A Novel Software Toolkit for Graph Edit Distance Computation

Kaspar Riesen[1], Sandro Emmenegger[1], and Horst Bunke[2]

[1] Institute for Information Systems, University of Applied Sciences and Arts Northwestern Switzerland,
Riggenbachstrasse 16, CH-4600 Olten, Switzerland
{kaspar.riesen,sandro.emmenegger}@fhnw.ch
[2] Institute of Computer Science and Applied Mathematics, University of Bern,
Neubrückstrasse 10, CH-3012 Bern, Switzerland
bunke@iam.ch

Abstract. Graph edit distance is one of the most flexible mechanisms for error-tolerant graph matching. Its key advantage is that edit distance is applicable to unconstrained attributed graphs and can be tailored to a wide variety of applications by means of specific edit cost functions. The computational complexity of graph edit distance, however, is exponential in the number of nodes, which makes it feasible for small graphs only. In recent years the authors of the present paper introduced several powerful approximations for fast suboptimal graph edit distance computation. The contribution of the present work is a self standing software tool integrating these suboptimal graph matching algorithms. It is about being made publicly available. The idea of this software tool is that the powerful and flexible algorithmic framework for graph edit distance computation can easily be adapted to specific problem domains via a versatile graphical user interface. The aim of the present paper is twofold. First, it reviews the implemented approximation methods and second, it thoroughly describes the features and application of the novel graph matching software.

1 Introduction to Graph Edit Distance

Graph matching refers to the process of evaluating the structural similarity of graphs. A large number of methods for graph matching have been proposed in recent years [1–5]. Compared to other graph matching methods, graph edit distance is very flexible. Due to its ability to cope with arbitrary structured graphs with unconstrained label alphabets for both nodes and edges. Therefore, graph edit distance has been used in the context of classification and clustering tasks in diverse applications [6–8].

Given two graphs, the source graph g_1 and the target graph g_2. The basic idea of graph edit distance is to transform g_1 into g_2 using some distortion operations. A standard set of distortion operations is given by *insertions*, *deletions*, and *substitutions* of both nodes and edges. We denote the substitution of two nodes

W.G. Kropatsch et al. (Eds.): GbRPR 2013, LNCS 7877, pp. 142–151, 2013.

u and v by $(u \to v)$, the deletion of node u by $(u \to \varepsilon)$, and the insertion of node v by $(\varepsilon \to v)$. For edges we use a similar notation. A sequence of edit operations $e_1, \ldots, e_k$ that transform g_1 completely into g_2 is called an *edit path* between g_1 and g_2.

Obviously, for every pair of graphs (g_1, g_2), there exist an infinite number of different edit paths transforming g_1 into g_2. Let $\Upsilon(g_1, g_2)$ denote the set of all possible edit paths between two graphs g_1 and g_2. To find the most suitable edit path out of $\Upsilon(g_1, g_2)$, one introduces a cost for each edit operation, measuring the strength of the corresponding operation. The idea of such a cost function is to define whether or not an edit operation represents a strong modification of the graph. Usually, the cost is defined with respect to the underlying node or edge labels, i.e. the cost $c(e)$ is a function depending on the edit operation e.

Clearly, between two similar graphs, there should exist an inexpensive edit path, representing low cost operations, while for dissimilar graphs an edit path with high cost is needed. Consequently, the *edit distance* of two graphs is defined by the minimum cost edit path between two graphs. Formally, the graph edit distance between g_1 and g_2 is defined by

$$d(g_1, g_2) = \min_{(e_1,\ldots,e_k)\in\Upsilon(g_1,g_2)} \sum_{i=1}^{k} c(e_i)$$

The possibility to parametrize graph edit distance by means of a cost function crucially amounts for the versatility of this particular dissimilarity model. That is, by means of graph edit distance it is possible to integrate domain specific knowledge about object similarity when defining the cost of the elementary edit operations. Thus, the concept of edit distance can be tailored to specific applications.

Traditionally, the computation of edit distance is carried out by means of a tree search algorithm which explores the space of all possible mappings of the nodes and edges of g_1 to the nodes and edges of g_2. Yet, a spate of other graph edit distance computation algorithms have been developed during the last years. The present paper introduces a flexible software package for various graph edit distance computation variants (including the traditional tree search algorithm). The graph edit distance software will be made publicly available soon under

`http://www.fhnw.ch/wirtschaft/iwi/gmt`

In Fig. 1 the main window of our novel graph matching software tool is shown. In ① in Fig. 1 the user of the framework is asked to define the *source graph set* $S = \{g_1, \ldots, g_n\}$ the *target graph set* $T = \{g'_1, \ldots, g'_m\}$[1], the folder where the individual graphs are locally stored (*graph folder*) and a *results folder* where the computed distance matrix $\mathbf{D} = (d(g_i, g_j))_{n\times m}$ is saved ($g_i \in S$ and $g_j \in T$). For more detailed and technical descriptions of the input formats of both *graph sets* and *graphs* as well as the output format of the distance matrix we refer to the above mentioned website. In ② in Fig. 1 the user defines whether or not to log meta information about the graphs being processed and the corresponding edit paths.

[1] Clearly, the source and target set might be the same sets in some applications.

The remainder of the present paper reviews different versions of graph edit distance available in our software tool and describes the user-defined parameters and options for graph edit distance computation in detail.

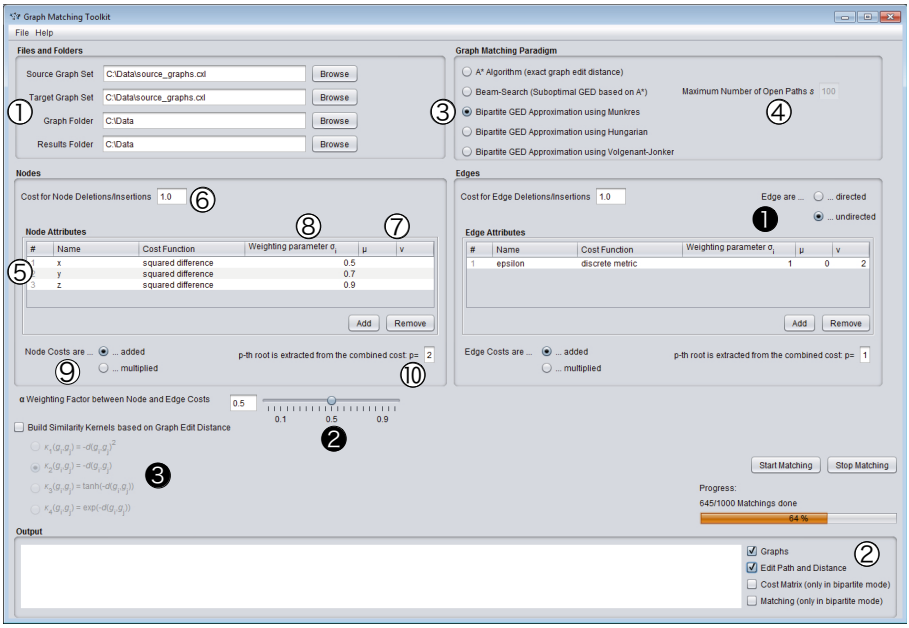


Fig. 1. The main window of our novel graph matching software tool

2 Graph Edit Distance Algorithms

In ③ in Fig. 1 the graph edit distance algorithm actually to be applied can be chosen by the user. The five available algorithms are briefly described in the next paragraphs.

A-Algorithm with Bipartite Heuristic (exact graph edit distance).* A widely used method for exact graph edit distance is based on the A* algorithm [9] which is a best-first search algorithm. The basic idea is to organize the underlying search space as an ordered tree. The root node of the search tree represents the starting point of our search procedure, inner nodes of the search tree correspond to partial solutions, and leaf nodes represent complete – not necessarily optimal – solutions. Such a search tree is constructed dynamically at runtime by iteratively creating successor nodes linked by edges to the currently considered node in the search tree. In order to determine the most promising node in the current search tree often a heuristic function is used. Formally, for a node p in the search tree, we use $g(p)$ to denote the cost of the optimal path from the root node to the current

node p, and we use $h(p)$ for denoting the estimated cost from p to a leaf node. The sum $g(p) + h(p)$ gives the total cost assigned to an open node in the search tree. Given that the estimation of the future cost $h(p)$ is lower than, or equal to, the real cost, it is guaranteed that the algorithm finds an optimal edit path [9]. To solve the problem of estimating a lower bound $h(p)$ for the future costs, one can map the unprocessed nodes and edges of graph g_1 to the unprocessed nodes and edges of graph g_2 such that the resulting costs are minimal. In [10] it is proposed to use a fast bipartite assignment algorithm of the unprocessed nodes and edges of the two graphs as heuristic function $h(p)$. This specific heuristic function $h(p)$ described in [10] is actually implemented in our software framework.

Beam Search. The method described in the previous paragraph finds an optimal edit path between two graphs g_1 and g_2 and thus returns the exact graph edit distance $d(g_1, g_2)$. Unfortunately, the computational complexity of any exact graph edit distance algorithm is exponential in the number of nodes of the involved graphs (whether or not a heuristic function $h(p)$ is deployed to govern the tree traversal process). This means that the running time and space complexity may be huge even for reasonably small graphs[2]. In [11] the issue of efficient graph edit distance computation is addressed by simple variants of a standard A* algorithm. One method presented in [11] is based on the idea of *beam search*. Instead of expanding all successor nodes in the search tree, only a fixed number s of nodes to be processed are kept in the set of open nodes at all times. Whenever a new partial edit path is added, only the s partial edit paths p with the lowest costs $g(p) + h(p)$ are kept, and the remaining partial edit paths are removed. This means that not the full search space is explored, but only those nodes are expanded that belong to the most promising partial matches.

For similar graphs, it is clear that edit operations of an optimal path have low costs. Therefore if only the partial edit paths with lowest costs are considered, we will obtain an edit path that is nearly optimal, which will result in a suboptimal distance close to the exact distance. For dissimilar graphs, the suboptimal distance will remain large. Note that this method requires the user to define the maximum number of open paths s (cp. ④ in Fig. 1). The parameter s controls both the degree of suboptimality and the computation time of the procedure. That is, increasing the parameter s simultaneously augments the probability of finding the true graph edit distance and the running time.

Bipartite Graph Edit Distance using Assignment Algorithms. In [12–14] the authors of the present paper introduced a novel algorithmic framework which allows us to approximately compute edit distance in a substantially faster way than traditional methods. The proposed algorithms consider only local, rather than global, edge structure during the optimization process. The method is based on an (optimal) fast bipartite assignment procedure mapping nodes and their local structure of one graph to nodes and their local structure of another graph.

[2] In practice we are able to compute the edit distance of graphs typically containing 12 nodes at most.

In [12] the algorithmic framework first substitutes all nodes of the smaller graph and the nodes remaining in the larger graph are either deleted (if they belong to g_1) or inserted (if they belong to g_2). In [13] this idea is extended by allowing insertions or deletions to occur not only in the larger, but also in the smaller of the two graphs under consideration. To this end, for two graphs g_1 and g_2 to be matched with nodes $V_1 = \{u_1, \ldots, u_n\}$ and $V_2 = \{v_1, \ldots, v_m\}$, respectively, a cost matrix $\mathbf{C}$ is defined as follows:

$$\mathbf{C} = \left[\begin{array}{cccc|cccc} c_{11} & c_{12} & \cdots & c_{1m} & c_{1\varepsilon} & \infty & \cdots & \infty \\ c_{21} & c_{22} & \cdots & c_{2m} & \infty & c_{2\varepsilon} & \ddots & \vdots \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \ddots & \infty \\ c_{n1} & c_{n2} & \cdots & c_{nm} & \infty & \cdots & \infty & c_{n\varepsilon} \\ \hline c_{\varepsilon 1} & \infty & \cdots & \infty & 0 & 0 & \cdots & 0 \\ \infty & c_{\varepsilon 2} & \ddots & \vdots & 0 & 0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \infty & \vdots & \ddots & \ddots & 0 \\ \infty & \cdots & \infty & c_{\varepsilon m} & 0 & \cdots & 0 & 0 \end{array}\right]$$

where c_{ij} denotes the cost of a node substitution $c(u_i \rightarrow v_j)$, $c_{i\varepsilon}$ denotes the cost of a node deletion $c(u_i \rightarrow \varepsilon)$, and $c_{\varepsilon j}$ denotes the cost of a node insertion $c(\varepsilon \rightarrow v_j)$.

Obviously, the left upper corner of the cost matrix represents the costs of all possible node substitutions, the diagonal of the right upper corner the costs of all possible node deletions, and the diagonal of the bottom left corner the costs of all possible node insertions. Note that each node can be deleted or inserted at most once. Therefore any non-diagonal element of the right-upper and left-lower part is set to ∞. The bottom right corner of the cost matrix is set to zero since substitutions of the form $(\varepsilon \rightarrow \varepsilon)$ should not cause any costs.

In the definition of cost matrix $\mathbf{C}$, to each entry c_{ij}, i.e. to each cost of a node substitution $c(u_i \rightarrow v_j)$, the minimum sum of edge edit operation costs, implied by node substitution $u_i \rightarrow v_j$, is added. That is, using a bipartite optimization procedure the cost of an optimal assignment of the adjacent edges of u_i and v_j is computed and added to entry c_{ij}. Clearly, to entry $c_{i\varepsilon}$ the cost of the deletion of all adjacent edges of u_i is added, and to the entry $c_{\varepsilon j}$ the cost of all insertions of the adjacent edges of v_j is added. Note that in ② in Fig. 1 one can define whether or not to log the cost matrix $\mathbf{C}$ in the output window.

On the basis of the quadratic cost matrix $\mathbf{C}$ any bipartite assignment algorithm can be executed. The result returned by bipartite optimization procedures applied to $\mathbf{C}$ corresponds to the minimum cost mapping of the nodes and their local edge structure of g_1 to the nodes and their local edge structure of g_2. In ② in Fig. 1 one can choose to log the optimal mapping of local structures found on matrix $\mathbf{C}$ in the output window. Given the optimal mapping between local structures, the edit operations on nodes and the implied edit operations of the edges can be inferred, and the accumulated costs of the individual edit operations on both nodes and edges can be computed. Note that assignment algorithms are not able to consider the global edge structure during the matching process. Hence, optimal matchings of nodes (considering the local edge structure) do not

necessarily lead to an optimal (i.e. minimum cost) edit path. That is, this procedure leads to a suboptimal graph edit distance, which is equal to or greater than the exact edit distance.

In [12, 13] we make use of Munkres' algorithm [15] as basic bipartite optimization procedure. In [14] not only *Munkres' Algorithm* but also a modern version of the *Hungarian Algorithm* as well as the algorithm of *Volgenant and Jonker* [16] are incorporated to solve the assignment problem. In our software package all three assignment algorithms are implemented.

3 Defining the Cost Function

The definition of adequate and application-specific cost functions is a key task in edit distance based graph matching. The definition of the cost is usually depending on the underlying label alphabets for nodes and edges. In our algorithmic framework, the labels for both nodes and edges can be given by the set of integers $L = \{1, 2, 3, \ldots\}$, real numbers $L = \mathbb{R}$, a set of symbolic labels $L = \{\alpha, \beta, \gamma, \ldots\}$, strings defined over an alphabet V $L = \{V^*\}$, or an arbitrary combination of different labels. Unlabeled graphs are obtained as a special case by assigning the same symbolic label λ to all nodes and edges. In our software tool a single node can be labeled with up to five different node attributes which can be arbitrarily named by the user (cp. ⑤ in Fig. 1). In Fig. 2 (a), for instance, the nodes are labeled with three attributes x, y and z (the nodes represent points in a three-dimensional space $\mathbb{R}^3$). In Fig. 2 (b) the nodes are labeled with two attributes, viz. a symbolic attribute named *type* and a string named *sequence*. Note that the label alphabets are implicitly defined by the distance function to be applied on them (see next paragraph).

The first step in cost definition is to define a non-negative parameter representing the cost of a deletion or insertion $c(u \to \varepsilon)$ or $c(\varepsilon \to u)$, respectively, of an arbitrary node u (cp. ⑥ in Fig. 1). For the sake of symmetry, an identical cost for deletions and insertions has to be defined. Second, for each attribute a distance function for node substitutions has to be chosen by the user. Typically, the cost of a node substitution $(u \to v)$ is measured by means of some distance function $d : L \times L \to \mathbb{R}$ defined on the node label alphabet L. For now we assume that the nodes are labeled with a single attribute from alphabet L. The attribute values of u and v are $u.A \in L$ and $v.A \in L$, respectively. In our software tool four different distance functions can be defined on each node attribute. Note that the first two distance functions are applicable to numerical attributes only. The last distance function is applicable to string attributes:

1. *absolute value of difference*: $d(u.A, v.A) = |u.A - v.A|$
2. *squared difference*: $d(u.A, v.A) = (u.A - v.A)^2$
3. *discrete metric*: $d(u.A, v.A) = \begin{cases} \mu, & \text{if } u.A = v.A \\ \nu, & \text{else} \end{cases}$

   where μ, ν are non-negative real values ($\mu, \nu \in \mathbb{R}^+$) to be defined by the user (cp. ⑦ in Fig. 1).

4. *Levenshtein distance*: $d(u.A, v.A)$ = minimal number of single-character edit operations (deletions, insertions, substitutions) required to change string $u.A$ into string $v.A$, also known as *string edit distance* (*sed*).

Assuming that the nodes are labeled with $k > 1$ attributes, the i-th attribute values of u and v are $u.A_i \in L_i$ and $v.A_i \in L_i$, respectively. For each attribute an individual distance function $d_i : L_i \times L_i \to \mathbb{R}$ has been defined by the user. The weighting parameter $\sigma_i \in]0, 1]$ (cp. ⑧ in Fig. 1) can be defined in order to scale the relative importance of an attribute distance value by means of

$$\sigma_i \cdot d_i(u.A_i, v.A_i)$$

In our framework there are two ways of combining the k individual weighted distance values $(\sigma_i \cdot d_i(u.A_i, v.A_i))_{1 \leq i \leq k}$, viz. by building the sum or the product (cp. ⑨ in Fig. 1). Finally, in ⑩ in Fig. 1 the user is asked to define a parameter p indicating that the p-th root is extracted from the combined node cost. Depending on whether the individual node costs are added or multiplied, we thus get the cost $c(u \to v)$ for node substitution $(u \to v)$ as

$$\left(\sum_{i=1}^{k} \sigma_i \cdot d_i(u.A_i, v.A_i)\right)^{1/p} \quad \text{or} \quad \left(\prod_{i=1}^{k} \sigma_i \cdot d_i(u.A_i, v.A_i)\right)^{1/p}$$

In Fig. 2 two examples of node cost functions are shown. In Fig. 2 (a) the cost for a node substitution is given as a weighted Euclidean distance between the nodes:

$$c(u \to v) = \sqrt{0.5 \cdot (u.x - v.x)^2 + 0.7 \cdot (u.y - v.y)^2 + 0.9 \cdot (u.z - v.z)^2}$$

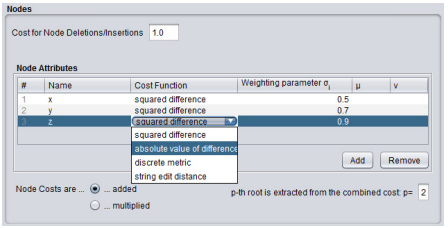
In Fig. 2 (b) the cost for a node substitution is defined by:

$$c(u \to v) = \begin{cases} 0.5 \cdot sed(u.sequence, v.sequence), & \text{if } u.type = v.type \\ sed(u.sequence, v.sequence), & \text{if } u.type \neq v.type \end{cases}$$

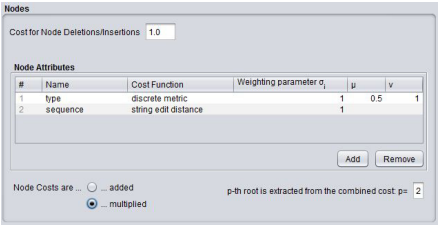
The edge attributes and their distance function can be defined analogously. Additionally, for edges the user has to define whether the edges are directed or undirected (cp. ❶ in Fig. 1). The weighting parameter $\alpha \in [0, 1]$ (cp. ❷ in Fig. 1) controls whether the edit operation cost on the nodes or on the edges is more important. That is, each cost of every node operation (deletion, insertion, substitution) is multiplied by α. In the case of edge operations the costs are multiplied by $(1 - \alpha)$. The default setting is $\alpha = 0.5$ leading to balanced importance between node and edge operation cost.

4 Similarity Kernel from Edit Distance

Kernel machines constitute a very powerful class of algorithms [17, 18]. As a matter of fact, kernel methods have become a rapidly emerging sub-field in intelligent information processing. As any kernel function can be regarded as a



(a)



(b)

Fig. 2. Two different parameter settings for defining the cost functions on node attributes

object similarity measure, the edit distance of graphs can also be interpreted as a pattern similarity measure in the context of kernel machines, which makes a large number of powerful methods applicable to graphs [19], including support vector machines for classification and kernel principal component analysis for feature space transformation and dimensionality reduction. In our algorithmic framework we provide four different transformations of graph edit distance $d(g_1, g_2)$ to a similarity measure $\kappa_i(g_1, g_2)$ (cp. ❸ in Fig. 1):

- $\kappa_1(g_1, g_2) = -d(g_1, g_2)^2$
- $\kappa_2(g_1, g_2) = -d(g_1, g_2)$
- $\kappa_3(g_1, g_2) = \tanh(-d(g_1, g_2))$
- $\kappa_4(g_1, g_2) = \exp(-d(g_1, g_2))$

Note that these similarity kernels are not positive definite and are therefore not valid kernels in the strict sense. Yet, there is theoretical and practical evidence that using kernel machines in conjunction with indefinite kernels may be both reasonable and beneficial [19, 20].

5 Conclusion and Future Work

In comparison with the great variety of software tools for statistical pattern recognition, the number of tools for structural pattern recognition is rather limited. There are some software tools available for manipulating graphs or exact

graph matching (e.g. the iGraph tool [21] or the VF2 library [22]), yet, a software tool for (approximate) graph edit distance computation is still missing. The present paper reviews three versions of graph edit distance which have been integrated in one publicly available software tool. We expect that the graph matching software tool introduced in this paper provides a major contribution towards promoting the use of graph based representations in pattern recognition and related fields.

In [23] a novel framework for graph isomorphism based on approximate graph edit distance computations has been introduced. It is planned to integrate these methods and thus the possibility of exact graph matching in our software tool in future work.

Acknowledgements. This work has been supported by the *Hasler Foundation* Switzerland.

References

1. Conte, D., Foggia, P., Sansone, C., Vento, M.: Thirty years of graph matching in pattern recognition. Int. Journal of Pattern Recognition and Artificial Intelligence 18(3), 265–298 (2004)
2. Luo, B., Wilson, R., Hancock, E.R.: Spectral embedding of graphs. Pattern Recognition 36(10), 2213–2223 (2003)
3. Wilson, R., Hancock, E.R.: Levenshtein distance for graph spectral features. In: Kittler, J., Petrou, M., Nixon, M. (eds.) Proc. 17th Int. Conf. on Pattern Recognition, vol. 2, pp. 489–492 (2004)
4. Boeres, M.C., Ribeiro, C.C., Bloch, I.: A randomized heuristic for scene recognition by graph matching. In: Ribeiro, C.C., Martins, S.L. (eds.) WEA 2004. LNCS, vol. 3059, pp. 100–113. Springer, Heidelberg (2004)
5. Sorlin, S., Solnon, C.: Reactive tabu search for measuring graph similarity. In: Brun, L., Vento, M. (eds.) GbRPR 2005. LNCS, vol. 3434, pp. 172–182. Springer, Heidelberg (2005)
6. Neuhaus, M., Bunke, H.: An error-tolerant approximate matching algorithm for attributed planar graphs and its application to fingerprint classification. In: Fred, A., Caelli, T.M., Duin, R.P.W., Campilho, A.C., de Ridder, D. (eds.) SSPR&SPR 2004. LNCS, vol. 3138, pp. 180–189. Springer, Heidelberg (2004)
7. Ambauen, R., Fischer, S., Bunke, H.: Graph edit distance with node splitting and merging and its application to diatom identification. In: Hancock, E., Vento, M. (eds.) GbRPR 2003. LNCS, vol. 2726, pp. 95–106. Springer, Heidelberg (2003)
8. Robles-Kelly, A., Hancock, E.R.: Graph edit distance from spectral seriation. IEEE Transactions on Pattern Analysis and Machine Intelligence 27(3), 365–378 (2005)
9. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions of Systems, Science, and Cybernetics 4(2), 100–107 (1968)
10. Riesen, K., Fankhauser, S., Bunke, H.: Speeding up graph edit distance computation with a bipartite heuristic. In: Frasconi, P., Kersting, K., Tsuda, K. (eds.) Proc. 5th. Int. Workshop on Mining and Learning with Graphs, pp. 21–24 (2007)

11. Neuhaus, M., Riesen, K., Bunke, H.: Fast suboptimal algorithms for the computation of graph edit distance. In: Yeung, D.-Y., Kwok, J.T., Fred, A., Roli, F., de Ridder, D. (eds.) SSPR & SPR 2006. LNCS, vol. 4109, pp. 163–172. Springer, Heidelberg (2006)
12. Riesen, K., Neuhaus, M., Bunke, H.: Bipartite graph matching for computing the edit distance of graphs. In: Escolano, F., Vento, M. (eds.) GbRPR. LNCS, vol. 4538, pp. 1–12. Springer, Heidelberg (2007)
13. Riesen, K., Bunke, H.: Approximate graph edit distance computation by means of bipartite graph matching. Image and Vision Computing 27(4), 950–959 (2009)
14. Fankhauser, S., Riesen, K., Bunke, H.: Speeding up graph edit distance computation through fast bipartite matching. In: Jiang, X., Ferrer, M., Torsello, A. (eds.) GbRPR 2011. LNCS, vol. 6658, pp. 102–111. Springer, Heidelberg (2011)
15. Munkres, J.: Algorithms for the assignment and transportation problems. Journal of the Society for Industrial and Applied Mathematics 5, 32–38 (1957)
16. Jonker, R., Volgenant, T.: A shortest augmenting path algorithm for dense and sparse linear assignment problems. Computing 38, 325–340 (1987)
17. Schölkopf, B., Smola, A.: Learning with Kernels. MIT Press (2002)
18. Shawe-Taylor, J., Cristianini, N.: Kernel Methods for Pattern Analysis. Cambridge University Press (2004)
19. Neuhaus, M., Bunke, H.: Bridging the Gap Between Graph Edit Distance and Kernel Machines. World Scientific (2007)
20. Haasdonk, B.: Feature space interpretation of SVMs with indefinite kernels. IEEE Transactions on Pattern Analysis and Machine Intelligence 27(4), 482–492 (2005)
21. Csárdi, G., Nepusz, T.: The igraph software package for complex network research. Inter. Journal Complex Systems 1695 (2006)
22. Cordella, L.P., Foggia, P., Sansone, C., Vento, M.: An improved algorithm for matching large graphs. In: Proc. of the 3rd IAPR TC-15 Workshop on Graphbased Representations in Pattern Recognition, pp. 149–159 (2001)
23. Riesen, K., Fankhauser, S., Bunke, H., Dickinson, P.: Efficient suboptimal graph isomorphism. In: Torsello, A., Escolano, F., Brun, L. (eds.) GbRPR 2009. LNCS, vol. 5534, pp. 124–133. Springer, Heidelberg (2009)