for the efficient computation of inner products in high-dimensional spaces regardless of their dimension.

In the context of practical pattern recognition methods, the kernel trick is of key importance because it allows for an extension of linear to non-linear algorithms in a very straightforward manner. Algorithms formulated in terms of inner products are of inherently linear nature. For instance, principal component analysis can be used to identify linear regularities in data sets and decorrelate data consisting of linearly dependent components. If the kernel trick is applied to principal component analysis, one obtains a modified principal component analysis, so-called kernel principal component analysis, that is able to detect non-linear regularities. Hence, kernel principal component analysis can be used to extract higher-order dominant directions in data sets. In practice, the non-linear modification of the standard principal component analysis method simply consists of feeding a kernel matrix into the algorithm instead of a matrix of inner products. Moreover, if any other kernel function is used, again a new variant of principal component analysis with completely different characteristics is obtained. Its conceptual simplicity is what makes the kernel approach extremely powerful.

The kernel trick is applicable to a large number of important algorithms, including support vector machines, principal component analysis, Fisher discriminant analysis, nearest-neighbor classifier, K-means clustering, self-organizing maps, canonical correlation analysis, partial least squares regression, and many more [Schölkopf and Smola (2002); Shawe-Taylor and Cristianini (2004)]. The first three of these methods will be discussed in greater detail in Sec. 4.3.

## 4.3 Kernel Machines

This section describes three common kernel machines: the support vector machine, kernel principal component analysis, and kernel Fisher discriminant analysis. The support vector machine is one of the most popular kernel machines for pattern classification. Support vector machines are interesting because their training procedure is based on results from statistical learning theory, and their performance in contrast to traditional classifiers is very convincing on various data sets. Kernel principal component analysis and kernel Fisher discriminant analysis are non-linear variants of standard feature extraction algorithms.

### 4.3.1 *Support Vector Machine*

Pattern classification by means of support vector machines (SVMs) is based on an intuitive geometrical method for the separation of two classes [Schölkopf and Smola (2002); Shawe-Taylor and Cristianini (2004); Vapnik (1998)]. The SVM training can be interpreted as a structural risk minimization process, so that the resulting classifier can be expected to generalize well on unseen data (see Sec. 4.1.2). Since the SVM training procedure can be formulated entirely in terms of inner products, the kernel trick can be applied to the basic SVM method (see Sec. 4.2.2), which makes SVMs extremely flexible and applicable to non-vectorial pattern domains.

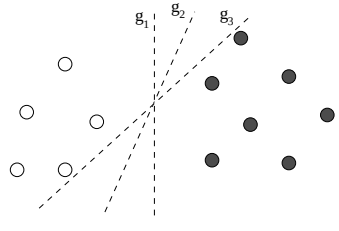



Fig. 4.7 Three example hyperplanes separating two classes

The basic idea of SVMs is to separate classes of patterns by hyperplanes. These hyperplanes are placed between two classes such that their distance to the closest pattern of either class is maximal. Such hyperplanes are commonly called maximum-margin hyperplane. In Fig. 4.7, three hyperplanes are shown correctly separating two classes. Hyperplane $g_1$ is the one with maximum margin, that is, with maximum distance to the closest patterns, and would intuitively be considered the most suitable hyperplane separating the two classes.

In the following, let us assume that a pattern space $\mathcal{X} = \mathbb{R}^n$, two classes $\mathcal{Y} = \{-1, +1\}$, and a labeled training set $\{(x_i, y_i)\}_{i=1\ldots m} \subseteq \mathcal{X} \times \mathcal{Y}$ is given. If the two classes are linearly separable, there exist parameters $w \in \mathbb{R}^n$ and $b \in \mathbb{R}$ such that the hyperplane

$$g(x) = \langle w, x \rangle + b \tag{4.6}$$

correctly separarates the two classes. That is, this hyperplane-defining function will evaluate to $g(x) = -1$ for patterns $x$ from the first class and to $g(x) = +1$ for patterns $x$ from the second class. Correspondingly, the classification of unknown elements $x'$ is performed according to the sign of $g(x')$. The condition of linear separability of the training set can also be

written

$$y_i \cdot (\langle w, x_i \rangle + b) > 0 \quad \text{for } i = 1, \ldots, m \ . \tag{4.7}$$

For each choice of parameters $w \in \mathbb{R}^n$ and $b \in \mathbb{R}$ satisfying the condition above, the training error will be zero, and the empirical risk optimal. Hence, the objective is to select from those classifiers separating the training patterns correctly the one that is expected to perform best on an independent test set.

In view of the structural risk minimization induction principle discussed in Sec. 4.1.2, the capacity of hyperplane classifiers defined according to Eq. (4.6) needs to be investigated for this purpose. First, since multiplying the parameters $w$ and $b$ with a positive constant does not change the hyperplane, a *canonical form* of hyperplanes is obtained by rescaling the parameters such that $\min_{i=1\ldots m} |\langle w, x_i \rangle + b| = 1$. For canonical hyperplanes, the VC dimension measure of capacity can explicitly be quantified [Vapnik (1998)].

**Theorem 4.2.** *Let* $\{(x_i, y_i)\}_{i=1\ldots m} \subseteq \mathcal{X} \times \mathcal{Y}$ *with* $\mathcal{X} = \mathbb{R}^n$ *and* $\mathcal{Y} = \{-1, +1\}$ *be a training set of patterns, and let* $g(x) = \langle w, x \rangle + b$ *be a hyperplane satisfying* $\|w\| \leq \Lambda$ *and being canonical with respect to the training data. If* $R > 0$ *is the radius of the smallest sphere around the training data, the VC dimension* $h$ *of the canonical hyperplane* $g(x)$ *satisfies*

$$h \leq \min(R^2 \Lambda^2 + 1, n + 1) \ .$$

This theorem states that by constraining the length of the hyperplane weight vector $w$, the VC dimension, and thus the capacity of the corresponding hyperplane classifier, can be bounded from above. Hence, following the structural risk minimization principle, the objective is to find a hyperplane with bounded length of the weight vector that performs well on the training data.

For a different perspective, the length of the weight vector can also be interpreted geometrically. The distance of any pattern $x' \in \mathcal{X}$ to the hyperplane $g(x) = \langle w, x \rangle + b$ is given by $g(x')/\|w\|$. Hence, if $g(x)$ is given in canonical form, the distance of the points from either class closest to the hyperplane amounts to $1/\|w\|$. The distance of the closest points to the hyperplane is commonly termed the *margin* of the hyperplane with respect to the training data. Hence, the smaller the length of the weight vector $\|w\|$, the larger is the margin, and the smaller is the capacity. Intuitively, large-margin hyperplanes are definitely to be favored over small-margin hyperplanes, since a larger margin corresponds to a better separability of

classes. An illustration of this idea is shown in Fig. 4.8. In Fig. 4.8a, a hyperplane with a large margin is placed between the two classes. Obviously, it is impossible to separate patterns of the given sample in a different manner by another hyperplane. On the other hand, the small margin hyperplanes in Fig. 4.8b are able to separate patterns in a large variety of ways. The VC dimension capacity measure is based on the number of labelings a classifier can achieve. Hence, the capacity of the hyperplane in Fig. 4.8a is significantly lower than that of the hyperplanes in Fig. 4.8b.

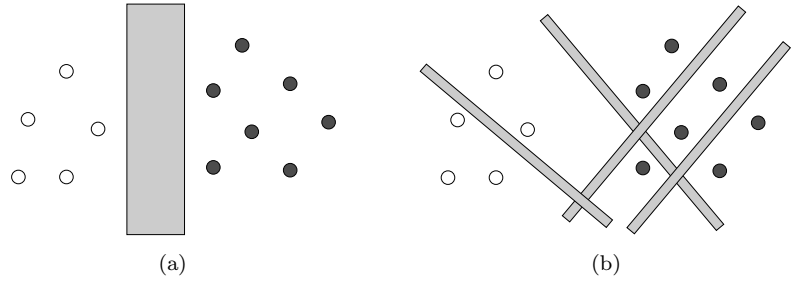



Fig. 4.8 Correspondence of margin and VC dimension. (a) Large margin and low capacity in contrast to (b) small margin and high capacity

The objective of structural risk minimization is to minimize the empirical risk using classifiers with a capacity suitable for the amount of training data available (see Sec. 4.1.2). More concretely, in the case of SVMs, the term to be optimized is

$$\begin{aligned} &\min_{w \in \mathbb{R}^n} \frac{1}{2}\|w\|^2 \\ &\text{subject to } y_i \cdot (\langle w, x_i \rangle + b) \geq 1 \quad \text{for } i = 1, \ldots, m \ , \end{aligned}$$

where the minimization of the squared length of the weight vector corresponds to keeping the capacity of the hyperplane classifier low (first line) while simultaneously classifying the training sample without error (second line). In order to be able to cope with linarly non-separable data as well, where the condition of the second line does not hold, so-called slack variables $\xi_i$ are used to allow for misclassification. For each training pattern $x_i$ being misclassified by the hyperplane, the corresponding slack variable $\xi_i$ is greater than zero, and the function to be optimized is penalized accordingly.

The resulting optimization term can then be written as

$$\min_{w \in \mathbb{R}^n} \frac{1}{2}\|w\|^2 + C \sum_{i=1}^{m} \xi_i \tag{4.8}$$
$$\text{subject to } y_i \cdot (\langle w, x_i \rangle + b) \geq 1 - \xi_i \text{ (where } \xi_i \geq 0) \quad \text{for } i = 1, \ldots, m \ .$$

Here, $C \geq 0$ denotes a regularization parameter to be defined before optimization. The regularization parameter controls whether the minimization of the capacity or the minimization of the empirical risk is more important. This minimization problem belongs to the class of linearly constrained convex optimization problems and can be solved by quadratic programming [More and Toraldo (1991); Burges and Vapnik (1995)] (see also Sec. 3.5). By reformulating the optimization problem, the regularization parameter $C$ can be replaced by a parameter $\nu \in [0, 1)$, from which upper and lower bounds on the fraction of training patterns acting as support vectors can be derived.

A theoretical investigation shows that the hyperplane classifier resulting from the solution of Eq. (4.8) meets the form [Schölkopf and Smola (2002)]

$$g(x) = \sum_{i=1}^{m} y_i \alpha_i \langle x, x_i \rangle + b \ , \tag{4.9}$$

where $(\alpha_1, \ldots, \alpha_m) \in \mathbb{R}^m$ and $b \in \mathbb{R}$ are obtained during optimization. The important observation is that the class-separating hyperplane can be defined by an expansion of the training patterns $\{(x_i, y_i)\}_{i=1\ldots m}$. Moreover, it turns out that only patterns located closely to the hyperplane contribute to the sum in Eq. (4.9); for all other patterns $x_i$ the corresponding weight is $\alpha_i = 0$. The patterns with non-zero weight are called support vectors, since the decision boundary resulting from SVM training depends on these support vectors only.

For a given test pattern $x \in \mathcal{X}$, the computation of the decision function in Eq. (4.9) depends on inner products of $x$ and training patterns $x_i$ only. This means that in order to determine in which subspace defined by the separating hyperplane an input point lies, one simply needs to know the parameters $\alpha_1, \ldots, \alpha_m, b$ and the inner product $\langle x, x_i \rangle$ of the input pattern and each training pattern. Hence, the kernel trick discussed in Sec. 4.2.2 is fully applicable. Instead of requiring that the pattern space satisfies $\mathcal{X} = \mathbb{R}^n$, we define a valid kernel function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ on a pattern space $\mathcal{X}$ of any kind and train the SVM in the corresponding kernel feature space $\mathcal{H}$ by replacing the dot product $\langle x, x_i \rangle$ by the kernel function $k(x, x_i)$. While the actual separating hyperplane is linear in the kernel feature space,

the decision boundary is usually more complex in the original pattern space because of the non-linearity of the feature mapping $\Phi : \mathcal{X} \rightarrow \mathcal{H}$.

SVM classification has been applied to a wide range of recognition problems and has frequently outperformed traditional classifiers [Schölkopf and Smola (2002); Shawe-Taylor and Cristianini (2004); Müller *et al.* (2001); Byun and Lee (2003)]. This demonstrates that SVMs are not only interesting in view of the structural risk minimization principle, the geometrical interpretation of the maximum-margin hyperplane, and the SVM formulation in terms of kernel functions, but also because of its convincing performance results obtained on difficult real-world data sets. An interesting observation is that the performance of SVMs is often surprisingly independent of the actual choice of kernel function in the case of vectorial patterns $\mathcal{X} = \mathbb{R}^n$ [Schölkopf and Smola (2002)]. Hence, in vector spaces, the powerful part of SVM based classification seems to be the classifier, not the feature extraction by kernel functions. The question whether a similar behavior can be observed in the case of graph kernels is to be investigated experimentally.

### 4.3.2 *Kernel Principal Component Analysis*

Principal component analysis (PCA) is a well-known technique for data analysis and dimensionality reduction [Hotelling (1933); Jolliffe (1986)]. The idea is to extract those features that best describe the dominant linear directions in a vectorial data set. In the resulting decorrelated data set, the components are ordered according to their importance, or contribution to the dominant direction.

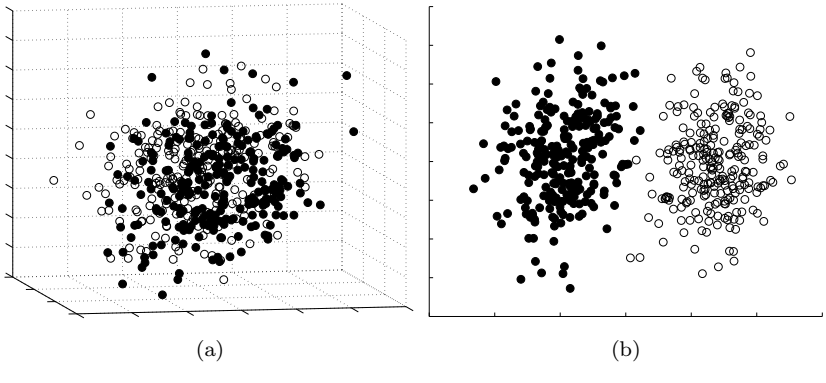



Fig. 4.9 (a) Original data set and (b) decorrelated data set after applying PCA

The basic idea of PCA is to find orthogonal directions describing the data at hand as accurately as possible by capturing as much of the data variance as possible. In the first step, PCA computes the subspace for which the original data projected onto leads to the highest variance. The resulting direction constitutes the first component of the decorrelated data set. In an iterative procedure, the maximum-variance subspace among all subspaces that are orthogonal to the previously computed directions are computed one at a time. The extracted components are ordered according to the amount of variance they capture. The maximal dimension of the resulting PCA feature space is bounded by the number of training elements.

More formally, the length of a vector $x$ projected onto a vector $w$ with $\|w\| = 1$ can be expressed by $\langle w, x \rangle$. For a given set of vectorial patterns $\{x_i\}_{i=1\ldots m}$ with zero mean, the vector $w_1$ for which the variance of the projection of $\{x_i\}_{i=1\ldots m}$ onto $w_1$ is maximal can be determined by [Duda *et al.* (2000)]

$$w_1 = \arg\max_{\|w\|=1} \sum_{i=1}^{m} \langle w, x_i \rangle^2 \ . \tag{4.10}$$

The resulting vector $w_1$ is the first principal component of the data $\{x_i\}_{i=1\ldots m}$. Successive principal components $w_2, w_3, \ldots$ can then recursively be computed by

$$w_k = \arg\max_{\|w\|=1} \sum_{i=1}^{m} \left\langle w, x_i - \sum_{j=1}^{k-1} w_j \langle w_j, x_i \rangle \right\rangle^2 \ . \tag{4.11}$$

In practice, these maximization problems are usually solved by means of an eigendecomposition of the covariance matrix of the original data [Duda *et al.* (2000)]. An illustration of a two-class data set before and after applying PCA is given in Fig. 4.9. While in Fig. 4.9a, a plane is required for the separation of the two classes in three dimensions, in the decorrelated data set in Fig. 4.9b it is sufficient to consider only the first component (horizontal axis) to obtain a reasonably accurate separation of the classes. Hence, in this example, the data has been reduced from three to one dimension without making classification substantially more difficult. It should be noted that PCA is an unsupervised method and therefore does not know which pattern belongs to which class.

The information PCA extracts from patterns can be formulated in terms of inner products of pairs of patterns. Replacing the inner product by a valid kernel function, according to the kernel trick described in Sec. 4.2.2, we obtain a kernelized variant of the PCA algorithm [Schölkopf *et al.* (1998,

1999)]. Kernel PCA differs from traditional PCA in that non-linear regularities in data can be detected as well, provided that a suitable kernel function is employed. Note that the actual PCA in the kernel feature vector space $\mathcal{H}$ is linear, but in the original space $\mathcal{X}$ the extracted features correspond to non-linear directions because of the non-linear feature space embedding $\Phi : \mathcal{X} \to \mathcal{H}$.

Kernel PCA can be carried out for any valid kernel function that can be defined on pairs of patterns. Hence, for a given recognition problem, it may be advantageous to apply a certain number of kernel functions simultaneously to the same data set, so that one obtains with minimal effort several feature extraction methods of different characteristics.

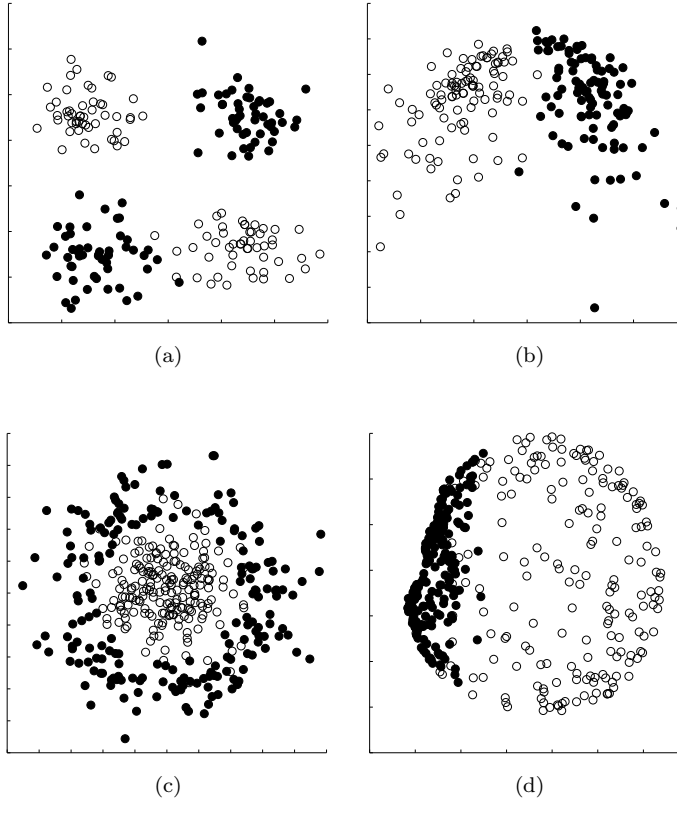



Fig. 4.10 (a) Data set and (b) its principal components using polynomial kernel PCA. (c) Data set and (d) its principal components using RBF kernel PCA

Kernel PCA can be understood as a visualization of the pattern distribution in the implicitly existing, but unknown kernel feature space. In fact, kernel PCA simply extracts from the hidden kernel feature space those components that best explain the variance of the data. A kernel PCA example is shown in Fig. 4.10a,b for a polynomial kernel and in Fig. 4.10c,d for an RBF kernel (see Table 4.1 for a definition of polynomial and RBF kernel functions). Clearly, in the kernel feature spaces (Fig. 4.10b,d) it is sufficient to consider the first feature component (horizontal axis) only, whereas in the original pattern spaces (Fig. 4.10a,c) more complex classifiers are required.

Kernel PCA is mainly used for pattern analysis. However, if kernel PCA extracts features that are relevant for an underlying classification problem, it may be reasonable to classify patterns in the resulting feature space, for instance by means of nearest-neighbor classifiers based on the Euclidean or any other standard distance measure.

### 4.3.3 *Kernel Fisher Discriminant Analysis*

The main limitation of PCA is that the method is unsupervised. That is, PCA does not exploit at all the valuable information which training pattern belongs to which class. Fisher discriminant anlysis (FDA) is a feature extraction method that can be regarded, to a certain extent, as supervised PCA. The FDA optimization term constitutes a trade-off between large inter-class distances and small intra-class distances. The aim is to linearly transform the input data such that classes are compact and well-separated in the resulting space [Duda *et al.* (2000)]. To this end, the data is projected onto a subspace such that the distance between the projected class means is large and the class variances are small. Again, the FDA computation can be formulated as an eigenvalue problem. An illustration of the difference between PCA and FDA is given in Fig. 4.11. Clearly, while PCA in Fig. 4.11a is only able to capture the global direction of the complete data set, FDA in Fig. 4.11b detects that the global direction is not optimal for the separation of the classes. Hence, classifying the FDA projected data is significantly easier than classifying the PCA projected data. In general, ignoring the information which training pattern stems from which class is not appropriate for many difficult pattern recognition problems. In view of this, FDA is supposed to be more suitable for pattern analysis than PCA.

In analogy to kernel PCA, FDA can be formulated in terms of inner products only, which makes kernel functions applicable to FDA [Mika *et al.*

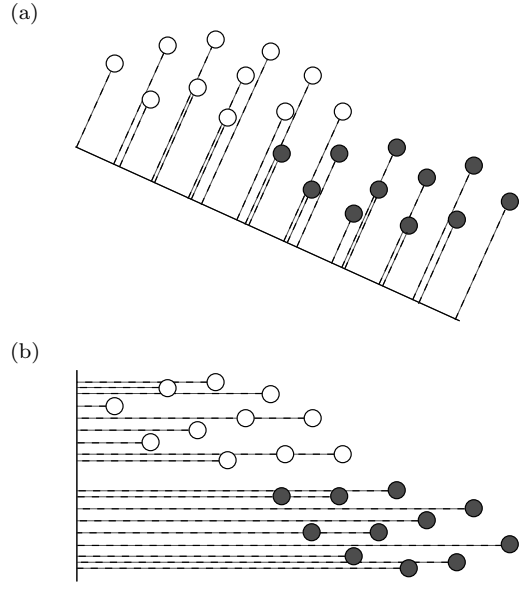



Fig. 4.11 (a) PCA, maximization of variance of projected data, and (b) FDA, maximization of distance between projected class means and class-wise minimization of variance of projected data

(1999); Baudat and Anouar (2000)]. In the kernel feature space $\mathcal{H}$, FDA projections are linear, but from the perspective of the original pattern space $\mathcal{X}$, kernel FDA in fact computes non-linear projections of the data.

For complex objects, for which a vectorial representation is not readily available, data visualization is of great importance, because more often than not visualizing the data to be analyzed provides significant insights into the distribution of the data. Clustering algorithms [Hartigan (1975); Jain *et al.* (1999a)] and distance-based techniques such as multi-dimensional scaling [Cox and Cox (1994)] may serve as a basic means of data visualization in these cases. Yet, kernel PCA and kernel FDA have the advantage over these methods that they extract decorrelated components of the high-dimensional feature space, and by plugging in various kernel functions, one obtains a multitude of extracted features reflecting different regularities of the hidden kernel feature space.

### 4.3.4 *Using Non-Positive Definite Kernel Functions*

In previous sections, kernel machines have been introduced for positive definite kernel functions. The main results from the theory of kernel functions, such as Theorem 4.1 about the equivalence of valid kernels and inner products in kernel feature spaces, require kernel functions to be symmetric and positive definite. In fact, it seems to be surprising that the single condition of positive definiteness implies a huge number of mathematical properties related to the underlying kernel function. The general kernel trick discussed in Sec. 4.2.2, for instance, requires positive definite kernels as well. However, in the case of SVMs (Sec. 4.3.1), kernel PCA (Sec. 4.3.2), kernel FDA (Sec. 4.3.2), and some other translation-invariant problems, it turns out that kernels only need to be conditionally positive definite in order to be applicable [Schölkopf and Smola (2002)]. Hence, this opens up the class of kernel functions that are suitable for the kernel machines described in this chapter. Although the complete theory of kernel functions might not be available if conditionally positive definite kernels are used, the interpretation of the kernel machines considered in this book remains valid, and inner products can be replaced by conditionally positive definite kernels.

From the theoretical point of view, valid kernel functions are particularly interesting because of their associated vector space structure. Yet, in practice it seems to be difficult to design valid kernel functions for certain applications. In such cases, it may be tempting to deliberately plug an invalid kernel function into a kernel machine. Using SVMs, for instance, the training procedure is not guaranteed to terminate if invalid kernels are used. On the other hand, if the training succeeds, one obtains an empirical classifier for which the theory of SVMs does not hold true, but which may nevertheless perform well. Accordingly, a number of successful applications of indefinite kernels functions to kernel machines have been reported [Bahlmann *et al.* (2002); DeCoste and Schölkopf (2002); Shimodaira *et al.* (2002); Moreno *et al.* (2004); Jain *et al.* (2005)].

In a recent analysis of indefinite kernels [Haasdonk (2005)], it was observed that SVM learning with indefinite kernels offers a reasonable geometrical interpretation. The main conclusion is that using SVMs in conjunction with arbitrary symmetric kernels can be understood as hyperplane classification in pseudo-Euclidean spaces. Pseudo-Euclidean spaces are defined by the direct product of two Euclidean spaces. In the case of valid kernels, the objective of SVM learning is to find a maximum-margin hyperplane separating two classes. If indefinite kernels are used, the SVM procedure aims at

maximizing the distance between the convex hulls of two classes. This geometrical interpretation is possible as distance, orthogonality, hyperplanes, and convex hulls can be defined in reasonable terms in pseudo-Euclidean spaces. Hence, even if kernel functions are applied that violate the conditions of (conditional) positive definiteness, training SVMs may nevertheless be a viable procedure with an intuitive and reasonable interpretation.

Given a kernel matrix derived from an arbitrary symmetric kernel function, its suitability for application to an SVM can be predicted to a certain degree [Haasdonk (2005)]. An increasing number of negative eigenvalues of the kernel matrix indicates that the convex hull separation of the training data is difficult. Another criterion is the distance of the class means in the pseudo-Euclidean space. If the Euclidean distance is negative, which is possible in pseudo-Euclidean spaces, a solution of the convex hull separation problem need not exist. Hence, indefinite kernels that are suitable for SVMs are characterized by a small number of negative eigenvalues of their kernel matrix and a positive distance of the class means. These criteria allow us to quantify how appropriate a given invalid kernel function may be for SVM classification.

### 4.4 Nearest-Neighbor Classification Revisited

This section will return to the nearest-neighbor classification briefly introduced in Sec. 3.6 and shed new light on the nearest-neighbor paradigm. For the formal definition of nearest-neighbor classifiers, let us assume that a pattern space $\mathcal{X}$, a space of class labels $\mathcal{Y}$, and a labeled training set of patterns $\{(x_i, y_i)\}_{i=1\ldots m} \subseteq \mathcal{X} \times \mathcal{Y}$ is given. The *1-nearest-neighbor classifier* (1NN) is defined by assigning a test pattern $x \in \mathcal{X}$ to the class of its most similar training pattern. Accordingly, the 1NN classifier $f : \mathcal{X} \to \mathcal{Y}$ is defined by

$$f(x) = y_j, \text{ where } j = \arg\min_{i=1\ldots m} d(x, x_i) \ , \tag{4.12}$$

where $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is a pattern dissimilarity measure. Clearly, the 1NN classifier assigns the same weight to each training pattern, no matter if a pattern constitutes an outlier or a true class representative. Its decision boundary is composed of piecewise linear functions, which makes the 1NN classifier very flexible and applicable to arbitrarily complex class distributions.

The training of a nearest-neighbor classifier consists of providing the classifier with a training set of prototypes — an actual process learning the

underlying classes and adapting free parameters does not occur in the case of nearest-neighbor classifiers. The capacity of the 1NN classifier, measured by the VC dimension introduced in Sec. 4.1.2, is infinite, since any training set can be learned without error by the 1NN classifier, provided that there are not contradicting training examples. Consequently, structural risk minimization cannot be applied to 1NN classifiers. For 1NN classifiers to generalize well on unseen data, the size of the underlying training set must therefore be large, as the classifier capacity cannot be limited in this case. Following the argumentation in Sec. 4.1, the nearest-neighbor paradigm corresponds to the empirical risk minimization principle. Intuitively, the more training patterns are sampled from the hidden underlying pattern distribution, the better is the resulting empirical estimation of the class boundaries of the 1NN classifier. If the training set is small, the few random pattern examples will never cover the pattern space sufficiently well. This idea is illustrated in Fig. 4.1, where a certain number of training patterns are needed to make the underlying class distribution apparent.

To render nearest-neighbor classification less vulnerable to outliers, the nearest-neighbor classifier can be extended to consider not only the single closest pattern, but several of the closest patterns. The *k-nearest-neighbor classifier* (kNN) assigns a test patterns to the class that occurs most frequently among its $k$ closest training patterns. Formally, if $\{(x_{(1)}, y_{(1)}), \ldots, (x_{(k)}, y_{(k)})\} \subseteq \{(x_i, y_i)\}_{i=1\ldots m}$ are those $k$ patterns in the training set that have the smallest distance $d(x, x_{(i)})$ to a test pattern $x$, the kNN classifier $f : \mathcal{X} \to \mathcal{Y}$ can be defined by

$$f(x) = \arg\max_{y \in \mathcal{Y}} \left| \left\{ (x_{(i)}, y_{(i)}) : y_{(i)} = y \right\} \right| \quad . \tag{4.13}$$

Hence, the kNN classifier reduces the influence of outliers by evaluating which class occurs frequently in a neighborhood around the test pattern. Although the kNN classifier can be regarded as a step towards a better generalization performance on unseen test data than the one offered by 1NN classifiers, the tradeoff between undertraining and overtraining cannot be explicitly addressed such as in the training process of SVMs.

Summarizing, the main limitation of nearest-neighbor classifiers is that a sufficiently large number of training samples must be available. For nearest-neighbor classifiers based on graph edit distance, this requirement is particularly cumbersome, since increasing the training set will always increase the running time of the classifier as well, which may be critical due to the computationally inefficient edit distance algorithm. On the other hand, nearest-neighbor classification provides a natural way to classify graphs

based on graph edit distance. Whether nearest-neighbor classification is sufficiently accurate on complex real-world data sets will be investigated empirically in Sec. 6.5 and Sec. 6.6.

This page intentionally left blank

## Chapter 5

# Graph Kernels

The main result of kernel theory is that it is perfectly sufficient to extract information about the pairwise similarity of patterns from the problem domain, by means of a kernel function, in order to apply non-linear versions of well-known algorithms for pattern analysis, classification, and regression to the pattern space under consideration. In fact, the definition of a valid kernel function in the problem domain allows us to formulate a pattern recognition problem in an inner product space instead of the original problem domain. The existence of such a vector space, or kernel feature space, as well as the definition of the inner product in this space can be derived from the kernel function.

Some kernel machines exhibit interesting theoretical properties related to statistical learning theory, as discussed in the previous chapter. These properties suggest that the application of kernel machines to pattern recognition problems, under certain general conditions, allows for the definition of classifiers generalizing well on unseen data. For instance, the training procedure of support vector machines (SVMs) explicitly aims at reducing the risk of overfitting, that is, not learning the class boundaries of the training set, but rather the class boundaries of the hidden underlying population. Kernel machines exhibit these beneficial properties regardless of the problem domain at hand.

Using kernel machines to address the problem of graph matching, the formulation in terms of kernel functions offers another crucial advantage over traditional methods: Graph kernels provide us with an embedding of the space of graphs into an inner product space according to the graph kernel similarity measure. To be able to address the graph matching problem in a vector space related to the space of graphs, we simply need to define a similarity measure on graphs that satisfies certain properties. The

tremendous benefit of such a vector space embedding is that it instantly eliminates the lack of mathematical structure in the space of graphs. That is, while a computation as simple as the average of a set of patterns cannot be performed a priori in the space of graphs, the same computation can readily be carried out in a vector space. Applying algorithms to graph matching that are defined with respect to quantities such as lengths, distances, angles, projections, linear combinations of patterns, etc. therefore requires a cumbersome and time-consuming redefinition of these quantities for graphs. These problems can be avoided by mapping patterns from the space of graphs into an inner product space using a kernel function. The pattern matching task can then be addressed in this vector space, where basic operations for pattern manipulation, such as those mentioned above, are readily available.

This chapter is concerned with the application of kernel machines to error-tolerant graph matching. The objective is to develop graph kernels that are capable of matching graphs of any kind and flexible enough to successfully cope with graphs extracted from real data, even if a significant amount of noise is present. For this reason, the focus is on graph kernels related to edit distance. In the following, the basic idea of graph kernels is illustrated, and related work is discussed. The remainder of the chapter is then devoted to a description of the proposed kernel functions.

## 5.1 Kernel Machines for Graph Matching

To enhance methods for graph matching, it is quite common to transfer ideas originally developed for statistical patterns represented by feature vectors to the domain of graphs. To give an example, self-organizing maps have been applied to the problem of graph clustering [Günter and Bunke (2002)]. The standard self-organization procedure is based on the idea that neurons of the self-organizing map migrate towards areas of the input space with high pattern densities. To make the self-organization algorithm applicable to graphs, the key operation that needs to be defined is a method to draw one graph (a neuron) by a certain amount closer to another graph (the input pattern). Hence, what is required is the computation of a weighted mean of two graphs $x, y$,

$$(1 - \alpha)x + \alpha y, \quad \text{where } \alpha \in [0, 1] \ . \tag{5.1}$$

If $x$ and $y$ were vectors, the weighted mean could easily be computed using vector addition and scalar multiplication. Yet, since $x$ and $y$ are actually

graphs, the same basic operations cannot be defined in a standardized way. To apply self-organizing maps to graphs — or any other algorithm defined for vectorial patterns, for that matter — the required operations need to be adapted for that specific task to the domain of graphs [Günter and Bunke (2002)], although vector space quantities, such as the unique mean of a set of vectors, may not even exist in the space of graphs. Hence, applying statistical pattern recognition algorithms to graphs may be possible, but often requires elaborate modifications.

Kernel machines offer a universal and more elegant approach to this fundamental problem. The definition of a single graph kernel function provides us with an embedding of graphs in an inner product space and thus enables us to apply a wide range of methods for the analysis and recognition of patterns. The graph kernel function, provided that the necessary conditions for valid kernels are satisfied, can be regarded as a graph similarity function. Thus, simply by defining a similarity function on graphs, we obtain an elegant method to perform, for instance, a Fisher discriminant analysis on graphs, or an SVM based classification of graphs, or a self-organizing clustering of graphs, or any other kernel method mentioned in the previous chapter. For statistical patterns, a kernel function is not required to apply these algorithms, since an inner product can easily be defined in the original vector space of patterns. But for graphs, the framework of kernel machines opens up a new class of graph matching algorithms and does so in a unified manner. While the actual graph matching process, that is, the evaluation of graph similarity, is actually effected by the kernel function, kernel algorithms can be used to interpret the resulting data by performing pattern classification, clustering, or feature extraction.

For an instructive example, let us consider a two-class pattern classification task. Assume that the classification is effected by assigning input patterns to the class whose mean element of the labeled training set is closer. That is, for an unknown input pattern $z$, a training set of patterns from the first class $\{x_1, \ldots, x_p\}$, and a training set of patterns from the second class $\{y_1, \ldots, y_q\}$, one has to compute the mean of the first class $\bar{x}$, the mean of the second class $\bar{y}$, and the distance from $z$ to $\bar{x}$ and $\bar{y}$. An illustration of this scenario is provided in Fig. 5.1, where patterns from the first class are depicted by gray dots, patterns from the second class by white dots, and the mean elements by black dots. The decision boundary of the classifier is the straight line perpendicular to the line connecting the two mean elements and centered between them. The corresponding

classification rule for an input pattern $z$ can be expressed by

$$\begin{aligned} D(z) &= \|z - \bar{y}\| - \|z - \bar{x}\| \\ &= \sqrt{\langle z, z\rangle + \langle \bar{y}, \bar{y}\rangle - 2\langle z, \bar{y}\rangle} \\ &\quad - \sqrt{\langle z, z\rangle + \langle \bar{x}, \bar{x}\rangle - 2\langle z, \bar{x}\rangle} \ , \end{aligned}$$

which is positive for patterns being closer to $\bar{x}$ than to $\bar{y}$, and negative for patterns being closer to $\bar{y}$ than to $\bar{x}$. Hence, in order to classify an input pattern $z$ according to this rule, that is, to determine which mean element is closer, one simply has to compute the value of the term $D(z)$ and evaluate its sign.

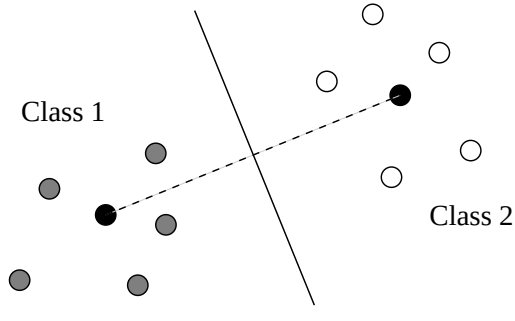



Fig. 5.1 A two-class classification problem and a simple minimum distance classifier

For patterns represented by feature vectors, the evaluation of the classification rule above is straightforward by means of the standard inner product or any other inner product. While it may be possible to define functions acting as inner products for graphs based on some graph similarity criterion, there exists by all means no well-established standard solution to this problem. Fortunately, by means of the kernel trick described in Sec. 4.2.2, an inner product definition in the space of graphs $\mathcal{X}$ is not required, as the inner product of two graphs can be evaluated in the kernel feature space $\mathcal{H}$ instead. If we denote the mean elements in the feature space by

$$\Phi\bar{x} = \frac{1}{p}\sum_{i=1}^{p}\Phi(x_i) \quad \text{and} \quad \Phi\bar{y} = \frac{1}{q}\sum_{i=1}^{q}\Phi(y_i) \ , \tag{5.2}$$

we obtain, for an input graph $z \in \mathcal{X}$ and a mapping $\Phi : \mathcal{X} \to \mathcal{H}$ from the graph space to the feature space, the classification rule

$$\begin{aligned} D(\Phi(z)) &= \sqrt{\langle \Phi(z), \Phi(z) \rangle + \langle \Phi\bar{y}, \Phi\bar{y} \rangle - 2\langle \Phi(z), \Phi\bar{y} \rangle} \\ &\quad - \sqrt{\langle \Phi(z), \Phi(z) \rangle + \langle \Phi\bar{x}, \Phi\bar{x} \rangle - 2\langle \Phi(z), \Phi\bar{x} \rangle} \\ &= \sqrt{\langle \Phi(z), \Phi(z) \rangle + \frac{1}{q^2} \sum_{i,j=1}^{q} \langle \Phi(y_i), \Phi(y_j) \rangle - \frac{2}{q} \sum_{i=1}^{q} \langle \Phi(z), \Phi(y_i) \rangle} \\ &\quad - \sqrt{\langle \Phi(z), \Phi(z) \rangle + \frac{1}{p^2} \sum_{i,j=1}^{p} \langle \Phi(x_i), \Phi(x_j) \rangle - \frac{2}{p} \sum_{i=1}^{p} \langle \Phi(z), \Phi(x_i) \rangle} \\ &= \sqrt{k(z,z) + \frac{1}{q^2} \sum_{i,j=1}^{q} k(y_i, y_j) - \frac{2}{q} \sum_{i=1}^{q} k(z, y_i)} \\ &\quad - \sqrt{k(z,z) + \frac{1}{p^2} \sum_{i,j=1}^{p} k(x_i, x_j) - \frac{2}{p} \sum_{i=1}^{p} k(z, x_i)} \ , \end{aligned}$$

due to the bilinearity of the inner product and the kernel trick. The important observation is that the classification of a graph $\Phi(z)$ in the feature space can be evaluated entirely in terms of the kernel function applied to graphs. While the classification rule in fact computes the distance of a pattern $\Phi(z)$ to the two class means $\Phi\bar{x}$ and $\Phi\bar{y}$, it is not necessary to explicitly compute those class means in the feature space, let alone to explicitly compute the mean of a set of graphs. The kernel feature space allows us to compute the Euclidean distance of a pattern to the two class means, but not the class means themselves.

For classification, it is sufficient to know how the class means are related to other patterns in terms of the inner product, which shows that kernel functions can be used to extract the information from a feature space that is relevant for recognition. This simple example of a classifier demonstrates how powerful the kernel mapping concept is and how elegantly the limitations of the graph domain can be overcome by application of kernel functions. A similar argumentation can of course also be formulated for all kernel algorithms mentioned in the previous chapter.

### 5.2 Related Work

In recent years, a number of kernel functions have been designed for graph matching [Shawe-Taylor and Cristianini (2004); Gärtner (2003)]. In this section, a brief description of some important classes of graph kernels is provided.

A general approach for complex objects based on *convolution kernels* has been proposed in [Haussler (1999); Watkins (2000)]. The basic idea is to define positive definite kernels on substructures of composite objects and combine these kernels, using the property that positive definite functions are closed under convolution, into a kernel function on the composite objects themselves [Haussler (1999)]. Hence, convolution kernels infer the similarity of complex objects from the similarity of their parts. Splitting a string into two substrings, for instance, can be regarded as a decomposition of a string. For string matching, several kernel functions have been proposed, most of them belonging to the class of convolution kernels. The basic idea is to map strings into a feature space whose coordinates are indexed by strings [Haussler (1999); Watkins (2000); Leslie *et al.* (2002, 2004)]. For instance, spectrum kernels compare two strings by counting the number of fixed-length substrings they have in common, which means that the coordinates of this feature space are indexed by the substrings occurring in any input string. The ANOVA kernel [Vapnik (1998); Burges and Vapnik (1995); Saunders *et al.* (1998); Watkins (1999)] is another convolution kernel, which uses a subset of the components of a composite object for comparison. Convolution kernels for graph matching related to edit distance are presented in Sec. 5.7 and Sec. 5.8.

Another class of graph kernels is based on the evaluation of *random walks* in graphs. These kernels measure the similarity of two graphs by the number of (possibly infinite) random walks in both graphs that have all or some labels in common [Gärtner (2002); Kashima and Inokuchi (2002); Gärtner *et al.* (2003); Kashima *et al.* (2003); Borgwardt *et al.* (2005); Borgwardt and Kriegel (2005)]. In [Gärtner *et al.* (2003)], the author demonstrates how the number of matching random walks in two graphs can be evaluated using the direct product graph of the two graphs. This graph kernel has then been extended to be able to cope with continuous labels [Borgwardt *et al.* (2005)]. The definition of this kernel is very elegant, but it performs poorly on some data sets containing a considerable amount of noise. For more details and a random walk kernel variant enhanced by graph edit distance, see Sec. 5.9.

Kernel functions from the class of *diffusion kernels* are constructed by turning a base similarity measure into a kernel matrix guaranteed to be positive definite [Kondor and Lafferty (2002); Kandola *et al.* (2002); Smola and Kondor (2003); Lafferty and Lebanon (2003); Vert and Kanehisa (2003); Lafferty and Lebanon (2005)]. The only requirement is that the base similarity measure is symmetric. If the base similarity measure is regarded as a graph (nodes are objects and edges are labeled with the similarity of the two adjacent objects), the diffusion kernel is equivalent to the sum of the weight product, or similarities, of fixed-length paths in this graph. Diffusion kernels can be related to the information diffusion process of random walks in graphs. For a diffusion kernel derived from graph edit distance, refer to Sec. 5.5.

There exist various other graph kernel functions. In [Jain *et al.* (2005); Jain (2005)], for instance, the *Schur-Hadamard inner product* is used to define a graph kernel based on attributed adjacency matrices. This kernel provides an explicit embedding of graphs in a vector space, but is not generally positive definite. For another example related to random walk kernels, *marginalized kernels* are a class of graph kernels evaluating the probability of labeled random walks in two graphs being similar [Kashima *et al.* (2003); Mahé *et al.* (2004)].

The main objective in this book is the definition of kernel functions that are derived from graph edit distance. Based on the assumption that graph edit distance is well suited for difficult graph matching problems, kernel functions are proposed that are sufficiently flexible for unconstrained graph representations. Regarding kernel functions as similarity measures, we obtain embeddings of the space of graphs into vector spaces, where the similarity of vectors is defined according to the edit distance of the original graphs. A description of the proposed kernel functions for graphs is given in the following sections.

## 5.3 Trivial Similarity Kernel from Edit Distance

The inner product in a vector space, and therefore any kernel function corresponding to an inner product in some kernel feature space as well, can be regarded as a pattern similarity measure. For instance, the standard inner product in a Euclidean vector space assigns maximum similarity to vectors pointing in the same direction and minimum similarity to vectors pointing into opposite directions. In the case of graphs, the edit distance

already provides us with a flexible and widely used graph dissimilarity measure. Hence, it may be tempting to simply turn the existing dissimilarity measure into a similarity measure by mapping low distance values to high similarity values and high distance values to low similarity values. In this section, simple transformations mapping edit distances into similarities are defined. These trivial kernel functions derived from graph edit distance can then be used as reference systems for the more complex kernel functions proposed in the remainder of this chapter.

### *Definition*

The trivial similarity kernels in this section are based on a number of simple transformations deriving the similarity of two graphs from their edit distance. Given the edit distance $d(g, g')$ of two graphs $g$ and $g'$, the similarity $k_i(g, g')$ is defined according to

$$\begin{aligned} k_1(g, g') &= -d(g, g')^2 \\ k_2(g, g') &= -d(g, g') \\ k_3(g, g') &= \tanh(-d(g, g')) \\ k_4(g, g') &= \exp(-d(g, g')) \ . \end{aligned}$$

For an illustration of these four transformations, see Fig. 5.2. The transformations mainly differ in the rate of decrease and the range the distance values are mapped into. Kernel $k_4$ for instance is the only function resulting in positive similarity values.

The computational complexity of the trivial kernels is equal to that of the edit distance algorithm, since for the computation of $k_i(g, g')$ the edit distance of two graphs $d(g, g')$ has to be determined first. For two graphs with $m$ and $m'$ nodes, respectively, we therefore obtain an exponential computational complexity of $O(m^{m'})$, where $m > m'$ is assumed.

### *Validity*

The trivial kernel functions defined in this section are not positive definite and are therefore not valid kernels. However, there is theoretical evidence [Haasdonk (2005)] that using kernel machines in conjunction with indefinite kernels may be reasonable if some conditions are fulfilled. Among the kernel functions described above, the kernel $k_1$ is explicitly suggested for classifying distance-based data [Haasdonk (2005)]. For a discussion about

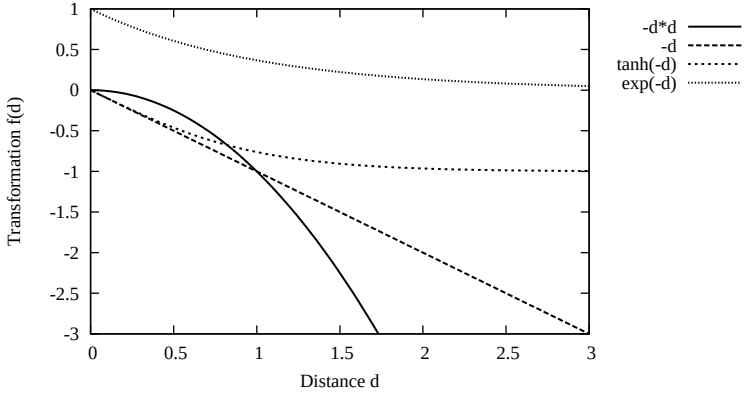



Fig. 5.2 Trivial distance to similarity transformations

using indefinite kernel functions, see Sec. 4.3.4.

***Discussion***

The advantage of trivial kernel approaches such as those described in this section is that the cumbersome definition of novel kernel functions measuring the similarity of graphs is not required. Instead, one simply relies on the widely used graph edit distance for graph matching. Using these functions, it is easy to investigate whether feeding the same distance information into a kernel machine leads to an improved classification accuracy over traditional distance based nearest-neighbor classifiers. Comparing in experiments the performance of these trivial kernel functions with the performance of more complex ones may be a first step towards understanding whether the power of graph kernel systems is primarily due to sophisticated kernel functions or rather the strength of kernel classifiers. An experimental evaluation of the trivial similarity kernels is provided in Sec. 6.6.1.

## 5.4 Kernel from Maximum-Similarity Edit Path

The most straightforward way to turn distances into similarities is to use simple monotonically decreasing transformations such as those described in the previous section. On the other hand, these approaches are defined with respect to any arbitrary kind of dissimilarity measure, without explicitly

taking the characteristics of graph edit distance into account. In these cases, the process of matching graphs is not adapted for the computation of similarities, but the resulting edit distance measure is simply turned into a similarity measure. In this section, a method is proposed to re-formulate graph edit distance as a graph similarity measure. The idea is to turn the minimum-cost edit path condition into a maximum-similarity edit path criterion.

***Definition***

According to Def. 3.1, the minimum-cost edit path between two graphs $g = (V, E, \mu, \nu)$ and $g' = (V', E'\mu', \nu')$ defines their edit distance,

$$d(g, g') = \min_{\substack{(e_1,\ldots,e_k) \\ \in \mathcal{P}(g,g')}} \sum_{i=1}^{k} c(e_i) \quad ,$$

where $c$ denotes the edit cost function and $\mathcal{P}(g, g')$ denotes the set of edit paths from $g$ to $g'$. In the same spirit, we define the edit similarity of two graphs by the maximum-similarity edit path between them,

$$k(g, g') = \max_{\substack{(u_1 \to u'_1,\ldots,u_k \to u'_k) \\ \in \mathcal{P}_s(g,g')}} \prod_{i=1}^{k} (\kappa_{rbf}(u_i, u'_i) + b) \quad , \tag{5.3}$$

where $\mathcal{P}_s(g, g')$ is obtained from $\mathcal{P}(g, g')$ by removing all deletions and insertions of nodes and edges. Hence, the proposed kernel function only considers the substitution of nodes and edges. The function $\kappa_{rbf}$ is the RBF kernel function introduced in Table 4.1 measuring the similarity of node and edge labels, and $b \in \mathbb{R}^+$. Note that for the sake of simplicity symbols $u_i, u'_i$ in the formula above denote both nodes and edges, and the similarity of labels is written $\kappa_{rbf}(u_i, u'_i)$ instead of the correct $\kappa_{rbf}(\mu(u_i), \mu'(u'_i))$ for nodes $u_i, u'_i$ and $\kappa_{rbf}(\nu(u_i), \nu'(u'_i))$ for edges $u_i, u'_i$. The computation of this kernel function can be carried out by means of a modified edit distance algorithm.

***Properties***

The kernel derived from the maximum-similarity edit path between two graphs is based on the idea that two graphs are similar if there exists an edit path between them consisting of a large number of substitutions with similar labels. In Eq. (5.3), the similarity of edit paths is measured by multiplying the similarities of node and edge labels involved in substitutions.

The resulting product will be large if the individual factors $\kappa_{rbf}(u_i, u'_i)$ are large, that is, if the substitutions of the edit path $u_i \rightarrow u'_i$ correspond to nodes (or edges) $u_i, u'_i$ with similar labels. Deletions and insertions are implicitly accounted for, since edit paths with more substitutions (corresponding to fewer insertions and deletions) will tend to result in higher similarity products. Parameter $b$ can be used to control how strongly substitutions are favored over deletions and insertions.

In analogy to the edit distance algorithm, the complexity of the maximum-similarity edit path computation amounts to $O(m^{m'})$ for two graphs consisting of $m$ and $m'$ nodes, respectively, assuming $m > m'$.

#### *Validity*

For reasons that will become apparent in Sec. 5.7, there is theoretical evidence that the similarity of graphs can be measured by evaluating the similarity of node and edge labels in edit paths according to the description above. However, the kernel function defined in Eq. (5.3) is not valid, which might limit its applicability to kernel machines (see Sec. 4.3.4 for a discussion).

#### *Discussion*

The maximum-similarity edit path kernel defined in this section offers a simple way to turn the edit distance measure into a kernel similarity measure by computing the similarity of edit operations instead of penalty costs. This kernel is mainly intended to serve as a link between graph edit distance and the definition of graph similarity kernels based on edit distance. In Sec. 6.6.2 on experimental results, the question will be addressed whether such a simple definition of graph similarity is sufficiently powerful for graph matching, or whether more complex valid kernel functions perform better in practice.

### 5.5 Diffusion Kernel from Edit Distance

Diffusion kernels [Kondor and Lafferty (2002); Kandola *et al.* (2002); Smola and Kondor (2003); Lafferty and Lebanon (2003); Vert and Kanehisa (2003); Lafferty and Lebanon (2005)] are defined with respect to a base similarity measure. This base similarity measure only needs to satisfy the condition of symmetry and can be defined for any kind of objects. For

error-tolerant graph matching, it is straightforward to use the edit distance as base dissimilarity measure and construct from an edit distance matrix a kernel similarity matrix. The diffusion kernel offers a convenient interpretation of the resulting similarity measure and is furthermore guaranteed to be positive definite, unlike the other kernel functions discussed so far.

***Definition***

Let $\{g_1, \ldots, g_n\} \subseteq \mathcal{X}$ be a set of graphs and let $d(g_i, g_j)$ denote the edit distance of graphs $g_i$ and $g_j$. In a manner similar to the trivial transformations in Sec. 5.3, the edit distance between patterns $\{g_1, \ldots, g_n\}$ is turned into a matrix of non-negative similarities,

$$B_{i,j} = \max_{1 \leq s,t \leq n} (d(g_s, g_t)) - d(g_i, g_j) \quad \text{for } 1 \leq i, j \leq n,$$

where $B_{i,j}$ denotes the similarity of graphs $g_i$ and $g_j$. The $n \times n$-matrix of similarities $B$ can then be turned into a positive definite kernel matrix. Given a decay factor $0 < \lambda < 1$, the *exponential diffusion kernel* [Kondor and Lafferty (2002)] is defined by

$$K = \sum_{k=0}^{\infty} \frac{1}{k!} \lambda^k B^k = \exp(\lambda B) \tag{5.4}$$

and the *von Neumann diffusion kernel* [Kandola *et al.* (2002)] by

$$K = \sum_{k=0}^{\infty} \lambda^k B^k \ . \tag{5.5}$$

Given a training set of graphs $\{g_1, \ldots, g_n\} \subseteq \mathcal{X}$ and any graph $g \in \mathcal{X}$, the value $k(g_i, g)$ can be evaluated by constructing the similarity matrix $B$ of patterns $\{g_1, \ldots, g_n, g\}$ and computing the value $K_{i,n+1}$ of the resulting kernel matrix. The decay factor $\lambda$ assures that for increasing $k$ the weight of the respective matrix $B^k$ decreases. Since $\lambda < 1$, the weighting factor $\lambda^k$ will be negligibly small for sufficiently large $k$. Therefore, only the first $t$ addends in the diffusion kernel sum will be evaluated in the experiments, based on the assumption that ignoring the remaining infinite number of addends will not have any effect on the kernel matrix because of the decay factor. An appropriate value of $t \in \{1, 2, 3, \ldots\}$ is to be empirically determined by a parameter validation process.

***Properties***

The idea behind diffusion kernels is based on the assumption that the base similarity measure $B$ can be enhanced by considering not only the distance of two patterns, but also the number of similar patterns they have in common. Computing powers of the base similarity matrix $B$ can be seen in equivalence to computing similarity products of sets of patterns. In the following, the similarity matrix $B$ is regarded as a similarity graph, where each node represents a graph from $\{g_1, \ldots, g_n\}$ and each pair of nodes $(g_i, g_j)$ is linked by an undirected edge labeled with the similarity $B_{i,j}$. The value $(B^k)_{i,j}$ can then be computed by traversing in this similarity graph all paths from $g_i$ to $g_j$ consisting of $k-1$ intermediate graphs and summing up the similarity products along the nodes of each path. Hence, the base similarity measure is enhanced by taking the number of similar graphs two graphs share into account. For instance, large values of $(B^2)_{i,j}$ indicate that there are many graphs $g_r \in \{g_1, \ldots, g_n\}$ that are similar to both $g_i$ and $g_j$, since $(B^2)_{i,j} = \sum_{r=1}^{n} B_{i,r} B_{r,j}$. The contribution of longer paths in the similarity graph is limited by means of the decay parameter $\lambda$. Diffusion kernels can also be interpreted in the context of stochastic processes on graphs. Under the assumption that nodes of a graph periodically send fractions of the data they contain to their neighboring nodes in the graph, diffusion kernels can be seen as models for the circulation, or diffusion, of information in graphs [Kondor and Lafferty (2002)].

Computing the edit distance of a test graph (consisting of $m$ nodes) to $n$ training graphs (consisting of $m'$ nodes) takes $O(nm^{m'})$, and the computation of the kernel matrix amounts to $O(tn^3)$, provided that only the first $t$ addends of the infinite sum are evaluated. Consequently, we obtain a computational complexity of $O(m^{m'} + tn^2)$ for a single evaluation of the diffusion kernel.

***Validity***

The exponential diffusion kernel and the von Neumann diffusion kernel are positive definite [Kondor and Lafferty (2002); Kandola *et al.* (2002)]. Hence, these diffusion kernels not only provide a strong link to graph edit distance, but they also constitute valid kernel functions, so that the theory of kernel functions is fully applicable in this case. It should be noted that the diffusion kernels are valid for any symmetric similarity measure, not just the modified edit distance of graphs.

### *Discussion*

Diffusion kernels are a unique class of kernels in the sense that they offer a generic way to turn an arbitrary symmetric similarity measure into a kernel matrix that satisfies the conditions of valid kernels. According to one interpretation, diffusion kernels evaluate if two graphs are similar to each other and similar to the same set of graphs. An open question is whether the positive definite diffusion kernels are advantageous over indefinite kernels, that is, whether the invalidity of a kernel function affects the kernel's applicability to recognition problems in practice. For an experimental evaluation of the diffusion kernel, see Sec. 6.6.3.

## 5.6 Zero Graph Kernel from Edit Distance

This section introduces a kernel function that can directly be derived from graph edit distance. The key idea is based on a result from the theory of positive definite functions. The similarity of two graphs is derived from their edit distance to a fixed set of training graphs acting, to a certain degree, as zero elements in the vector space. Provided that the edit distance is available, the kernel function is simple to compute and exhibits several convenient geometrical properties in the kernel feature space.

### *Definition*

Throughout this section, graphs will be denoted by symbols $x, x', \ldots$ rather than symbols $g, g', \ldots$ to make clear that the kernel function described below is in fact applicable to various kinds of patterns, not only graphs. Let $\mathcal{X}$ denote the set of graphs and $d(x, x')$ denote the edit distance of two graphs $x, x' \in \mathcal{X}$. Given a training set of graphs $\mathcal{X}^t \subseteq \mathcal{X}$ and a single graph $x_0 \in \mathcal{X}^t$ called *zero graph*, the basic kernel function is defined for all pairs of graphs $x, x' \in \mathcal{X}$ by

$$k_{x_0}(x, x') = k(x, x') = \frac{1}{2}\left(d(x, x_0)^2 + d(x_0, x')^2 - d(x, x')^2\right) \quad . \tag{5.6}$$

This kernel function can be regarded as a measure of the squared distance from graph $x$ to $x_0$ and from $x_0$ to $x'$ in relation to the squared distance from $x$ to $x'$ directly. From this definition it is clear that the kernel function assigns high similarity to graphs $x$ and $x'$ that are close to each other and both suffciently distant to the selected zero graph $x_0$, which is equivalent to $d(x, x')^2$ being small and $d(x, x_0)^2$ and $d(x_0, x')^2$ being large.

***Properties***

As the kernel function given above is defined in closed form, standard geometrical properties in the implicitly existing kernel feature space can easily be derived from the kernel function, as demonstrated in Sec. 4.2.2. For instance, the Euclidean distance of two vectors, that is, the length of the difference vector, can be expressed in terms of the inner product,

$$\begin{aligned}\|\Phi(x) - \Phi(x')\|_{\mathcal{H}}^2 &= \langle \Phi(x) - \Phi(x'), \Phi(x) - \Phi(x') \rangle \\ &= \langle \Phi(x), \Phi(x) \rangle + \langle \Phi(x'), \Phi(x') \rangle - 2\langle \Phi(x), \Phi(x') \rangle \\ &= k(x,x) + k(x',x') - 2k(x,x') \\ &= d(x,x')^2 - \frac{1}{2}\left(d(x,x)^2 + d(x',x')^2\right) \\ &= d(x,x')^2 \quad ,\end{aligned}$$

using the bilinearity and symmetry of the inner product, the kernel trick, and the symmetry of graph edit distance. This means that the Euclidean distance in the kernel feature space is equal to the edit distance in the space of graphs. By means of the kernel mapping $\Phi : \mathcal{X} \to \mathcal{H}$, we therefore obtain a perfect embedding of the space of graphs $\mathcal{X}$ in the vector space $\mathcal{H}$, where the edit distance measure from the graph space is preserved in the vector space. Hence, any kernel algorithm evaluating the Euclidean distance of vectors in the feature space actually uses the edit distance of graphs. This implies that a nearest-neighbor classifier in the kernel feature space $\mathcal{H}$ will behave exactly like a nearest-neighbor classifier in the graph space $\mathcal{X}$.

Similarly, the length of a vector $\Phi(x) \in \mathcal{H}$ turns out to be equal to the edit distance of the corresponding graph to the zero graph,

$$\|\Phi(x)\|_{\mathcal{H}}^2 = \langle \Phi(x), \Phi(x) \rangle = d(x,x_0)^2 - \frac{1}{2} d(x,x)^2 = d(x,x_0)^2 \quad .$$

The computation of angles of vectors in the kernel feature space can also be expressed in terms of the inner product. The angle $\alpha$ between any pattern $\Phi(x) \in \mathcal{H}$ and the zero graph mapped to the feature space $\Phi(x_0) \in \mathcal{H}$ is undefined, which follows from

$$\cos\alpha = \frac{\langle \Phi(x), \Phi(x_0) \rangle}{\|\Phi(x)\|_{\mathcal{H}} \, \|\Phi(x_0)\|_{\mathcal{H}}} \quad \text{and} \quad \|\Phi(x_0)\|_{\mathcal{H}} = 0 \quad .$$

Moreover, two graphs in their feature space representation are orthogonal if the edit distances $d(x,x_0)$, $d(x_0,x')$, and $d(x,x')$, interpreted as straight line segments in the Euclidean plane, form a right triangle. For two vectors

$\Phi(x), \Phi(x') \in \mathcal{X}$ being orthogonal, $\Phi(x) \perp \Phi(x')$, we obtain

$$\begin{aligned}\Phi(x) \perp \Phi(x') &\iff \langle \Phi(x), \Phi(x') \rangle = 0 \\ &\iff k(x, x') = 0 \\ &\iff d(x, x_0)^2 + d(x_0, x')^2 = d(x, x')^2 \ .\end{aligned}$$

The zero vector of any vector space has zero length, the length of an arbitrary vector is defined by its Euclidean distance to the zero vector, and the angle between the zero vector and any other vector is undefined. We hence conclude that the fixed graph $x_0 \in \mathcal{X}$ in its feature space representation $\Phi(x_0) \in \mathcal{H}$ exhibits properties that are characteristic of zero vectors. Calling $x_0 \in \mathcal{X}^t$ zero graph is therefore legitimate. In view of this property the zero graph can be understood, in a way, as a graph defining the origin of the space of graphs, which is obviously needed for embedding the distance-based space of graphs into a vector space.

From closure properties of positive definite functions discussed in Lemma 4.1, the simple kernel function defined in Eq. (5.6) can easily be extended to allow for a more complex modeling of graph similarity in the kernel feature space. The pointwise sum or product of several positive definite functions, for instance, is known to be positive definite as well. It is therefore straightforward to define, based on Eq. (5.6), the sum kernel function and the product kernel function given by

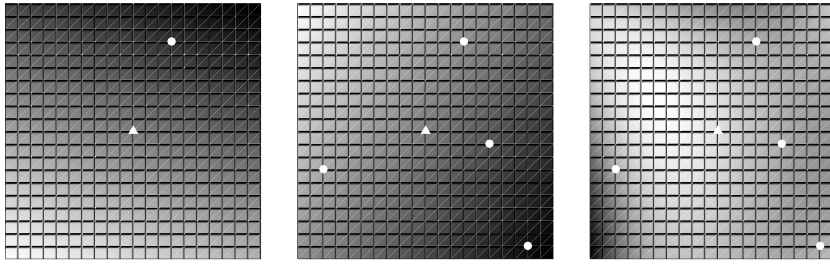
$$k_I^+(x, x') = \sum_{x_0 \in I} k_{x_0}(x, x') \ , \tag{5.7}$$

$$k_I^*(x, x') = \prod_{x_0 \in I} k_{x_0}(x, x') \ , \tag{5.8}$$

where $I \subseteq \mathcal{X}^t \subseteq \mathcal{X}$ denotes a set of zero graphs. Hence, instead of evaluating the similarity of two graphs in relation to a single zero graph $x_0 \in \mathcal{X}^t$, we combine the result of several kernel functions with distinct zero graphs $x_0 \in I$.

By evaluating different sets of zero graphs, this combination allows us to determine a set of zero graphs that corresponds to the vector space embedding that is best suited for classification. Unfortunately, a reliable method to derive from the pattern distribution a well-performing set of zero graphs, without evaluating the performance on a validation set, has not yet been found. Although such a selection criterion would certainly be desirable, we have to resort in our experiments to a simple greedy selection strategy for zero graphs. Given a parameter $n \in \{1, 2, 3, \ldots\}$, the idea is to first determine the $n$ zero graphs $I^{(1)} \subseteq \mathcal{X}^t$ best performing on a

validation set. In the next step, the $n$ best performing pairs of zero graphs $I^{(2)} \subseteq I^{(1)} \times \mathcal{X}^t$ are identified, and in iteration 3, we compute the $n$ best zero sets $I^{(3)} \subseteq I^{(2)} \times \mathcal{X}^t$. This selection strategy is pursued until $n$ zero sets of a predefined maximum size $z_{max}$ are obtained. Finally, the set of zero graphs from $I^{(1)} \cup I^{(2)} \cup I^{(3)} \cup \ldots \cup I^{(z_{max})}$ that performs best on the validation set is chosen.



(a) (b) (c)

Fig. 5.3 Zero graph kernel function using (a) the single kernel function in Eq. (5.6), (b) the sum kernel in Eq. (5.7), and (c) the product kernel in Eq. (5.8)

For the purpose of illustration, let us first consider the two-dimensional Euclidean vector space $\mathcal{X} = \mathbb{R}^2$ instead of the space of graphs and the Euclidean distance instead of the graph edit distance to measure the dissimilarity of patterns. To visualize the kernel functions, we choose a constant zero graph $x_0 \in \mathbb{R}^2$ and a set of zero patterns $I \subseteq \mathbb{R}^2$, fix the first argument $x \in \mathbb{R}^2$ of the kernel function, and evaluate the kernel function for various values of the second argument $x' \in \mathbb{R}^2$. More concretely, we iteratively select vectors $x'$ from the area $[-10, 10] \times [-10, 10] \subseteq \mathbb{R}^2$ and compute the value of the kernel function $k_{x_0}(x, x')$, $k_I^+(x, x')$, and $k_I^*(x, x')$ with respect to the constant set $I$ and the constant vectors $x_0, x$. The resulting kernel values can then be visualized in a figure, where the brightness of the pixel at position $x'$ represents the value of the kernel function with second argument $x'$. The kernel functions defined in Eq. (5.6), Eq. (5.7), and Eq. (5.8) are visualized in Fig. 5.3a–c. Note that the triangle mark at the center of each image represents the first argument $x$ of the kernel function and the circle marks represent the elements of the corresponding zero element $x_0$ or zero set $I$, respectively. Brighter colors indicate higher values of the kernel function, which means that in all three figures it can be observed that the kernel function assigns high values to patterns $x'$ being close to

$x$ and distant from all zero elements. In particular the product kernel is able to reflect non-linear similarity conditions in the vector space and may therefore be better suited to model complex data sets.

For two graphs consisting of $m$ and $m'$ nodes, respectively, the computational complexity of the zero graph kernel amounts to $O(|I|m^{m'})$, where $|I|$ denotes the number of zero graphs and $m > m'$.

***Validity***

A kernel function needs to satisfy the property of positive definiteness in order to be applicable to kernel machines. The kernel functions proposed in this section are based on the edit distance of graphs, but in the following a more general setting will be considered by regarding the underlying distance measure $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}^+ \cup \{0\}$ in Eq. (5.6), Eq. (5.7), and Eq. (5.8) as an arbitrary symmetric pattern dissimilarity measure. This allows us to derive from the theory of positive definite kernel functions conditions on the dissimilarity measure $d$ that imply the positive definiteness of the kernel function in Eq. (5.6), and hence the validity of the kernel functions in Eq. (5.7) and Eq. (5.8).

For an arbitrary pattern space $\mathcal{X}$, an element $x_0 \in \mathcal{X}$, and a symmetric function $\bar{k} : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$, it can be shown that the kernel function defined by

$$k(x, x') = \frac{1}{2}\left(\bar{k}(x, x') - \bar{k}(x, x_0) - \bar{k}(x_0, x') + \bar{k}(x_0, x_0)\right) \tag{5.9}$$

is positive definite if, and only if, the function $\bar{k}$ is conditionally positive definite [Berg *et al.* (1984)]. Recall that the class of conditionally positive definite functions is a larger class of kernels containing all positive definite kernels, as discussed in Sec. 4.2.1. If we substitute in Eq. (5.9)

$$\bar{k}(x, x') = -d(x, x')^2 \ ,$$

we obtain the proposed kernel function in Eq. (5.6), assuming that the distance of pattern $x_0$ to itself is zero. Consequently, the proposed kernel function is positive definite if the negated square of the underlying dissimilarity measure, $-d^2$, is conditionally positive definite. Examples of distance functions satisfying this condition are $d(x, x') = \|x - x'\|^p$ (for $0 < p \leq 1$) and $d(x, x') = \sin(x - x')$ [Berg *et al.* (1984)]. Hence, in some cases the validity of the kernel function can be proved. For the graph edit distance measure, unfortunately, general definiteness properties cannot be inferred.

The validity of the graph kernel functions presented in this section therefore cannot be established. However, as discussed in Sec. 4.3.4, there is strong evidence that indefinite kernel functions can sucessfully be applied to pattern classification under certain conditions. This issue will further be investigated in experiments in Sec. 6.6.4.

#### *Discussion*

The main advantage of the zero graph kernel is that it can easily be computed once the edit distance of graphs is available and does not require the underlying graphs to satisfy particular conditions. Hence, for fully developed graph matching systems based on edit distance, the application of this kernel function is straightforward and simple. The mapping of the graphs to the kernel feature space preserves the edit distance of graphs as the Euclidean distance, and a number of interesting geometrical properties can be derived from the embedding. Since the edit distance algorithm already copes with the time-consuming graph matching process, the evaluation of the kernel function afterwards is very efficient, and a novel graph matching procedure need not be defined by the kernel function. The definition of the kernel functions with respect to an underlying dissimilarity function makes the proposed kernel applicable to any data for which a distance measure can be defined. The only drawback of this kernel function is that its validity cannot be proved.

### 5.7 Convolution Edit Kernel

Convolution kernels are a class of kernel functions defined for composite objects [Haussler (1999)]. The idea is to decompose complex objects into smaller parts, for which a similarity function can more easily be defined or more efficiently be computed. Using a convolution operation, these similarities are then turned into a kernel function on the composite objects. For some example applications of convolution kernels, see Sec. 5.2. In this section, a convolution graph kernel based on the decomposition of two graphs into edit paths is proposed.

#### *Preliminaries*

In the following, assume that a composite object $x \in \mathcal{X}$ is given. The concept of decomposing a composite object into its parts is mathematically

denoted by a relation $R$, where $R(x_1, \ldots, x_d, x)$ represents the decomposition of $x$ into parts $\{x_1, \ldots, x_d\}$. This notation allows us to denote by $R^{-1}(x) = \{(x_1, \ldots, x_d) : R(x_1, \ldots, x_d, x)\}$ the set of decompositions of any $x \in \mathcal{X}$. For the definition of the convolution kernel, a kernel function $\kappa_i$ is required for each part of a decomposition $x_i$ $(1 \leq i \leq d)$. The general convolution kernel function can then be written as [Haussler (1999)]

$$k(x, x') = \sum_{\substack{(x_1,\ldots,x_d)\in R^{-1}(x) \\ (x'_1,\ldots,x'_d)\in R^{-1}(x')}} \prod_{i=1}^{d} \kappa_i(x_i, x'_i) \ . \tag{5.10}$$

This kernel function defines the similarity of two patterns $x$ and $x'$ by the sum, over all decompositions, of the similarity product of the parts of $x$ and $x'$. For a simple example, assume that the set of all decompositions of a graph $g = (V, E, \mu, \nu) \in \mathcal{X}$ is defined by $R^{-1}(g) = V$. This means that each of its nodes is a valid decomposition of $g$. If we use the function assigning 1 to nodes with the same label and 0 to nodes with a different label as underlying similarity measure $\kappa(u, v)$, the corresponding convolution kernel $k(g, g')$ for graphs $g = (V, E, \mu, \nu)$ and $g' = (V', E', \mu', \nu')$,

$$k(g, g') = \sum_{u\in V, v\in V'} \kappa(u, v) \ , \tag{5.11}$$

simply returns the number of pairs of nodes with matching label.

The kernel proposed in this section is based on the decomposition of pairs of graphs into edit paths. Technically, the decomposition of a graph is defined by an arbitrarily ordered sequence of some of its nodes and edges. For two graphs, the basic idea is to derive from decompositions of this kind valid edit paths, if possible. For the sake of simplicity, we first consider decompositions consisting of nodes only. That is, for a graph $g = (V, E, \mu, \nu)$ and a parameter $s_{max} \in \{1, 2, \ldots\}$, we define $R^{-1}(g) = \{(s, u_1, \ldots, u_s) : 1 \leq s \leq s_{max}$ and $u_i \in V$ and $u_i \neq u_j$ for $i \neq j\}$. A corresponding convolution kernel for graphs $g = (V, E, \mu, \nu)$ and $g' = (V', E', \mu', \nu')$ can then be defined by

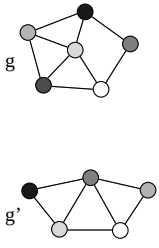
$$k(g, g') = \sum_{\substack{(s,u_1,\ldots,u_s)\in R^{-1}(g) \\ (s',u'_1,\ldots,u'_s)\in R^{-1}(g')}} \kappa_\delta(s, s') \prod_{i=1}^{s} \kappa_{rbf}(u_i, u'_i) \ , \tag{5.12}$$

using the standard RBF kernel $\kappa_{rbf}$ introduced in Table 4.1 for comparing node labels, and the Dirac kernel $\kappa_\delta$, that is, $\kappa_\delta(s, s') = 1$ if $s = s'$ and

$\kappa_\delta(s, s') = 0$ otherwise. Although the kernel function is defined for arbitrary decompositions of graphs into sequences of nodes, the Dirac kernel $\kappa_\delta$ eliminates all pairs of sequences with different length, which makes the product in the formula well-defined. Note that in the convolution kernel above, we use an RBF kernel to evaluate the similarity of node labels, but any other valid kernel function could be used as well.

Consequently, this kernel evaluates, for all combinatorial node sequences not larger than $s_{max}$, how similar pairs of nodes of two graphs are. Comparing the node labels of two node sequences $(u_1, \ldots, u_s)$ and $(u'_1, \ldots, u'_s)$, each from one of two graphs, can therefore be understood as determining how strong the distortion associated with the corresponding substitution of node labels $u_i \to u'_i$ is: If the labels $\mu(u_i)$ and $\mu'(u'_i)$ differ significantly, the resulting similarity $\kappa_{rbf}(u_i, u'_i)$ will be low, whereas the similarity for two nodes with similar label will be high. The product in the equation above can therefore be understood as a measure of similarity of node substitutions between graphs. In view of the edit distance concept, we simply define node substitutions to be explicitly encoded in the graph decomposition, through $u_i \to u'_i$ for all $i \in \{1, \ldots, s\}$, so that node deletions and insertions can implicitly be inferred from non-substituted nodes, $u \to \varepsilon$ for $u \in V \setminus \{u_1, \ldots, u_s\}$ and $\varepsilon \to u'$ for $u' \in V' \setminus \{u'_1, \ldots, u'_s\}$. In this manner, the decomposition of two graphs into ordered sequences of nodes can be interpreted as a node edit path between the two graphs. The product in Eq. (5.12) computes the similarity of an edit path by measuring the similarity of substituted nodes, and the sum of all edit path similarities is finally returned. This has the effect that the value of the kernel function is in fact determined by the number of high-similarity edit paths between two graphs.

This convolution kernel defined for nodes only is illustrated in Fig. 5.4. For two graphs $g$ and $g'$, four sample decompositions and their similarities are shown. Note that nodes are assumed to be labeled with a brightness value ranging from 1 (white) to 6 (black). The example in Fig. 5.4a illustrates two decompositions under consideration having different length. In this case, the Dirac kernel evaluates to $\kappa_\delta(5, 4) = 0$, and the decomposition is therefore not taken into account at all in the convolution sum in Eq. (5.12). The two decompositions in the second example (Fig. 5.4b) have the same length, but the brightness labels of corresponding nodes seem to be fairly different. To compare the brightness of nodes, we use an RBF kernel function with parameter $\sigma^2 = 1$. The resulting product amounts to a similarity value very close to zero. Note that in the similarity product, the




(a) (5,○,○,●,●,○) (4,●,●,○,○) $\kappa_\delta(5,4) = 0$

(b) (4,○,○,●,●) (4,●,●,○,○) $4! \cdot \kappa_{rbf}(3,6)\,\kappa_{rbf}(2,4)\,\kappa_{rbf}(6,1)\,\kappa_{rbf}(5,3) = 24 \cdot 5 \cdot 10^{-19} \simeq 0.0$

(c) (4,●,○,●,○) (4,○,○,●,○) $4! \cdot \kappa_{rbf}(5,4)\,\kappa_{rbf}(3,2)\,\kappa_{rbf}(6,6)\,\kappa_{rbf}(2,1) = 24 \cdot 0.049 \simeq 1.2$

(d) (5,●,○,●,○,○) (5,●,○,●,○,○) $5! \cdot \kappa_{rbf}(5,4)\,\kappa_{rbf}(3,2)\,\kappa_{rbf}(6,6)\,\kappa_{rbf}(2,1) \cdot \kappa_{rbf}(4,3) = 120 \cdot 0.018 \simeq 2.2$

Fig. 5.4 Illustration of the node convolution kernel in Eq. (5.12): Four example decompositions of $g$ and $g'$ into sequences of nodes

permutations of the decomposition of the four nodes are included as well (hence the factor 4!), since any simultaneous permutation of the two node sequences will result in exactly the same similarity value. The decompositions in Fig. 5.4c clearly constitute a better matching of nodes, since the brightness of corresponding nodes seems to be quite consistent, resulting in a similarity value of 1.2. Finally, the example in Fig. 5.4d demonstrates that the similarity of longer decompositions tends be higher. If five nodes are considered instead of four nodes, we obtain a similarity value of 2.2.

It is clear, from the example and the definition of the kernel function, that a single pair of nodes with completely different labels in a decomposition, which is equivalent to a very small factor in the product in Eq. (5.12), may decrease the product of similarities substantially, no matter how well the remaining nodes of the decomposition match. Conversely, since all combinations of node sequences are considered in the convolution kernel, it is sufficient to have a single high-similarity decomposition in order to obtain a high value of the convolution kernel.

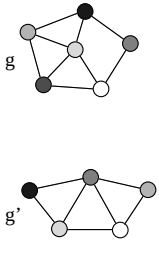
### *Definition*

In the following, we extend the convolution approach described above to edit paths consisting not only of node substitutions, but of edge substitutions as well. To this end, we define the decomposition of a graph

$g = (V, E, \mu, \nu)$ as a sequence of nodes and edges,

$$R^{-1}(g) = \{(s, c, C_1, C_2, w_1, \ldots, w_s) : \\ 1 \leq s \leq s_{max} \text{ and } c \in \{0,1\}^s \text{ and } C_1, C_2 \in \{0,1\}^{s \times s} \text{ and} \\ w_i \in V \cup E \text{ and } w_i \neq w_j \text{ for } i \neq j\} \ ,$$

where $c$ is a binary vector of length $s$ and $C_1, C_2$ are binary matrices of dimension $s \times s$. The role of $c$, $C_1$, and $C_2$ is similar to the role of the length parameter $s$ in the first component. The idea is to encode the structure of the decomposition in a single parameter, which can be used in turn to make sure that only consistent decompositions are compared. Vector $c = (c_1, \ldots, c_s)$ encodes the position of nodes and edges in the sequence $(w_1, \ldots, w_s)$, where $c_i = 1$ if $w_i$ is a node and $c_i = 0$ if $w_i$ is an edge. The compatibility of two sequences $(w_1, \ldots, w_s)$ and $(w'_1, \ldots, w'_s)$ can then be checked by computing the product of Dirac kernels $\prod_i \kappa_\delta(c_i, c'_i)$, which is 1 for compatible sequences and 0 for incompatible ones. Matrices $C_1$ and $C_2$ are similarly used to encode the adjacency structure of nodes and edges, where $C_1$ relates edges to their source node and $C_2$ relates edges to their target node. That is, a value 1 in $C_1$ indicates that an edge belongs to a source node, where row indices represent nodes of the decomposition and column indices represent edges of the decomposition. Again, a Dirac kernel can be applied to individual entries of $C_1$ and $C_2$ to determine whether the node and edge structure of two decompositions are compatible.




(a) $(5, c, C_1, C_2, \circ, \bullet, \bullet, \bullet)$ $\quad \kappa_{rbf}(2,1)\kappa_{rbf}(6,4)\kappa_{rbf}(5,3)$

$(5, c', C'_1, C'_2, \circ, \bullet, \bullet, \bullet)$ $\quad \simeq 0.0001$

$c = (1, 0, 0, 1, 1)$

$$(C_1)_{ij} = \begin{cases} 1 \text{ for } (i,j) \in \{(1,2),(1,3),(4,3),(5,2)\} \\ 0 \text{ otherwise} \end{cases}$$

$c = c'$ and $C_1 = C_2 = C'_1 = C'_2$

(b) $(5, c, C_1, C_2, \circ, \bullet, \bullet, \bullet)$ $\quad \kappa_{rbf}(2,2)\kappa_{rbf}(6,6)\kappa_{rbf}(5,4)$

$(5, c', C'_1, C'_2, \circ, \bullet, \bullet, \bullet)$ $\quad \simeq 0.37$

$c, c', C_1, C_2, C'_1, C'_2$ like in (a)

Fig. 5.5 Illustration of the convolution kernel in Eq. (5.13): Two example decompositions of $g$ and $g'$

Based on this graph decomposition, the corresponding convolution ker-

nel for graphs $g = (V, E, \mu, \nu)$ and $g' = (V', E', \mu', \nu')$ can be defined by

$$k(g, g') = \sum_{\substack{(s,c,C_1,C_2,w_1,\ldots,w_s)\in R^{-1}(g) \\ (s',c',C_1',C_2',w_1',\ldots,w_s')\in R^{-1}(g')}} \kappa_\delta(s,s')\kappa_\delta(c,c')\kappa_\delta(C_1,C_1')\kappa_\delta(C_2,C_2')\prod_{i=1}^{s}\kappa_{rbf}(w_i,w_i') \quad , \tag{5.13}$$

The various functions $\kappa_\delta$ are Dirac kernels returning 1 if the numbers ($s$ and $s'$), vectors ($c$ and $c'$), or matrices ($C_1$ and $C_1'$, $C_2$ and $C_2'$) are completely identical, and 0 otherwise. That is, if any of the consistency parameters $s, c, C_1, C_2$ of two decompositions does not match, the resulting value will be 0, and the two decompositions will not be taken into account. If the sequence of nodes and edges in two decompositions $(s, c, C_1, C_2, w_1, \ldots, w_s)$ and $(s', c', C_1', C_2', w_1', \ldots, w_s')$ does not correspond, for instance, because at position $i$ there is a node $w_i$ in the first decomposition and an edge $w_i'$ in the second decomposition, the vectors $c = (c_1, \ldots, c_i = 1, \ldots, c_s)$ and $c' = (c_1', \ldots, c_i' = 0, \ldots, c_s')$ will differ, and hence $\kappa_\delta(c, c') = 0$. Similarly, if the node and edge structure of the decompositions, encoded in $C_1, C_2, C_1', C_2'$ is not consistent, the decompositions will not contribute anything to the convolution sum, due to $\kappa_\delta(C_1, C_1')\kappa_\delta(C_2, C_2') = 0$. Again, RBF kernels are used to measure the similarity of substituted node and edge labels, possibly with different width parameter of the RBF kernel for nodes and edges. The parameter $s_{max}$ limiting the size of edit paths can be used to control the running time of the kernel computation. For the evaluation of the kernel function, it is sufficient to generate all possible valid edit paths up to a certain size without permutations.

In analogy to Fig. 5.4, two example decompositions of two graphs and the resulting similarity values are illustrated in Fig. 5.5. Note that in this illustration nodes are labeled with a brightness attribute (ranging from 0 for white to 6 for black) and edges are unlabeled. The decomposition in Fig. 5.5b results in a significantly higher similarity than the one in Fig. 5.5a, which corresponds very well with the observation that the node-to-node mapping in Fig. 5.5b is clearly better than that in Fig. 5.5a.

### *Properties*

Technically, the cumbersome encoding of the decomposition structure allows us to eliminate inconsistent decomposition from the kernel evaluation using only positive definite functions. Intuitively, the consistency check can

be understood as a procedure detecting whether two decompositions are actually equivalent to a valid set of node and edge substitutions. Provided that two decompositions are consistent, it is guaranteed that the sequences of nodes and edges can be turned into a set of node and edge substitutions such that edge substitutions are in accordance with node substitutions, by substituting each element $w_i$ of the first sequence with the corresponding element $w'_i$ of the second sequence. While only substitutions are modeled explicitly in decompositions, deletions and insertions of nodes and edges are implicitly accounted for by assuming that the remaining nodes and edges are deleted from the first graph and inserted into the second graph. In this manner, we obtain a valid edit path between two graphs from their decomposition.

In general, longer decompositions correspond to more combinatorial permutations of nodes and edges. Hence the similarity resulting from longer decompositions, representing edit paths with many substitutions, will tend to be higher. This behavior can also be observed in Fig. 5.4, where the two decompositions in Fig. 5.4c exhibit, in fact, a higher similarity (0.049) than the ones in Fig. 5.4d (0.018), but the larger number of permutations of five compared to four nodes finally results in a higher similarity of 2.2 in Fig. 5.4d compared to 1.2 in Fig. 5.4c. Consequently, the influence of edit paths with more substitutions (and therefore fewer deletions and insertions) in contrast to edit paths with fewer substitutions (and therefore more deletions and insertions) will tend to be stronger. In the context of edit distance, this effect can be seen in equivalence to penalty costs assigned to deletions and insertions. Hence, the convolution edit kernel aims at finding a preferably large set of substitutions between nodes and edges with similar labels, which is conceptually equivalent to computing the minimum-cost edit path in graph edit distance.

The evaluation of the convolution kernel is computationally inefficient because of the combinatorially large number of decompositions to be considered. Assume that two graphs under consideration consist of $m$ and $m'$ nodes, respectively, where $m > m'$ holds true. In the worst case, decompositions are generated by selecting $s$ nodes out of the $m$ (or $m'$) nodes of the graphs. In order to take all node-to-node correspondences into account, the sequence of nodes of one graph additionally have to be permuted, resulting in a factor of $s!$. The overall computational complexity of the convolution edit kernel can then be shown to be $O(m! \cdot m'^{s-1})$.

***Validity***

The convolution edit kernel defined in Eq. (5.13) is symmetric and positive definite and hence constitutes a valid graph kernel. The proof follows from the positive definiteness of the RBF kernel function [Shawe-Taylor and Cristianini (2004)], the positive definiteness of the Dirac kernel, $\sum_{i,j} c_i c_j \kappa_\delta(x_i, x_j) = \sum_i c_i^2 \geq 0$ for all $c_i, c_j \in \mathbb{R}$, and the positive definiteness of convolution kernels [Haussler (1999)].

To increase the penalty cost implicitly associated with deletions and insertions, and hence make the preference of substitutions over deletions and insertions even more significant, one can also extend the label similarity function by adding a positive constant $b > 0$ to the RBF function, that is, use $\kappa_{rbf}(w_i, w_i') + b$ instead of $\kappa_{rbf}(w_i, w_i')$ in Eq. (5.13). The result is an increase of the similarity of longer edit paths. With this extension, the kernel function will no longer be positive definite, but at least conditionally positive definite, which means that the kernel function is still applicable to some important, but not all kernel machines (see discussion in Sec. 4.3.4).

***Discussion***

The convolution graph kernel described in this section is mainly interesting because it makes edit distance based graph matching suitable for kernel machines. Among the valid graph kernels proposed in this book, this kernel function models the structural matching of edit distance most accurately. The kernel function is positive definite and therefore applicable to kernel machines without restriction. The key idea of the proposed graph kernel is to decompose two graphs into sequences of nodes and edges such that a valid edit path between them can be derived. Unlike in the standard graph edit distance algorithm, the kernel function does not compute the minimum cost edit path, but rather accumulates the similarity of all possible edit paths between two graphs. If two graphs are similar, there will be at least one high-similarity edit path between them, whereas for dissimilar graphs, there will only exist low-similarity edit paths. In analogy to deletion and insertion penalty costs in the edit distance context, the convolution kernel provides a way to control how strongly substitutions are favored over deletions and insertions by means of parameters of underlying kernel functions. The only drawback is the exponential complexity of the kernel computation. However, a complexity parameter can be used to reduce the running time of the computation. An open question to be addressed in

experiments is whether it is beneficial to re-define graph edit distance in terms of a positive definite kernel function, such as the convolution kernel in this section, instead of simply turning graph edit distance into a (possibly indefinite) similarity function, as proposed in previous sections.

## 5.8 Local Matching Kernel

The graph kernel function proposed in this section is closely related to the convolution edit kernel described in the previous section. The local matching kernel belongs to the class of convolution kernels [Haussler (1999)] as well, based on the same idea of decomposing graphs into sequences of nodes and edges and evaluating the similarity of the resulting substructures. The main difference is that the local matching kernel does not require edit paths to be consistent, but matches small subgraphs of the two graphs. The interpretation in the previous section, where the decompositions of two graphs can be regarded as an edit path between them, cannot be adopted for the local matching kernel, since node and edge substitutions need not be consistent in this case. This kernel function is simpler to define and compute, and its comparison to the previously described convolution edit kernel may serve as a first step towards investigating if the node and edge structure is relevant for graph matching, or if accumulating local substructure similarities is sufficient.

Convolution kernels are usually defined with respect to a decomposition of patterns expressed in terms of a relation between patterns and their parts. In Eq. (5.10) in the previous section, the convolution kernel function is given in its most general form. The local matching kernel described in this section is based on a decomposition of graphs into sequences of some nodes and the edges these nodes are connected to, which is to a certain degree analogous to the previously described convolution edit kernel. Hence, the idea is to select nodes from each graph and compare the labels of these nodes and their edges.

### *Definition*

In the following, let $g = (V, E, \mu, \nu)$ be a graph, and for each $u \in V$ define the set of edges originating in $u$ by $E_u = \{u\} \times \{v \in V : (u, v) \in E\}$. Formally, the local matching decomposition of $g$ is expressed in terms of a

relation $R$ between the parts of $g$ and graph $g$ itself by defining

$$R^{-1}(g) = \Big\{ (s, (u_1, n_1, e_{u_1}^{(1)}, \ldots, e_{u_1}^{(n_1)}), \ldots, (u_s, n_s, e_{u_s}^{(1)}, \ldots, e_{u_s}^{(n_s)})) : \\ 1 \leq s \leq s_{max} \text{ and } u_i \in V \text{ and } u_i \neq u_j \text{ for } i \neq j \text{ and} \\ 0 \leq n_i \leq |E_{u_i}| \text{ and } e_{u_i}^{(p)} \in E_{u_i} \text{ and } e_{u_i}^{(p)} \neq e_{u_i}^{(q)} \text{ for } p \neq q \Big\} \ .$$

The first component $s$ specifies the number of nodes $u_1, \ldots, u_s$ present in the decomposition, up to a maximum number of nodes $s_{max}$. The remaining components of type $(u_i, n_i, e_{u_i}^{(1)}, \ldots, e_{u_i}^{(n_i)})$ represent a node $u_i$, together with $n_i$ of the $|E_{u_i}|$ edges originating in $u_i$. Note that edges terminating in $u_i$ are not directly associated with the decomposition of their target node $u_i$, but are rather assigned to decompositions of their source nodes. Clearly, components $(u_i, n_i, e_{u_i}^{(1)}, \ldots, e_{u_i}^{(n_i)})$ can be regarded as substructures of $g$ each consisting of a single node and some, possibly not all, of its adjacent edges. These local substructures of graphs, or partial subgraphs, can then be used for graph matching.

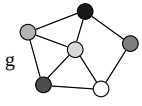
For two graphs $g = (V, E, \mu, \nu)$ and $g' = (V', E', \mu', \nu')$, the corresponding local matching convolution kernel is defined by

$$k(g, g') = \tag{5.14}$$
$$\sum_{\substack{(s,(u_1,n_1,e_{u_1}^{(1)},\ldots,e_{u_1}^{(n_1)}),\ldots)\in R^{-1}(g) \\ (s',(u_1',n_1',e_{u_1'}^{\prime(1)},\ldots,e_{u_1'}^{\prime(n_1')}),\ldots)\in R^{-1}(g')}} \kappa_\delta(s, s') \prod_{i=1}^{s} \kappa_{rbf}(u_i, u_i') \kappa_\delta(n_i, n_i') \prod_{p=1}^{n_i} \kappa_{rbf}(e_{u_i}^{(p)}, e_{u_i'}^{\prime(p)}) \,,$$

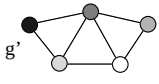
It is clear that for two decompositions with a different number of nodes $s \neq s'$, the resulting product will be zero, due to the Dirac kernel, $\kappa_\delta(s, s') = 0$. Similarly, if a node $u_i$ in the decomposition contains a different number of edges than the corresponding node in the other graph $u_i'$, $n_i \neq n_i'$, the resulting similarity will be zero as well, due to $\kappa_\delta(n_i, n_i') = 0$. Hence, only the similarity of decompositions that are consistent in terms of the number of nodes and edges are taken into account, that is, contribute a non-zero similarity value to the convolution kernel.

In this kernel function, the consistency of decompositions can explicitly be modeled in the definition of the decomposition relation $R$. Since no information about common edges needs to be shared between nodes of the decomposition, there is no need for more complex consistency parameters, such as the matrices required in the previous section. Again, the similarity of nodes and edges is determined by the similarity of their labels, measured with an RBF kernel function $\kappa_{rbf}$. Two example decompositions are shown

in Fig. 5.6. The local matching kernel reflects the obvious fact very well that the matching in Fig. 5.6a is worse than that in Fig. 5.6b. In this particular example, the similarity of graphs only depends on node labels and the presence of a certain number of edges, but not on the actual edges themselves, since all edges are unlabeled.




(a) $(2, (\circ, 2, \ldots), (\bullet, 1, \ldots))$ $(2, (\bullet, 2, \ldots), (\circ, 1, \ldots))$ $\kappa_{rbf}(2,6)\kappa_{rbf}(4,3) \simeq 4{\cdot}10^{-8}$




(b) $(2, (\circ, 2, \ldots), (\bullet, 1, \ldots))$ $(2, (\circ, 2, \ldots), (\circ, 1, \ldots))$ $\kappa_{rbf}(2,2)\kappa_{rbf}(4,3) \simeq 0.37$

Fig. 5.6 Illustration of the local matching kernel in Eq. (5.14): Two example decompositions of $g$ and $g'$

### *Properties*

The interpretation of the local matching kernel is less persuasive than that of the previously described convolution edit kernel, where the kernel similarity of graphs is closely related to edit paths between graphs. The local matching kernel is based on the idea to derive a graph similarity measure from the accumulation of the similarity of local substructures. For dissimilar graphs, it is assumed that only a few, if any at all, local substructures exhibit similar node and edge labels. Two similar graphs, on the other hand, will contain a substantial amount of nodes and edges with similar labels, which should be reflected in a higher graph similarity according to the local matching kernel. The local matching kernel does not fully exploit the node and edge structure of a graph, since the structural matching is limited to nodes and some edges originating in these nodes, not to complete subgraphs or valid edit paths. Hence, the local matching kernel will only be successful in cases where the node and edge labels are more important for the matching of graphs than the graph structure given by nodes and edges.

In analogy to the convolution edit kernel, the computational complexity of the local matching kernel amounts to $O(m! \cdot m'^{s-1})$, where $s$ denotes the length of decompositions and $m$ and $m'$ denote the size of the two involved graphs (where $m > m'$).

### *Validity*

The local matching kernel in Eq. (5.14) is a valid kernel function. The validity of the kernel can be derived from the symmetry and the positive definiteness of the underlying kernel functions $\kappa_\delta, \kappa_{rbf}$ and the validity of the convolution kernel [Haussler (1999)].

In analogy to the kernel function described in the previous section, the RBF kernel functions measuring the similarity of node and edge labels in Eq. (5.14) can be modified to favor longer sequences over shorter ones, by adding a positive constant to the RBF function. While the motivation for such a modification may be less evident for the local matching kernel than the convolution edit kernel, the two methods will be evaluated based on the same set of extended RBF kernels for the sake of consistency.

### *Discussion*

The difference between the convolution edit kernel in the previous section and the local matching kernel in this section is that the decomposition of the former is more restrictive than that of the latter. In cases where the convolution edit kernel outperforms the local matching kernel, one can conclude that knowing which node is linked to which other node by an edge is relevant for the matching. In this case, simply comparing isolated nodes and their edges is not a sufficiently powerful model of graph similarity. Conversely, in cases where the two methods perform equally well, it will be more important that the labels of nodes and their edges are actually present in a graph, but not which node is connected to which node. Summarizing, the graph kernel described in this section is simpler to define and pursues a more basic matching of graphs than the convolution edit kernel. On the other hand, it will only be useful for graph matching problems, where the structure of graphs is of minor importance, and is therefore not universally applicable. An open issue, to be addressed in experiments, is whether the convolution edit kernel provides us not only theoretically, but also in practice with a stronger graph matching model than the local matching kernel. This question is of particular interest in view of the fact that the classification performance of standard vector kernels, such as the RBF kernel, the polynomial kernel, and the sigmoid kernel, is often comparable, independent of the actual kernel function used, although their mathematical definition and properties are quite different [Schölkopf and Smola (2002)].

## 5.9 Random Walk Edit Kernel

In this section, a graph kernel based on random walks is proposed. The basic idea of random walk kernels is to define the similarity of graphs by comparing random walks in two graphs [Gärtner (2002); Kashima and Inokuchi (2002); Gärtner *et al.* (2003); Kashima *et al.* (2003); Borgwardt *et al.* (2005); Borgwardt and Kriegel (2005)]. For instance, one of the most elegant random walk kernels computes the number of matching random walks in two graphs. A key observation is that this computation can efficiently be realized by means of the direct product of two graphs, without having to explicitly enumerate random walks in graphs [Gärtner *et al.* (2003)]. This allows us to consider random walks of arbitrary length. Random walk kernels are undoubtedly very efficient, but on noisy data their accuracy is often unsatisfactory. For this reason, an extension to a standard random walk kernel is proposed in this section to make the kernel more robust and therefore applicable to noisy graph data as well. The idea is to integrate information from the global matching of graphs into the otherwise locally defined random walk kernel. To this end, we first compute the edit distance of graphs and use the optimal edit path to define the adjacency matrix of the direct product in an extended way to enhance the robustness of the random walk kernel.

### *Preliminaries*

First, a standard random walk kernel for discrete attributes [Gärtner *et al.* (2003)] and its extension to continuously labeled graphs [Borgwardt *et al.* (2005)] is described. For this purpose, the definition of the direct product of two graphs is required.

**Definition 5.1 (Direct Product Graph).** *The direct product of two graphs* $g = (V, E, \mu, \nu)$ *and* $g' = (V', E', \mu', \nu')$ *is the graph* $(g \times g') = (V_\times, E_\times, \mu_\times, \nu_\times)$ *defined by*

$$
\begin{aligned}
V_\times &= \{(u, u') \in V \times V' : \mu(u) = \mu'(u')\} \\
E_\times &= \{((u, u'), (v, v')) \in V_\times \times V_\times : \\
&\qquad (u, v) \in E \text{ and } (u', v') \in E' \text{ and } \nu(u, v) = \nu'(u', v')\} \ .
\end{aligned}
$$

*The labeling functions of the direct product graph are defined according to* $\mu_\times(u, u') = \mu(u) = \mu'(u')$ *and* $\nu_\times((u, u'), (v, v')) = \nu(u, v) = \nu'(u', v')$.

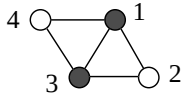
From two graphs, the direct product graph is obtained by identifying nodes and edges in $g$ and $g'$ having the same label. The set of nodes of the direct product graph $(g \times g')$ consists of all pairs of nodes $(u, u')$, one from $g$ and one from $g'$, with identical labels. Edges are inserted between nodes $(u, u')$ and $(v, v')$ of the direct product graph if $u$ and $v$ are connected by an edge in $g$, $u'$ and $v'$ are connected by an edge in $g'$, and both edges have the same label. Hence, the direct product graph, by definition, identifies the compatible nodes and edges in two graphs. So far, determining the similarity of labels is restricted to evaluating whether two discrete labels are identical or not. In this sense, the direct product of two graphs is similar, but not identical, to the association graph [Levi (1972)], as the association graph encodes which pairs of nodes are compatible in terms of present or absent edges, while in the direct product graph only the presence of edges is considered.

In Fig. 5.7, four example direct product graphs are illustrated. In this example, nodes are labeled with a single discrete attribute (*black* or *white*) and edges are unlabeled. Note that the graphs $g$ and $g'$ become increasingly similar from Fig. 5.7a to 5.7d. Accordingly, one can observe that the more similar two graphs are, the larger is their direct product graph. By convention, unlabeled edges are modeled by a single dummy label assigned to every edge, so that in the definition of the direct product graph, all pairs of edges satisfy the condition of identical labels.

The adjacency matrix of a graph is a matrix consisting of one row and one column per node. The idea is to represent the structure of a graph in a matrix. Formally, the adjacency matrix $A_\times$ of the direct product $(g \times g')$ of two graphs is defined by

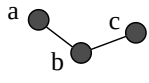
$$[A_\times]_{(u,u'),(v,v')} = \begin{cases} 1 \text{ if } ((u,u'),(v,v')) \in E_\times \ , \\ 0 \text{ otherwise } . \end{cases} \tag{5.15}$$

Hence, rows and columns of the adjacency matrix of the direct product graph are indexed by nodes $(u, u')$ of the direct product $(g \times g')$, that is, pairs of nodes of the underlying graphs $g$ and $g'$. From the definition above it is clear that $A_\times$ is a $|V_\times| \cdot |V_\times|$-matrix containing at the position in row $(u, u')$ and column $(v, v')$ value 1 if node $(u, u')$ is connected to node $(v, v')$ by an edge in $(g \times g')$, and value 0 otherwise. Given a weighting parameter $\lambda \geq 0$, one can then derive a kernel function for graphs $g$ and $g'$ from the adjacency matrix $A_\times$ of their direct product $(g \times g')$ by defining [Gärtner

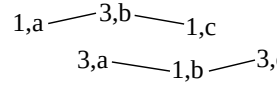



$g$

(a)



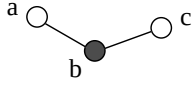

$g'$



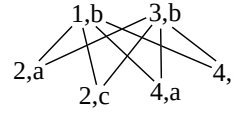

$g \times g'$

$k(g, g') = 6.9$

(b)



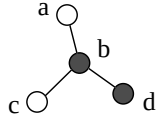

$g'$



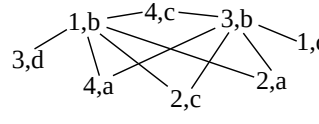

$g \times g'$

$k(g, g') = 8.3$

(c)



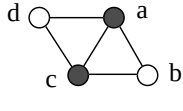

$g'$



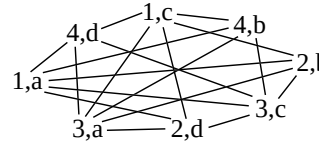

$g \times g'$

$k(g, g') = 11.0$

(d)




$g'$




$g \times g'$

$k(g, g') = 14.6$

Fig. 5.7 Discrete random walk kernel based on the direct product

*et al.* (2003)]

$$k(g, g') = \sum_{i,j=1}^{|V_\times|} \left[ \sum_{n=0}^{\infty} \lambda^n A_\times^n \right]_{ij} . \tag{5.16}$$

To evaluate this kernel for two graphs, one has to construct the direct product of the two graphs, sum up weighted powers of the direct product adjacency matrix, and return the sum of all elements of the resulting matrix. For infinite sums, it is sufficient to consider a finite number of addends only, since choosing $\lambda < 1$ will make the contribution of $\lambda^n A_\times^n$ to the overall sum insignificant for large $n$. In the experiments, the sum will therefore only be evaluated for $n = 0, 1, \ldots, t$, where $t$ is a parameter to be defined. In Fig. 5.7, the random walk kernel similarity of four pairs of graphs is illustrated ($\lambda = 0.1$). Note that the resulting kernel similarity reflects the intuitive observation that the similarity increases from Fig. 5.7a to 5.7d.

From the validity of a kernel function $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ one can derive that there exists a vector space $\mathcal{H}$ with an inner product being equal to the kernel function and a mapping $\Phi : \mathcal{X} \to \mathcal{H}$ from the original pattern space into this vector space (see Sec. 4.2.2). While the mapping into the vector space is know to exist, it is usually not possible to explicitly construct the function mapping patterns into vectors in closed form. In view of this, the random walk kernel defined above is particularly interesting for one reason: The definition of the random walk kernel is not only very elegant, but also offers an intuitive interpretation of the underlying graph similarity measure and provides an explicit construction of the corresponding mapping from the space of graphs into a vector space [Gärtner *et al.* (2003)]. For the construction of the mapping function $\Phi$, the definition of walks in graphs is required. Given a graph $g = (V, E, \mu, \nu)$, let the function $w_n(g)$ denote the set of walks in graph $g$ consisting of $n$ edges. That is,

$$\begin{aligned}
w_0(g) &= V \\
w_1(g) &= \{(u_1, e_1, u_2) \in V \times E \times V : e_1 = (u_1, u_2)\} \\
w_2(g) &= \{(u_1, e_1, u_2, e_2, u_3) \in V \times E \times V \times E \times V : \\
&\qquad e_1 = (u_1, u_2), e_2 = (u_2, u_3)\} \\
w_n(g) &= \{(u_1, e_1, u_2, e_2, \ldots, u_n, e_n, u_{n+1}) \in V \times E \times \ldots \times E \times V : \\
&\qquad e_i = (u_i, u_{i+1}) \text{ for } i = 1, \ldots, n\} .
\end{aligned}$$

A walk in a graph is obtained by starting at a node and walking through the graph by randomly traversing edges. The sequence of node and edge labels corresponding to such a walk is defined by

$$\rho : (u_1, e_1, , \ldots, e_n, u_{n+1}) \mapsto (\mu(u_1), \nu(e_1), \ldots, \nu(e_n), \mu(u_{n+1})) .$$

A feature vector space $\mathcal{H}$ can then be constructed by defining one feature for each possible sequence of node and edge labels. That is, the components of vectors in $\mathcal{H}$ are related to sequences of node and edge labels. For a graph $g = (V, E, \mu, \nu) \in \mathcal{X}$, the mapping from the space of graphs to the vector space can be denoted by $\Phi : \mathcal{X} \to \mathcal{H}$, $g \mapsto \Phi(g) = (\Phi_{s_1}(g), \Phi_{s_2}(g), \ldots)$, where each vector component $\Phi_s(g)$ is indexed by a single sequence $s$ of node and edge labels. Since there are obviously infinitely many different node and edge label sequences, the dimension $|\{s_1, s_2, \ldots\}|$ of the resulting vector space if infinite. More concretely, for a label sequence $s = (s_1, s_2, \ldots, s_{2n+1})$ containing $n$ edge labels, define the vector component $\Phi_s(g)$ by

$$\Phi_s(g) = \lambda^{\frac{n}{2}} \left|\{w \in w_n(g) : \rho(w) = s\}\right| \quad , \tag{5.17}$$

where $\lambda \geq 0$ is a weighting factor. Hence, $\Phi_s(g)$ is the number of walks in graph $g$ with node and edge label sequence $s$, additionally weighted by $\lambda^{\frac{n}{2}}$. Each component of a vector in $\mathcal{H}$ therefore represents the number of walks with corresponding node and edge label sequence. For this mapping $\Phi : \mathcal{X} \to \mathcal{H}$, it can be shown [Gärtner *et al.* (2003)] that the inner product of graphs mapped to the feature space is equal to $k$ in Eq. (5.16), $\langle \Phi(g), \Phi'(g) \rangle = k(g, g')$, where the parameter $\lambda$ in Eq. (5.16) and 5.17 is the same. That is, the vector space $\mathcal{H}$ defined by the mapping $\Phi$ is the kernel feature space corresponding to $k$. For the random walk kernel in Eq. (5.16), it is therefore possible to explicitly construct the function $\Phi$ mapping graphs to the kernel feature space.

For an example, consider the computation of the random walk kernel for the graphs $g$ and $g'$ in Fig. 5.7a. Note that, in this example, a color label *black* or *white* is attached to nodes, and edges are unlabeled. The adjacency matrix of the corresponding direct product is

$$A_\times = \begin{pmatrix} 0&1&0&0&0&0 \\ 1&0&1&0&0&0 \\ 0&1&0&0&0&0 \\ 0&0&0&0&1&0 \\ 0&0&0&1&0&1 \\ 0&0&0&0&1&0 \end{pmatrix} \quad ,$$

or any permutation thereof. Computing the infinite sum in Eq. (5.16) ($\lambda = 0.1$), we obtain $k(g, g') = 6 \cdot 0.1^0 + 8 \cdot 0.1^1 + 12 \cdot 0.1^2 + 16 \cdot 0.1^3 + 24 \cdot 0.1^4 + 32 \cdot 0.1^5 + 48 \cdot 0.1^6 + 64 \cdot 0.1^7 + \ldots = 6.93877$. On the other hand, we can also directly compute the inner product in the vector space $\langle \Phi(g), \Phi(g') \rangle$ instead of the kernel function $k(g, g')$, since the explicit mapping into the

kernel feature space is known. Graph $g$ in Fig. 5.7 obviously contains a large number of walks, but graph $g'$ in Fig. 5.7a consists of three black nodes only. Hence, no walk in $g'$ will contain nodes labeled *white*, and therefore $\Phi_s(g') = 0$ for all sequences $s$ containing a node label *white*. Accordingly, since the objective is the computation of $\langle \Phi(g), \Phi(g') \rangle = \sum_s \Phi_s(g) \cdot \Phi_s(g')$, it is perfectly sufficient to consider walks consisting of black nodes only, in $g'$ as well as in $g$. In the remainder of this example, considerations will therefore be restricted to nodes labeled *black*. In $g$, there are only two walks of a fixed length: The walk alternating between node 1 and node 3, either starting at node 1 or node 3. Hence, $\Phi_s(g) = 2\lambda^{\frac{n}{2}}$ for a sequence $s$ containing $n$ edges, using Eq. (5.17). The number of walks of length $n$ in $g'$ turns out to be $2 \cdot 2^{\frac{n+1}{2}}$ if $n$ is odd and $3 \cdot 2^{\frac{n}{2}}$ if $n$ is even, which can be shown using a simple inductive argument. Hence, for a sequence $s$ of odd length $n$, we get $\Phi_s(g') = 2 \cdot 2^{\frac{n+1}{2}} \cdot \lambda^{\frac{n}{2}}$, and for a sequence $s$ of even length $n$, we get $\Phi_s(g') = 3 \cdot 2^{\frac{n}{2}} \cdot \lambda^{\frac{n}{2}}$. Consequently,

$$
\begin{aligned}
\langle \Phi(g), \Phi(g') \rangle &= \sum_s \Phi_s(g) \cdot \Phi_s(g') = \\
&= \sum_{n=1,3,5,\ldots} 2\lambda^{\frac{n}{2}} \cdot 2 \cdot 2^{\frac{n+1}{2}} \lambda^{\frac{n}{2}} + \sum_{n=0,2,4,\ldots} 2\lambda^{\frac{n}{2}} \cdot 3 \cdot 2^{\frac{n}{2}} \lambda^{\frac{n}{2}} \\
&= 2 \sum_{n=0}^{\infty} \left( 4\lambda \cdot \lambda^{2n} 2^n + 3 \cdot \lambda^{2n} 2^n \right) \\
&= 2 \frac{4\lambda + 3}{1 - 2\lambda^2} = 6.93877 \quad \text{(for } \lambda = 0.1) \ .
\end{aligned}
$$

As expected, the inner product $\langle \Phi(g), \Phi(g') \rangle$ is equal to the kernel function $k(g, g')$. Due to the simple structure of the graph in Fig. 5.7a, it is possible, in this particular example, to enumerate all walks that are relevant for the computation of the inner product. If we consider the slightly different graph in Fig. 5.7b, however, it becomes substantially more difficult to find a formula for explicitly computing the inner product, while the computation of the kernel function only requires the construction of the adjacency matrix of the direct product graph and a few matrix computations.

Summarizing, the random walk kernel offers a solid interpretation of the underlying kernel similarity measure — high similarity means a large number of matching walks in two graphs — and simultaneously an elegant and efficient way to compute the kernel function using the direct product of graphs. Yet, the applicability of the random walk kernel defined above is seriously limited in that the kernel only takes into account whether labels are equal or different, but not whether they are similar. This means that

the kernel will only be successful on discretely labeled graphs, but not on continuously labeled graphs. If two walks differ only in a single label by a small amount, the two walks are considered completely different and are therefore not taken into account. For this reason, an extension to the random walk kernel has been proposed [Borgwardt *et al.* (2005)] to allow for a certain amount of error-tolerance in the matching of graphs. The extended kernel is assumed to be more suitable for graphs containing a significant amount of noise and continuously labeled attributes extracted from real-world data. The basic idea is not to evaluate if two walks are identical, but rather if they are similar. To this end, the modified direct product of two graphs needs to be defined first, which differs from the direct product in Def. 5.1 only in the absence of the equality conditions of node and edge labels.

**Definition 5.2 (Modified Direct Product Graph).** *The modified direct product of two graphs* $g = (V, E, \mu, \nu)$ *and* $g' = (V', E', \mu', \nu')$ *is the graph* $(g \times g') = (V_\times, E_\times, \mu_\times, \nu_\times)$ *defined by*

$$V_\times = V \times V'$$
$$E_\times = \{((u,u'),(v,v')) \in V_\times \times V_\times : (u,v) \in E, (u',v') \in E'\} \ .$$

The adjacency matrix of this modified direct product graph can then be defined according to [Borgwardt *et al.* (2005)]

$$[A_\times]_{(u,u'),(v,v')} = \begin{cases} k_{step}((u,u'),(v,v')) & \text{if } ((u,u'),(v,v')) \in E_\times \\ 0 & \text{otherwise} \ , \end{cases} \tag{5.18}$$

where the kernel function $k_{step}$ measuring the similarity of pairs of nodes $(u, u')$ and $(v, v')$ is given by

$$k_{step}((u,u'),(v,v')) = k_{rbf}(u,u') \cdot k_{rbf}((u,v),(u',v')) \cdot k_{rbf}(v,v') \ . \tag{5.19}$$

Here, we use RBF kernels for measuring the similarity of node labels and edge labels, but any other valid kernel function could be used instead. The modified adjacency matrix defined above can be interpreted as a fuzzy adjacency matrix assigning higher adjacency values to nodes of the product graph $(g \times g')$ if the corresponding pairs of nodes and pairs of edges of the original graphs $g$ and $g'$ have similar labels, and lower adjacency values otherwise [Borgwardt *et al.* (2005)]. The modified random walk kernel is obtained by plugging the adjacency matrix $A_\times$ from Eq. (5.18) into the kernel function $k$ in Eq. (5.16).

The modified random walk kernel is significantly more powerful than the standard random walk kernel, but may still perform poorly on certain

data sets in contrast to edit distance based nearest-neighbor classifiers. The modified random walk kernel is defined by accumulating the similarity of walks in two graphs. For some graph representations, however, there may be global matching constraints that need to be taken into account for a successful graph matching. In such as case, the optimization of a global matching criterion, such as the minimum-cost edit path in graph edit distance, may be more appropriate than summing up local graph similarities. Experiments confirm that the modified random walk kernel and graph edit distance address the graph matching problem in complementary ways, and one approach usually performs significantly worse or better than the other one. Therefore a method is proposed in the following to bring together the best from both worlds: The flexibility of graph edit distance and the power and efficiency of random walk kernels.

### *Definition*

The key idea of the random walk edit kernel is to enhance the random walk kernel defined above by information obtained from the minimum-cost edit path between graphs. This allows us to integrate global matching information into the locally defined matching process of the random walk kernel. To this end, assume that an optimal edit path from $g$ to $g'$ has already been computed, and let $S = \{u_1 \to u'_1, u_2 \to u'_2, \ldots\}$ denote the set of node substitutions present in the optimal edit path. The enhanced adjacency matrix of the modified direct product graph $(g \times g')$ in Def. 5.2 is then defined by

$$[A_\times]_{(u,u'),(v,v')} = \begin{cases} k_{step}((u,u'),(v,v')) & \text{if } ((u,u'),(v,v')) \in E_\times \\ & \text{and } u \to u' \in S \\ & \text{and } v \to v' \in S \ , \\ 0 & \text{otherwise} \ , \end{cases} \tag{5.20}$$

where $k_{step}$ is the function defined in Eq. (5.19). The random walk kernel enhanced by edit distance is then defined according to the standard random walk kernel function

$$k(g,g') = \sum_{i,j=1}^{|V_\times|} \left[ \sum_{n=0}^{\infty} \lambda^n A_\times^n \right]_{ij} . \tag{5.21}$$

The edit distance enhanced random walk kernel differs from the random walk kernel for discretely labeled graphs and the modified random walk kernel for continuously labeled graphs only in the definition of the adjacency matrix $A_\times$ of the direct product.

### *Properties*

The random walk kernel enhanced by edit distance evaluates the similarity of two graphs by the number of walks in both graphs with similar labels. The similarity of labels is evaluated by means of RBF kernels, or any other valid kernel function. The modification proposed in this section eliminates from the kernel computation all pairs of random walks whose nodes violate the optimal node-to-node correspondence identified by the minimum-cost edit path between the two graphs. That is, the global matching constraint is that only those node-to-node mappings are considered in the random walk in either graph that correspond to node substitutions in the optimal edit path.

Once the direct product graph $(g \times g')$ and its adjancency matrix $A_\times$ are computed, the evaluation of the kernel function $k(g, g')$ is quite efficient. In contrast to the inefficient edit distance computation, which is needed for the definition of $A_\times$ in Eq. (5.20), computing the kernel function will be substantially faster. The overall computational complexity of the random walk edit kernel is therefore equal to that of the edit distance algorithm: $O(m^{m'})$ for two graphs of size $m$ and $m'$ (where $m > m'$).

### *Validity*

The random walk kernel for discretely labeled graphs [Gärtner *et al.* (2003)] and the random walk kernel for continuously labeled graphs [Borgwardt *et al.* (2005)] can be understood as convolution kernels [Haussler (1999)] and are both positive definite, which means that they constitute valid kernel functions. The proposed random walk kernel enhanced by edit distance uses the optimal edit path resulting from an edit distance computation, which is not positive definite, and hence this kernel is not valid in general. However, because of its close connection to the valid random walk kernel, the proposed kernel is nevertheless assumed to be successfully applicable to difficult graph matching problems. For a more thorough discussion about applying invalid kernels to kernel machines, the reader is referred to Sec. 4.3.4.

### *Discussion*

The random walk kernel enhanced by edit distance is mainly intended to serve as an alternative to using either the modified random walk kernel for continuously labeled graphs or the edit distance method. The definition of

the proposed kernel is based on the observation that in some cases it may be advantageous to include global information into the graph matching process. For this purpose, the edit distance enhanced random walk kernel first computes the edit distance of two graphs and then uses the node-to-node correspondences of the optimal edit path to drive the evaluation of random walks. This procedure guarantees that only those random walks that satisfy the edit distance matching contribute a certain amount to the overall graph similarity. Thus, in the modified random walk kernel, the direct product graph consists of any pair of nodes, while the proposed enhanced kernel is restricted to those pairs of nodes that correspond to node substitutions. The proposed modifications are assumed to make the random walk kernel more robust against large amounts of noise and structural variation between graphs from the same class.

# Chapter 6

# Experimental Results

In this chapter, an experimental evaluation of the graph kernels described in the previous chapter is given. The main objective is to confirm empirically that the proposed graph kernels are applicable to difficult graph classification problems. A key issue to be investigated is the question whether theoretical properties of the kernel functions are also reflected in practical experiments. The proposed kernels will be compared to the performance of traditional edit distance based nearest-neighbor classifiers. These standard classifiers have proved to be sufficiently flexible and accurate for graphs extracted from noisy data. The most straightforward and most important classifier performance measure is certainly the classification accuracy, that is, the fraction of correctly classified patterns. A second measure of performance is the running time of a classifier. In graph matching, the computational time and space complexity of matching algorithms is of crucial importance, because computationally inefficient graph classifiers will only be applicable to small graphs.

The experimental results reported in this chapter are all derived on strictly independent test sets. The training of classifiers is performed on a training set and the optimization of classifier parameters on a validation set. Finally, the validated systems are applied to the patterns of an independent test set, and the resulting performance is reported. The pattern classification tasks considered in this book include the recognition of line drawings, the recognition of scene images, the classification of diatoms, the classification of fingerprints, and the classification of molecules. The line drawing data set is based on semi-artificial data, but the four other data sets have been extracted from real-world data. The data sets are considered to be difficult because of heavily overlapping graph classes and significant amounts of distortion.

In the following, the graph data sets used in the experiments are first described in detail. For a summary of the data set characteristics, see Tables 6.2 and 6.3 (p. 146f). Note that a larger number of sample patterns from the data sets are illustrated in Appendix A. Next, an experimental evaluation of a standard edit distance based nearest-neighbor classifier is given, and finally the performance of the graph kernels proposed in the previous chapter is addressed. A summary of the experimental results is provided in Sec. 6.7.

## 6.1 Line Drawing and Image Graph Data Sets

### 6.1.1 *Letter Line Drawing Graphs*

The letter graph data set is based on one of the simplest and most intuitive ways to define graphs. Graphs are usually visualized by drawing circles representing nodes and lines representing edges. Hence, it seems to be natural to interpret points in the plane as nodes (implicitly associated with their coordinates) and lines connecting these points by edges. The idea of the letter graph data set is to interpret line drawings as graphs. To arrive at an intuitive and simple classification problem, line drawings representing capital letters consisting of straight lines only are considered.

The letter graph data set consists of 1,050 graphs extracted from line drawings. The data set is evenly split into 15 classes. In a first step, 15 prototype line drawings representing capital letters, one for each class, are constructed in a manual fashion. An illustration of these prototype drawings is provided in Fig. 6.1. To obtain a noisy sample set of letter line drawings, distortions randomly removing, inserting, and displacing lines are repeatedly applied to these prototypes. Based on this procedure, 70 noisy patterns are generated from each clean prototype. Several examples of distorted line drawings representing a letter $A$ are shown in Fig. 6.2. For more examples, refer to Sec. A.1.

In a second step, distorted line drawings are transformed into attributed graphs. To this end, ending points of lines are represented by nodes endowed with a two-dimensional label specifying its position. The lines themselves are represented by unlabeled edges. The semi-artificial letter database hence consists of simple graphs with an intuitive interpretation of nodes and edges. The resulting 1,050 graphs are split into a training set and a validation set each of size 150 and a test set of size 750. The letter data set is mainly used to investigate how the running time and recogni-

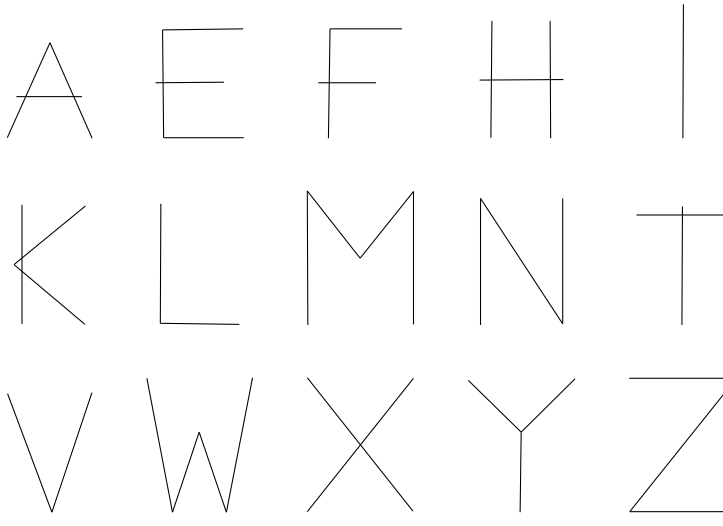


Fig. 6.1 Letter line drawing prototypes

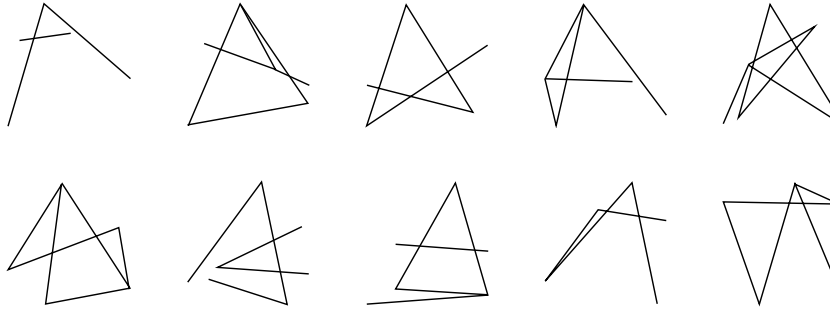


Fig. 6.2 Distorted instances of a letter *A*

tion accuracy of the various graph kernels depend on different parameter settings before applying the kernels to real-world data sets.

### 6.1.2 *Image Graphs*

The image database [Le Saux and Bunke (2005)] is used for the classification of images. First, images are segmented into regions of homogeneous colors using a mean shift algorithm. An example of an original image and the resulting color segmentation is provided in Fig. 6.3. The segmentation algorithm typically extracts less than 10 regions per image. The main idea of this segmentation process is that color seems to be a strong aspect of image recognition. In a second step, those regions that are relevant for the recognition of a particular class are selected, and all other regions are discarded. To this end, a mutual information criterion is optimized that estimates how strongly individual regions of images correspond to image classes. Segmented images can then be transformed into region adjacency graphs by representing regions by nodes labeled with attributes specifying the color histogram of the respective segment. Two nodes are connected by an unlabeled edge if the two corresponding regions are adjacent.

(a) (b)

Fig. 6.3 (a) Original image from the database and (b) segmented image

The image database [Le Saux and Bunke (2005)] contains 162 images from the five classes *snowy*, *countryside*, *people*, *city*, and *streets*. On the average, these graphs contain 2.8 nodes and 2.5 edges. For a few example images, see Fig. 6.4. Note that a larger number of sample images from this data set are provided in Sec. A.2. The database is finally split into a training set, validation set, and test set of equal size.

(a) (b) (c)

Fig. 6.4 Example images from the classes (a) countryside, (b) city, and (c) streets

### 6.1.3 *Diatom Graphs*

Diatoms are unicellular algae found in humid places where light provides the basis for photosynthesis. The identification of diatoms is useful for various applications such as environmental monitoring and forensic medicine [du Buf and Bayer (2002)]. The huge estimated number of more than 10,000 diatom classes makes the classification of diatoms very difficult. For a few examples of diatoms, see Fig. 6.5; the full data set is illustrated in Sec. A.3. Experts classify diatoms by analyzing their size, shape, and texture. Consequently, for the automatic identification of diatoms, a graph extraction procedure has been developed [Ambauen *et al.* (2003)] based on texture and geometry. First, diatom images are segmented into foreground and background. Using an edge detection algorithm, vertical stripes are extracted from diatom images and the direction of these stripes is computed. A region growing procedure finally segments diatoms into regions with homogeneous texture. Segmented diatoms can then be transformed into attributed graphs by representing regions by nodes and the adjacency of regions by edges. In this representation, nodes are labeled with an attribute specifying the relative size of regions and the dominant direction and density of their texture stripes, and edges are labeled with an attribute corresponding to the neighborhood degree of the two adjacent regions.

The diatom data set consists of 22 diatom classes and 5 graphs per class. The 110 diatom graphs are split into a training set and validation set each of size 37 and a test set of size 36. Note that the number of training patterns per class appears to be rather small for the difficult diatom identification problem.

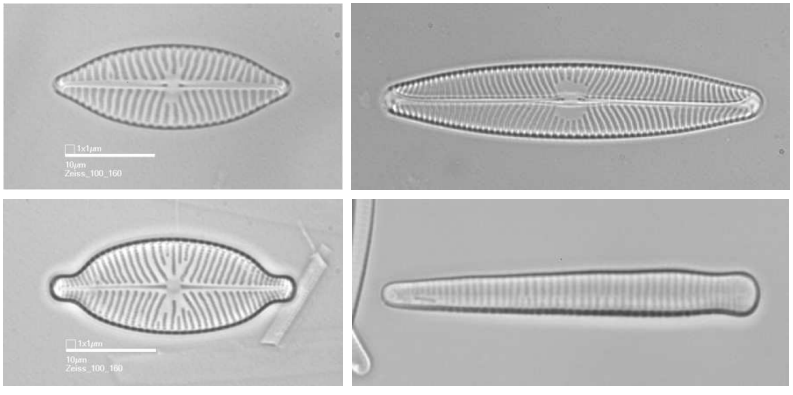


Fig. 6.5 Diatom image examples from four different classes

## 6.2 Fingerprint Graph Data Set

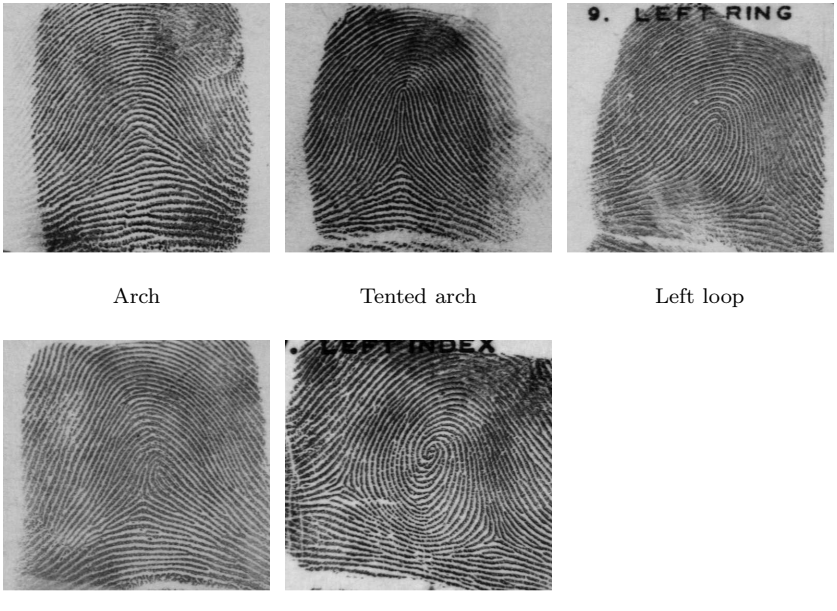
### 6.2.1 *Biometric Person Authentication*

Biometric person authentication refers to the task of automatically recognizing the identity of a person from his or her physiological or behavioral characteristics. For instance, there exist biometric systems based on fingerprint identification, signature verification, voice recognition, face image recognition, iris recognition, hand geometry recognition, and many more. In order to be suitable for personal authentication, biometric measurements are required to exhibit a number of properties. For instance, each person should actually possess the biometric, biometric measurements from different persons should be sufficiently different, the biometric should mostly be permanent over time, and it should be easy to capture the biometric of a person. In view of this, fingerprints seem to be a natural choice for biometric identification, since each person is believed to have unique fingerprints and fingerprints do not change over time. According to a recently published industry report [International Biometric Group (2006)], fingerprint recognition is expected to gain 45% of the biometrics market this year, followed by face recognition at 19%.

The idea of fingerprint recognition is to capture fingerprints by means of fingerprint sensors and process the resulting images to recognize the identity of the person corresponding to the fingerprint [Maltoni *et al.* (2003)]. If the identity of the person is known, the captured fingerprint only needs to be compared to a stored template, which is commonly referred to as *finger-*

*print verification*. For example, a biometric system might read the claimed identity of a person from an electronic badge and hence would only have to confirm the identity by analyzing the person's fingerprint. In forensic applications, on the other hand, the identity of the person belonging to a fingerprint is not known beforehand and must be established by searching large databases of fingerprints. This process is called *fingerprint identification*. The FBI database of fingerprints, for example, reportedly contains more than 200 million fingerprints. Given such huge databases, the need for efficient querying techniques is obvious.

### 6.2.2 *Fingerprint Classification*



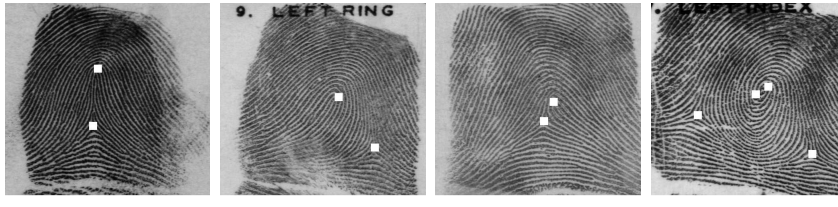

Arch Tented arch Left loop

Right loop Whorl

Fig. 6.6 Examples of the five main Galton-Henry fingerprint classes

One common approach to render fingerprint identification more efficient is to reduce the number of stored fingerprints to be compared. The basic idea is to define classes of fingerprints sharing global characteristics [Yager and Amin (2004)], so that for a given input fingerprint one only needs to

consider those fingerprints from the database that have the same characteristics. One of the first fingerprint classification systems was developed by Francis Galton more than 100 years ago and later refined by Edward Henry [Henry (1900)]. Many fingerprint classification systems used today by law enforcement agencies around the world are still based on the traditional Galton-Henry system. The Galton-Henry classification system is based on the observation that fingerprints can be grouped into classes of global ridge line patterns, where the five main classes are arch, tented arch, left loop, right loop, and whorl. For examples of these fingerprint classes, see Fig. 6.6. Further fingerprint examples are given in Sec. A.4.




Tented arch Left loop Right loop Whorl

Fig. 6.7 Fingerprints with singular delta points and singular core points

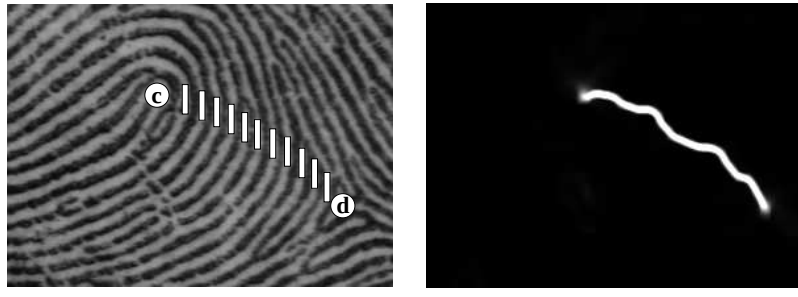
A common method for fingerprint classification is based on the detection of *singular points* in fingerprints. In the following, fingerprints will be regarded as orientation fields, where each pixel of a fingerprint image is assigned the direction of the local ridge line. In orientation fields, singular points are those points for which it is impossible to determine a unique orientation. For instance, in the whorl example in Fig. 6.6, a single direction of the ridge lines at the center of the whorl cannot be defined. Singular points that are located at the center of a whorl or at the center of a loop are called *core points*, while singular points at positions where ridges from three different directions meet are called *delta points*. It turns out that the number, type, and location of singular points in a fingerprint uniquely describe the fingerprint's class [Kawagoe and Tojo (1984); Karu and Jain (1996)]. Arch fingerprints, for instance, have no singular points, whereas whorl fingerprints contain two delta points and two core points. For an illustration of singular points in fingerprints, see the five fingerprint examples in Fig. 6.7. Hence, the problem of fingerprint classification based on the five Galton-Henry classes can be reduced to the problem of reliably detecting singular points in fingerprints. In order to cope with errors in-

troduced by singular point detection algorithms, a large number of pattern recognitions methods have been applied to fingerprint classification, including rule-based, syntactic, statistical, and neural network based approaches [Maltoni *et al.* (2003)]. Yet, since structure plays a crucial role in the analysis of fingerprints, graph matching seems to be particularly appropriate for fingerprint classification. Consequently, a number of graph matching classifiers have been proposed [Maio and Maltoni (1996); Marcialis *et al.* (2003); Lumini *et al.* (1999); Cappelli *et al.* (1999); Serrau *et al.* (2005)] based on segmenting fingerprints into regions of homogeneous orientation.

### 6.2.3 *Fingerprint Graphs*

The graph extraction approach proposed in this book is closely related to the traditional classification method based on the detection of singular points. The basic idea is to apply a filter to fingerprint images extracting regions that are relevant for classification. The rationale behind this procedure is that the region extraction procedure is expected to be substantially more robust against noise than the singular point detection. The basic idea is to detect where ridges in fingerprints have almost vertical orientation. A closer examination of the five fingerprint classes reveals that these regions are related to singular points in such a way that each core point is connected to a delta point by a region consisting only of vertical orientations. An illustration of a fingerprint, a pair of core and delta points, and the vertical ridge lines connecting the two singular points is provided in Fig. 6.8a. From the role of core and delta points in the global ridge flow of fingerprints, it can easily be shown that there must exist such a connection between core and delta point, because delta points in fact indicates where ridges belonging to a loop or whorl pattern meet regular ridges going from left to right.

A preliminary investigation shows that a modified directional variance measure seems to be best suited for the computation of vertical orientation regions. This modified directional variance is based on a vector averaging procedure and an orientation variance measure that is by definition sensitive to vertical orientations. The modified directional variance $\sigma^2(i,j)$ indicates for a pixel $(i,j)$ in a fingerprint image how likely it is that the local orientation at $(i,j)$ is vertical. In a first step, a fingerprint image under consideration is preprocessed by computing at each pixel the variance of the grayscale values in a local window [Maltoni *et al.* (2003)]. If the grayscale variance is low, the brightness around that pixel is largely con-




(a) (b)

Fig. 6.8 Left loop fingerprint image (a) with core point (marked by $c$), delta point (marked by $d$), and indicated vertical orientations, and (b) visualization of the modified directional variance

stant, and the corresponding part of the fingerprint can therefore be considered background. If the variance is high, one can assume that there must be fingerprint ridge and valley structures, or noise, in the local window. After segmenting the foreground of the fingerprint from the background, the ridge orientation at each foreground pixel is estimated using a Sobel filter [Maltoni *et al.* (2003)]. Additionally, an averaging procedure may be applied to the resulting orientations in order to obtain a smooth orientation field.

The result of Sobel filtering is a gradient vector $(\Delta_x(i,j), \Delta_y(i,j)) \in \mathbb{R} \times \mathbb{R}$ at each foreground pixel $(i,j)$ of the fingerprint image indicating the direction of the maximum grayscale intensity change. The length of the gradient vector $(\Delta_x(i,j)^2 + \Delta_y(i,j)^2)^{1/2}$ can be interpreted as the likelihood of the corresponding gradient direction. Obviously, if $(\Delta_x(i,j), \Delta_y(i,j))$ indicates the gradient direction of the grayscale surface, the perpendicular direction $(-\Delta_y(i,j), \Delta_x(i,j))$ can be regarded as the orientation of the ridge line at pixel $(i,j)$. For the computation of the directional variance measure, the local orientations $(-\Delta_y(i,j), \Delta_x(i,j)) \in \mathbb{R} \times \mathbb{R}$ are then additionally normalized to the half space $\mathbb{R}^+ \times \mathbb{R}$ by turning gradient vectors that point into the other half space into the opposite direction. For a set of orientation vectors pointing largely in horizontal direction, the resulting sum of vectors will point in horizontal direction as well; for a set of orientation vectors after normalization pointing more or less in vertical direction, the resulting sum of vectors after normalization may point anywhere depending on the number of vectors pointing upwards and downwards. For an illustration, see the orientation vectors in Fig. 6.9. The orientation vectors

in Fig. 6.9a are normalized to those shown in Fig. 6.9b, and the resulting vector sum is illustrated in Fig. 6.9c. In this example, the vector sum apparently represents the average orientation well. Conversely, the unnormalized and normalized vertical orientations in Fig. 6.9d and 6.9e result in the vector sum shown in Fig. 6.9f, which substantially differs from the true average orientation. The reason for this behavior is that the normalization procedure does not take the cyclic nature of orientation vectors into account. The resulting average vector will therefore point in the intuitively correct average direction only for non-vertical orientations.

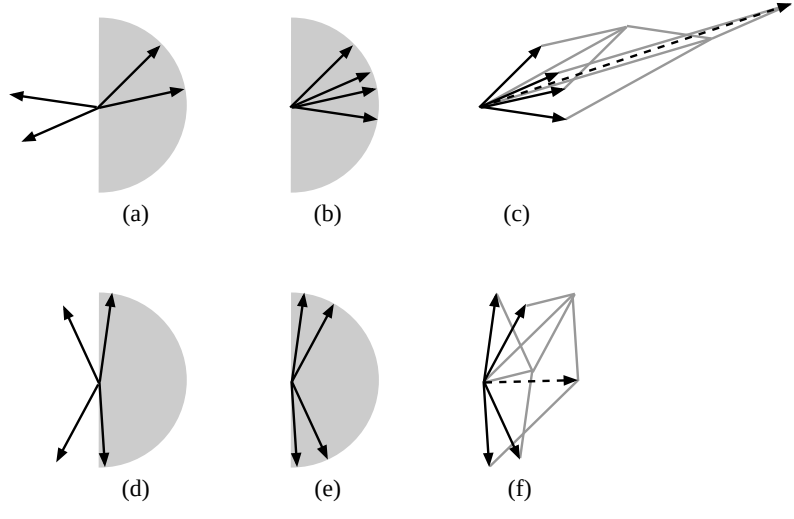


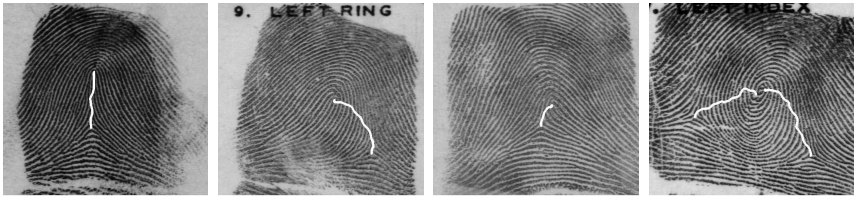

Fig. 6.9 (a), (d) Unnormalized orientation vectors, (b), (e) normalized orientation vectors, and (c), (f) computation of vector sum

In the following, assume that a fingerprint image under consideration has been preprocessed and the normalized orientation vectors have been computed for each foreground pixel. Denoting by $\alpha(p,q) \in [-\pi/2, +\pi/2]$ the angle corresponding to the normalized orientation vector at pixel $(p,q)$, and by $\bar{\alpha}(i,j) \in [-\pi/2, +\pi/2]$ the angle corresponding to the sum of normalized orientation vectors of a local window of size $n$ around pixel $(i,j)$, the modified directional variance is defined according to

$$\sigma^2(i,j) = \frac{1}{1-n} \sum_{p,q} \sin^2(\alpha(p,q) - \bar{\alpha}(i,j)) \ , \tag{6.1}$$

where the summation is performed over the window of size $n$ around $(i,j)$. This expression, which is closely related to the standard statistical vari-

ance, can be used to determine whether or not the ridge orientation vectors around pixel $(i, j)$ point in vertical direction. Because of the normalization procedure, this modified directional variance measure is sensitive to vertical orientations, since for vertical orientations the computed average direction $\bar{\alpha}(i, j)$ will differ significantly from the underlying set of orientation vectors $\alpha(p, q)$, which will result in a higher variance $\sigma^2(i, j)$. The modified directional variance turns out to be more accurate than other intuitive measures of verticality. An illustration of the modified directional variance is given in Fig. 6.8b. Clearly, the detection of vertically oriented parts of ridge lines between core point and delta point appears to be quite reliable.

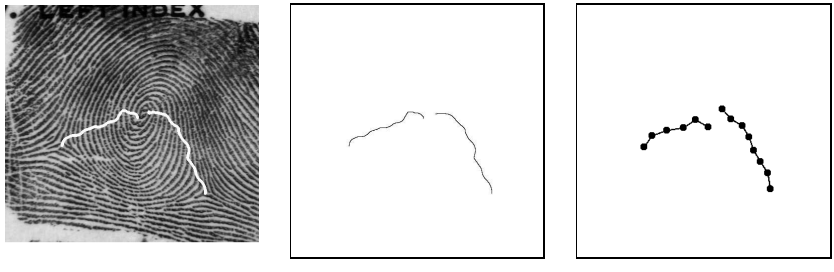



Tented arch Left loop Right loop Whorl

Fig. 6.10 Fingerprints with singular regions

The advantage of using regions of vertical orientations instead of singular points is that these regions are less prone to noise and extraction errors. For instance, if a singular point happens to lie outside a captured image, the singular point itself cannot be detected, but vertically oriented ridge lines can easily be extracted. Or, if the area around a singular point is covered by noise rendering the singular point detection impossible, the corresponding region of vertical orientations can nevertheless be used for classification. For an illustration of vertical regions detected in fingerprints, see Fig. 6.10. Criteria that are relevant for fingerprint classification include the number, position, and direction of vertical orientation regions. After computing the modified directional variance at each foreground pixel of a fingerprint image, the resulting image of extracted regions is binarized and undergoes a noise removal and thinning procedure [Zhou *et al.* (1995)], resulting in a skeletonized representation of the extracted regions. Ending points and bifurcation points of the skeletonized regions are then represented by nodes, and additional nodes are inserted along the skeleton (between ending points and bifurcation points) at regular intervals. Each node is then assigned an attribute giving the position of the corresponding

pixel in the image. Finally, undirected edges are inserted to link nodes that are directly connected through a ridge in the skeleton. An angle attribute is assigned to each edge denoting the orientation of the edge with respect to the horizontal direction. An example of extracted vertical orientation regions, the corresponding skeleton, and the resulting graph are shown in Fig. 6.11.



(a) (b) (c)

Fig. 6.11 (a) Vertical orientation region in a fingerprint image, (b) the extracted skeleton, and (c) the corresponding graph

The graph extraction procedure described above is applied to the NIST-4 reference database of fingerprints [Watson and Wilson (1992)], consisting of 4,000 fingerprint images labeled according to the five main Henry-Galton classes *arch*, *tented arch*, *left loop*, *right loop*, and *whorl*. On the average, the extracted graphs contain 6.1 nodes and 10.3 edges. For this graph database, 61 reference patterns have been constructed in a manual fashion. For instance, since fingerprints from the *arch* class are characterized by the absence of vertically oriented ridges, the empty graph is chosen as prototype graph for the *arch* class. Fingerprints from the *left loop* class are characterized by extracted regions located in the right half of the fingerprint, running from the delta point in an anticlockwise direction upwards to the core point of the loop. Prototypes for the *left loop* class are therefore constructed to reflect this characteristic property. Similarly, the *whorl* class is represented by graph prototypes consisting of one large or two regions running from left to right. Note that the fingerprints and extracted regions shown in Fig. 6.10 constitute typical representatives of their classes. For an illustration of some example class prototypes, see Fig. 6.12.

The validation of the constructed prototype graphs is carried out on the first 250 graphs from the database. Fingerprints can then be classified

Table 6.1 Fingerprint classification rate on the NIST-4 database of fingerprints

| Classifier | Accuracy |
|---|---|
| RNN [Marcialis *et al.* (2003)] | 76.75 |
| MASKS [Cappelli *et al.* (1999)] | 71.45 |
| GM [Serrau *et al.* (2005)] | 65.15 |
| MLP [Jain *et al.* (1999b)] | 86.01 |
| Proposed single | 80.25 |
| Proposed combined | 88.80 |

by computing the edit distance of the corresponding fingerprint graph to each prototype graph and assigning the unknown fingerprint to the class of the best matching prototype (nearest-neighbor classification). On the last 2,000 fingerprints of the NIST-4 database, this method results in a classification accuracy of 80.25%. If we use the method for the data-level fusion of graphs presented in Sec. 3.7, the accuracy can further be increased to 88.8% (cf. Table 3.6). A comparison of these results to some other fingerprint classifiers is given in Table 6.1. In this table, RNN, MASKS, GM, and MLP refer to graph matching methods based on recursive neural networks [Marcialis *et al.* (2003)], dynamic masks [Cappelli *et al.* (1999)], graph edit distance [Serrau *et al.* (2005)], and a non-structural multi-layer neural network approach [Jain *et al.* (1999b)], respectively. Note that the edit distance method proposed in this book clearly outperforms the other graph matching systems. According to these results, the proposed method for the extraction of vertically oriented ridges seems to be suitable for graph classification.

The basic assumption of statistical learning theory is that patterns from the training set and test set are distributed according to the same hidden joint probability density of patterns and class labels, as discussed in Sec. 4.1. Clearly, manually constructing a set of training prototype

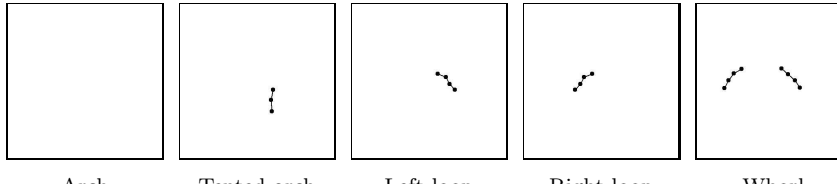



Fig. 6.12 Example class prototypes

graphs, in such a way that class characteristics relevant for classification are reflected, severely violates the condition that the set of training patterns should be randomly sampled from the underlying pattern distribution. Certainly, the pattern distribution arising from the manual construction process will be different from the distribution of the full fingerprint graph population. For an example, the constructed prototype graphs tend to be smaller than graphs extracted from fingerprints. Hence, if an SVM is trained on the prototype graphs, the resulting class-separating hyperplanes will only be appropriate for graphs from the training set, but not for graphs from an independent test set. For these reasons, the manually constructed training set of prototypes will not be used in the remainder of this book. Instead, the fingerprint database is split into a training set, validation set, and test set, such that all graphs involved in the training, validation, and testing process stem from the same population. In addition, all fingerprint images from the NIST-4 database that are labeled with two classes, instead of a single one, are removed, so that for classification each graph is uniquely labeled with a single class. Such class label ambiguities occur when fingerprints exhibit characteristics from more than one class. Replacing the manually constructed set of training graphs by a randomly selected set of graphs from the database renders classification more difficult; removing ambiguously labeled fingerprints from the database renders classification easier. Finally, a database of 3,300 fingerprint graphs is obtained and split into a training set and validation set each of size 150 and a test set of size 3,000. This data set of fingerprint graphs is then applicable to nearest-neighbor classification as well as SVM classification without further modifications.

## 6.3 Molecule Graph Data Set

For over fifty years, the US National Cancer Institute (NCI) has been conducting experiments aimed at identifying materials exhibiting a certain anti-cancer activity. Since 1999, the NCI has also been carrying out AIDS antiviral screen tests to discover chemical compounds that might be capable of inhibiting the HIV virus [Weislow *et al.* (1989)]. The screen tests were set up to measure how strongly the compounds under consideration were able to protect human cells from infection by the HIV-1 virus. Compounds that were able to provide a protection from an HIV infection in at least half the cases were tested in a second experiment. Those chemi-

cal compounds that reproducibly provided a perfect protection from HIV were labeled *confirmed active* (CA), while compounds that were only able to protect cells from infection in at least 50% of the cases in the second test were labeled *moderately active* (MA). All other compounds were labeled *confirmed inactive* (CI). From the 42,438 screened compounds, 406 were found to belong to the CA category, 1056 to the MA category, and 40,976 to the CI category. The chemical structure of the tested compounds and their HIV antiviral screen test results have been made available to the public in the NCI Open Database [Milne and Miller (1986); Voigt *et al.* (2001)].

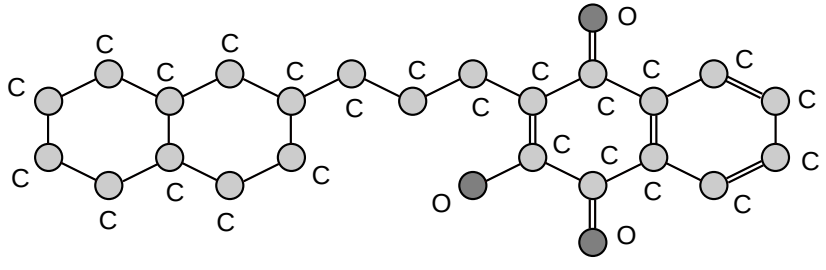



Fig. 6.13 Example molecule

Chemical compounds constitute a good example of patterns for which a structural pattern description is obviously better suited than a statistical one. An example molecule consisting of carbon atoms and oxygen atoms is illustrated in Fig. 6.13. When dealing with molecules, we are usually interested in the function of the corresponding chemical compound in terms of its reaction to, or effect on, other materials. Since functional properties of chemical compounds are well-known to be tightly coupled with the presence, or absence, of certain substructures in its molecular structure, the representation of molecules by graphs provides for a suitable data description technique. The molecule data set used in this book consists of 1,600 graphs representing chemical compounds that are inactive (CI) and 400 graphs representing chemical compounds that are active against the HIV virus (CA). The data set is split into a training set and validation set of size 250 each, and a test set of size 1,500.

The representation of molecules by graphs is straightforward by turning atoms into nodes and bonds into edges. Nodes are labeled with their chemical symbols and charge, and edges are labeled with their valence. In order to render the matching of molecules easier and decrease the average

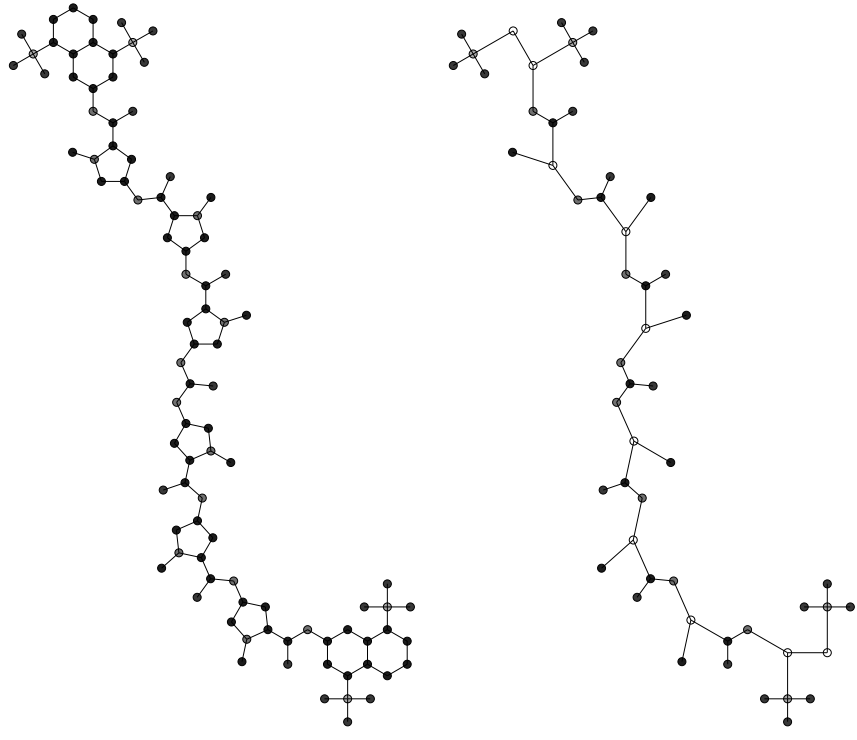



Fig. 6.14 Elimination of cycles, a) original molecule and b) the same molecule after cycle elimination

size of the molecule graphs, we additionally perform a cycle elimination procedure. The idea is to identify cycles of nodes and replace these cycles by single nodes. Edges connected to one of the nodes of a cycle will be replaced by an edge linked to the single node representing the full cycle. The resulting single node is labeled with the sequence of chemical symbols and the sum of charges of involved nodes. For an illustration of a molecule with a rather large number of cycles and the molecule resulting from cycle elimination, refer to Fig. 6.14. The graphs in the molecule data set contain 9.5 nodes and 10.1 edges on the average; the largest graph consists of 85 nodes, and the largest number of edges amounts to 328. An illustration of sample graphs from both classes is provided in Sec. A.5.

Table 6.2 Summary of the characteristics of the letter data set, the image data set, and the diatom data set

| Letter graph data set | |
|---|---|
| Patterns | Line drawings representing capital letters |
| Classes | 15 (A, E, F, H, I, K, L, M, N, T, V, W, X, Y, Z) |
| Size of training set | 150 |
| Size of validation set | 150 |
| Size of test set | 750 |
| Average per graph | 4.6 nodes, 4.5 edges |

| Image graph data set | [Le Saux and Bunke (2005)] |
|---|---|
| Patterns | Images taken with a digital camera |
| Classes | 5 (snowy, countryside, city, people, streets) |
| Size of training set | 54 |
| Size of validation set | 54 |
| Size of test set | 54 |
| Average per graph | 2.8 nodes, 2.5 edges |

| Diatom image graph data set | [Ambauen *et al.* (2003)] |
|---|---|
| Patterns | Microscopic diatom images |
| Classes | 22 (Caloneis amphisbaena, Cocconeis neodiminuta,...) |
| Size of training set | 37 |
| Size of validation set | 37 |
| Size of test set | 36 |
| Average per graph | 4.4 nodes, 6.2 edges |

## 6.4 Experimental Setup

The graph data sets described above are split into a labeled training set, labeled validation set, and labeled test set. A summary of the main data set characteristics is provided in Tables 6.2 and 6.3. The training set is used to construct and learn the classifiers. If an SVM is used for classification, the support vectors and class-separating hyperplane are solely determined based on the training set, that is, the set of support vectors is a subset of the training set. In the case of kernel PCA and kernel FDA, the training set is used to compute the transformation matrix mapping patterns from the high-dimensional kernel feature space into a low-dimensional subspace. If nearest-neighbor classifiers are used, the set of prototype patterns is in fact the training set. Meta-parameters that cannot be determined during

Table 6.3 Summary of the characteristics of the fingerprint image data set and the molecule data set

| NIST-4 Fingerprint graph data set | [Watson and Wilson (1992)] |
|---|---|
| Patterns | Fingerprint images from the NIST-4 database |
| Classes | 5 (arch, tented arch, left loop, right loop, whorl) |
| Size of training set | 150 |
| Size of validation set | 150 |
| Size of test set | 3,000 |
| Average per graph | 6.2 nodes, 10.3 edges |

| NCI Molecule data set | [Milne and Miller (1986)] |
|---|---|
| Patterns | Chemical compounds from the NCI database |
| Classes | 2 (confirmed inactive, confirmed active) |
| Size of training set | 250 |
| Size of validation set | 250 |
| Size of test set | 1,500 |
| Average per graph | 9.5 nodes, 10.1 edges |

learning are optimized by means of the validation set. The idea is to train classifiers for various meta-parameter configurations and apply the resulting trained classifiers to the patterns from the validation set in order to obtain a recognition performance measure. Finally, the meta-parameter setting that performs best on the validation set is kept. Examples of meta-parameters include cost function parameters of graph edit distance, parameters of graph kernel functions, the number of neighbors in kNN classification, and the regularization parameter $C$ in SVM classification. The classifier resulting from the optimization of meta-parameters on the validation set is finally applied to the independent test set. The test set recognition accuracy can be seen as an estimate of the performance on test patterns randomly sampled from the underlying pattern distribution.

The main objective in this book is the comparison of traditional edit distance classifiers with classifiers based on graph kernels. In the following sections, these classifiers will be compared in terms of running time and recognition accuracy on the independent test set.

## 6.5 Evaluation of Graph Edit Distance

This section gives an experimental evaluation of traditional edit distance based nearest-neighbor classification. For an introduction to nearest-

neighbor classification, see Sec. 3.6 and Sec. 4.4. In Sec. 6.6, the proposed graph kernels will be compared to the reference results obtained in this section.

### 6.5.1 *Letter Graphs*

The first task to be considered is the classification of graphs from the letter data set. The edit distance of letter graphs is computed according to the Euclidean cost function defined in Eq. (3.2). Since the edges of letter graphs are unlabeled, edge substitutions can be carried out for free in this cost model, and the edge substitution parameter $\alpha_{edge}$ consequently has no influence on the resulting edit distance. The remaining parameters to be validated are the node insertion and deletion penalty $\gamma_{node}$, the edge insertion and deletion penalty $\gamma_{edge}$, and the node substitution factor $\alpha_{node}$. In the following, we proceed by computing the approximate edit distance (Sec. 3.4) of graphs from the validation set to graphs from the training set and classify the graphs from the validation set according to a $k$-nearest-neighbor (kNN) classifier. It is clear that the edit cost parameters will strongly affect the edit distance measure, and therefore the accuracy of the resulting classifier as well.

If we set the node substitution factor to $\alpha_{node} = 1$ and compute for each configuration of insertion and deletion penalty costs $\gamma_{node} = 1, 2, \ldots, 20$ and $\gamma_{edge} = 1, 2, \ldots, 20$ the classification accuracy on the validation set, we obtain the illustration shown in Fig. 6.15a. The accuracy (the vertical axis in this figure) is defined by the fraction of correctly classified graphs out of the 150 graphs from the validation set. Focusing on the interesting part of the plot close to the origin, we find that values of $\gamma_{node} = 2$ and $\gamma_{edge} = 2$ seem to be best suited for classifying the letter data set. The corresponding detailed illustration of the classification accuracy in Fig. 6.15b shows that the optimal parameters are close to, but greater than, zero. Next, setting $\gamma_{edge} = 1$ and computing the classifier accuracy for $\gamma_{node} = 0, 0.2, 0.4, \ldots, 3$ and node substitution factor $\alpha_{node} = 0.2, 0.4, 0.6, \ldots, 3$ results in the illustration shown in Fig. 6.16. Again, an optimal area of the recognition surface can clearly be determined. These results suggest that the parameterization of the Euclidean edit cost function is smoothly related to the recognition accuracy on the validation set. To obtain a configuration of parameters $(\alpha_{node}, \gamma_{node}, \gamma_{edge}) \in \mathbb{R}^+ \times \mathbb{R}^+ \times \mathbb{R}^+$ that is as well-performing on the validation set as possible, a steepest-ascent search in the three-dimensional parameter space with respect to the recognition rate on the validation set

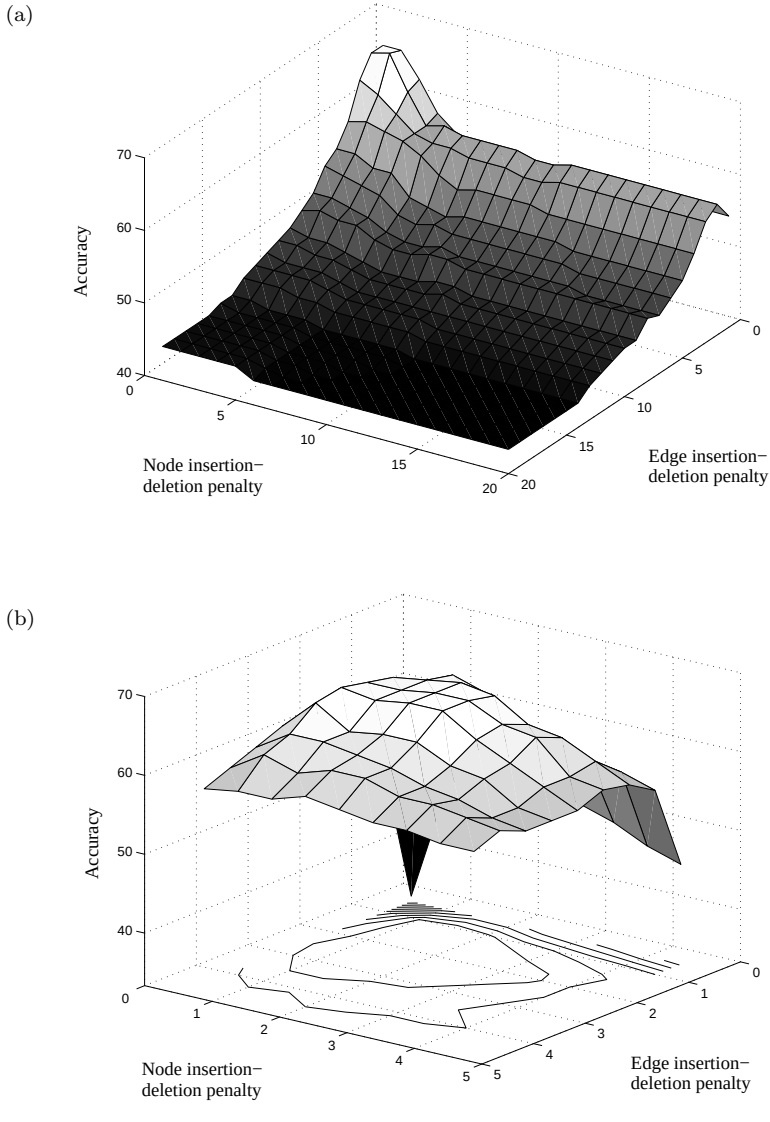



Fig. 6.15 Letter data set: Accuracy with respect to node insertion and deletion penalty $\gamma_{node}$ and edge insertion and deletion penalty $\gamma_{edge}$

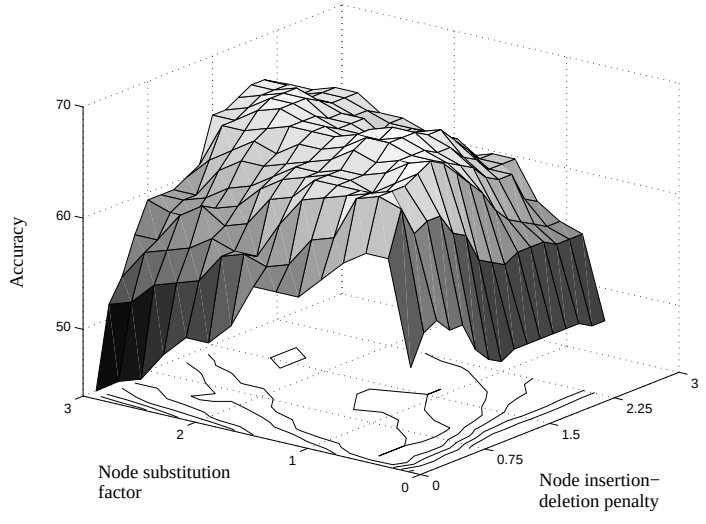



Fig. 6.16 Letter data set: Accuracy with respect to node insertion and deletion penalty $\gamma_{node}$ and node substitution factor $\alpha_{node}$

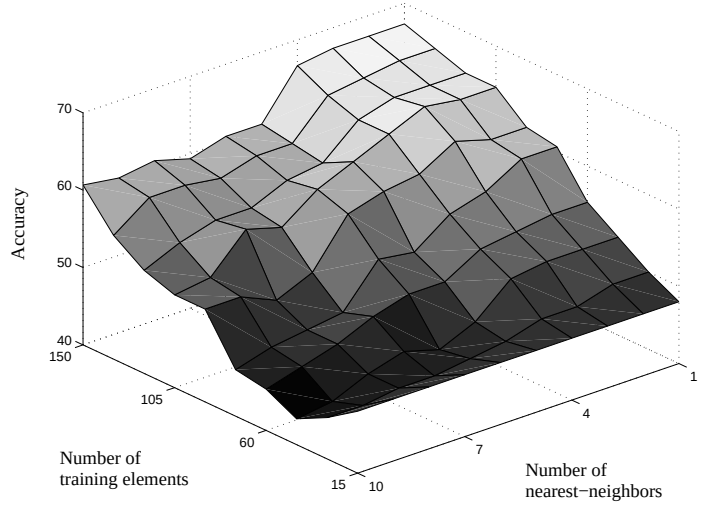



Fig. 6.17 Letter data set: Accuracy with respect to number of training elements $s$ and number of nearest-neighbors $k$

is employed. The empirically optimal parameters $\alpha_{node} = 1$, $\gamma_{node} = 0.3$, and $\gamma_{edge} = 1.1$ result in a classification accuracy of 67.3% on the validation set using a 3-nearest-neighbor classifier. On the independent test set, a classification accuracy of 69.3% is obtained.

The training set of the letter database consists of 15 classes and 10 patterns per class. In the following it is investigated how smaller training sets affect the classification accuracy. To this end, using the empirically best edit cost parameters obtained above, the size $s$ of the training set and the number of nearest-neighbors $k$ are varied. The resulting classification performance for $k = 1, 2, \ldots, 10$ and $s = 15, 30, \ldots, 150$ is shown in Fig. 6.17. As expected, the classification accuracy increases monotonically for an increasing number of training elements $s$. Hence, reducing the training set seriously affects the ability of the nearest-neighbor classifier to estimate the boundaries of the underlying classes. The optimal performance is obtained for large training sets and few nearest-neighbors, which corresponds well to the value of $k = 3$ determined in the parameter optimization process.

### 6.5.2 *Image Graphs*

The graphs extracted from images contain node labels representing color histograms and edge labels representing the common boundary of two regions. The resulting graphs are small enough for the exact edit distance algorithm (Sec. 3.3). Since the Euclidean distance of color histograms reflects the dissimilarity of the respective colors, the Euclidean edit cost model from Eq. (3.2) appears to be well suited for the image data set. In the original system [Le Saux and Bunke (2005)], the authors suggest to assign lower costs to the deletion of nodes in large graphs than those in small graphs, by defining $c(u \rightarrow \varepsilon) = 1/n$ for deleting node $u$ in a graph with $n$ nodes. The insertion and deletion costs of edges are defined equivalently. In experiments it turns out that neither the Euclidean cost model nor the cost model from [Le Saux and Bunke (2005)] performs significantly better, and hence the cost model proposed in the original paper is used in the following experiments. The node substitution factor $\alpha_{node}$ and the edge substitution factor $\alpha_{edge}$ are the only parameters to be optimized.

The recognition performance on the validation set for parameters $\alpha_{node} = 0, 0.1, 0.2, \ldots, 3$ and $\alpha_{edge} = 0, 0.1, 0.2, \ldots, 3$ is illustrated in Fig. 6.18a. Obviously, the best performance is obtained for very small values of the edge substitution parameter. To further investigate the optimal weight of edge substitutions, the accuracy for $\alpha_{node} = 0, 0.1, 0.2, \ldots, 3$

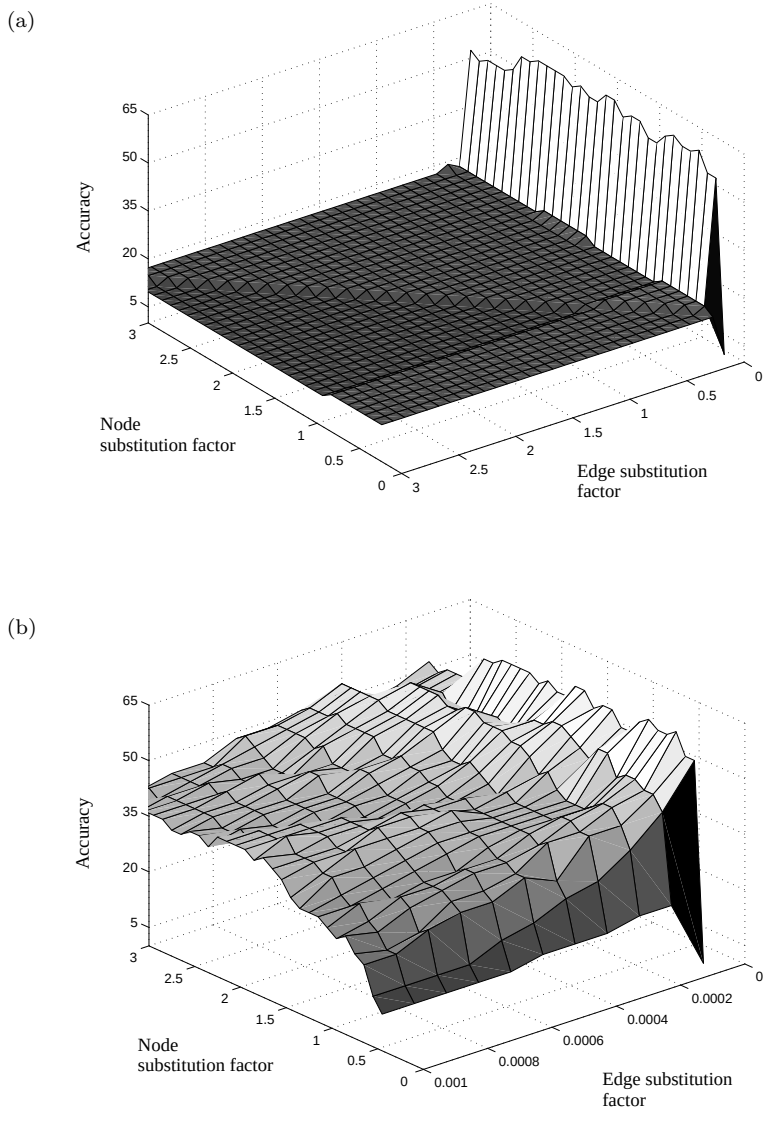



Fig. 6.18 Image data set: Accuracy with respect to node substitution factor $\alpha_{node}$ and edge substitution factor $\alpha_{edge}$

and the range $\alpha_{edge} = 0, 0.0001, 0.0002, \ldots, 0.001$ close to zero is evaluated in a second experiment. The resulting performance is shown in Fig. 6.18b. These results clearly demonstrate that the classification accuracy is maximum if edge substitutions can be carried out for free. Hence, the information present in edge labels, that is, the size of the common boundary of two regions, is not useful for classification and can safely be ignored. This observation shows that other edge labels should be defined for this graph representation that are more relevant for the considered classification task. In view of this, graph matching paradigms that put the main emphasis on an optimal node-to-node correspondence might perform well on the image data set. Another idea might be to ignore all edges and address the image classification problem in the context of point set matching [Caetano *et al.* (2006)].

Using a steepest-ascent search procedure, the empirically optimal parameters $\alpha_{node} = 0.3$, $\alpha_{edge} = 0$, and number of neighbors $k = 12$ are determined on the validation set. The resulting recognition accuracy is 64.8% on the validation set and 48.1% on the test set. The significantly higher recognition rate on the validation set indicates that a serious amount of overfitting occurs in the optimization process of the parameters. That is, the optimal parameters are too strongly adapted to the validation set and are not suitable for the independent test set. This observation indicates that a nearest-neighbor classifier, requiring that the training set covers a large part of the pattern space, may not be suitable for this particular classification problem.

### 6.5.3 *Diatom Graphs*

The graphs extracted from segmented diatom images are region adjacency graphs consisting of nodes labeled with information about the texture of regions and edges labeled with the neighborhood degree of adjacent regions. Again, the Euclidean edit cost model from Eq. (3.2) and the approximate edit distance algorithm described in Sec. 3.4 are used in the experiments. Note that this approach differs slightly from the edit cost model that has originally been employed [Ambauen *et al.* (2003)], where the node insertion and deletion cost were set proportional to the pixel size of the respective region instead of constant values.

First, the node and edge insertion and deletion penalty costs are set to $\gamma_{node} = \gamma_{edge} = 0.5$, and the recognition accuracy is computed for node and edge substitution factors $\alpha_{node} = 0, 10, \ldots, 100$ and $\alpha_{edge} = 0, 10, \ldots, 100$.

The resulting illustration is provided in Fig. 6.19. A thorough examination reveals that the best results are obtained for medium node substitution factors and large edge substitution factors. In a second experiment, the accuracy is computed for fixed parameters $\alpha_{node} = 20$ and $\alpha_{edge} = 75$ and for all combinations of the node insertion and deletion penalty $\gamma_{node} = 0, 0.1, \ldots, 1$ and edge insertion and deletion penalty $\gamma_{edge} = 0, 0.1, \ldots, 1$. In the resulting illustration in Fig. 6.20, it can clearly be observed that an optimal recognition rate is obtained for rather small values of $\gamma_{edge}$.

On the validation set, the steepest-ascent search procedure determines optimal parameters $\alpha_{node} = 25$, $\alpha_{edge} = 95$, $\gamma_{node} = 0.5$, $\gamma_{edge} = 0.2$, and $k = 1$. The recognition rate obtained on the validation set amounts to 86.5% and the one on the test set to 66.7%. Note that the training set consists of 37 training patterns from 22 classes only. This means that there are only 2 (in 15 cases) or 1 (in 7 cases) prototypes per class in the training set, which is obviously a very low number of prototypes, especially in the context of empirical risk minimization. Also, the significantly higher recognition rate on the validation set than on the test set indicates that the nearest-neighbor classifier strongly overfits the validation data. If a pattern in the training set happens to be an outlier, or even if a pattern is located at the boundary rather than at the center of its class (with respect to the underlying distribution), a nearest-neighbor classifier will never be able to estimate the true distribution of this pattern's class.

### 6.5.4 *Fingerprint Graphs*

The fingerprint data set contains graphs describing class-characteristic regions in fingerprints. The fingerprint graphs consist of nodes labeled with a position attribute and edges labeled with an angle attribute. Because of the cyclic nature of angular measurements, the Euclidean distance is not appropriate for measuring the dissimilarity of two angles. In fact, the angle attributes attached to edges in fingerprint graphs specify an undirected orientation, that is, the angle value $\nu(e)$ of each edge $e$ is in the interval $\nu(e) \in (-\pi/2, +\pi/2]$. If the dissimilarity of two angle attributes $\nu(e) = 1.5$ and $\nu(e') = -1.5$ is measured by means of the Euclidean distance, $|\nu(e) - \nu(e')| = 3$, the two attributes are found to be quite dissimilar. However, the angle $\nu(e) = 1.5$ actually specifies a vector pointing almost vertically upwards and $\nu(e') = -1.5$ a vector pointing almost vertically downwards. Hence, the two undirected orientations are in fact very similar, since the direction, upwards or downwards, is

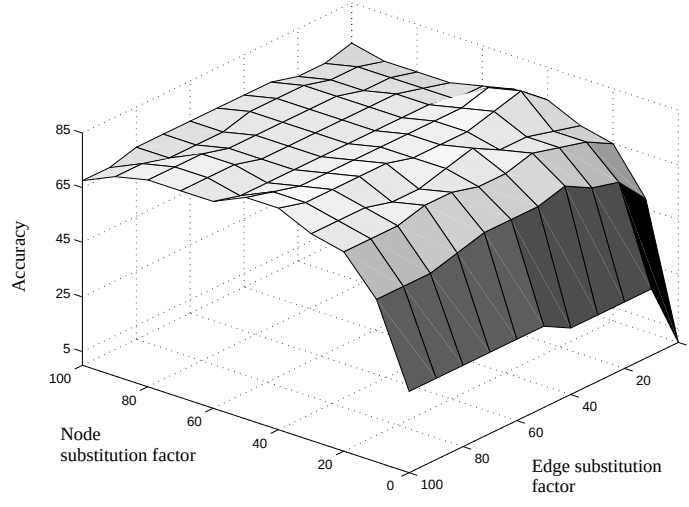



Fig. 6.19 Diatom data set: Accuracy with respect to node substitution factor $\alpha_{node}$ and edge substitution factor $\alpha_{edge}$

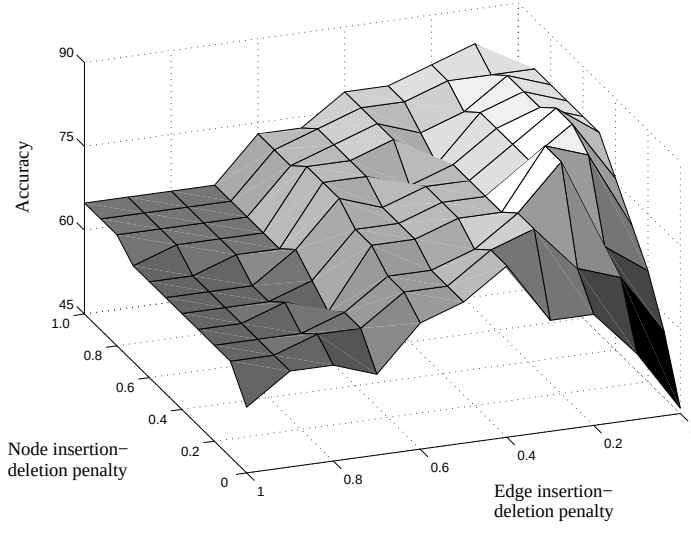



Fig. 6.20 Diatom data set: Accuracy with respect to node insertion and deletion penalty $\gamma_{node}$ and edge insertion and deletion penalty $\gamma_{edge}$

not taken into account in ridge orientations at all. For these reasons, a modified dissimilarity measure for undirected orientations is used, defined by $d : (-\pi/2, +\pi/2] \times (-\pi/2, +\pi/2] \rightarrow [0, \pi/2]$, $d(\nu(e), \nu(e')) = \min(\pi - |\nu(e) - \nu(e')|, |\nu(e) - \nu(e')|)$. For the example above, a dissimilarity of $d(1.5, -1.5) = \min(0.14, 3) = 0.14$ is obtained, which indicates that the undirected orientations are quite similar. Correspondingly, edge substitution costs are defined by replacing the Euclidean distance in Eq. (3.2) by $d$ defined above. Node operation costs and edge insertion and deletion costs are defined according to Eq. (3.2). In the experiments, the approximate edit distance algorithm is applied (Sec. 3.4).

A first experimental evaluation shows that the highest classification accuracy is achieved for large edge insertion and deletion penalty costs $\gamma_{edge}$. This means that if the edit distance is forced to apply mostly edge substitutions rather than insertions and deletions, the classification of fingerprint graphs works best. Conversely, the accuracy is maximum for low values of node insertion and deletion penalty costs $\gamma_{node}$. In Fig. 6.21, the fingerprint classification accuracy is shown for parameters $\alpha_{node} = 0.1, 0.2, \ldots, 1$, $\alpha_{edge} = 1$, number of nearest neighbors $k = 1, 2, \ldots, 20$, $\gamma_{node} = 0$, and $\gamma_{edge} = 80$. Obviously, a small number of nearest-neighbors is sufficient for an accurate classification. The accuracy for parameters $\alpha_{node} = 0.5, 0.55, 0.6, \ldots, 1$, $\alpha_{edge} = 1$, $\gamma_{node} = 0$, $\gamma_{edge} = 500, 550, 600, \ldots, 1000$, and $k = 3$ is illustrated in Fig. 6.22. For each choice of the edge insertion and deletion penalty $\gamma_{edge}$ there seems to be a specific value of the node substitution factor $\alpha_{node}$ resulting in the highest recognition accuracy. If the edge insertion and deletion cost is increased, the node substitution costs must be increased as well to make sure that the classification rate does not decrease.

The highest classification accuracy in Fig. 6.21 (for $\alpha_{node} = 0.1$ and $k = 5$) coincides with the best performance on the validation set returned by a steepest-ascent search procedure. Note that the maximum recognition rate is obtained for many parameter configurations, which means that the nearest-neighbor classifier appears to be quite insensitive against parameter changes on this data set. The highest recognition rate amounts to 84.0% on the validation set and 76.2% on the independent test set.

It should be noted that this classification rate is not directly comparable to the classification rates reported in Sec. 6.2: In Sec. 6.2 the performance is evaluated with a nearest-neighbor classifier based on a set of carefully selected class prototypes, whereas in the current section the set of training graphs is randomly selected from the full set of fingerprint graphs. Further-

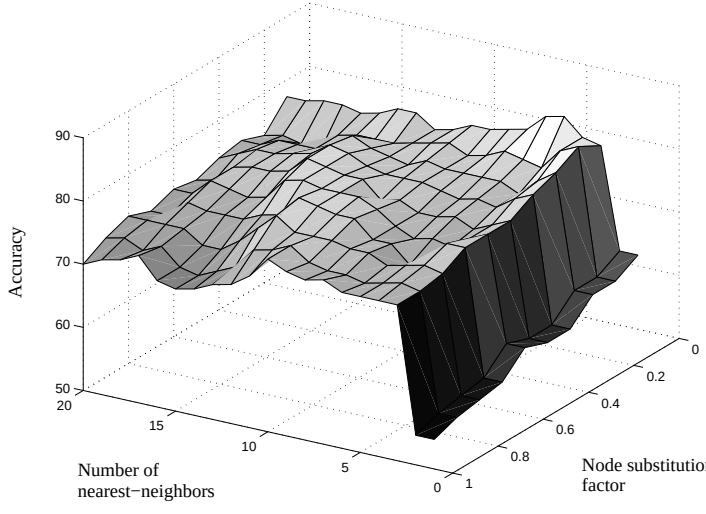



Fig. 6.21 Fingerprint data set: Accuracy with respect to number of nearest neighbors and node substitution factor $\alpha_{node}$

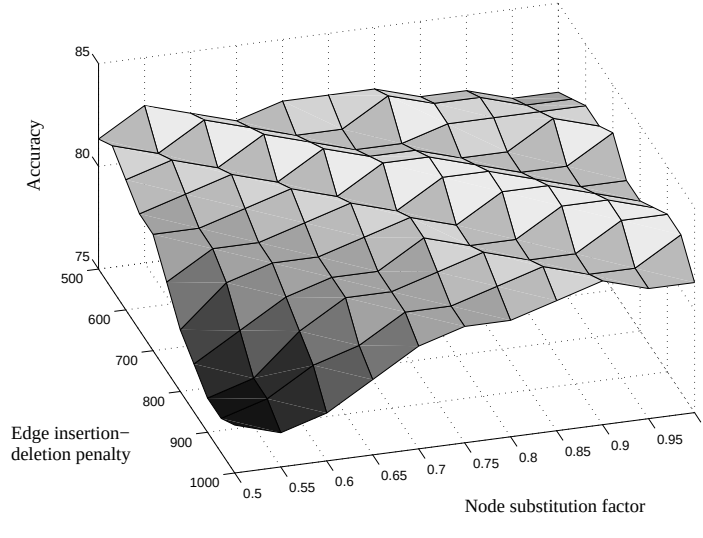



Fig. 6.22 Fingerprint data set: Accuracy with respect to edge insertion and deletion penalty $\gamma_{edge}$ and node substitution factor $\alpha_{node}$

more, ambiguously labeled fingerprint graphs, which are usually considered difficult in the context of fingerprint classification, have been removed from the database for the experiments described in this section, whereas the results in Sec. 6.2 have been obtained on the full data set.

### 6.5.5 *Molecule Graphs*

The molecule graph data set consists of graphs representing chemical compounds. In these molecule graphs, nodes representing a single atom are labeled with the respective chemical symbol and atomic charge, and nodes representing an atomic ring of several elements are labeled with the chemical symbols and sum of atomic charge of all involved atoms. Edges are labeled with a valence attribute. In the experiments, we use a variant of the Euclidean edit cost model from Eq. (3.2). The costs of node and edge insertions and deletions are set to constant values, and edge substitution costs are set proportional to the Euclidean distance of the corresponding valence labels. For node substitutions, we measure the dissimilarity of two chemical symbols with a Dirac function (returning 0 if the two symbols are equal, and 1 otherwise) and the dissimilarity of the two charge attributes with the Euclidean distance. The resulting dissimilarity is then weighted with a node substitution factor, analogously to the Euclidean cost model.

In a first experiment on the molecule data set, the classification accuracy is computed for node substitution factors $\alpha_{node} = 0.5, 1, 1.5, \ldots, 5$, edge substitution factors $\alpha_{edge} = 1, 2, 3, \ldots, 10$, node insertion and deletion penalty $\gamma_{node} = 3$, edge insertion and deletion penalty $\gamma_{edge} = 2$, and $k = 5$ nearest-neighbors. The resulting illustration is provided in Fig. 6.23. Obviously, the classification rate is maximal for rather small values of node and edge substitution factors. Assigning higher costs to node and edge substitutions, that is, increasingly replacing substitutions by deletions followed by insertions, leads to an inferior matching performance. In other words, the edit distance process relies on the substitution operation to match nodes and edges considered similar under the edit cost model at hand.

In a second experiment, we compute the classification rate for substitution factors $\alpha_{node} = 0.5$ and $\alpha_{edge} = 3$ and for insertion and deletion penalty costs $\gamma_{node}, \gamma_{edge} = 1, 2, 3, \ldots, 10$. The corresponding illustration is shown in Fig. 6.24. In consistence with the results of the first experiment, we find it is advantageous to enforce the application of substitutions by assigning large penalty costs to insertions and deletions for edges and for nodes as well.

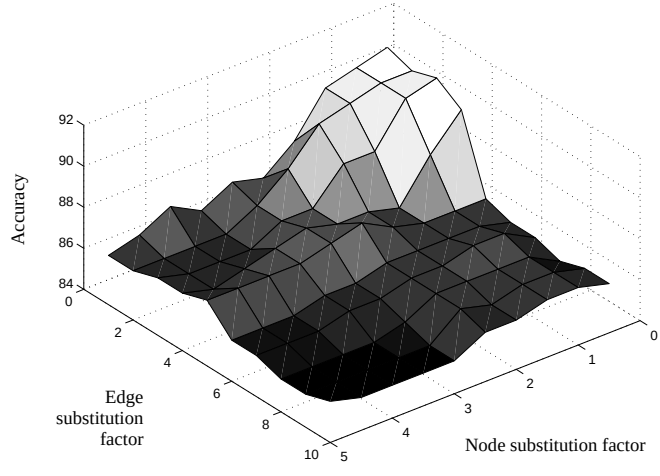



Fig. 6.23 Molecule data set: Accuracy with respect to node substitution factor $\alpha_{node}$ and edge substitution factor $\alpha_{edge}$

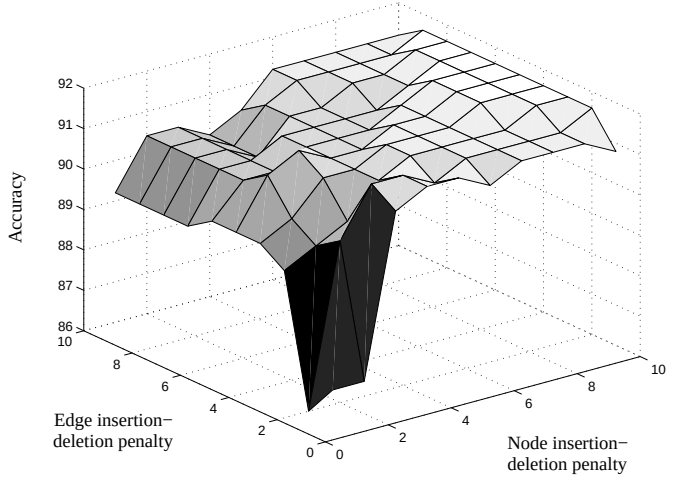



Fig. 6.24 Molecule data set: Accuracy with respect to node insertion and deletion penalty $\gamma_{node}$ and edge insertion and deletion penalty $\gamma_{edge}$

On the molecule data set, the steepest-ascent parameter optimization procedure results in a maximum classification rate of 95.2% on the validation set and 92.6% on the independent test set. The fact that we achieve a classification rate above 90% using the reference method demonstrates that the graph matching approach is very well suited for the structural analysis of molecule data.

## 6.6 Evaluation of Graph Kernels

In this section, the performance of the graph kernels proposed in Chap. 5 is investigated. Since the overall objective is to develop graph kernels that find better decision boundaries than traditional graph classifiers, the main focus is on the classification accuracy. Yet, a second performance criterion, the running time, is of interest as well. In the context of graph matching, the size of graphs and training sets are often limited in practical applications because of inefficient matching methods. The classification results in this section are obtained by applying the kernel functions under consideration to SVMs.[1] A summary of the recognition accuracy on all data sets is provided in Table 6.8, Fig. 6.33, and Fig. 6.34 at the end of this chapter, and the relations between running time and recognition rate are illustrated in Fig. 6.35.

### 6.6.1 *Trivial Similarity Kernel from Edit Distance*

The trivial kernel functions introduced in Sec. 5.3 constitute simple monotonically decreasing transformations mapping edit distance to similarities. These kernels are mainly used for the purpose of providing us with a very basic way to apply edit distance to kernel machines. In view of this, the application of these trivial kernels to SVMs is merely regarded as a reference classification for more complex kernels.

In the case of the trivial kernel functions, there are no parameters to be optimized. However, it seems to be intuitively clear that the graph similarity characteristics that will correspond to an optimal SVM classification will be different from those that will be optimal in a nearest-neighbor classification. Therefore, it is certainly reasonable to perform an independent optimization process of the edit cost parameters for each of the trivial ker-

[1] The SVM implementation used in the experiments is based on LIBSVM [Chang and Lin (2001)].

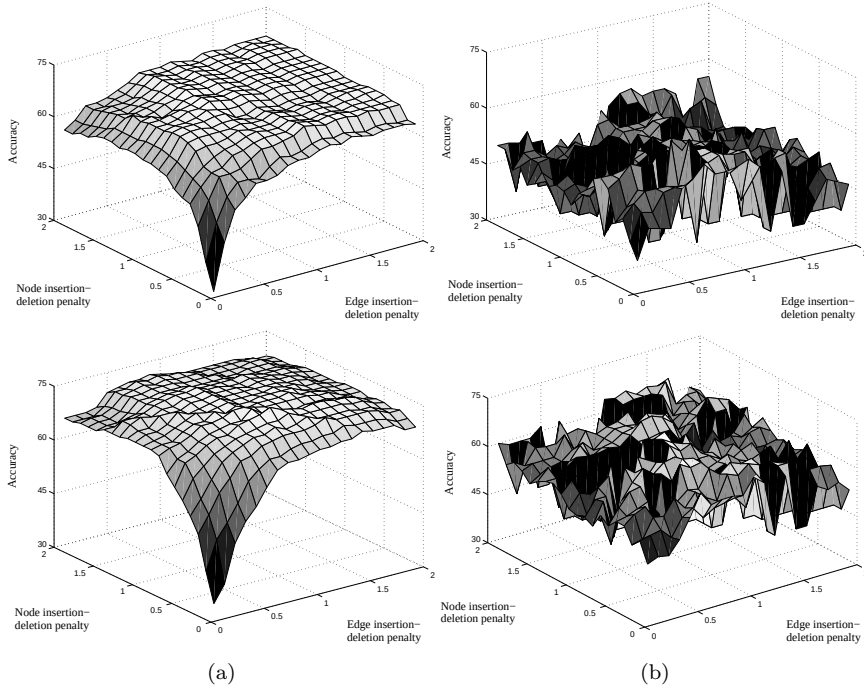



Fig. 6.25 Letter data set: Accuracy on validation set (upper row) and test set (lower row) of (a) an edit distance kNN classifier and (b) a trivial-kernel SVM classifier

Table 6.4 Optimizing parameters for edit distance kNN classifier or trivial-kernel SVM classifier

| | Edit-distance kNN | | Trivial-kernel SVM | |
|---|---|---|---|---|
| Parameter configuration | Validation | Test set | Validation | Test set |
| Optimal for edit distance based kNN classifier | 67.3% | 69.3% | 57.3% | 58.7% |
| Optimal for trivial-kernel SVM classifier | | | 64.0% | 57.3% |

nel functions. Yet, in practice, parameter validation seems to be difficult. In Fig. 6.25, the classification accuracy on the letter database is illustrated for edit cost parameters $\gamma_{node} = 0.1, 0.2, \ldots, 2$ and $\gamma_{edge} = 0.1, 0.2, \ldots, 2$. While the recognition surface of the edit distance based nearest-neighbor classifier in Fig. 6.25a is extremely smooth, the recognition surface of the

trivial kernel function $k_1$ in Fig. 6.25b is much noisier and less regular. Also, comparing the recognition accuracy on the validation set (upper row) with the corresponding recognition accuracy on the test set (lower row), it appears that edit cost parameters of the trivial kernel that are optimal on the validation set will not necessarily perform well on an independent test set. Therefore, it may be more appropriate to apply those edit cost parameters to the trivial kernel functions that correspond to an optimal edit distance based nearest-neighbor classification, so that exactly the same edit distance matrices are fed into the nearest-neighbor classifier and the trivial-kernel SVM. The absolute recognition rates in Table 6.4 confirm that optimizing parameters with respect to the trivial kernel function does not lead to a superior performance on the test set compared to optimizing parameters with respect to the nearest-neighbor classifier, even if the resulting parameter configuration is then applied to an SVM using the trivial kernel. The recognition rates 58.7% of the nearest-neighbor optimal parameter configuration and 57.3% of the trivial kernel optimal parameter configuration do not differ significantly (on a significance level of $\alpha = 0.05$). For these reasons, a trivial kernel specific parameter optimization is not carried out.

The performance of the baseline nearest-neighbor classifier and the trivial similarity kernels, among other graph kernels, are summarized in Table 6.8 (p. 177). Obviously, not even in a single scenario, one of the trivial kernels significantly outperforms the nearest-neighbor classifier. Although trivial kernels are in several cases able to reach and exceed the accuracy of the baseline system, the improvement is never statistically significant. This observation indicates that turning distance matrices into kernel matrices is not sufficient to improve classification. Hence, more sophisticated graph kernels are needed.

### 6.6.2 *Kernel from Maximum-Similarity Edit Path*

The maximum-similarity edit path kernel is based on intuitive arguments and provides a straightforward formulation of the edit distance concept in terms of a similarity measure. However, in practice, the method turns out to be incapable of processing the structural similarity of graphs in such a way that the classification performance can be improved. In fact, the maximum-similarity kernel performs poorly throughout all experiments. The accuracy of the maximum-similarity kernel is shown in Table 6.8 (p. 177). The reason for the disappointing performance of the maximum-similarity kernel is primarily due to the fact that the kernel results in a few disproportionately

larger similarity values than the average. On the letter database, for instance, the largest similarity of graphs from the validation set to graphs from the training set is around 100,000, whereas the average similarity amounts to only 72. Hence, the simple maximum-similarity kernel is not applicable to the problems considered in this book.

In the following, the trivial kernels discussed in the previous section and the maximum-similarity kernel discussed in this section will be considered as reference systems for more complex graph kernels, in addition to the baseline edit distance based nearest-neighbor classifier.

### 6.6.3 *Diffusion Kernel from Edit Distance*

The diffusion kernel described in Sec. 5.5 is based on the idea of transforming edit distance matrices into positive definite kernel matrices. The only parameter of the diffusion kernel is the decay factor $\lambda \geq 0$. The computation of the diffusion kernel matrix involves infinite sums, but the evaluation of the diffusion kernel is sufficiently accurate if only a finite number of addends are taken into account. Hence, the two parameters to be optimized are the decay factor $\lambda$ and the number of addends $t$ in the sum. Since none of the two diffusion kernel variants significantly outperforms the other one in any experiment, only the exponential diffusion kernel is considered in the following.

Performing the matrix exponentiation operation in the computation of the diffusion kernel is inefficient for large matrices, that is, for large training sets. In order to accelerate the kernel computation, the size of the kernel matrix can be reduced by removing patterns from the training set. However, a smaller training set renders the classification of patterns from an independent test set not only faster, but also substantially more difficult. In Fig. 6.26, the running time and accuracy of a diffusion kernel based SVM is compared to the running time and accuracy of a nearest-neighbor classifier. Note that the running time of the nearest-neighbor classifier includes the time needed to compute the edit distance matrix, while the running time of the diffusion kernel only includes the time it takes to turn the edit distance matrix into a kernel matrix and perform the SVM classification. Hence, to correctly classify as many graphs of the validation set as the nearest-neighbor classifier, the diffusion kernel takes more than four times longer. For small training sets, a serious amount of overfitting can be observed.

While the small training set of the image data set is considered a drawback in the general case, it is an advantage in the context of the ineffi-

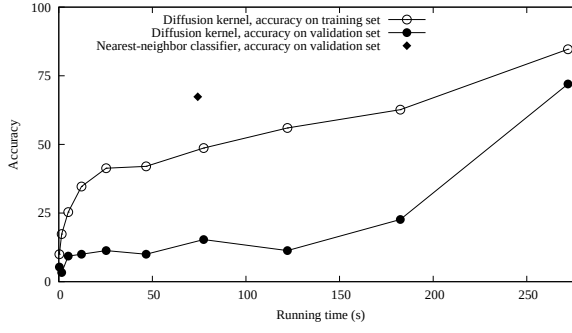



Fig. 6.26 Letter data set, diffusion kernel: Running time and accuracy for training sets of various size

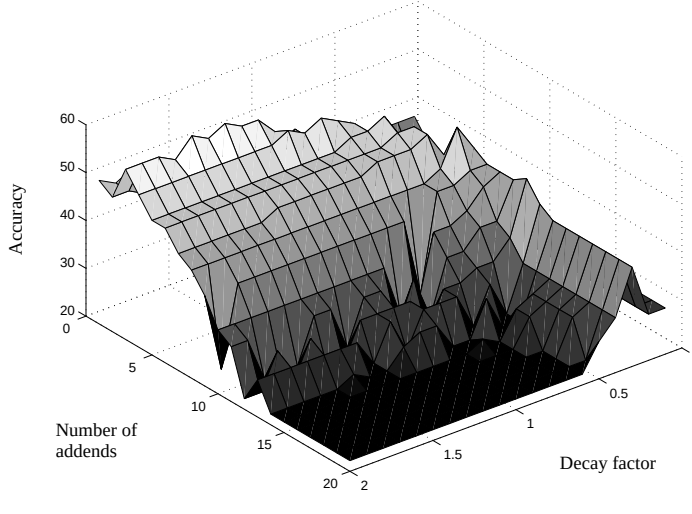



Fig. 6.27 Image data set, diffusion kernel: Accuracy with respect to number of addends $t$ and decay factor $\lambda$

cient diffusion kernel. On the image data set, the recognition accuracy obtained for parameters $\alpha_{node} = 0.1$, $\alpha_{edge} = 0$, $\lambda = 0.1, 0.2, \ldots, 2$, and $t = 1, 2, \ldots, 20$ is illustrated in Fig. 6.27. The best recognition rates are obtained for a small number of addends around $t = 4$ and high decay factors $\lambda$, or for a large number of addends and low decay factors. Using a steepest-ascent search procedure, the overall optimal recognition rate of 70.4% on the validation set is obtained for parameters $t = 6$, $\lambda = 0.142$, and the SVM regularization parameter $\nu = 0.65$. On the independent test set, the diffusion kernel SVM achieves a classification rate of 66.7%.

For the diatom data set, a thorough analysis reveals that those edit cost parameters that are optimal for edit distance based nearest-neighbor classifier perform sufficiently well in conjunction with the diffusion kernel, too. Therefore, the diffusion kernel is applied in conjunction with the edit distance matrix that is optimal for the nearest-neighbor classifier to the diatom data set — a specific optimization of the underlying edit cost parameters is not carried out for the diffusion kernel. The highest recognition rate on the validation set amounts to 75.7% for parameters $t = 10$, $\lambda = 0.0001$, and the SVM regularization parameter $\nu = 0.6$. On the independent test set, an accuracy of 77.8% is obtained. Note that low optimal decay parameters indicate that the diffusion kernel puts the main emphasis on short walks in the graphs under consideration. Also, for low decay parameters, the number of addends to be considered in the infinite sum of the diffusion kernel function can be reduced, since even moderate powers of the decay parameter will be negligibly small.

Unfortunately, the diffusion kernel matrix cannot be computed for the fingerprint data set, because the training set, and hence the corresponding kernel matrix, is too large. Reducing the size of the training set by ignoring several training patterns renders the computation feasible, but makes the classification based on the remaining training elements more difficult. In view of this, it appears to be reasonable that the diffusion kernel only achieves a disappointing recognition rate of 74.0% on the validation set and 67.1% on the independent test set (on a reduced training set of 69 out of 150 patterns). Note that all parameters are optimized on the validation set, including the optimal size of the training set for which the computation of the diffusion kernel is feasible.

Using the diffusion kernel to derive a kernel matrix from an edit distance matrix and applying the kernel matrix to an SVM is clearly less efficient than applying the edit distance matrix to a nearest-neighbor classifier. In some cases, however, the diffusion kernel leads to significantly better recog-

nition rates. The diffusion kernel is therefore not only interesting from the theoretical point of view — providing a simple way to turn a matrix of edit distances into a valid kernel matrix — but performs surprisingly well on difficult graphs extracted from real-world data.

### 6.6.4 *Zero Graph Kernel from Edit Distance*

The zero graph kernel is defined with respect to a set of zero graphs. From the theoretical point of view, it seems to be clear that using more than one zero graph will lead to more complex kernel functions, or similarity measures in the space of graphs, than using only one. Hence, for difficult graph classification problems, kernel functions defined with respect to several zero graphs are intuitively expected to perform better than those defined with respect to a single zero graph. As described in Sec. 5.6, the optimal number of zero graphs is determined on the validation set.

In the following results, the edit cost parameters from the optimal nearest-neighbor classifier are used. The classification performance of an SVM classifier on the validation set and test set for various number of zero graphs is illustrated in Fig. 6.28a. As expected, using several zero graphs instead of a single one is advantageous in terms of the accuracy. The optimal number of zero graphs is rather low compared to the total number of graphs in the training set. Hence, selecting a few suitable zero graphs from the training set seems to be more appropriate than using a single zero graph or the whole training set of graphs. It should also be noted that the zero graph kernel in conjunction with an SVM clearly outperforms the nearest-neighbor classifier based on exactly the same graph edit distance matrix.

The validity of the zero graph kernel cannot be established in general. In Sec. 4.3.4, two criteria have been mentioned that can be used to predict whether the application of invalid kernels to SVMs is reasonable. If the number of negative eigenvalues of the kernel matrix is small and the mutual distances of involved class means are mostly positive (which is not guaranteed in pseudo-Euclidean spaces), the SVM training can be interpreted in a meaningful way as a separation of convex hulls [Haasdonk (2005)]. For the letter data sets, the fraction of negative eigenvalues and the average distance between class means is illustrated in Fig. 6.28b,c for zero sets of size $1, 2, \ldots, 50$. For small numbers, the fraction of negative eigenvalues is large and the average distance between class means is negative, which indicates that using a certain minimum number of zero graphs is advisable.

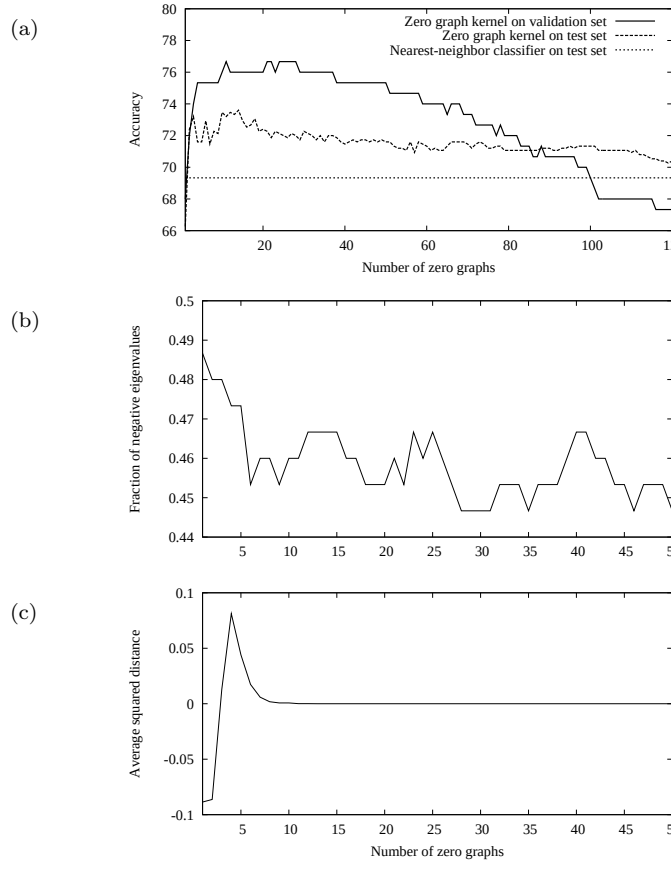



Fig. 6.28 Letter data set, zero graph kernel: (a) Accuracy for zero sets of various size. Criteria for the suitability of invalid kernels, (b) fraction of negative eigenvalues and (c) average squared distance of class means

Hence, the suitability criteria for indefinite kernels correspond quite well to the classification accuracy on the validation set. The iterative greedy algorithm for selecting zero graphs appears to be appropriate for graph matching applications.

The zero graph kernel is defined with respect to an arbitrary dissimilarity measure, which means that the kernel can be applied to any data for which a distance measure can be defined. For instance, instead of considering the edit distance of graphs, one can as well use the edit distance of strings [Wagner and Fischer (1974)] and define a zero string kernel in

Table 6.5 Performance of zero graph kernel on strings and graphs

| Type | Data set | kNN | SVM | Zero patterns | Kernel |
|---|---|---|---|---|---|
| Strings | Chicken pieces | 74.3 | 81.1 • | 4 | $k_I^+$ |
| | Toolset | 53.3 | 66.7 | 3 | $k_I^*$ |
| | Pendigits | 74.2 | 89.7 • | 4 | $k_I^*$ |
| | Chromosomes | 90.7 | 91.1 | 8 | $k_I^+$ |
| Graphs | Letters | 69.3 | 73.2 • | 11 | $k_I^+$ |
| | Images | 48.1 | 59.3 • | 5 | $k_I^+$ |
| | Diatoms | 66.7 | 66.7 | 5 | $k_I^+$ |
| | Fingerprints | 76.2 | 73.9 | 33 | $k_I^+$ |
| | Molecules | 92.6 | 92.7 | 5 | $k_I^+$ |

• Statistically significant improvement on a significance level of $\alpha = 0.05$

analogy to the zero graph kernel. In the following experiment, we use four data sets of strings representing silhouettes of chicken pieces (5 classes, 446 patterns) [Andreu *et al.* (1997)], images of tools (8 classes, 47 patterns) [Siddiqi *et al.* (1999)], handwritten digits (10 classes, 701 patterns) [Alpaydin and Alimoglu (1998)], and chromosomes (21 classes, 280 patterns) [Lundsteen *et al.* (1980)]. For a detailed description of the original data sets, the string extraction procedures, and the string edit distance computation, see [Spillmann (2005)]. Again, the set of zero strings that performs best on the validation set is selected. The classification rates on these four string data sets and the five graph data sets described at the beginning of this chapter are given in Table 6.5. Note that *kNN* indicates the performance of the $k$-nearest-neighbor classifier and *SVM* indicates the performance of the zero graph (zero string) kernel in conjunction with an SVM. It turns out that the nearest-neighbor classifier is outperformed in four out of nine cases by the zero graph (zero string) kernel. Hence, in some cases, the zero graph (zero string) kernel is able to process the edit distance information in such a way that an SVM finds more robust class boundaries than a nearest-neighbor classifier.

### 6.6.5 *Convolution Edit Kernel*

The convolution edit kernel is based on a decomposition of graphs into edit paths. The similarity of two graphs is basically derived from the similarity of their node and edge labels. If RBF kernel functions are used to determine the similarity of labels, the only parameters of the convolution kernel function to be optimized are node and edge RBF kernel parameters $\sigma_{node}$

and $\sigma_{edge}$ (see Table 4.1). In Sec. 5.7, where the validity of the convolution kernel is discussed, it is mentioned that from the theoretical point of view it may be advantageous to include an additive positive constant $b > 0$ in the label similarity function, resulting in $k_{rbf}(x, x') + b$ instead of $k_{rbf}(x, x')$, which can be interpreted as an insertion and deletion penalty cost similar to insertion and deletion edit costs. This extension does not invalidate the property of conditional positive definiteness, and the corresponding kernel function is therefore applicable to SVMs, kernel PCA, and kernel FDA (see Sec. 4.2.1 and Sec. 4.3.4). In the following, we will therefore also consider additive substitution constants $b_{node}$ for node labels and $b_{edge}$ for edge labels, so that the similarity of node labels is specified by $(\sigma_{node}, b_{node})$ and the similarity of edge labels by $(\sigma_{edge}, b_{edge})$.

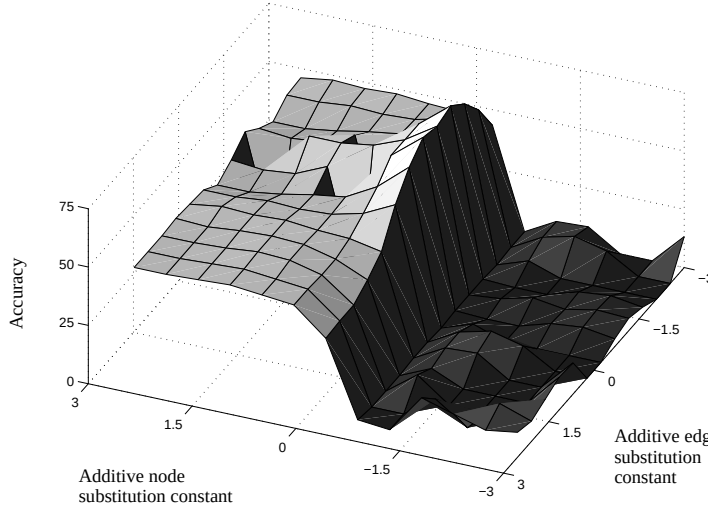



Fig. 6.29 Letter data set, convolution kernel: Accuracy with respect to additive node substitution constant $b_{node}$ and additive edge substitution constant $b_{edge}$

In the first experiment, the SVM classification accuracy on the letter data set is computed for parameters $\sigma_{node} = 2.5$, $\sigma_{edge} = 1$, $b_{node} = -3, -2.5, \ldots, 2.5, 3$, and $b_{edge} = -3, -2.5, \ldots, 2.5, 3$. Note that for the sake of illustration, negative values of $b_{node}$ and $b_{edge}$ are evaluated as well, although these parameters should in fact be positive. The resulting classification accuracy depicted in Fig. 6.29 clearly indicates that the recognition is best for small positive values for $b_{node}$ and values around 1 for $b_{edge}$. Hence, extending the positive definite kernel function to a conditionally

positive definite function leads to an improved recognition performance. Consequently, considering only the parts of two graphs that are structurally similar is not sufficient (standard RBF kernel), but the non-matching parts must be taken into account as well (extended RBF kernel). For a summary of the SVM results of the convolution kernel on all data sets compared to other graph kernels, refer to Table 6.8 (p. 177).

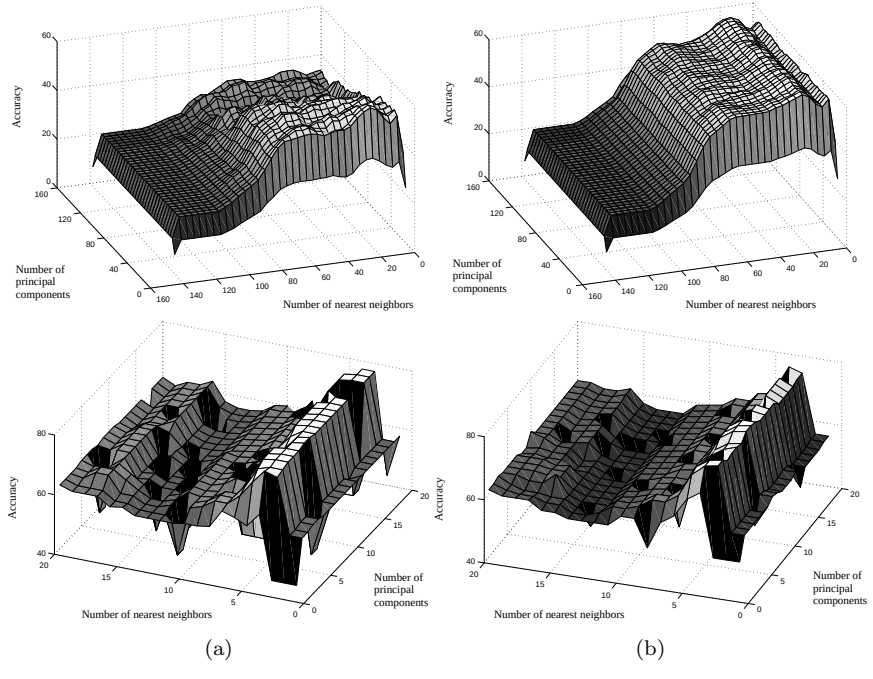



Fig. 6.30 Fingerprint data set, convolution kernel: Performance of (a) kernel PCA and (b) kernel FDA for $d, k = 1 \dots 150$ (upper row) and $d, k = 1 \dots 20$ (lower row)

The most prominent kernel machine for classification is certainly the SVM. In Sec. 4.3.2 and Sec. 4.3.3 kernel PCA and kernel FDA for feature extraction have briefly been introduced. Whereas SVMs aim at separating data in the kernel feature space by hyperplanes, kernel PCA and kernel FDA search for dominant directions in the data. Hence, unlike SVMs, kernel PCA and kernel FDA belong to the class of methods for feature extraction, or dimensionality reduction. The simplest way to turn such feature extraction methods into classifiers is to apply a nearest-neighbor classifier after feature extraction. That is, in a first step kernel PCA or

Table 6.6 Convolution kernel: Performance of SVM, kNN after kernel PCA, and kNN after kernel FDA on validation set (V) and test set (T)

| Data set | SVM | Kernel PCA | Kernel FDA | |
|---|---|---|---|---|
| Letters | 77.3 | 50.7 | 42.7 | (V) |
| | 75.2 • | 46.4 | 37.3 | (T) |
| Images | 72.2 | 55.6 | 59.3 | (V) |
| | 68.5 • | 53.7 | 40.7 | (T) |
| Diatoms | 64.9 | 45.9 | 45.9 | (V) |
| | 72.2 • | 41.7 | 44.4 | (T) |
| Fingerprints | 86.0 | 77.3 | 78.0 | (V) |
| | 80.5 • | 72.2 | 71.7 | (T) |
| Molecules | 98.8 | 91.2 | 91.6 | (V) |
| | 98.0 • | 90.7 | 90.1 | (T) |

• Statistically significant improvement over unmarked entries ($\alpha = 0.05$)

kernel FDA is applied to the graph data under consideration, which results in a feature space of finite dimension where each graph is represented by a vector. The extracted feature vectors can then be classified according to a labeled training set of patterns using a nearest-neighbor classification rule.

In the following, kernel PCA and kernel FDA are computed on the training set and then applied to the validation set. The two parameters of kernel PCA and kernel FDA, the number of principal components $d = 1, \ldots, m$ to be extracted and the number of nearest-neighbors $k = 1, \ldots, m$ to be considered, where $m$ denotes the size of the training set, are optimized on the validation set. The best performing set of parameters is finally applied to the independent test set. The performance of kernel PCA and kernel FDA in conjunction with a nearest-neighbor classifier applied to the fingerprint data set for $d, k = 1, \ldots, 150$ and $d, k = 1, \ldots, 20$ is illustrated in Fig. 6.30. Note that the data in the upper row figures has been averaged to render overall tendencies apparent, which means that absolute values are rather insignificant. Obviously, it is sufficient to consider a rather small number of nearest neighbors (around $k = 3$). The dimension of the space of extracted features does not strongly affect the classification accuracy; optimal values can be found around $d = 5$.

The classification performance of kernel PCA and kernel FDA in conjunction with a nearest-neighbor classifier on the five graph data sets are reported in Table 6.6. Kernel PCA and kernel FDA appear to be considerably less suitable for classification than SVMs. Hence, if a feature extraction algorithm is explicitly needed, that is, if the aim is to obtain

vectorial representations of graphs, kernel PCA and kernel FDA may be an appropriate choice; but if the feature extraction process is only used in combination with a classifier, it is probably advantageous to use a kernel classifier in the first place, such as an SVM. The kernel PCA and kernel FDA results obtained with the convolution kernel are typical for the other kernel functions described in this book as well. The two feature extraction methods are in all cases unable to get close to the performance of SVMs. A more detailed examination of kernel PCA and kernel FDA for classification is therefore omitted.

### 6.6.6 *Local Matching Kernel*

The definition of the local matching kernel is very similar to the definition of the convolution edit kernel in that it is based on a decomposition of graphs into substructures. In the case of the local matching kernel, the decomposition process is not directly related to edit paths, but the similarity of graphs is rather defined by accumulating similarities of local substructures consisting of a single node and its adjacent edges. Whereas the number of decompositions to be considered by the convolution edit kernel is constrained by the condition that two decompositions must correspond to a valid partial edit path, the local matching kernel virtually evaluates all combinatorial possibilities to map nodes from one graph to nodes from the other graph. Hence, the number of decompositions to be considered will be substantially higher for the local matching kernel than for the convolution edit kernel. This effect can very well be observed by limiting the size of decompositions, that is, by taking into account decompositions of graphs into edit paths (convolution kernel) and local substructures (local matching kernel) of a certain maximum length only, and counting the respective number of decompositions in the convolution formula. In Fig. 6.31 the number of decompositions is illustrated for various maximum lengths of decompositions. Clearly, the condition of the convolution edit kernel about valid edit paths substantially constrains the number of edit paths to be considered in contrast to the local matching kernel. It can therefore be expected that the convolution edit kernel will be feasible for longer decompositions in shorter time compared to the local matching kernel.

In practice, the local matching kernel is able to reach the recognition performance of the baseline nearest-neighbor classifier on the letter data set and the image data set, but not on the diatom data set. The graphs from the fingerprint data set, on the other hand, seem to be too large for the local

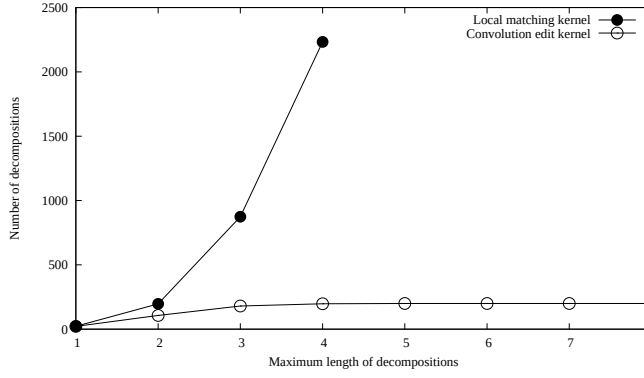



Fig. 6.31 Letter data set: Number of decompositions to be considered by the local matching kernel and convolution edit kernel

matching kernel, so that the huge number of combinatorial combinations of nodes and edges render the kernel computation intractable. Therefore, the local matching kernel fails to classify the fingerprint data set. The recognition rates of all proposed graph kernels are provided in Table 6.8. Comparing the local matching kernel to the convolution edit kernel, it turns out that they exhibit the same tendencies, but the performance of the local matching kernel is constantly inferior. Moreover, while the performance improvement of the convolution edit kernel over the baseline method is statistically significant (in three out of four cases), the local matching kernel never outperforms the baseline system significantly. Hence, it is clear that using the concept of edit paths between graphs in the definition of the convolution edit kernel is one key aspect of the successful application of the convolution edit kernel to graph classification.

### 6.6.7 *Random Walk Edit Kernel*

The random walk kernel is based on the number of similar random walks in two graphs that do not violate the node-to-node correspondence given by the optimal edit path between the two graphs. In this section, the focus is on the performance of the proposed edit distance enhanced random walk kernel compared to a nearest-neighbor classifier and the traditional random walk kernel. The traditional random walk kernel differs from the random walk kernel proposed in this book in that the former does not impose any constraints on nodes of two graphs to be matched, while the latter matches only those pairs of nodes that correspond to a node substitution in the

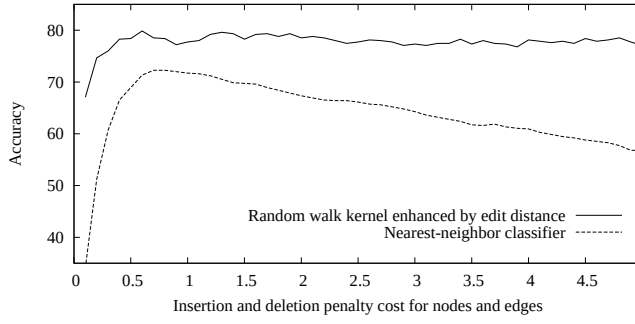



Fig. 6.32 Letter data set, random walk edit kernel: Accuracy for various insertion and deletion penalty costs

optimal edit path between the two graphs.

The traditional random walk kernel is defined with respect to a single weighting parameter $\lambda$, whereas the enhanced random walk kernel additionally depends on the underlying edit cost parameters. In a first experiment, the influence of the edit operation costs on the resulting performance is studied both for a nearest-neighbor classifier and the enhanced random walk kernel. Node (and edge) insertion and deletion costs determine how likely a node (edge) is to be substituted by another node (edge). For low insertion and deletion costs, only a few inexpensive substitutions will occur in a minimum-cost edit path, whereas for high insertion and deletion costs, the edit distance algorithm will tend to substitute as many nodes (edges) as possible. The resulting cost of an optimal edit path, that is, the resulting edit distance, is obviously essential for the performance of a nearest-neighbor classifier. Insertion and deletion costs, as well as other edit cost parameters, therefore have to be carefully optimized to guarantee maximum recognition performance. In the case of the random walk kernel enhanced by edit distance, on the other hand, the main emphasis is on the search for promising node-to-node correspondences rather than a particular distance value. The recognition accuracy of a nearest-neighbor classifier and the enhanced random walk kernel applied to an SVM is illustrated in Fig. 6.32 for various node and edges insertion and deletion costs. The strong influence of the edit cost parameter on the performance of the nearest-neighbor classifier can clearly be observed, which suggests that edit cost parameters should undergo a thorough optimization process. Conversely, the enhanced random walk kernel exhibits a roughly constant

Table 6.7 Performance of nearest-neighbor classifier and random walk kernel

| Data set | Edit distance and kNN | Random walk kernel and SVM | Enhanced random walk kernel and SVM |
|---|---|---|---|
| Letters | 69.3 | 75.7 • | 74.7 • |
| Image | 48.1 • | 33.3 | 51.9 • |
| Diatoms | 66.7 • | 44.4 | 58.3 • |
| Fingerprints | 76.2 | 78.2 • | 80.4 •∘ |
| Molecules | 92.6 | — | — |

• Statistically significant improvement over unmarked entries ($\alpha = 0.05$)
•∘ Statistically significant improvement over all other entries ($\alpha = 0.05$)

performance behavior for edit costs above a certain threshold and outperforms the nearest-neighbor classifier for all values of the edit cost parameter. Hence, for the proposed random walk kernel, an extensive optimization of edit cost parameters seems to be less important than for nearest-neighbor classifiers.

In the following, the performance of the proposed enhanced random walk kernel is compared to the performance of a nearest-neighbor classifier and the traditional random walk kernel. In Table 6.7, the corresponding recognition rates on independent test sets are given for all five graph data sets. It turns out that the nearest-neighbor classifier outperforms the traditional random walk kernel on the image data set and the diatom data set, whereas the latter outperforms the former on the letter data set and the fingerprint data set. Also, the random walk kernel cannot be applied to the molecule data set, since the rather large molecule graphs render the computation unfeasible. Hence, the characteristics of the underlying data set determine whether the edit distance or the traditional random walk kernel is more suitable for graph matching. In view of this, the two methods obviously emphasize complementary aspects of matching the structure of two graphs and therefore address the graph matching problem in different ways. The proposed random walk kernel enhanced by edit distance, on the other hand, never performs significantly worse than the two baseline methods throughout the experiments on all data sets (except for the molecule database). In each row of Table 6.7, entries that are marked with the same symbol are not significantly different from each other (on a statistical significance level of $\alpha = 0.05$). Clearly, the performance of the proposed method is as good as that of the better baseline method on four data sets. On the fingerprint data set, the enhanced random walk kernel even results in

a statistically significant improvement of the recognition performance over the other methods, which means that the combination of the two classifiers outperforms the individual ones. Hence, the proposed extension of random walk kernels can be regarded as a combination of edit distance and random walk based graph matching that performs well on all data sets, provided that the random walk kernel is applicable at all. Enhancing the locally defined random walk kernel by global graph matching information is therefore advantageous for graph classification.

## 6.7 Summary and Discussion

In this section, a concluding summary of the experimental results is provided. The most important aspect of classifier performance is a classifier's ability to correctly predict the class of patterns from an independent test set. In Table 6.8, Fig. 6.33, and Fig. 6.34, the recognition rates on the validation set and on the independent test set are shown for the classifiers discussed in previous chapters.

The *edit distance based nearest-neighbor classifier* described in Sec. 4.4 constitutes the baseline graph classification system. Referring to the first line of Table 6.8, the recognition rates obtained on the letter data set indicate that the nearest-neighbor classifier does not overfit the training data. Also, only a weak tendency towards overfitting can be observed on the molecule data set. However, on the three other data sets, the accuracy on the validation set is substantially higher than that on the independent test set. In these cases, the nearest-neighbor classifier is too strongly focused on the characteristics of the training set instead of learning the class boundaries of the underlying graph population. The rather small training sets of the image data set and diatom data set render the classification of unknown patterns additionally difficult, since in these cases nearest-neighbor classification is particularly prone to misclassifications due to outliers.

The *trivial similarity kernels from edit distance* have mainly been introduced as reference systems for more complex graph kernels. The idea is to evaluate whether turning edit distance matrices into similarity matrices and applying them to SVMs may be more suitable for classification than distance based nearest-neighbor classifiers. The results in Table 6.8, Fig. 6.33, and Fig. 6.34 clearly demonstrate that the performance cannot significantly be improved with this approach. The trivial kernels seem to exhibit the same overfitting tendencies as the nearest-neighbor classifier.

Table 6.8 Summary of the accuracy on the validation set (V) and test set (T)

| | Letter | Image | Diatom | Fingerprint | Molecule | |
|---|---|---|---|---|---|---|
| Edit distance | 67.3 | 64.8 | 86.5 | 84.0 | 95.2 | (V) |
| and kNN (4.4) | 69.3 | 48.1 | 66.7 | 76.2 | 92.6 | (T) |
| Trivial similarity | 57.3 | 59.3 | 70.3 | 69.3 | 82.8 | (V) |
| kernel $k_1$ (5.3) | 58.7 | 53.7 | 61.1 | 64.1 | 84.3 | (T) |
| Trivial similarity | 58.7 | 63.0 | 78.4 | 76.7 | 90.0 | (V) |
| kernel $k_2$ (5.3) | 66.1 | 55.6 | 66.7 | 71.3 | 90.4 | (T) |
| Trivial similarity | 36.7 | 64.8 | 54.1 | 80.0 | 80.0 | (V) |
| kernel $k_3$ (5.3) | 42.1 | 57.4 | 55.6 | 76.1 | 80.3 | (T) |
| Trivial similarity | 50.7 | 64.8 | 56.8 | 77.3 | 80.0 | (V) |
| kernel $k_4$ (5.3) | 53.7 | 55.6 | 58.3 | 74.7 | 80.3 | (T) |
| Max. similarity | 31.3 | 22.2 | 40.5 | 25.3 | 80.0 | (V) |
| kernel (5.4) | 26.8 | 22.2 | 36.1 | 22.6 | 79.9 | (T) |
| Diffusion edit | 72.0 | 70.4 | 75.7 | 74.0 | 80.0 | (V) |
| kernel (5.5) | 75.2 •○ | 66.7 • | 77.8 • | 67.1 | 80.0 | (T) |
| Zero graph | 76.7 | 72.2 | 86.5 | 81.3 | 92.8 | (V) |
| kernel (5.6) | 73.2 •○ | 59.3 • | 66.7 | 73.9 | 92.7 | (T) |
| Convolution edit | 77.3 | 72.2 | 64.9 | 86.0 | 98.8 | (V) |
| kernel (5.7) | 75.2 •○ | 68.5 • | 72.2 | 80.5 •○ | 98.0 •○ | (T) |
| Local matching | 74.7 | 66.7 | 54.1 | — | — | (V) |
| kernel (5.8) | 72.4 | 57.4 | 44.4 | — | — | (T) |
| Random walk | 74.7 | 64.8 | 73.0 | 86.0 | — | (V) |
| edit kernel (5.9) | 74.7 •○ | 51.9 | 58.3 | 80.4 •○ | — | (T) |

• Stat. significant improvement over edit distance based kNN classifier ($\alpha = 0.05$)
•○ Stat. significant improvement over the first six classifiers in this table ($\alpha = 0.05$)

On the letter data set, the diatom data set, the fingerprint data set, and the molecule data set, the trivial kernels are not even able to reach the recognition rate of the baseline classifier. Hence, we find that the trivial kernel functions do not provide us with an appropriate way to apply graph edit distance to kernel machines.

On all data sets, the *maximum similarity edit path kernel* performs extremely poorly and is only able to reach about half the recognition rate of

the baseline system and the other graph kernels (except for the molecule data set). Therefore, the maximum similarity kernel is not included in Fig. 6.33 and Fig. 6.34 for the sake of readability. Multiplying individual node and edge label similarities to obtain a similarity measure on edit paths is not a reasonable approach to graph matching. Summarizing, it appears that re-formulating the edit distance concept in a similarity context — either by transforming distances into similarities or by combining individual label similarities into a graph similarity measure — does not lead to improvements over nearest-neighbor classification. This observation strongly suggests that more elaborate graph kernels are needed. Still, the underlying edit distance concept seems to be a promising approach to error-tolerant graph matching, due to its flexibility and universal applicability.

On the first three data sets, the *diffusion kernel* clearly outperforms the baseline classifier. Not only does the diffusion kernel reach a higher overall classification accuracy, but it is also less vulnerable to overfitting. In Fig. 6.35, the running time and accuracy of the edit distance based nearest-neighbor classifier is compared to the running time and accuracy of some of the proposed graph kernels. The time it takes to compute the edit distance matrix, or the kernel matrix, and perform the nearest-neighbor classification, or the SVM classification, is indicated on the horizontal axis, while the vertical axis represents the resulting classification accuracy. Note that the running time and accuracy are measured for the classification of the independent test set — the training and validation process is not accounted for in these computations. Obviously, the upper left corner of each plot in Fig. 6.35 is the optimal area of performance, corresponding to a fast (low running time) and well-performing (high accuracy) classifier. Comparing the running time and accuracy of the baseline classifier ($\circ$) and the diffusion kernel ($\triangle$) on the first three data sets, we find that the diffusion kernel is not constantly faster or slower than the baseline system, but achieves a higher recognition rate. On the letter data set (together with the convolution edit kernel) and on the diatom data set, the diffusion kernel even performs better than all other evaluated classifiers. The improvement of the classification accuracy on the letter data set over the baseline system and the trivial reference kernels is statistically significant. Concluding, the diffusion kernel seems to be a reliable and reasonably efficient alternative to nearest-neighbor classification in cases where an edit distance matrix is given. On the fingerprint data set, the matrix computation needed for the derivation of the diffusion kernel matrix fails due to memory constraints, and the resulting classification performance based on a reduced training set

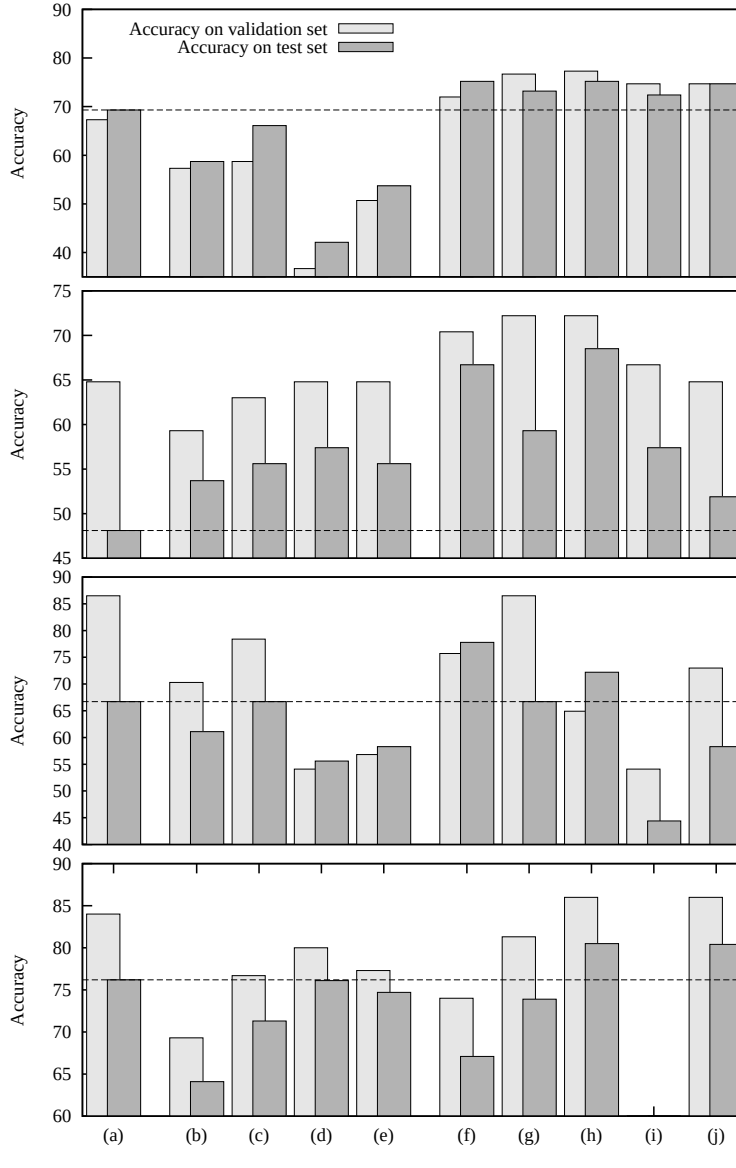



Fig. 6.33 Summary of the accuracy on the letter, image, diatom, and fingerprint data sets (from top to bottom). (a) Nearest-neighbor classifier, (b)–(e) trivial kernels, (f) diffusion kernel, (g) zero graph kernel, (h) convolution edit kernel, (i) local matching kernel, and (j) random walk edit kernel

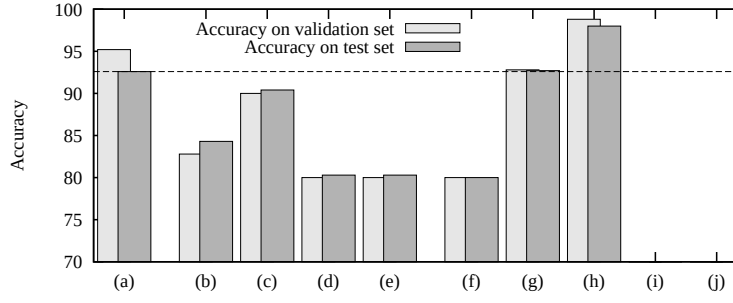



Fig. 6.34 Summary of the accuracy on the molecule data set. (a) Nearest-neighbor classifier, (b)–(e) trivial kernels, (f) diffusion kernel, (g) zero graph kernel, (h) convolution edit kernel, (i) local matching kernel, and (j) random walk edit kernel

is significantly lower, as expected, than that of the baseline system. On the molecule data set, the diffusion kernel performs equally poorly.

Similarly to the diffusion kernel, the *zero graph kernel* is based on the idea of interpreting edit distance data such that similarities are obtained. While the diffusion kernel is in fact a valid kernel function, the zero graph kernel is not. In this regard, it seems to be comprehensible that the performance of the zero graph kernel is lower than that of the diffusion kernel on three out of five data sets. On the letter data set and the image data set, the zero graph kernel significantly outperforms the baseline system. Compared to the other graph kernels, a rather large amount of overfitting is involved in the training of the zero graph kernel. In Fig. 6.35, it can be observed that the running time of the zero graph kernel is largely identical to the running time of the baseline system, since the computational bottleneck of the zero graph kernel is the computation of the edit distance matrix. To further study the tradeoff between running time and accuracy, those classifiers whose time complexity can be controlled to a certain extent are investigated in greater detail. For each of these classifiers, the basic idea is to apply several classifier instances of various degrees of complexity to the letter validation set and measure the running time and recognition rate. (The validation set is used instead of the test set because the letter validation set is substantially smaller than the letter test set, which makes computations more efficient, particularly for classifiers of increased complexity.) The time complexity of the baseline system is controlled by limiting the number of initial seed substitutions in the approximate edit distance algorithm (see Sec. 3.4). Similarly, the time complexity of the

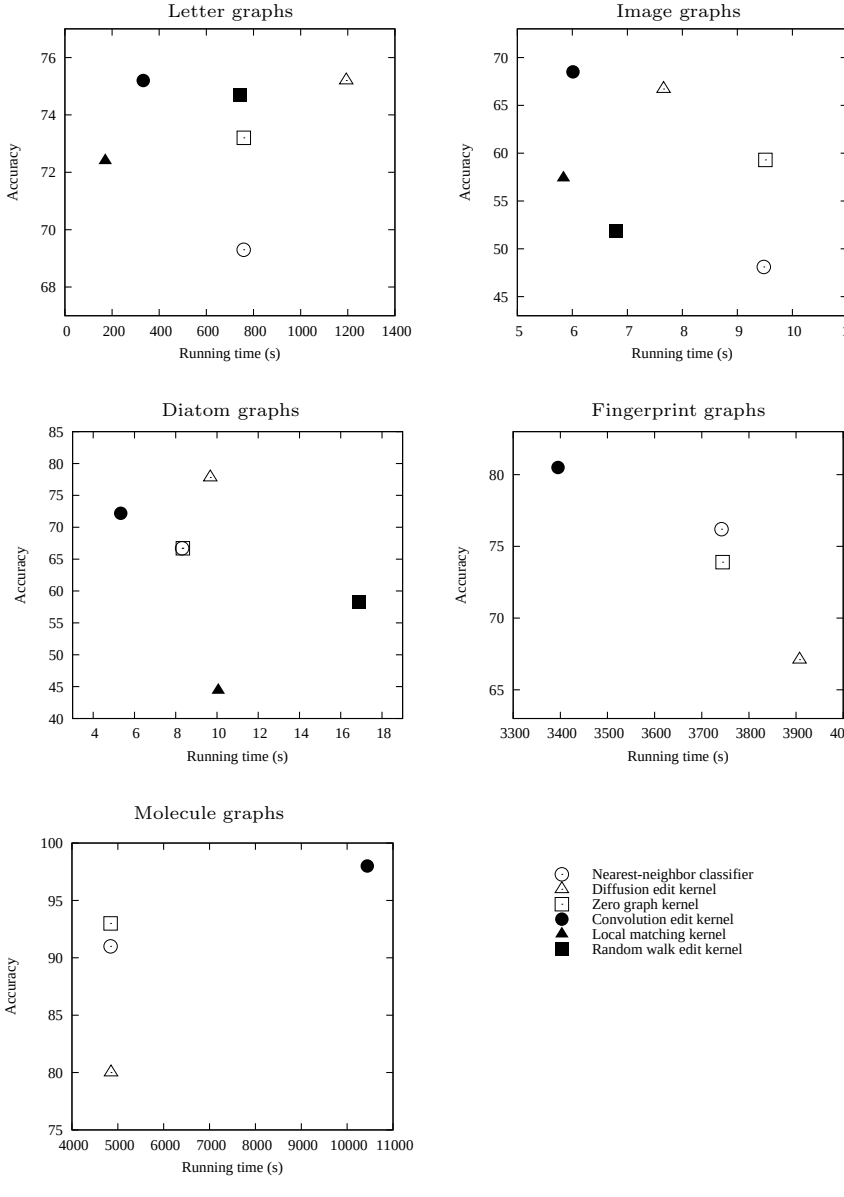



Fig. 6.35 Running times and classification rates on the test set

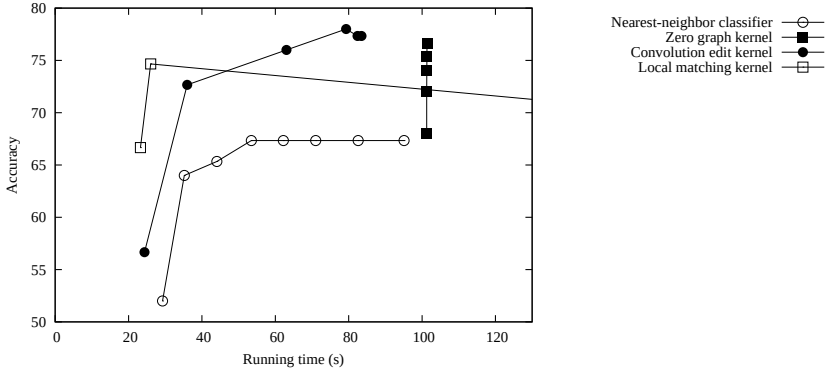



Fig. 6.36 Tradeoff between low running time and high accuracy (on the validation set)

zero graph kernel is controlled by limiting the number of zero graphs (see Sec. 5.6). It appears to be clear that allowing for a larger number of seed substitutions (or a larger set of zero graphs) will make the resulting edit distance classifier (or SVM classifier) more accurate, but slower. In Fig. 6.36, the resulting running times and recognition rates on the letter data set are illustrated. Note that the results illustrated in this figure are not directly comparable to the results illustrated in Fig. 6.35, as the former are related to the test set and the latter, for reasons of efficiency, to the validation set. The tradeoff between low running times and high recognition rates can very well be observed for the nearest-neighbor classifier. Increasing the number of zero graphs, on the other hand, does not visibly affect the running time of the zero graph kernel, since the evaluation of the kernel function and the SVM classification are both extremely efficient in contrast to the time-consuming edit distance computation, which does not depend on the number of zero graphs. Overall, the zero graph kernel achieves a higher recognition rate than the baseline system, but is not competitive in terms of running time.

The *convolution edit kernel* performs convincingly well on all data sets. Not only does the convolution edit kernel outperform the baseline classifier and all trivial reference kernels in every single scenario, but the convolution edit kernel can also be computed faster than any other of the described classifiers (except for the local matching kernel, which is faster on two data sets). Only on the molecule data set, the convolution kernel seems to be lacking in terms of efficient computation. On four out of five data sets, the convolution edit kernel achieves a statistically significant improvement

Table 6.9 Accuracy and running time of the edit distance based nearest-neighbor classifier and the convolution edit kernel

| | Letter | Image | Diatom | Fingerprint | Molecule |
|---|---|---|---|---|---|
| Nearest-neighbor classifier | | | | | |
| Accuracy | 69.3 | 48.1 | 66.7 | 76.2 | 92.6 |
| Running time | 13' | 9" | 8" | 62' | 81' |
| Convolution edit kernel | | | | | |
| Accuracy | 75.2 • | 68.5 • | 72.2 | 80.5 • | 98.0 • |
| Running time | 6' | 6" | 5" | 57' | 174' |

• Statistically significant improvement over other classifier ($\alpha = 0.05$)

of the accuracy over the baseline system. In the only non-significant case, the increase of the recognition rate from 66.7% to 72.2% is not statistically significant mostly because of the very small diatom test set, which makes it difficult to achieve significant improvements. Moreover, the amount of overfitting of this kernel function appears to be rather small. Figure 6.35 and Fig. 6.36 show that the convolution edit kernel is faster than the baseline system on four out of five data sets and more accurate on all of them. The speedup in computational time of the convolution kernel over the baseline system is mainly due to the limitation of the length of edit path decompositions, in contrast to complete edith paths considered in the standard or approximate edit distance algorithm. The actual performance numbers are given in Table 6.9. In view of the overall objective of this book, which is the definition of graph kernels that perform better than the baseline system, it turns out that the convolution kernel offers a powerful alternative to nearest-neighbor classification.

The *local matching kernel* is closely related to the convolution edit kernel. The number of decompositions to be considered in the local matching kernel increases exponentially with an increasing length of decompositions (see Fig. 6.31). Therefore, in constrast to the convolution edit kernel, the computation of the local matching kernel is often limited to rather short decompositions, which makes the local matching kernel faster, but also less accurate than the convolution kernel. This behavior can clearly be observed in Fig. 6.35. On the validation set of the letter database, the local matching kernel is quite competitive with the convolution edit kernel in terms of running time for short decompositions, as shown in Fig. 6.36, but falls back if the length of decompositions is increased. Because of its computational complexity, the local matching kernel is not applicable to larger graphs, such as the graphs extracted from the fingerprint database

and the molecule database. Recall that the convolution edit kernel differs from the local matching kernel in the condition that only valid edit paths between graphs are considered instead of combinatorial node-to-node mappings. Hence, it turns out that the edit path concept is relevant for successfully applying the convolution kernel to graph matching, and simply convolving node and edge label similarities is not sufficient.

On all data sets, the *random walk edit kernel* does not perform significantly worse than the baseline system, but achieves a statistically significant improvement of the classification accuracy on two data sets only. On the molecule data set containing large graphs, the random walk kernel cannot be computed. Overall it appears that the random walk kernel performs well if a sufficient number of training elements are present, which is satisfied in the case of the letter data set and the difficult fingerprint data set. It should be noted that, on the fingerprint data set, only the convolution edit kernel and the random walk kernel are able to achieve a higher recognition rate than the baseline system. Yet, the running time of the random walk kernel on this data set amounts to 23h compared to the running time of less than 1h of the convolution edit kernel and has therefore been left out in the fingerprint plot in Fig. 6.35. Also, since the running time of the random walk kernel cannot be controlled by a parameter, such as the length of decompositions in the case of the convolution edit kernel, the random walk kernel is not included in Fig. 6.36.

# Chapter 7

# Conclusions

This book addresses the issue of matching graphs by means of kernel functions that are to a certain degree related to graph edit distance. The main advantage of using graphs for the representation of patterns over the traditional feature vector approach in pattern recognition is that graphs constitute a more powerful class of data structures than vectors. For example, in statistical pattern recognition, each pattern is represented by a feature vector of constant dimension, regardless of the complexity of the underlying object. In order to capture all characteristic properties of a pattern that may be relevant for classification, it is therefore often necessary to identify a large number of descriptive features in a time-consuming manual process. The definition of appropriate features is particularly difficult in cases where patterns are composed of several objects, for instance resulting from an interest point detection or region segmentation process. Especially if structure plays an important role, graphs constitute a versatile alternative to feature vectors. In structural pattern recognition based on graphs, the idea is to transform patterns into graphs and then perform the analysis and classification of patterns in the domain of graphs. In the general case, relevant properties of patterns under consideration can be stored in labels attached to nodes and edges, in addition to the node and edge structure reflecting the structure of the underlying patterns.

The process of evaluating the structural similarity of graphs is commonly referred to as graph matching. For quite some years, a large number of methods for graph matching have been proposed based on various notions of structural error and models of optimal node and edge correpondence between graphs. Many of these methods are only applicable to a restricted set of graph structures or weakly distorted graphs. One of the most powerful graph matching paradigms for unconstrained graphs with arbitrarily

labeled nodes and edges is based on graph edit distance. The edit distance of graphs is defined by the amount of distortion that is needed to turn one graph into the other. Graph edit distance can be adapted to various graph matching scenarios by means of underlying edit cost functions. For these reasons, as far as the structural matching of graphs is concerned, the edit distance is considered to be extremely flexible and has found widespread application. However, the subsequent processing of the resulting distance data is often limited to a small number of pattern recognition methods. This limitation is due to the fact that, if graph edit distance is used, the only mathematical structure that is known in the space of graphs is the dissimilarity of graphs. Thus, edit distance based pattern classification is essentially restricted to methods based on the nearest-neighbor paradigm.

In recent years, a novel class of methods for pattern recognition, called kernel machines, have been proposed. Kernel machines are based on kernel functions defining the pairwise similarity of patterns. The intuitive idea is to generalize basic algorithms to more powerful and more widely applicable methods by means of an elegant mathematical result from kernel theory. Given a pattern space and a kernel function assigning similarities to pairs of patterns, the kernel function is known to infer the existence of a related feature vector space endowed with an inner product, provided that the kernel function satisfies certain properties. In addition to the existence of this kernel feature space, a mapping from the pattern space to this feature space is known to exist such that evaluating the kernel function for two patterns is equal to mapping the two patterns to the feature space and computing their inner product. Hence, the kernel feature space is related to the original pattern space in that the kernel function (defined in the pattern space) is equal to the inner product (defined in the kernel feature space). The definition of a pattern similarity measure, or kernel function, in a pattern space therefore implies a rich mathematical structure in a related inner product space. Since a large number of algorithms for pattern analysis and recognition are formulated in terms of inner products only, it is sufficient to know the inner product of patterns to apply these algorithms in the kernel feature space. Consequently, since the kernel function in the original pattern space is equal to the inner product in the kernel feature space, it is not even necessary to explicitly construct the kernel feature space to compute inner products. Kernel machines are characterized by the fact that they only need to know how patterns are related in terms of an inner product. The class of kernel machines includes kernel principal component analysis for unsupervised feature extraction, kernel Fisher discriminant analysis

for supervised feature extraction, and support vector machines (SVMs) for classification and regression.

In the graph matching context, the advantage of using kernel machines for classification is two-fold. First of all, for structural and non-structural patterns, kernel machines offer a straight-forward extension of standard linear algorithms to non-linear ones, which can be effected simply by replacing the inner product in any kernel algorithm by a kernel function. Theoretical considerations suggest that transforming data non-linearly into high-dimensional vector spaces may be advantageous for classification. Secondly, kernel machines provide us with a universal approach for the application of standard algorithms defined in vector spaces to the space of graphs. By means of graph kernel functions, the recognition problem can be mapped from the sparse space of graphs to an inner product space, where each graph is represented by a vector. The kernel approach thus allows us to overcome the fundamental limitation of graph matching by embedding the space of graphs into a vector space and in this way renders powerful methods from statistical pattern recognition applicable to graphs. For graph classification, the use of SVMs seems to be particularly appealing, since the SVM training is based on principles from statistical learning theory and is expected to provide for a balanced condition between overfitting and underfitting.

In this book, a number of graph kernel functions related to graph edit distance are proposed. The basic rationale is to combine the flexibility of edit distance with the power of kernel machines. The edit distance is considered one of the best performing methods for unconstrained graph matching, and SVMs are expected to be better suited for pattern classification than nearest-neighbor classifiers, which is based on theoretical arguments and practical evidence. An experimental evaluation of the recognition performance of traditional and the novel methods proposed in this book is carried out on five challenging graph data sets. The semi-artificial letter data set consists of 1,050 graphs from 15 classes representing line drawings of capital letters. This data set has been generated by applying random distortions to manually constructed class prototypes. The second data set, the image data set, contains 162 graphs from five classes extracted from images. These graphs are region adjacency graphs resulting from a color-based segmentation process on images. The diatom data set consists of 110 microscopic images of diatoms from 22 classes. This data set is considered to be very difficult for classification since each class consists of only five sample graphs (each to be assigned either to a training set, validation set, or independent

test set). The fourth data set contains 3,300 graphs representing fingerprints from the five main Henry-Galton fingerprint classes. The last data set contains 2,000 molecules from the class of molecules that are inactive against HIV and the class of molecules that are confirmed active against HIV. These five graph data sets can be considered quite diverse, based on descriptive characteristics such as the number of graphs, the number of classes, and the average number of nodes and edges. In view of this, it is clear that the development of graph matching methods performing well on all four data sets must be considered a difficult problem.

The first set of graph kernels defined in this book are kernels based on monotonically decreasing transformations mapping distances to similarities. The idea behind this approach is to investigate whether edit distance data can successfully be applied to SVMs with minimal modifications. In experiments it turns out that none of these trivial kernels is able to significantly outperform the baseline system, which is an edit distance based $k$-nearest-neighbor classifier. The actual choice of transformation does have a certain influence on the performance, but an overall tendency that one transformation is more suitable than the others cannot be observed. Next, another graph kernel is constructed by re-formulating the minimum-cost edit path of the edit distance algorithm into a maximum-similarity edit path method. The basic idea is to evaluate the similarity of node and edge labels instead of penalty costs of edit operations. As far as the recognition performance is concerned, the maximum-similarity kernel fails in almost all experiments and is typically only able to achieve half the classification accuracy of the baseline classifier. Hence, simple approaches to turn edit distance into similarities in conjunction with SVMs bear no advantage over edit distance based nearest-neighbor classification.

A collection of more complex graph kernels is proposed in the second part of this book. The *diffusion kernel* turns edit distance matrices into valid kernel matrices by evaluating not only the distance between two patterns, but also the number of similar patterns they have in common. The computation of the diffusion kernel involves a matrix exponentiation operation, which makes the evaluation of diffusion kernels inefficient for large sets of training patterns. On the fingerprint data set, the computation of the diffusion kernel is unfeasible due to memory constraints, and on a reduced training set, the diffusion kernel's performance is inferior to that of other kernels. On the molecule data set involving large graphs, the diffusion kernel can be computed, but is not able to reach the classification accuracy of a standard nearest-neighbor classifier. On the other three graph data

sets, however, the diffusion kernel performs convincingly well and leads to a significant improvements of the classification accuracy over the baseline system. The *zero graph kernel* is a graph kernel defining the similarity of graphs by their distance to a set of zero graphs based on a theoretical result from kernel function theory. The zero graph kernel is not generally valid, but an analysis of two criteria for the applicability of invalid kernel functions to SVMs suggests that it may nevertheless be reasonable to use this kernel for classification. The kernel function is not limited to graph data, but can be applied to any data for which a distance measure is available. The zero graph kernel outperforms the baseline classifier on four of the five graph datsets and on four string data sets, using string edit distance, but the improvement is statistically significant in only four cases. Hence, at least on some data sets, the zero graph kernel might be a viable alternative to nearest-neighbor classification.

The *convolution edit kernel* belongs to the class of convolution kernels and is based on ideas from graph edit distance. The similarity of two graphs is obtained by convolving products of similarities of pairs of node labels and pairs of edge labels, so that high similarity values indicate that two graphs contain many similar nodes and edges. The concept of substitution costs, insertion costs, and deletion costs can be modeled, to a certain extent, by parameters of the label similarity functions, and only valid edit paths between two graphs are considered in the convolution operation. The convolution edit kernel function is valid and performs very well in terms of recognition rate and running time. In the experiments, the convolution edit kernel is more accurate on all five graph data sets and faster than the baseline classifier on four of them, resulting in a significant improvement on the letter data set, the image data set, the fingerprint data set, and the molecule data set. The *local matching kernel* is an alternative convolution kernel that differs from the convolution edit kernel in that graphs are not decomposed into edit paths, but into arbitrary sets of nodes and their edges. Hence, the local matching kernel is conceptually simpler, but because of fewer constraints, a larger number of decompositions must be considered in comparison to the convolution edit kernel. Accordingly, it turns out that the local matching kernel is often slower and constantly less accurate than the convolution edit kernel. On all five data sets, the local matching kernel does not achieve significantly higher classification rates than the baseline system. On the two data sets with rather large graphs, the local matching kernel cannot even be computed. The *random walk edit kernel* is based on the notion of random walks in graphs. The idea is to define the similarity

of two graphs by the number of similar random walks in these graphs. The computation of the random walk kernel can be effected efficiently by turning pairs of graphs into direct product graphs and analyzing direct product adjacency matrices. In contrast to traditional random walk kernels, the random walk edit kernel proposed in this book combines the locally defined random walk kernel with edit distance information. The idea is to identify only those pairs of nodes in the random walk kernel that correspond to a node substitution in the optimal edit path. While the baseline classifier outperforms the traditional random walk kernel on the image and diatom data sets, and vice versa on the letter and fingerprint data sets, the recognition rate of the proposed random walk edit kernel is always at least as good as the higher recognition rate of the baseline classifier and the traditional random walk kernel. On the molecule data set, the computation of the traditional random walk kernel as well as that of the proposed extended random walk kernel are unfeasible.

The experimental results clearly demonstrate that edit distance based graph kernels are advantageous for graph classification over traditional nearest-neighbor classifiers. Among the proposed graph kernels, the convolution edit kernel offers the best tradeoff between conceptual complexity and close relationship to graph edit distance on one hand and computational efficiency and applicability to real world recognition problems on the other hand. The convolution kernel is more accurate on all five data sets and faster than the baseline system, and most other graph kernels, on four of them. In cases where the convolution edit kernel fails, the diffusion edit kernel or the random walk edit kernel might be viable alternatives.

The present book is focused on the problem of graph classification. However, the pattern recognition methods applicable in conjunction with kernel functions comprise a larger set of algorithms for pattern analysis, feature extraction, clustering, data mining, regression, and many more. Convincing results obtained in pattern classification warrant further research in applying graph kernels to a wider range of algorithms. So far, it appears that the advantage of using graph kernels, allowing us to address the recognition problem in a rich inner product space instead of the sparse space of graphs, has not yet fully been exploited. Also, constructing novel graph kernels by applying ideas from edit distance to well-known classes of kernel

functions, such as convolution kernels or random walk kernels, seems to be a promising way to approach unconstrained graph matching. Certainly, the basis for novel graph kernels is not limited to graph edit distance, but a huge number of error-tolerant graph matching methods might potentially serve as starting points for the development of graph kernels.

This page intentionally left blank

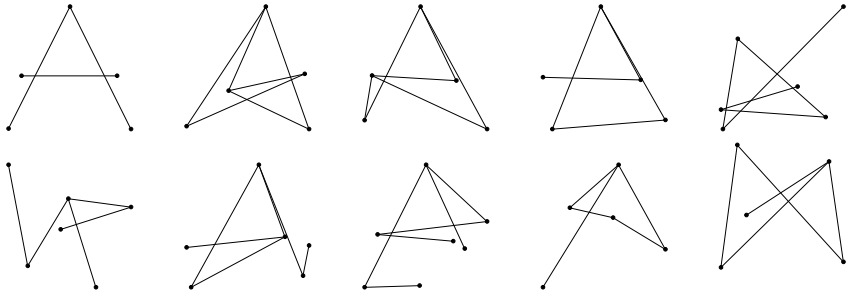
# Appendix A

# Graph Data Sets

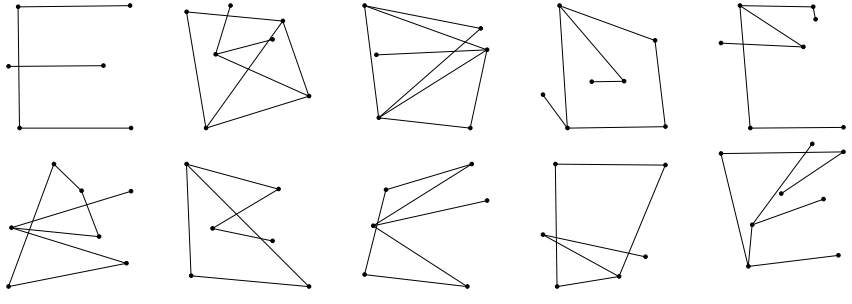
In Chap. 6, the graph kernel functions proposed in this book are evaluated in terms of recognition performance and running time in experiments on graphs from five different data sets representing line drawings, pictures, microscopic images, fingerprints, and molecules. The structural properties of the graphs extracted from the underlying patterns and the data set characteristics are discussed in greater detail in Sects. 6.1, 6.2, and 6.3. For a better understanding of the data and the difficulty of the corresponding classification tasks, we provide in this appendix a comprehensive overview on the actual structure and image data our graph classification experiments are based on. To this end, a large number of sample patterns from the five data sets are illustrated in the following sections.

## A.1 Letter Data Set

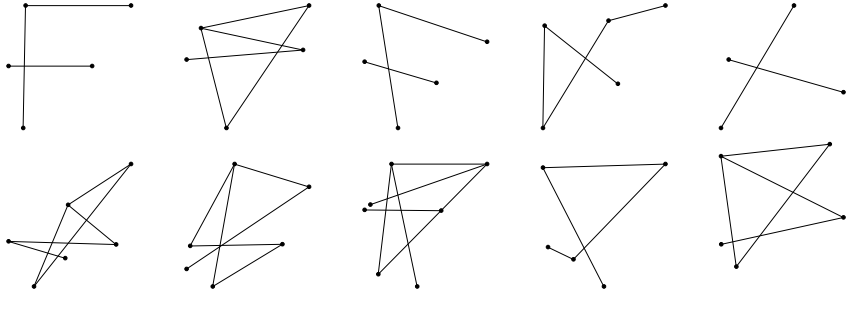
The letter data set consists of 1,050 line drawings representing distorted capital letters *A, E, F, H, I, K, L, M, N, T, V, W, X, Y*, and *Z*. A detailed description of the graph generation process and the data set is given in Sec. 6.1.1. In the following, we provide for each class an illustration of a clean class prototype and nine distorted samples randomly selected from the graph data set. Note that generally rather strong distortions have been applied resulting in line drawings that differ significantly from the letter prototype.



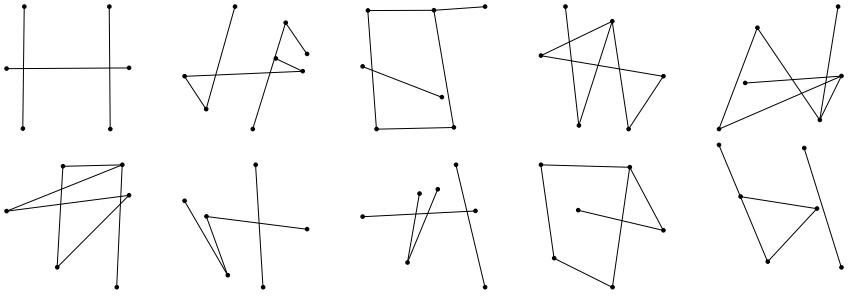
Letter A (class 1 of 15)



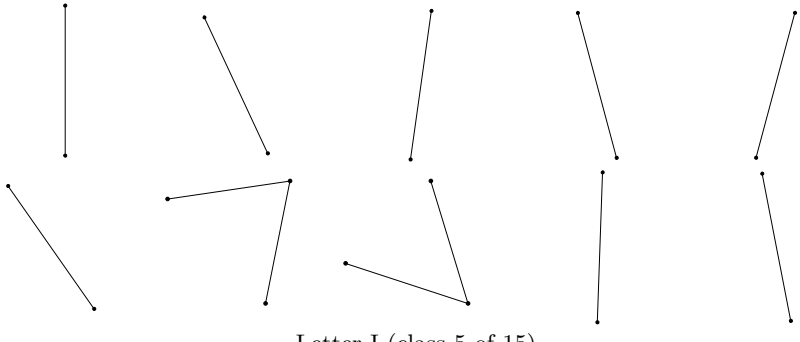
Letter E (class 2 of 15)



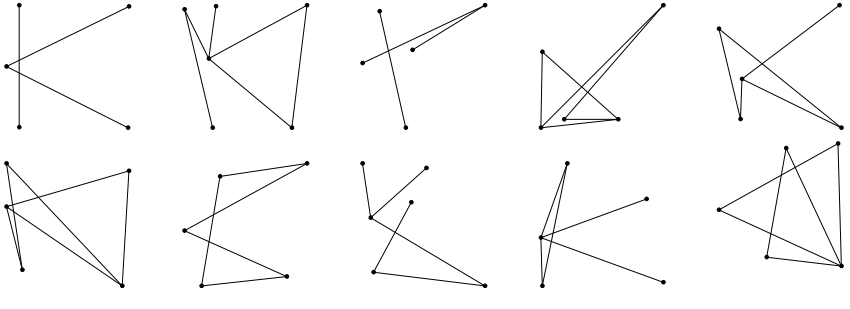
Letter F (class 3 of 15)



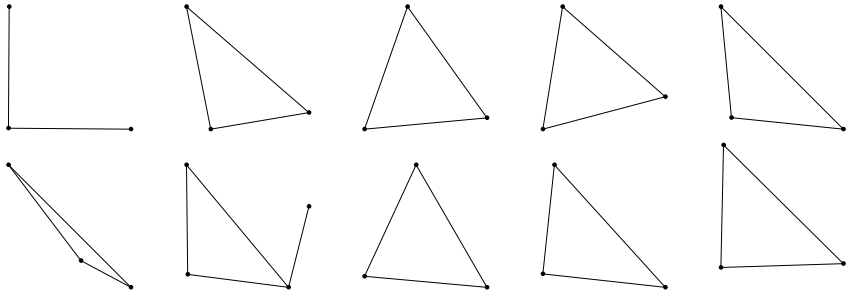
Letter H (class 4 of 15)



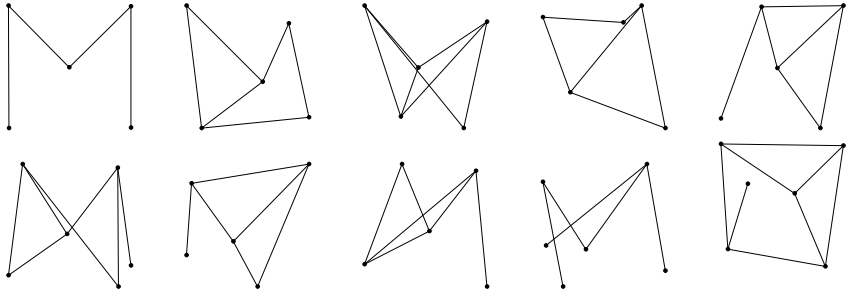
Letter I (class 5 of 15)



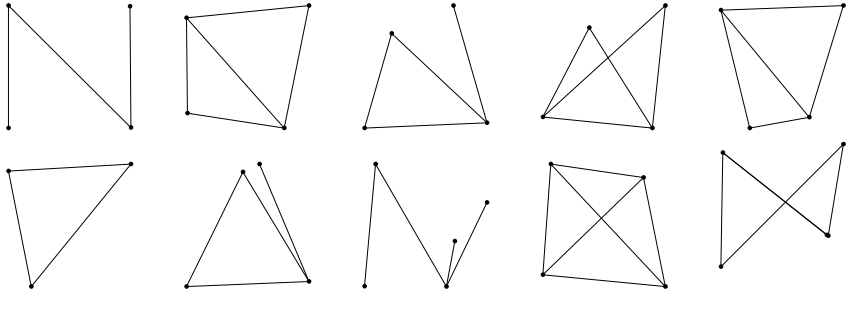
Letter K (class 6 of 15)



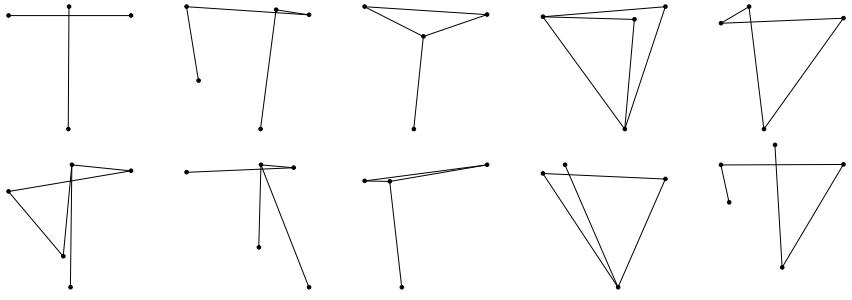
Letter L (class 7 of 15)



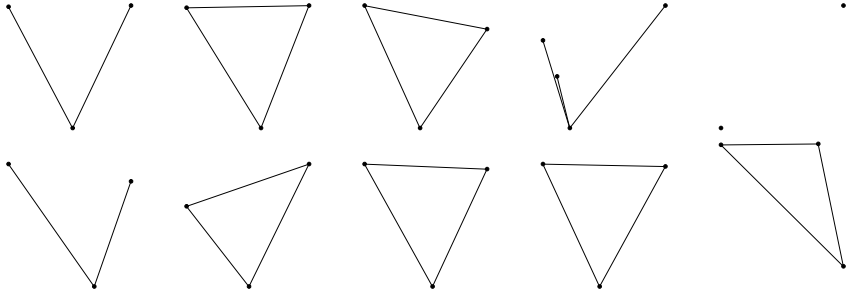
Letter M (class 8 of 15)



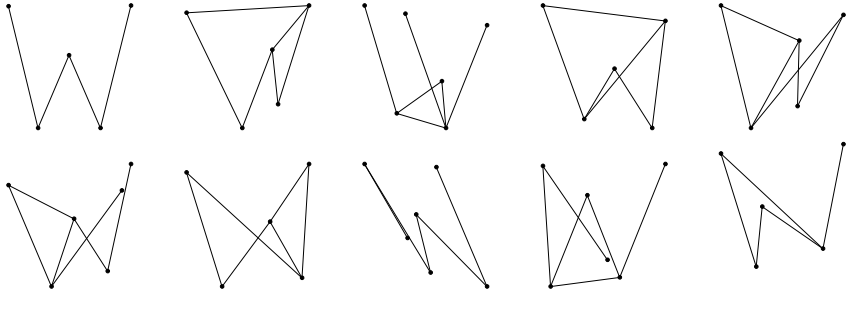
Letter N (class 9 of 15)



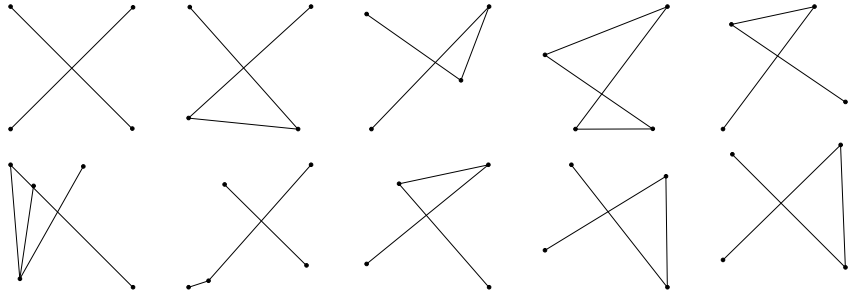
Letter T (class 10 of 15)



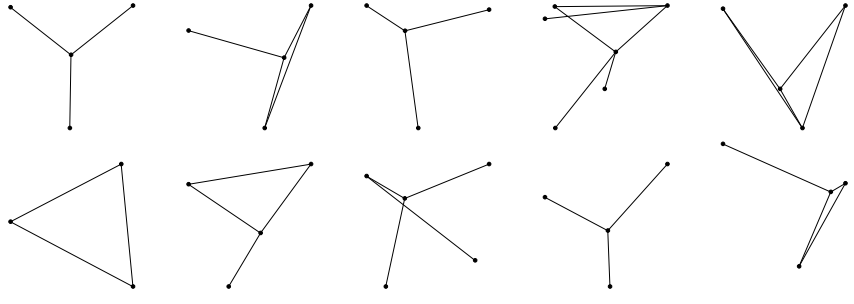
Letter V (class 11 of 15)



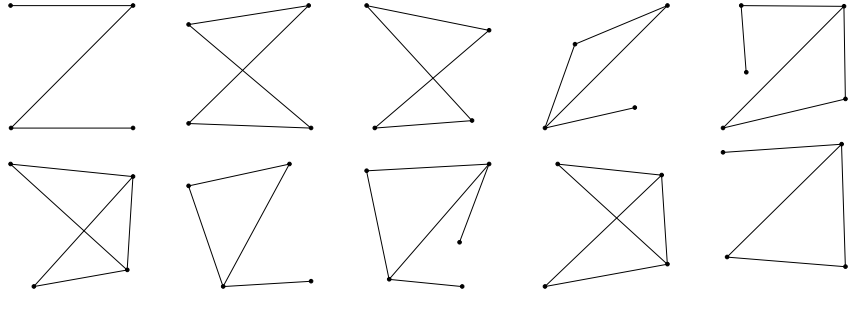
Letter W (class 12 of 15)



Letter X (class 13 of 15)

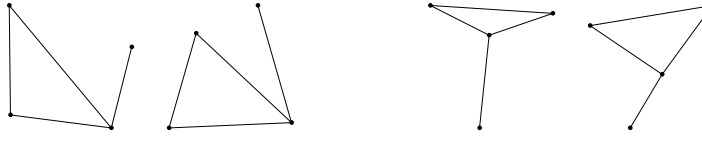


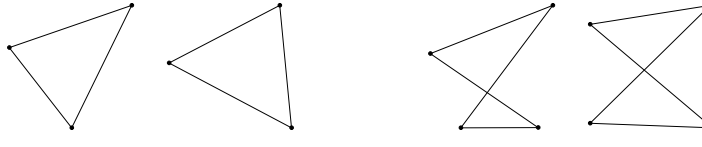
Letter Y (class 14 of 15)



Letter Z (class 15 of 15)

Even in this small sample set of 135 distorted line drawings out of 1,050 in total, it is easy to identify patterns from the same class that look structurally completely different and pairs of patterns from two different classes that look extremely similar. This indicates that the line drawing classes are in fact non-compact and overlapping, which means that a correct classification of unknown patterns will be difficult for any classifier. It should be noted that the line drawings shown above have been randomly selected from the full sample set and represent typical examples. A few example pairs of similar graphs from different classes out of the graphs illustrated above are shown below.



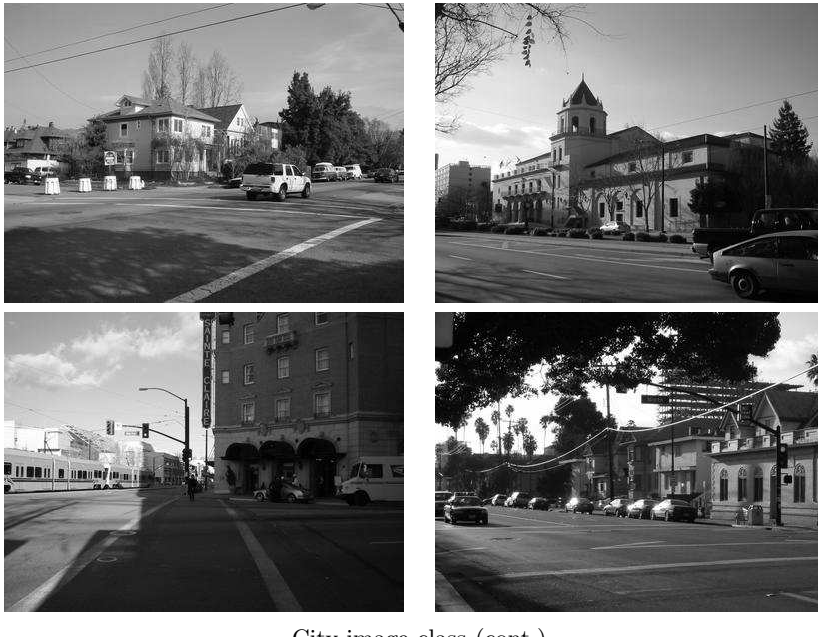




## A.2 Image Data Set

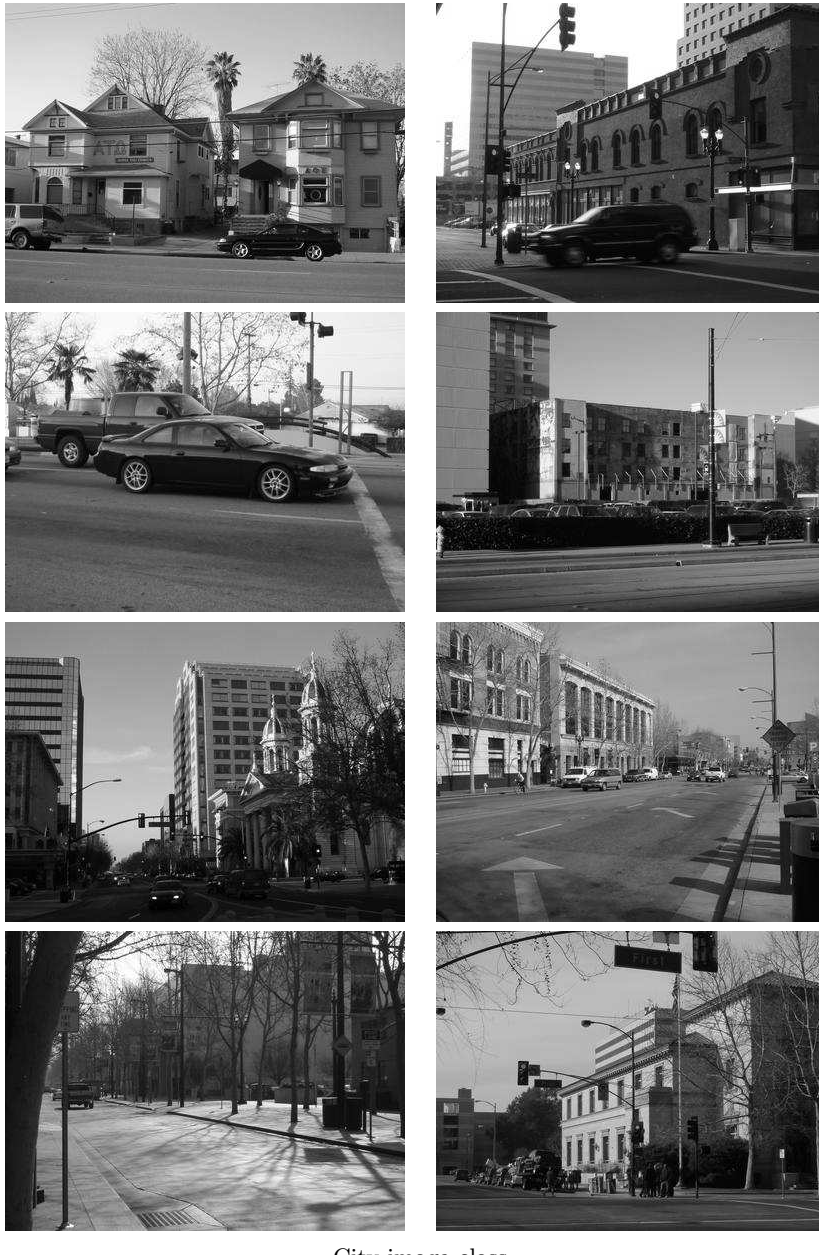
The image data set contains 162 images from the five classes *snowy, countryside, city, people*, and *streets*. For a detailed description of the graph extraction process and a discussion of data set characteristics, refer to Sec. 6.1.2. Below, we provide a few examples from the three classes *countryside, city*, and *streets*. Samples from the other two classes cannot be reproduced for copyright reasons.

Countryside image class (cont.)

Countryside image class



City image class (cont.)



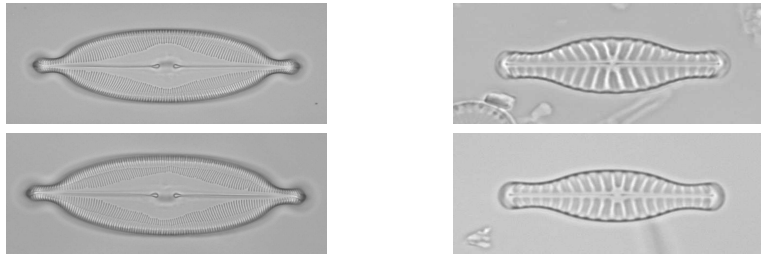
City image class

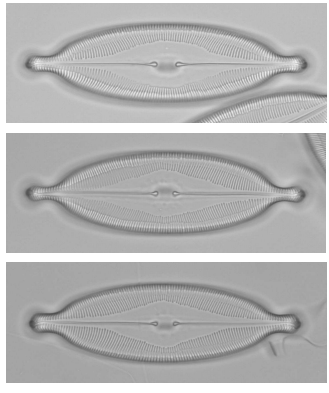
Streets image class (cont.)

Streets image class

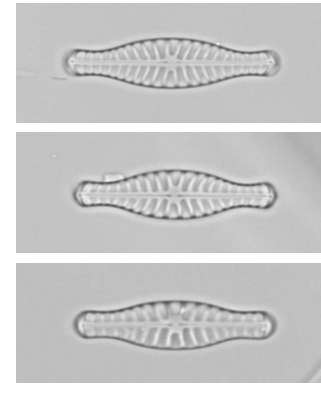
### A.3 Diatom Data Set

The diatom data set consists of 110 microscopic images of diatoms. The process of extracting region adjacency graphs from these images is described in Sec. 6.1.3. In the following, the full diatom image data set, that is, five patterns for each of the 22 classes, is illustrated.

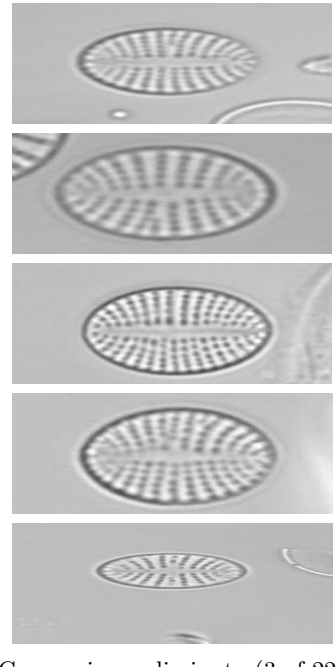




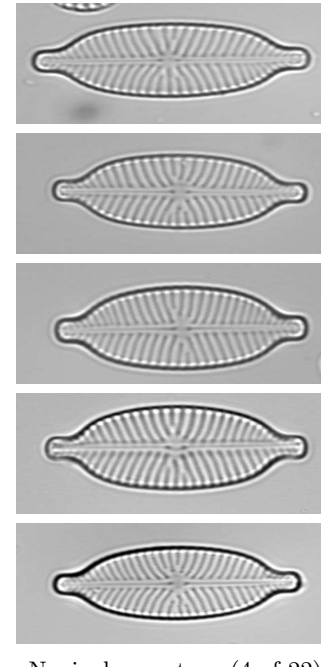
Caloneis amphisbaena (1 of 22)



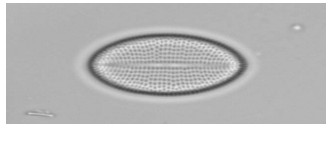
Navicula capitata (2 of 22)

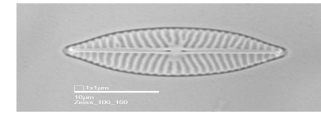


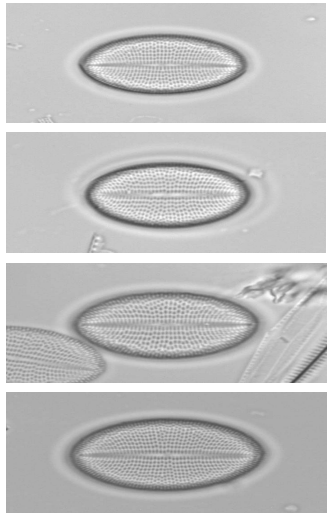
Cocconeis neodiminuta (3 of 22)



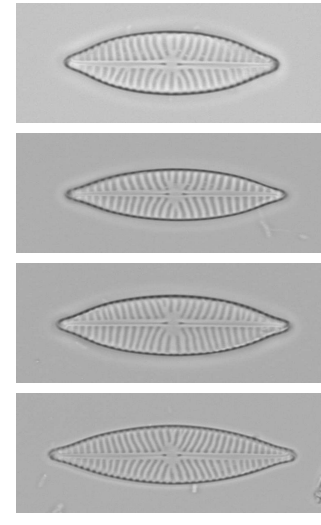
Navicula constans (4 of 22)



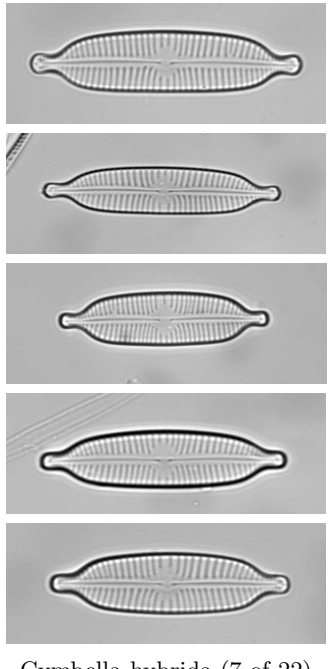




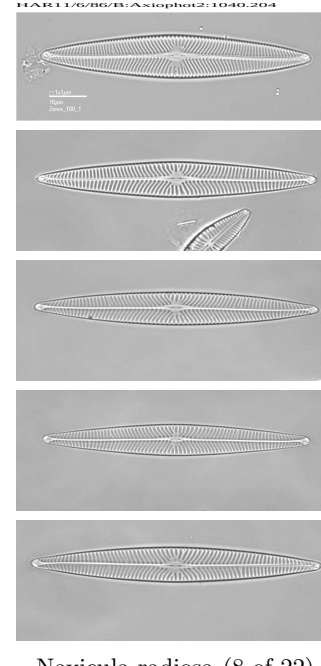
Cocconeis placentula (5 of 22)



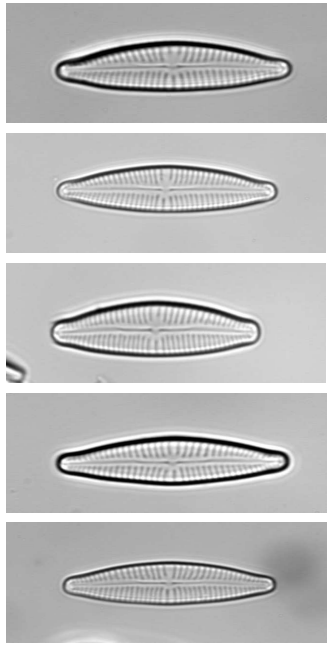
Navicula menisculus (6 of 22)



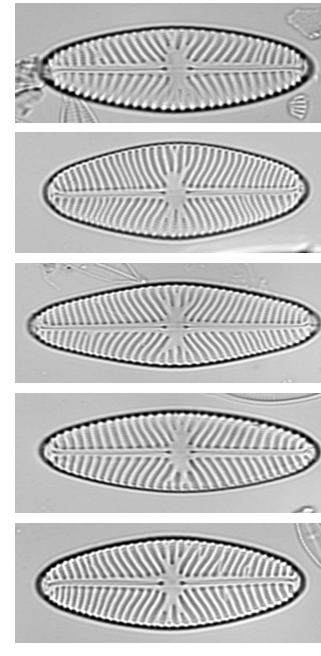
Cymbella hybrida (7 of 22)



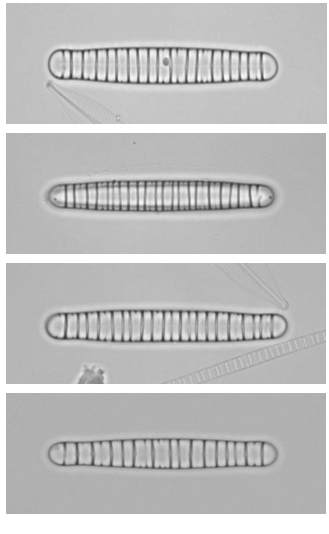
Navicula radiosa (8 of 22)

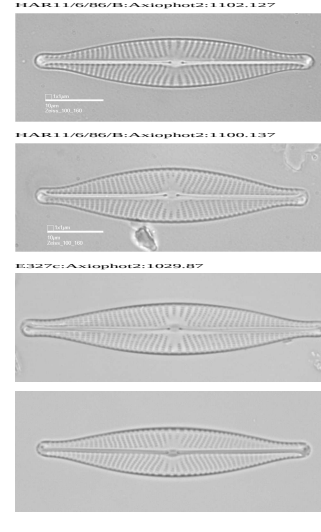


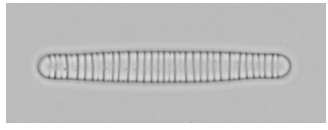
Cymbella subequalis (9 of 22)



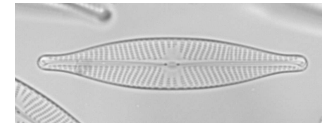
Navicula reinhardtii (10 of 22)



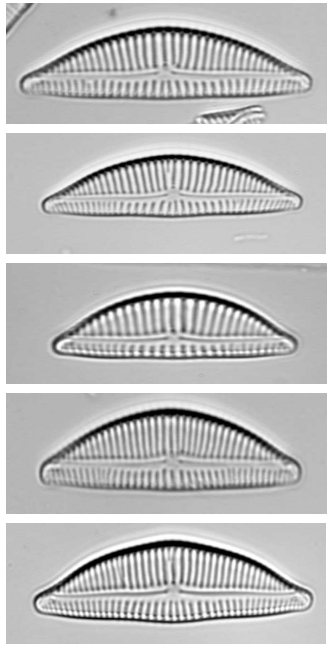




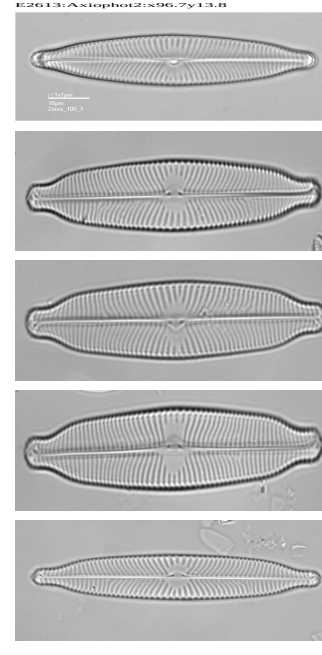
Diatoma moniliformis (11 of 22)



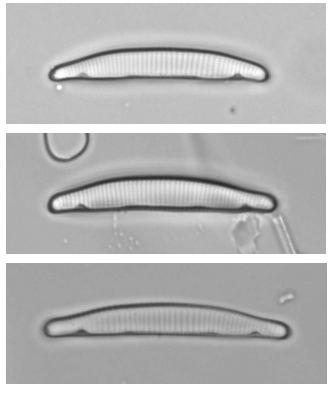
Navicula rhynchocephala (12 of 22)

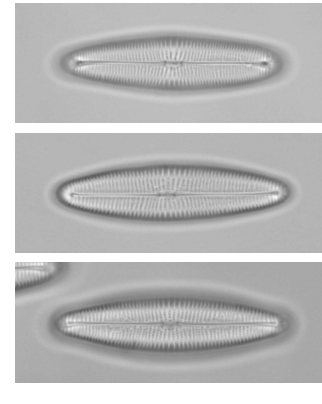


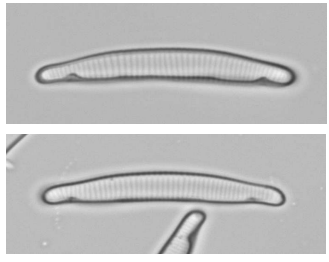
Encyonema silesiacum (13 of 22)



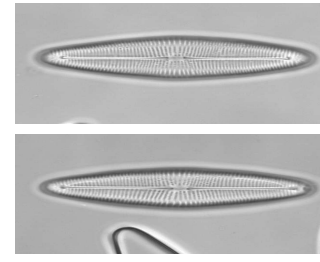
Navicula viridula (14 of 22)



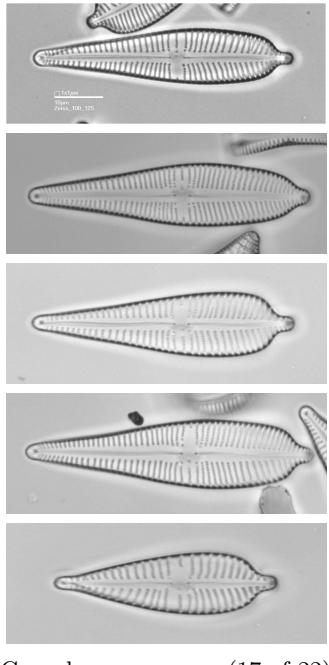




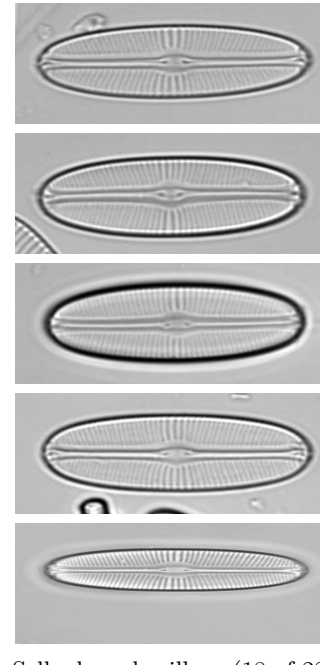
Eunotia incisa (15 of 22)



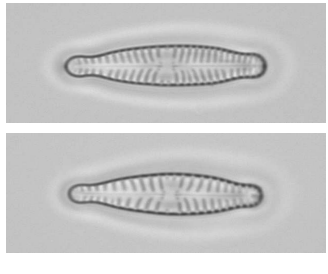
Parlibellus delognei (16 of 22)

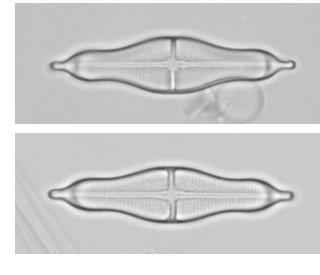


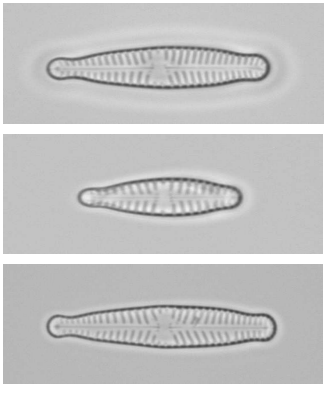
Gomphonema augur (17 of 22)



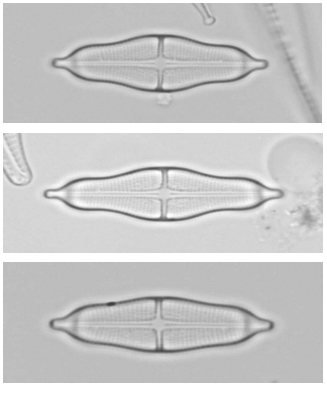
Sellaphora bacillum (18 of 22)



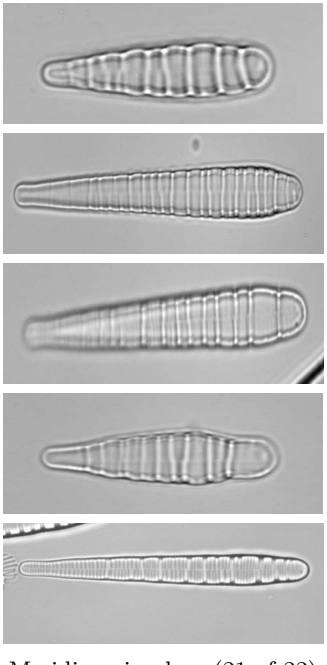




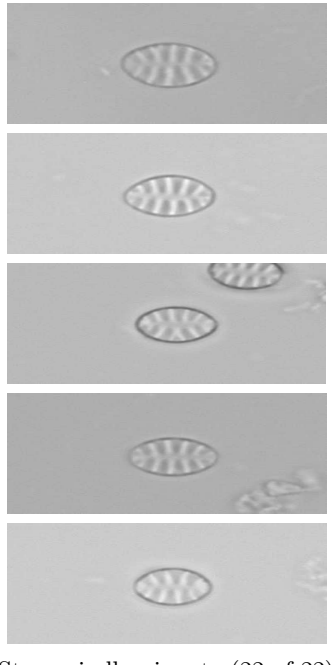
Gomphonema sp. 1 (19 of 22)



Stauroneis smithii (20 of 22)



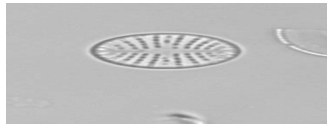
Meridion circulare (21 of 22)



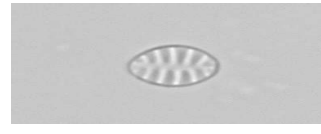
Staurosirella pinnata (22 of 22)

Compared to the total number of more than 10,000 diatom classes known to this date, our data set consisting of 22 classes appears to be rather small. Yet, even in this small subset of diatoms, it is easy to identify samples from different classes that look similar. This observation clearly shows that, in analogy to the letter data set, the correct classification of

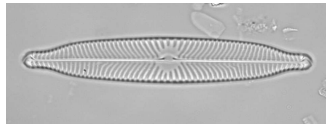
diatoms will be difficult for any classification method, on the one hand because of the presence of similar patterns from different classes, and on the other hand because of the overall small size of the data set. In the following, a few diatom samples from different classes that look similar in terms of shape and texture are illustrated.



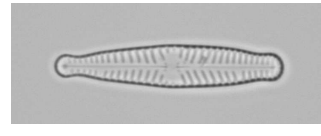
Cocconeis neodiminuta



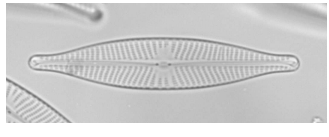
Staurosirella pinnata



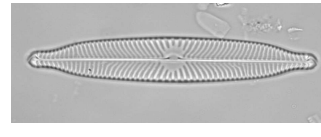
Navicula viridula



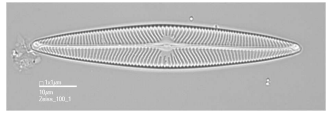
Gomphonema sp.1



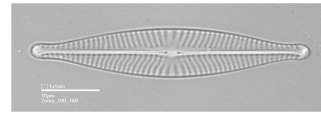
Navicula rhynchocephala



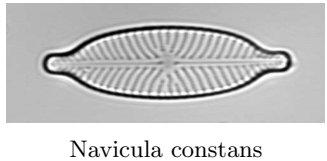
Navicula viridula



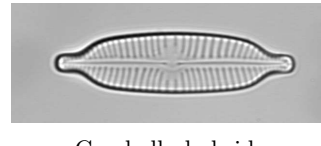
Navicula radiosa



Navicula rhynchocephala



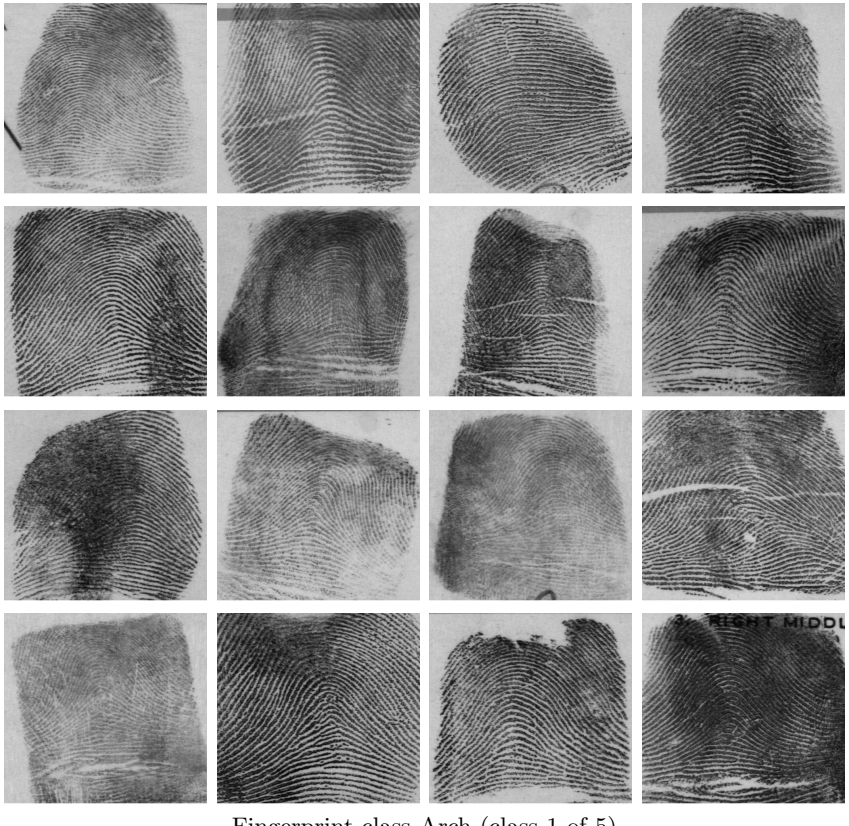
Navicula constans



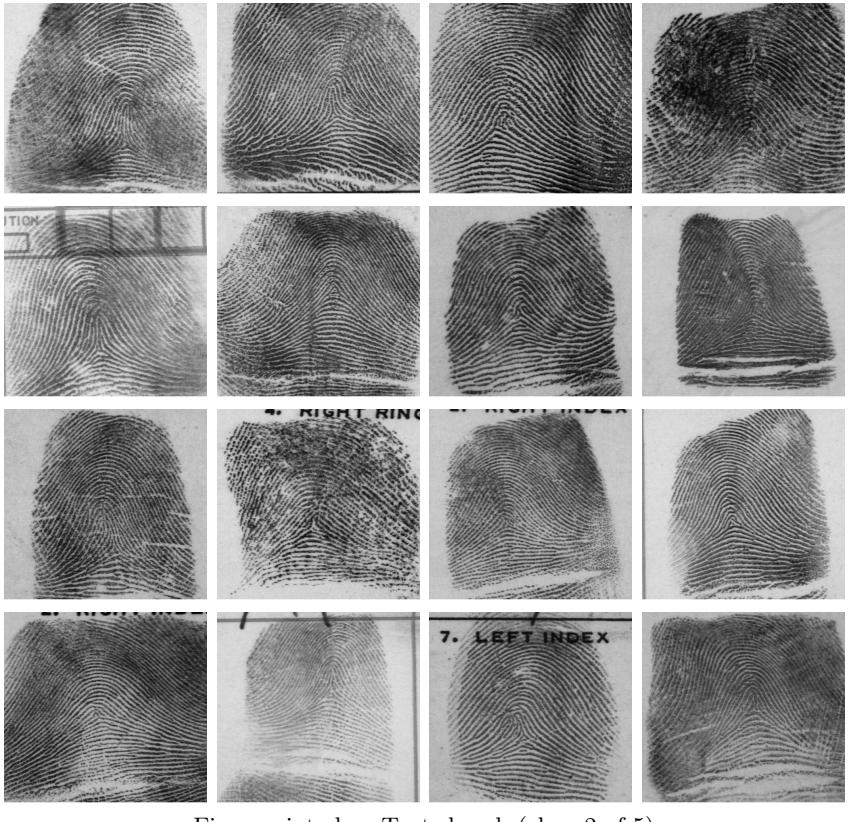
Cymbella hybrida

### A.4 Fingerprint Data Set

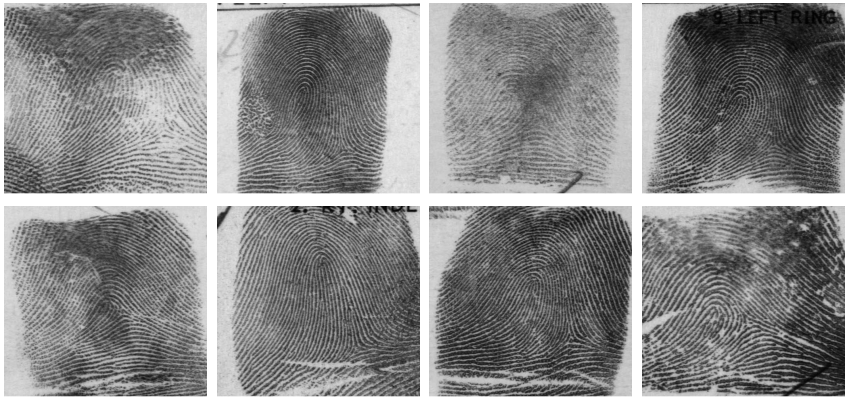
The fingerprint data set is based on the NIST special database 4 and consists of 3,300 graphs extracted from fingerprint images belonging to the classes *arch, tented arch, left loop, right loop*, and *whorl*. For a detailed description of the graph extraction procedure and the task of fingerprint classification in general, the reader is referred to Sec. 6.2. In the following, some example fingerprints from the five classes are illustrated.



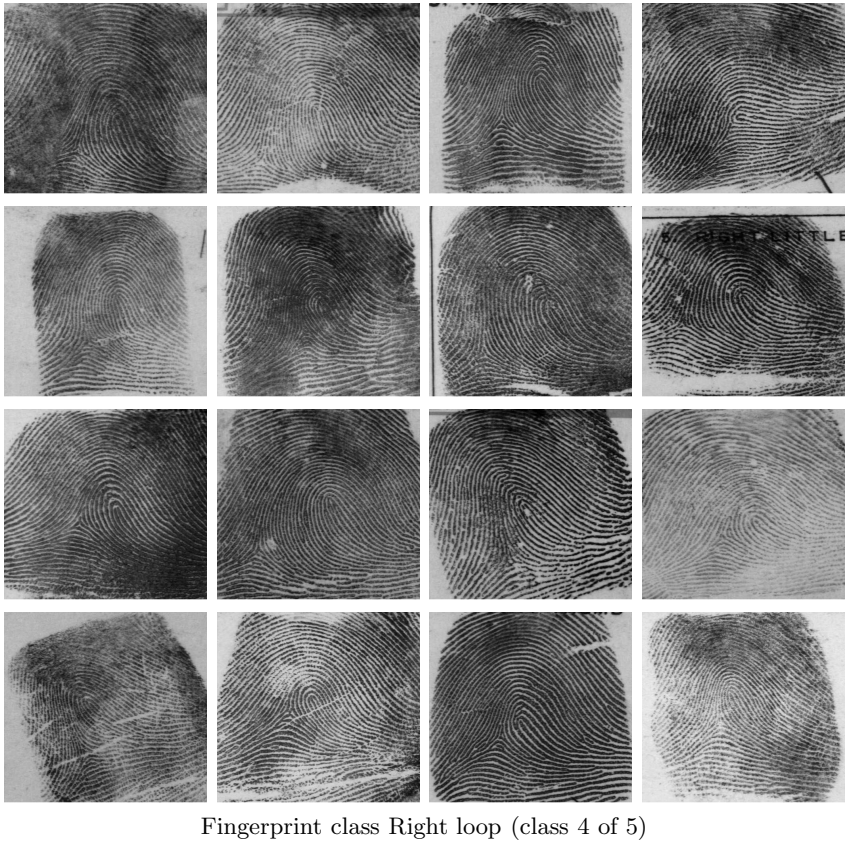
Fingerprint class Arch (class 1 of 5)



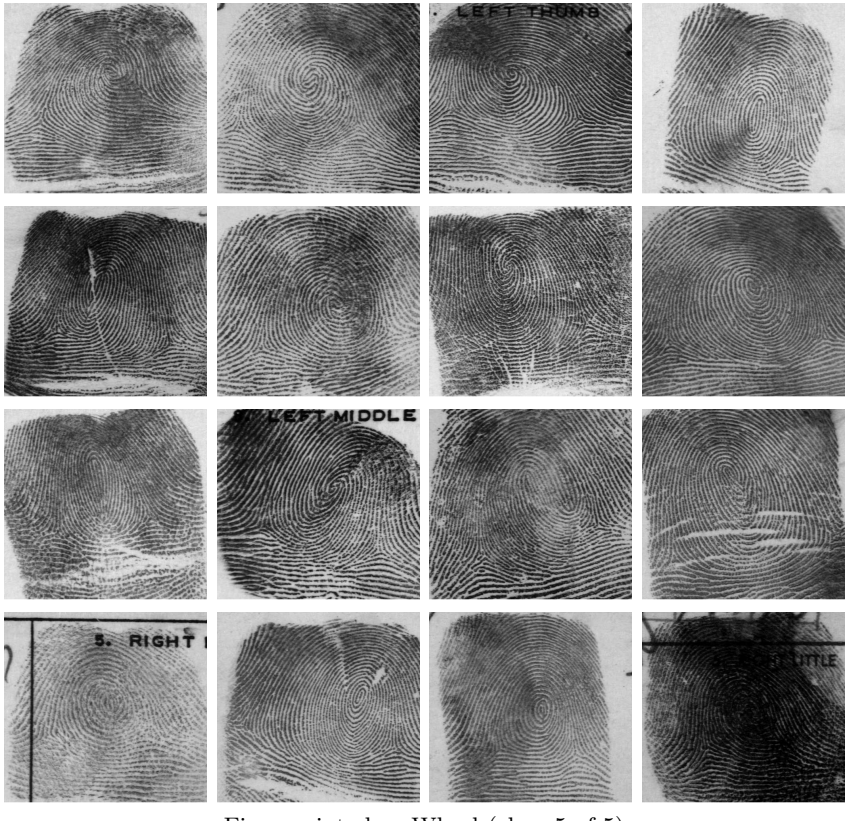

Fingerprint class Tented arch (class 2 of 5)




Fingerprint class Left loop (class 3 of 5)



Fingerprint class Right loop (class 4 of 5)



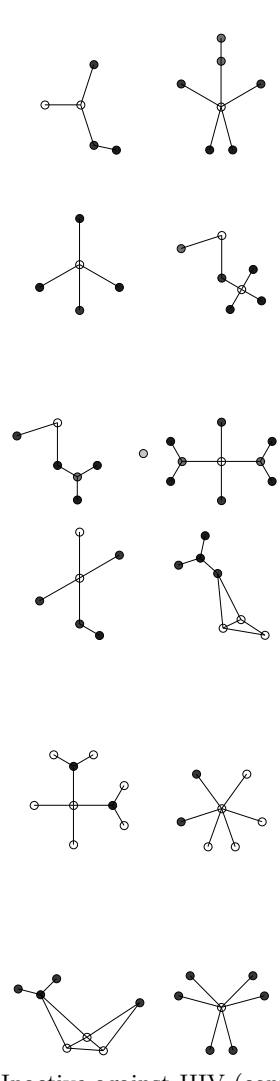

Fingerprint class Whorl (class 5 of 5)

These sample fingerprints randomly selected from the full data set make clear that it is in many cases extremely difficult to detect those ridge lines that are relevant for fingerprint classification, even so for a human observer. The major difficulties arise from the presence of distortion and noise introduced during the fingerprint capturing process and actual irregularities in fingerprints such as scars and skin injuries.

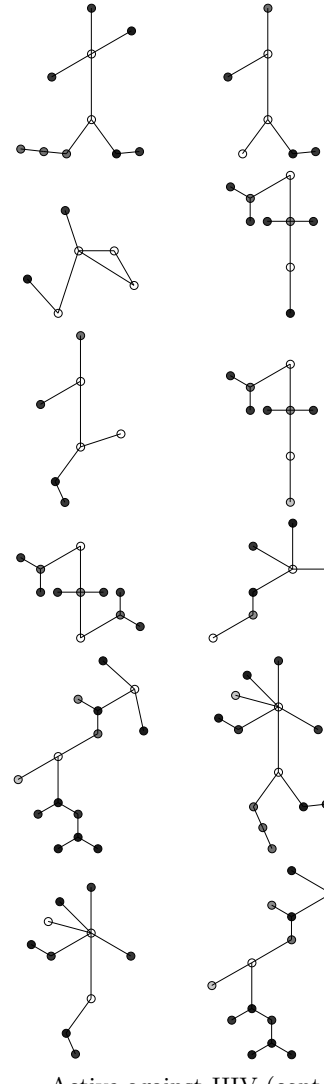
### A.5 Molecule Data Set

The molecule data set consists of 1,600 graphs that are *inactive against HIV* and 400 graphs that are *confirmed active against HIV*. A description of the molecule graph representation is given in Sec. 6.3. A few sample molecule

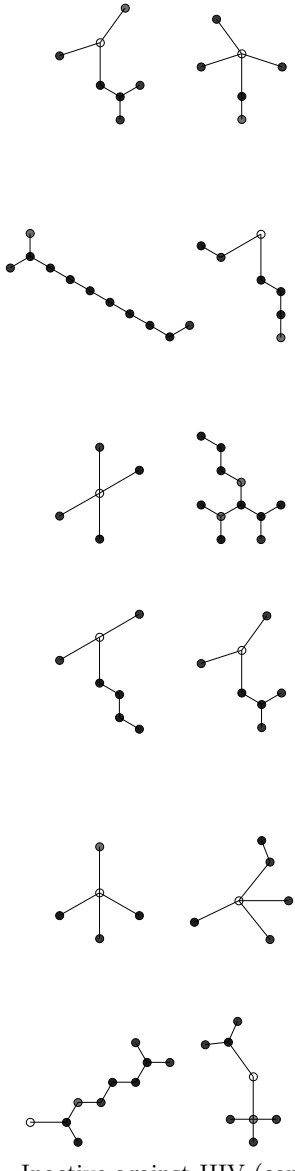
graphs after cycle elimination from both classes are illustrated below (four molecules per line first, and two molecules per line further down). Note that an automatic graph drawing application has been used by the authors of the database to assign two-dimensional coordinates to nodes; however, these coordinates are not used in the graph matching process.



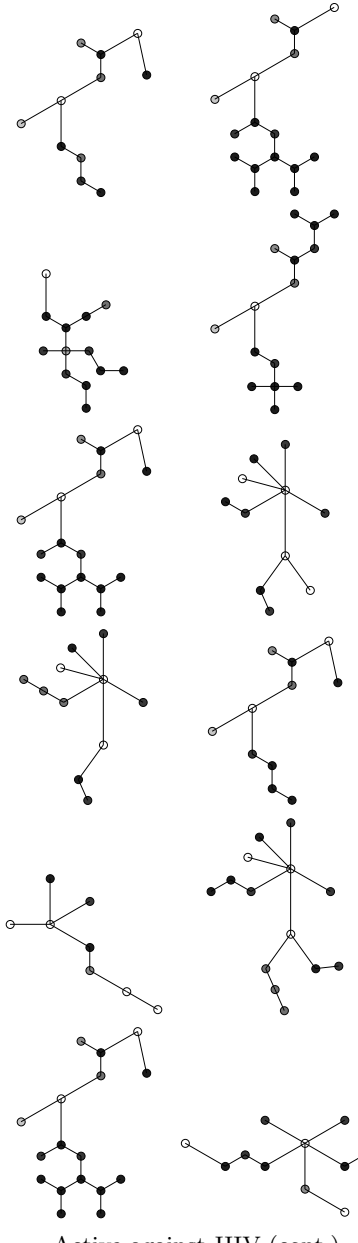
Inactive against HIV (cont.)



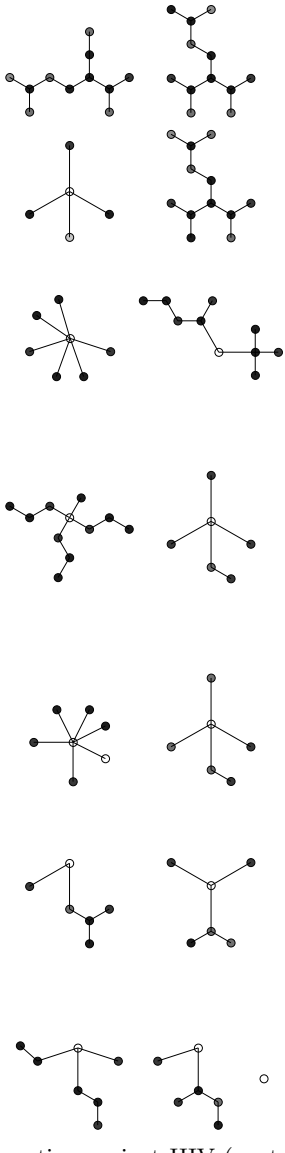
Active against HIV (cont.)



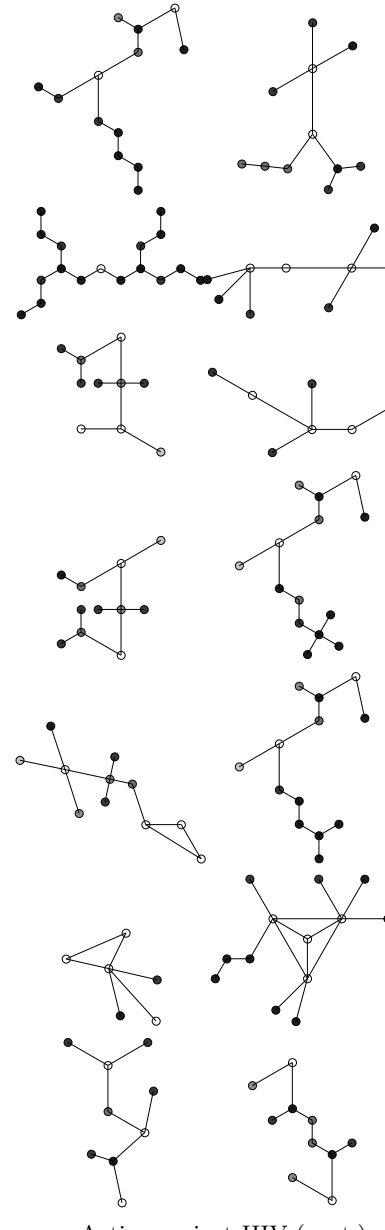
Inactive against HIV (cont.)



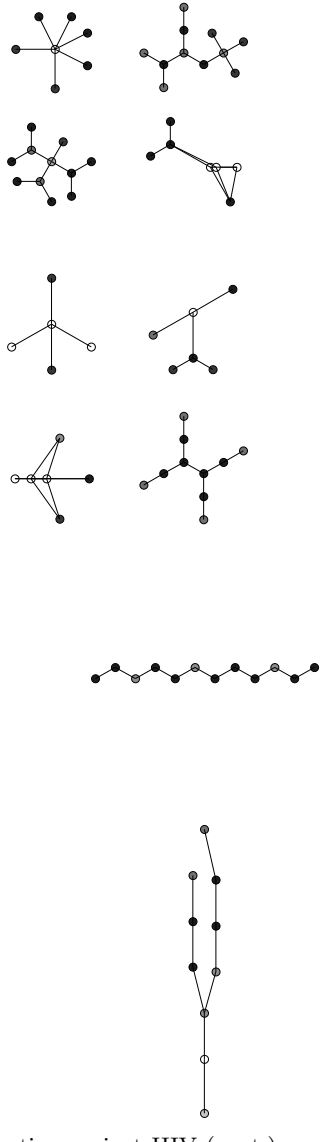
Active against HIV (cont.)



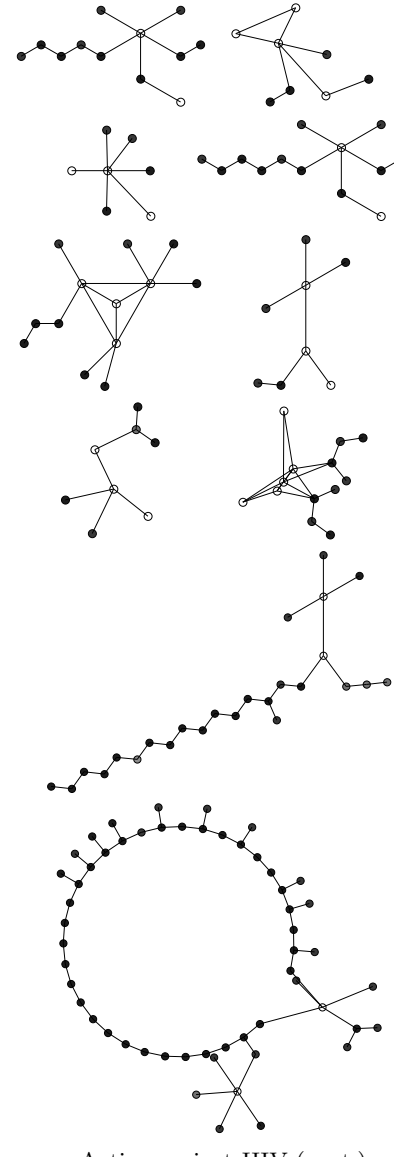
Inactive against HIV (cont.)



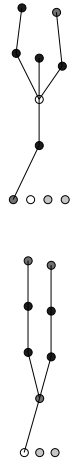
Active against HIV (cont.)



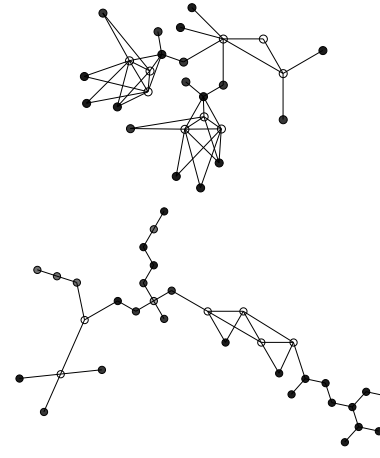
Inactive against HIV (cont.)



Active against HIV (cont.)



Inactive against HIV



Active against HIV

# Bibliography


Alpaydin, E. and Alimoglu, F. (1998). *Pen-Based Recognition of Handwritten Digits*, Dept. of Computer Engineering, Bogazici University.

Ambauen, R., Fischer, S. and Bunke, H. (2003). Graph edit distance with node splitting and merging and its application to diatom identification, in E. Hancock and M. Vento (eds.), *Proc. 4th Int. Workshop on Graph Based Representations in Pattern Recognition*, LNCS 2726 (Springer), pp. 95–106.

Andreu, G., Crespo, A. and Valiente, J. (1997). Selecting the toroidal self-organizing feature maps (TSOFM) best organized to object recognition, in *Proc. IEEE Int. Conf. on Neural Networks*, Vol. 2, pp. 1341–1346.

Bahlmann, C., Haasdonk, B. and Burkhardt, H. (2002). On-line handwriting recognition with support vector machines, in *Proc. Int. Workshop on Frontiers in Handwriting Recognition*, pp. 49–54.

Barsi, A. (2003). Neural self-organization using graphs, in *Machine Learning and Data Mining in Pattern Recognition*, LNCS 2734 (Springer), pp. 343–352.

Baudat, G. and Anouar, F. (2000). Generalized discriminant analysis using a kernel approach, *Neural Computations* **12**, 10, pp. 2385–2404.

Baxter, K. and Glasgow, J. (2000). Protein structure determination, combining inexact graph matching and deformable templates, in *Proc. Vision Interface*, pp. 179–186.

Berg, C., Christensen, J. and Ressel, P. (1984). *Harmonic Analysis on Semigroups* (Springer).

Bishop, C. (1996). *Neural Networks for Pattern Recognition* (Oxford University Press).

Borgwardt, K. and Kriegel, H.-P. (2005). Shortest-path kernels on graphs, in *Proc. 5th Int. Conf. on Data Mining*, pp. 74–81.

Borgwardt, K., Ong, C., Schönauer, S., Vishwanathan, S., Smola, A. and Kriegel, H.-P. (2005). Protein function prediction via graph kernels, *Bioinformatics* **21**, 1, pp. 47–56.

Bunke, H. and Allermann, G. (1983). Inexact graph matching for structural pattern recognition, *Pattern Recognition Letters* **1**, pp. 245–253.

Bunke, H. and Bühler, U. (1993). Applications of approximate string matching to 2D shape recognition, *Pattern Recognition* **26**, 12, pp. 1797–1812.

Bunke, H., Foggia, P., Guidobaldi, C. and Vento, M. (2003). Graph clustering using the weighted minimum common supergraph, in *Proc. 4th Int. Workshop on Graph Based Representations in Pattern Recognition*, LNCS 2726 (Springer), pp. 235–246.

Bunke, H., Jiang, X. and Kandel, A. (2000). On the minimum common supergraph of two graphs, *Computing* **65**, 1, pp. 13–25.

Bunke, H. and Shearer, K. (1998). A graph distance metric based on the maximal common subgraph, *Pattern Recognition Letters* **19**, 3, pp. 255–259.

Burges, C. and Vapnik, V. (1995). A new method for constructing artificial neural networks, Tech. Rep. N00014-94-C-0186, AT&T Bell Laboratories.

Byun, H. and Lee, S. (2003). A survey on pattern recognition applications of support vector machines, *Int. Journal of Pattern Recognition and Artificial Intelligence* **17**, 3, pp. 459–486.

Caelli, T. and Kosinov, S. (2004). Inexact graph matching using eigen-subspace projection clustering, *Int. Journal of Pattern Recognition and Artificial Intelligence* **18**, 3, pp. 329–355.

Caetano, T., Caelli, T., Schuurmans, D. and Barone, D. (2006). Graphical models and point pattern matching, *IEEE Trans. on Pattern Analysis and Machine Intelligence* **28**, 10, pp. 1646–1663.

Cappelli, R., Lumini, A., Maio, D. and Maltoni, D. (1999). Fingerprint classification by directional image partitioning, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **21**, 5, pp. 402–421.

Cesar, R., Bengoetxea, E. and Bloch, I. (2002). Inexact graph matching using stochastic optimization techniques for facial feature recognition, in *Proc. 16th Int. Conf. on Pattern Recognition*, pp. 465–468.

Chang, C. and Lin, C. (2001). *LIBSVM: A Library for Support Vector Machines*, software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

Christmas, W., Kittler, J. and Petrou, M. (1995). Structural matching in computer vision using probabilistic relaxation, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **17**, 8, pp. 749–764.

Conte, D., Foggia, P., Sansone, C. and Vento, M. (2004). Thirty years of graph matching in pattern recognition, *Int. Journal of Pattern Recognition and Artificial Intelligence* **18**, 3, pp. 265–298.

Cordella, L., Foggia, P., Sansone, C. and Vento, M. (2000). Fast graph matching for detecting CAD image components, in *Proc. 15th Int. Conf. on Pattern Recognition*, Vol. 2, pp. 1038–1041.

Cover, T. (1965). Geometrical and statistical properties of systems of linear inequalities with applications in pattern recognition, *IEEE Trans. on Electronic Computers* **14**, pp. 326–334.

Cover, T. and Hart, P. (1967). Nearest neighbor pattern classification, *IEEE Trans. on Information Theory* **13**, 1, pp. 21–27.

Cox, T. and Cox, M. (1994). *Multidimensional Scaling* (Chapman and Hall).

Cross, A., Wilson, R. and Hancock, E. (1997). Inexact graph matching using genetic search, *Pattern Recognition* **30**, 6, pp. 953–970.

DeCoste, D. and Schölkopf, B. (2002). Training invariant support vector machines, *Machine Learning* **46**, 1, pp. 161–190.

Dickinson, P., Kraetzl, M., Bunke, H., Neuhaus, M. and Dadej, A. (2004). Similarity measures for hierarchical representations of graphs with unique node labels, *Int. Journal of Pattern Recognition and Artificial Intelligence* **18**, 3, pp. 425–442.

du Buf, H. and Bayer, M. (eds.) (2002). *Automatic Diatom Identification* (World Scientific).

Duda, R., Hart, P. and Stork, D. (2000). *Pattern Classification*, 2nd edn. (Wiley-Interscience).

Ehrig, H. (1992). Introduction to graph grammars with applications to semantic networks, *Computers and Mathematics with Applications* **23**, pp. 557–572.

Eshera, M. and Fu, K. (1984). A graph distance measure for image analysis, *IEEE Transactions on Systems, Man, and Cybernetics (Part B)* **14**, 3, pp. 398–408.

Feng, J., Laumy, M. and Dhome, M. (1994). Inexact matching using neural networks, in E. Gelsema and L. Kanal (eds.), *Pattern Recognition in Practice IV: Multiple Paradigms, Comparative Studies, and Hybrid Systems* (North-Holland), pp. 177–184.

Fernandez, M.-L. and Valiente, G. (2001). A graph distance metric combining maximum common subgraph and minimum common supergraph, *Pattern Recognition Letters* **22**, 6–7, pp. 753–758.

Frasconi, P., Gori, M. and Sperduti, A. (1998). A general framework for adaptive processing of data structures, *IEEE Transactions on Neural Networks* **9**, 5, pp. 768–786.

Friedman, M. and Kandel, A. (1999). *Introduction to Pattern Recognition* (World Scientific).

Garey, M. and Johnson, D. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness* (Freeman and Co.).

Gärtner, T. (2002). Exponential and geometric kernels for graphs, in *Proc. NIPS Workshop on Unreal Data: Principles of Modeling Nonvectorial Data*.

Gärtner, T. (2003). A survey of kernels for structured data, *SIGKDD Explorations* **5**, 1, pp. 49–58.

Gärtner, T., Flach, P. and Wrobel, S. (2003). On graph kernels: Hardness results and efficient alternatives, in B. Schölkopf and M. Warmuth (eds.), *Proc. 16th Annual Conf. on Learning Theory*, pp. 129–143.

Gold, S. and Rangarajan, A. (1996). A graduated assignment algorithm for graph matching, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **18**, 4, pp. 377–388.

Gomila, C. and Meyer, F. (2001). Tracking objects by graph matching of image partition sequences, in *Proc. 3rd Int. Workshop on Graph Based Representations in Pattern Recognition*, pp. 1–11.

Gori, M., Maggini, M. and Sarti, L. (2005). Exact and approximate graph matching using random walks, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **27**, 7, pp. 1100–1111.

Günter, S. and Bunke, H. (2002). Self-organizing map for clustering in the graph domain, *Pattern Recognition Letters* **23**, 4, pp. 405–417.

Haasdonk, B. (2005). Feature space interpretation of SVMs with indefinite ker-

nels, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **27**, 4, pp. 482–492.

Hart, P., Nilsson, N. and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths, *IEEE Transactions of Systems, Science, and Cybernetics* **4**, 2, pp. 100–107.

Hartigan, J. (1975). *Clustering Algorithms* (John Wiley & Sons).

Haussler, D. (1999). Convolution kernels on discrete structures, Tech. Rep. UCSC-CRL-99-10, University of California, Santa Cruz.

Henry, E. (1900). *Classification and Uses of Finger Prints* (Routledge, London).

Hopcroft, J. and Wong, J. (1974). Linear time algorithm for isomorphism of planar graphs, in *Proc. 6th Annual ACM Symposium on Theory of Computing*, pp. 172–184.

Hotelling, H. (1933). Analysis of a complex of statistical variables into principal components, *Journal of Educational Psychology* **24**, pp. 417–441 and 498–520.

International Biometric Group (2006). Biometrics market and industry report 2006-2010, .

Jain, A., Murty, M. and Flynn, P. (1999a). Data clustering: A review, *ACM Computing Surveys* **31**, 3, pp. 264–323.

Jain, A., Prabhakar, S. and Hong, L. (1999b). A multichannel approach to fingerprint classification, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **21**, 4, pp. 348–359.

Jain, B. (2005). *Structural Neural Learning Machines*, Ph.D. thesis, Technical University Berlin.

Jain, B., Geibel, P. and Wysotzki, F. (2005). SVM learning with the Schur-Hadamard inner product for graphs, *Neurocomputing* **64**, pp. 93–105.

Jain, B. and Wysotzki, F. (2003). Automorphism partitioning with neural networks, *Neural Processing Letters* **17**, 2, pp. 205–215.

Jain, B. and Wysotzki, F. (2004). Solving inexact graph isomorphism problems using neural networks, *Neurocomputing* **63**, 1–3, pp. 169–207.

Jiang, X., Münger, A. and Bunke, H. (2001). On median graphs: Properties, algorithms, and applications, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **23**, 10, pp. 1144–1151.

Jolliffe, I. (1986). *Principal Component Analysis* (Springer).

Justice, D. and Hero, A. (2006). A binary linear programming formulation of the graph edit distance, *IEEE Trans. on Pattern Analysis and Machine Intelligence* **28**, 8, pp. 1200–1214.

Kandola, J., Shawe-Taylor, J. and Cristianini, N. (2002). Learning semantic similarity, *Neural Information Processing Systems* **15**.

Karu, K. and Jain, A. (1996). Fingerprint classification, *Pattern Recognition* **29**, 3, pp. 389–404.

Kashima, H. and Inokuchi, A. (2002). Kernels for graph classification, in *Proc. ICDM Workshop on Active Mining*, pp. 31–36.

Kashima, H., Tsuda, K. and Inokuchi, A. (2003). Marginalized kernels between labeled graphs, in *Proc. 20th Int. Conf. on Machine Learning*, pp. 321–328.

Kawagoe, M. and Tojo, A. (1984). Fingerprint pattern classification, *Pattern*

*Recognition* **17**, pp. 295–303.

Kondor, R. and Lafferty, J. (2002). Diffusion kernels on graphs and other discrete input spaces, in *Proc. 19th Int. Conf. on Machine Learning*, pp. 315–322.

Kuncheva, L. (2004). *Combining Pattern Classifiers: Methods and Algorithms* (John Wiley).

Lafferty, J. and Lebanon, G. (2003). Information diffusion kernels, in *Advances in Neural Information Processing Systems*, Vol. 15 (MIT Press), pp. 375–382.

Lafferty, J. and Lebanon, G. (2005). Diffusion kernels on statistical manifolds, *Journal of Machine Learning Research* **6**, pp. 129–163.

Le Saux, B. and Bunke, H. (2005). Feature selection for graph-based image classifiers, in *Proc. 2nd Iberian Conf. on Pattern Recognition and Image Analysis*, LNCS 3523 (Springer), pp. 147–154.

Leslie, C., Eskin, E., Cohen, A., Weston, J. and Noble, W. (2004). Mismatch string kernels for discriminative protein classification, *Bioinformatics* **20**, 4, pp. 467–476.

Leslie, C., Eskin, E. and Noble, W. (2002). The spectrum kernel: A string kernel for SVM protein classification, in *Proc. Pacific Symposium on Biocomputing* (World Scientific), pp. 564–575.

Levenshtein, V. (1966). Binary codes capable of correcting deletions, insertions and reversals, *Soviet Physics Doklady* **10**, 8, pp. 707–710.

Levi, G. (1972). A note on the derivation of maximal common subgraphs of two directed or undirected graphs, *Calcolo* **9**, pp. 341–354.

Lladós, J., Martí, E. and Villanueva, J. (2001). Symbol recognition by error-tolerant subgraph matching between region adjacency graphs, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **23**, 10, pp. 1137–1143.

Lladós, J. and Sánchez, G. (2004). Graph matching versus graph parsing in graphics recognition, *Int. Journal of Pattern Recognition and Artificial Intelligence* **18**, 3, pp. 455–475.

Lu, S., Ren, Y. and Suen, C. (1991). Hierarchical attributed graph representation and recognition of handwritten chinese characters, *Pattern Recognition* **24**, 7, pp. 617–632.

Luks, E. (1982). Isomorphism of graphs of bounded valence can be tested in ploynomial time, *Journal of Computer and Systems Sciences* **25**, pp. 42–65.

Lumini, A., Maio, D. and Maltoni, D. (1999). Inexact graph matching for fingerprint classification, *Machine Graphics and Vision, Special Issue on Graph Transformations in Pattern Generation and CAD* **8**, 2, pp. 231–248.

Lundsteen, C., Phillip, J. and Granum, E. (1980). Quantitative analysis of 6985 digitized trypsin G-banded human metaphase chromosomes, *Clinical Genetics* **18**, pp. 355–370.

Luo, B. and Hancock, E. (2001). Structural graph matching using the EM algorithm and singular value decomposition, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **23**, 10, pp. 1120–1136.

Luo, B., Wilson, R. and Hancock, E. (2003). Spectral embedding of graphs, *Pattern Recognition* **36**, 10, pp. 2213–2223.

Mahé, P., Ueda, N., Akutsu, T., Perret, J.-L. and Vert, J.-P. (2004). Extensions of marginalized graph kernels, in *Proc. 21st Int. Conf. on Machine Learning*, pp. 552–559.

Maio, D. and Maltoni, D. (1996). A structural approach to fingerprint classification, in *Proc. 13th Int. Conf. on Pattern Recognition*, pp. 578–585.

Maltoni, D., Maio, D., Jain, A. and Prabhakar, S. (2003). *Handbook of Fingerprint Recognition* (Springer).

Marcialis, G., Roli, F. and Serrau, A. (2003). Fusion of statistical and structural fingerprint classifiers, in J. Kittler and M. Nixon (eds.), *4th Int. Conf. Audio- and Video-Based Biometric Person Authentication*, LNCS 2688 (Springer), pp. 310–317.

Markowitz, H. (1952). Portfolio selection, *Journal of Finance* **7**, pp. 77–91.

Mika, S., Rätsch, G., Weston, J., Schölkopf, B. and Müller, K.-R. (1999). Fisher discriminant analysis with kernels, in *Proc. IEEE Workshop on Neural Networks for Signal Processing* (IEEE), pp. 41–48.

Milne, G. and Miller, J. (1986). The NCI drug information system. 1. system overview, *Journal of Chemical Information and Computer Sciences* **26**, 4, pp. 154–159.

Mollineda, R., Vidal, E. and Casacuberta, F. (2002). A windowed weighted approach for approximate cyclic string matching, in R. Kasturi, D. Laurendeau and C. Suen (eds.), *Proc. 16th Int. Conf. on Pattern Recognition*, pp. 188–191.

Montes-y-Gómez, M., López-López, A. and Gelbukh, A. (2000). Information retrieval with conceptual graph matching, in *Proc. 11th Int. Conf. on Database and Expert Systems Applications*, LNCS 1873 (Springer), pp. 312–321.

More, J. and Toraldo, G. (1991). On the solution of quadratic programming problems with bound constraints, *SIAM Journal on Optimization* **1**, pp. 93–113.

Moreno, P., Ho, P. and Vasconcelos, N. (2004). A Kullback-Leibler divergence based kernel for SVM classification in multimedia applications, in *Proc. Advances in Neural Information Processing Systems*, Vol. 16, pp. 1385–1392.

Müller, K., Mika, S., Rätsch, G., Tsuda, K. and Schölkopf, B. (2001). An introduction to kernel-based learning algorithms, *IEEE Transactions on Neural Networks* **12**, 2, pp. 181–202.

Munkres, J. (1957). Algorithms for the assignment and transportation problems, *Journal of the Society for Industrial and Applied Mathematics* **5**, pp. 32–38.

Neuhaus, M. and Bunke, H. (2004). A probabilistic approach to learning costs for graph edit distance, in J. Kittler, M. Petrou and M. Nixon (eds.), *Proc. 17th Int. Conference on Pattern Recognition*, Vol. 3, pp. 389–393.

Neuhaus, M. and Bunke, H. (2005). Self-organizing maps for learning the edit costs in graph matching, *IEEE Transactions on Systems, Man, and Cybernetics (Part B)* **35**, 3, pp. 503–514.

Neuhaus, M. and Bunke, H. (2007). Automatic learning of cost functions for graph edit distance, *Information Sciences* **177**, 1, pp. 239–247.

Nocedal, J. and Wright, S. (2000). *Numerical Optimization* (Springer).

Parzen, E. (1962). On the estimation of a probability density function and mode, *Annals of Mathematical Statistics* **33**, pp. 1064–1076.

Pelillo, M., Siddiqi, K. and Zucker, S. (1999). Matching hierarchical structures using association graphs, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **21**, 11, pp. 1105–1120.

Peris, G. and Marzal, A. (2002). Fast cyclic edit distance computation with weighted edit costs in classification, in R. Kasturi, D. Laurendeau and C. Suen (eds.), *Proc. 16th Int. Conf. on Pattern Recognition*, Vol. 4, pp. 184–187.

Robles-Kelly, A. and Hancock, E. (2004). String edit distance, random walks and graph matching, *Int. Journal of Pattern Recognition and Artificial Intelligence* **18**, 3, pp. 315–327.

Robles-Kelly, A. and Hancock, E. (2005). Graph edit distance from spectral seriation, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **27**, 3, pp. 365–378.

Rocha, J. and Pavlidis, T. (1994). A shape analysis model with applications to a character recognition system, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **16**, 4, pp. 393–404.

Rouvray, D. and Balaban, A. (1979). Chemical applications of graph theory, in R. Wilson and L. Beineke (eds.), *Applications of Graph Theory* (Academic Press), pp. 177–221.

Sanfeliu, A. and Fu, K. (1983). A distance measure between attributed relational graphs for pattern recognition, *IEEE Transactions on Systems, Man, and Cybernetics (Part B)* **13**, 3, pp. 353–363.

Sanfeliu, A., Serratosa, F. and Alquézar, R. (2004). Second-order random graphs for modeling sets of attributed graph and their application to object learning and recognition, *Int. Journal of Pattern Recognition and Artificial Intelligence* **18**, 3, pp. 375–396.

Saunders, C., Gammerman, A. and Vovk, V. (1998). Ridge regression learning algorithm in dual variables, in *Proc. 15th Int. Conf. on Machine Learning* (Morgan Kaufmann Publishers), pp. 515–521.

Schädler, K. and Wysotzki, F. (1999). Comparing structures using a Hopfield-style neural network, *Applied Intelligence* **11**, pp. 15–30.

Schenker, A., Bunke, H., Last, M. and Kandel, A. (2004a). Building graph-based classifier ensembles by random node selection, in F. Roli, J. Kittler and T. Windeatt (eds.), *Proc. 5th Int. Workshop on Multiple Classifier Systems*, LNCS 3077 (Springer), pp. 214–222.

Schenker, A., Last, M., Bunke, H. and Kandel, A. (2004b). Classification of web documents using graph matching, *Int. Journal of Pattern Recognition and Artificial Intelligence* **18**, 3, pp. 475–496.

Schölkopf, B. and Smola, A. (2002). *Learning with Kernels* (MIT Press).

Schölkopf, B., Smola, A. and Müller, K.-R. (1998). Nonlinear component analysis as a kernel eigenvalue problem, *Neural Computation* **10**, pp. 1299–1319.

Schölkopf, B., Smola, A. and Müller, K.-R. (1999). Kernel principal component analysis, in *Advances in Kernel Method — Support Vector Learning* (MIT

Press), pp. 327–352.

Selkow, S. (1977). The tree-to-tree editing problem, *Information Processing Letters* **6**, 6, pp. 184–186.

Serrau, A., Marcialis, G., Bunke, H. and Roli, F. (2005). An experimental comparison of fingerprint classification methods using graphs, in *Proc. 5th Int. Workshop on Graph-based Representations in Pattern Recognition*, LNCS 3434 (Springer), pp. 281–290.

Shawe-Taylor, J. and Cristianini, N. (2004). *Kernel Methods for Pattern Analysis* (Cambridge University Press).

Shimodaira, H., Noma, K., Nakai, M. and Sagayama, S. (2002). Dynamic time-alignment kernel in support vector machine, in *Proc. Advances in Neural Information Processing Systems*, Vol. 14, pp. 921–928.

Shokoufandeh, A. and Dickinson, S. (2001). A unified framework for indexing and matching hierarchical shape structures, in *Proc. 4th Int. Workshop on Visual Form*, LNCS 2059 (Springer), pp. 67–84.

Shokoufandeh, A., Macrini, D., Dickinson, S., Siddiqi, K. and Zucker, S. (2005). Indexing hierarchical structures using graph spectra, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **27**, 7, pp. 1125–1140.

Siddiqi, K., Shokoufandeh, A., Dickinson, S. and Zucker, S. (1999). Shock graphs and shape matching, *Int. Journal of Computer Vision* **30**, pp. 1–24.

Singh, M., Chatterjee, A. and Chaudhury, S. (1997). Matching structural shape descriptions using genetic algorithms, *Pattern Recognition* **30**, 9, pp. 1451–1462.

Smola, A. and Kondor, R. (2003). Kernels and regularization on graphs, in *Proc. 16th. Int. Conf. on Comptuational Learning Theory*, pp. 144–158.

Spillmann, B. (2005). *Klassifikation von Zeichenketten durch Transformation in n-dimensionale reelle Vektorräume mittels Prototyp-Selektion*, Master's thesis, University of Bern, Switzerland, in German.

Suganthan, P. (2002). Structural pattern recognition using genetic algorithms, *Pattern Recognition* **35**, 9, pp. 1883–1893.

Suganthan, P., Teoh, E. and Mital, D. (1995a). Pattern recognition by graph matching using the potts MFT neural networks, *Pattern Recognition* **28**, 7, pp. 997–1009.

Suganthan, P., Teoh, E. and Mital, D. (1995b). Pattern recognition by homomorphic graph matching using Hopfield neural networks, *Image Vision Computing* **13**, 1, pp. 45–60.

Suganthan, P. and Yan, H. (1998). Recognition of handprinted chinese characters by constrained graph matching, *Image and Vision Computing* **16**, 3, pp. 191–201.

Tsai, W. and Fu, K. (1979). Error-correcting isomorphism of attributed relational graphs for pattern analysis, *IEEE Transactions on Systems, Man, and Cybernetics (Part B)* **9**, 12, pp. 757–768.

Ullman, J. (1976). An algorithm for subgraph isomorphism, *Journal of the Association for Computing Machinery* **23**, 1, pp. 31–42.

Umeyama, S. (1988). An eigendecomposition approach to weighted graph matching problems, *IEEE Transactions on Pattern Analysis and Machine Intel-*

*ligence* **10**, 5, pp. 695–703.

van Wyk, M. and Clark, J. (2000). An algorithm for approximate least-squares attributed graph matching, *Problems in Applied Mathematics and Computational Intelligence* , pp. 67–72.

van Wyk, M., Durrani, T. and van Wyk, B. (2003). A RKHS interpolator-based graph matching algorithm, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **24**, 7, pp. 988–995.

Vapnik, V. (1982). *Estimation of Dependencies Based on Empirical Data* (Springer).

Vapnik, V. (1998). *Statistical Learning Theory* (John Wiley).

Vapnik, V. and Chervonenkis, A. (1971). On the uniform convergence of relative frequencies of events to their probabilities, *Theory of Probability and its Applications* **16**, 2, pp. 264–280.

Vert, J.-P. and Kanehisa, M. (2003). Graph-driven features extraction from microarray data using diffusion kernels and kernel CCA, in *Advances in Neural Information Processing Systems*, Vol. 15 (MIT Press), pp. 1425–1432.

Voigt, J., Bienfait, B., Wang, S. and Nicklaus, M. (2001). Comparison of the NCI open database with seven large chemical structural databases, *Journal of Chemical Information and Computer Sciences* **41**, 3, pp. 702–712.

Wagner, R. and Fischer, M. (1974). The string-to-string correction problem, *Journal of the Association for Computing Machinery* **21**, 1, pp. 168–173.

Wallis, W., Shoubridge, P., Kraetzl, M. and Ray, D. (2001). Graph distances using graph union, *Pattern Recognition Letters* **22**, 6, pp. 701–704.

Wang, I., Fan, K.-C. and Horng, J.-T. (1997). Genetic-based search for error-correcting graph isomorphism, *IEEE Transactions on Systems, Man, and Cybernetics (Part B)* **27**, 4, pp. 588–597.

Watkins, C. (1999). Kernels from matching operations, Tech. Rep. CSD-TR-98-07, Royal Holloway College.

Watkins, C. (2000). Dynamic alignment kernels, in A. Smola, P. Bartlett, B. Schölkopf and D. Schuurmans (eds.), *Advances in Large Margin Classifiers* (MIT Press), pp. 39–50.

Watson, C. and Wilson, C. (1992). *NIST special database 4, fingerprint database.*

Weislow, O., Kiser, R., Fine, D., Bader, J., Shoemaker, R. and Boyd, M. (1989). New soluble formazan assay for HIV-1 cytopathic effects: Application to high flux screening of synthetic and natural products for AIDS antiviral activity, *Journal of the National Cancer Institute* **81**, pp. 577–586.

Wilson, R. and Hancock, E. (1997). Structural matching by discrete relaxation, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **19**, 6, pp. 634–648.

Wilson, R. and Hancock, E. (2004). Levenshtein distance for graph spectral features, in *Proc. 17th Int. Conf. on Pattern Recognition*, Vol. 2, pp. 489–492.

Wong, A. and You, M. (1985). Entropy and distance of random graphs with application to structural pattern recognition, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **7**, 5, pp. 599–609.

Xu, L. and Oja, E. (1990). Improved simulated annealing: Boltzmann machine, and attributed graph matching, in L. Almeida (ed.), *Proc. Int. Workshop*

*on Neural Networks*, LNCS 412 (Springer), pp. 151–161.

Yager, N. and Amin, A. (2004). Fingerprint classification: A review, *Pattern Analysis and Applications* **7**, 1, pp. 77–93.

Yao, Y., Marcialis, G., Pontil, M., Frasconi, P. and Roli, F. (2003). Combining flat and structured representations for fingerprint classification with recursive neural networks and support vector machines, *Pattern Recognition* **36**, 2, pp. 397–406.

Zhang, X. and Ye, Y. (1998). *Computational Optimization Program Library: Convex Quadratic Programming*, University of Iowa.

Zhou, R., Quek, C. and Ng, G. (1995). A novel single-pass thinning algorithm and an effective set of performance criteria, *Pattern Recognition Letters* **16**, 12, pp. 1267–1275.

# Index