Shortest-path kernels on graphs

Karsten M. Borgwardt and Hans-Peter Kriegel
Institute for Computer Science
Ludwig-Maximilians-University Munich
Oettingenstr. 67, 80538 Munich, Germany
{kb}|{kriegel}@dbs.ifi.lmu.de

Abstract

Data mining algorithms are facing the challenge to deal with an increasing number of complex objects. For graph data, a whole toolbox of data mining algorithms becomes available by defining a kernel function on instances of graphs. Graph kernels based on walks, subtrees and cycles in graphs have been proposed so far. As a general problem, these kernels are either computationally expensive or limited in their expressiveness. We try to overcome this problem by defining expressive graph kernels which are based on paths. As the computation of all paths and longest paths in a graph is NP-hard, we propose graph kernels based on shortest paths. These kernels are computable in polynomial time, retain expressivity and are still positive definite. In experiments on classification of graph models of proteins, our shortest-path kernels show significantly higher classification accuracy than walk-based kernels.

1 Introduction

Kernel methods are a popular method from statistical learning theory [18] with numerous applications in data mining. Kernels allow to perform tasks such as classification via Support Vector Machines [21], regression [5], clustering [1] and principal component analysis [19] using nonlinear hypotheses and on a wide variety of different data types.

Early studies on kernel methods dealt almost exclusively with vector-based descriptions of input data. Haussler [10] was the first to define a principled way of designing kernels on structured objects, the so-called *R-convolution kernel*.

Over recent years, kernels on structured objects such as strings and trees, transducers, dynamical systems, on nodes in graphs [14] and on graphs [8, 13] have been defined. Generally spoken, graph kernels are based on the comparison of graph-substructures via kernels. Walks [8, 13], subtrees [17] and cyclic patterns [11] have been considered for this purpose. However, kernels on these substructures are either computationally expensive, sometimes even NP-hard to determine, or limited in their expressiveness.

These disadvantages of existing kernel methods are due to competing requirements in graph kernel design: first, the kernel should be a good measure of similarity for graphs; second, its computation should be possible in polynomial time; third, the kernel must still be positive definite; fourth, ideally, it should be applicable to all graphs, not just a small subset of graphs. Existing graph kernels have difficulties with reaching at least one of these four goals. In this article, we present a class of graph kernels that measure similarity based on shortest paths in graphs, that are computable in polynomial time, that are positive definite and that are applicable to a wide range of graphs.

Outline of this paper In section 2, we will review existing kernels on graphs with respect to their expressivity and efficiency. In section 3, we define a graph kernel based on all paths which is positive definite, yet its computation is NP-hard. In section 4, we define an algorithm for calculating shortest-path kernels on graphs which are positive definite, computationally feasible and still expressive. We then test our shortest-path kernel on a classification task on graph models of proteins in section 5. We conclude with a discussion and conclusions in section 6.

2 Existing graph kernels

Before we will review existing graph kernels, we will state the essential definitions from graph theory necessary to follow our argumentation.

2.1 Primer on graph theory

A *graph* G consists of a set of *nodes* (or *vertices*) V and *edges* E. In this article, n denotes the number of nodes in a graph and m the number of edges in a graph.

An *attributed* graph is a graph with labels on nodes and/or edges; we refer to labels as *attributes*. In our case, attributes will consist of pairs of the form (*attribute-name*, *value*). The adjacency matrix A of G is defined as

$$[A]_{ij} = \begin{cases} 1 & \text{if } (v_i, v_j) \in E, \\ 0 & \text{otherwise} \end{cases}$$

where v_i and v_j are nodes in G. A *walk* w of length $k-1$ in a graph is a sequence of nodes $v_1, v_2, \cdots, v_k$ where $(v_{i-1}, v_i) \in E$ for $1 < i \leq k$.

w is a *path* if $v_i \neq v_j$ iff $i \neq j \forall i, j \in \{1, \ldots, k\}$. Alternatively, *walks* are often referred to as *paths*; *paths* are then named *simple, unique* or *loopless paths*, which may lead to some confusion. To clarify the difference for the remainder of this article, we define a path to be a walk without repetitions of nodes. A *cycle* is a walk with $v_1 = v_k$, a *simple cycle* does not have any repeated nodes except for v_1.

A Hamilton path is a path that visits every node in a graph exactly once. An Euler path is a path that visits every edge in a graph exactly once.

2.2 Random walk kernel

Random walk kernels are based on the idea to count the number of matching walks in two input graphs. Gärtner et al. [9] define an elegant approach to determine all pairs of matching walks in two input graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ via a direct product graph $G_\times$:

$$k_\times(G_1, G_2) = \sum_{i,j=1}^{|V_\times|} [\sum_{n=0}^{\infty} \lambda_n A_\times^n]_{ij},$$

where $A_\times$ is the adjacency matrix of $G_\times = (V_\times, E_\times)$, defined via

$$V_\times(G_1 \times G_2) = \{(v_1, w_1) \in V_1 \times V_2 : label(v_1) = label(w_1)\}$$

$$E_\times(G_1 \times G_2) = \{((v_1, w_1), (v_2, w_2)) \in V^2(G_1 \times G_2) : (v_1, v_2) \in E_1 \wedge (w_1, w_2) \in E_2 \wedge (label(v_1, v_2) = label(w_1, w_2))\}$$

λ_n must be chosen appropriately for $k_\times$ to converge.

Based on ideas in [13], this kernel is redefined in [3] to measure similarities between walks that are not identically labeled. For this purpose, pairs of walks up to a fixed length k are observed in product graphs consisting of all pairs of nodes and edges -independently from their labels- from both input graphs. Node and edge labels along the walks are compared via kernel functions.

Despite its clear definition, these random walk kernels bear difficulties. The direct product graph may contain $|V_1| \times |V_2|$ nodes. Even if its adjacency matrix is sparse, it might become full by taking powers of this matrix. Handling such a huge full matrix leads to enormous runtime and memory requirements.

Besides computational issues, one observes the problem that small identical substructures in input graphs can lead to high similarity scores. As walks allow for repetitions of nodes, these graph kernels lead to the problem of "tottering", i.e. by iteratively visiting the same cycle of nodes, a walk can generate artifically high similarity values. Thus tottering limits the expressiveness of random walk graph kernels.

Analyzing current random walk kernels, it is obvious that the approach in [3] measuring partial similarity leads to huge product graphs. Keeping to the original random walk kernel by Gärtner et al. [9], the ability to find partial similarities is lost, reducing the expressivity of the graph kernel. Both approaches suffer from tottering.

2.3 Label enrichment and redefinition of kernel

In [16], steps are undertaken to improve the expressivity and efficency of walk-based graph kernels. Additional labels are added to each graph to reduce the number of matching nodes in two graphs. This measurement reduces the effort to compute their kernel values, as the number of pairs of walks that have to be considered is reduced.

As a strategy to avoid tottering, the walk-based graph kernel from [13] is redefined such that walks cannot contain sub-cycles consisting of two nodes. However, while the label enrichment has a positive impact on classification accuracy in the experimental evaluation, forbidding two-nodes-cycles did not lead to a significant improvement.

2.4 Subtree kernel and cyclic pattern kernel

As an alternative to random walk kernels, kernels based on subtrees and cyclic patterns have been defined.

The kernel in [17] compares subtree patterns in graphs. Starting from a node v, a tree is created by adding all nodes that can be reached from v in $1, \ldots, h$ steps where h is the height of this tree. Obviously, as there is often more than one walk between two nodes, the same node must be included several times into the subtree. These "copies" of the same node are treated as if they were distinct nodes; otherwise the pattern would not be a tree any more. This repeated visiting of the same nodes leads to tottering as in walk-based kernels. Furthermore, the number of nodes to consider grows exponentially with the height of the subtree under study. While it seems attractive to deal with more expressive substructures such as subtrees, they increase computational costs and do not reduce tottering.

Computing kernels based on cyclic and tree patterns [11] is a further approach to define graph kernels. Instead of

counting the frequency of these cycles in two input graphs, an intersection kernel is applied that counts the number of cycles that appear in both graphs. However, the computation of general cycles is NP-hard. To overcome this problem, only graphs are considered that do not contain more than k simple cycles. As a consequence, it is only applicable to datasets that fulfill this constraint.

Our approach to define an expressive and efficient graph kernel is to compare graphs based on paths instead of walks. The challenge is to ensure positive definiteness of the graph kernel and to keep its runtime complexity polynomial.

3 Graph kernel on all paths

In [17], for directed graphs, a proof is given that graph kernels based on subpaths are not positive definite. In this proof, a path is -as usually - defined as a sequence of nodes, consisting of at least 1 node and without any repetitions of nodes. We will show in the following that defining paths as sequences of neighboring pairwise distinct edges allows to define kernels based on subpaths that are still positive definite.

We will first define paths as sequences of edges. Second, we will prove that our edge-based path kernel on graphs is a positive definite R-convolution kernel.

3.1 Edge walks and edge paths

Definition 1 (Edge walk and edge path) *Given a graph G=(V, E) with $\{e_1, \ldots, e_l\} \subset E$ and $\{v_{i_1}, v_{i_2}, v_{j_1}, v_{j_2}\} \subset V$. An edge walk $w = (e_1, e_2, \ldots, e_l)$ is defined as a sequence of edges e_1 to e_l where e_i with $1 \leq i \leq l$ is a neighbor of $e_{i+1} = e_j$, i.e. $e_i = (v_{i_1}, v_{i_2})$ and $e_j = (v_{j_1}, v_{j_2})$ are neighbors if $v_{i_2} = v_{j_1}$. An edge path p is defined as an edge walk without repetitions of the same edge.*

Note that in the above definition, an edge path may contain the same node multiple times, but every edge only once. An edge path p is an Euler path in the graph exactly consisting of the edges of p. In the remainder of the paper, we will refer to edge paths and edge walks as paths and walks, respectively, unless explicitly stated otherwise.

3.2 All-paths kernel

After redefining paths as edge paths, we are now able to define a kernel on all paths in two graphs.

Definition 2 (All-paths kernel) *Given two graphs G_1 and G_2. Let $P(G_i)$ be the set of all paths in graph G_i where $i \in \{1, 2\}$. Let k_{path} be a positive definite kernel on two paths, defined as the product of kernels on edges and nodes along the paths. We then define an all-paths kernel $k_{all\ paths}$ as*

$$k_{all\ paths}(G_1, G_2) = \sum_{p_1 \in P(G_1)} \sum_{p_2 \in P(G_2)} k_{path}(p_1, p_2),$$

i.e. we define the all-paths kernel as the sum over all kernels on pairs of paths from G_1 and G_2.

In the following lemma, we prove that the all-paths kernel is a valid kernel.

Lemma 1 *The all-paths kernel is positive definite.*

Proof:

We define a relation $R(x', x'', x)$, where x' is a path and x'' and x are graphs. $R(x', x'', x) = 1$ iff x'' is the graph that is created when removing all edges in x' from x. $R^{-1}(x)$ is then the set of all possible decompositions of graph x via R into x' and x''. R is finite, as there is only a finite number of paths in a graph, since their length is upper bounded by the number of edges. We define a kernel k_{path} on paths as a product of kernels on nodes and edges in these paths; this is a positive definite tensor product kernel [18]. We also define a trivial graph kernel $k_{one} = 1$ for all pairs of graphs.

We can then define an all-paths kernel as a positive definite R-convolution defined as in [10]

$$\begin{aligned} &k_{all\ paths}(G_1, G_2) = \\ &= \sum_{R^{-1}(G_1)} \sum_{R^{-1}(G_2)} k_{path}(x'_1, x'_2) * k_{one}(x''_1, x''_2) = \\ &= \sum_{p_1 \in P(G_1)} \sum_{p_2 \in P(G_2)} k_{path}(p_1, p_2) \end{aligned} \quad (1)$$

with $P(G_i)$ as the set of all paths in G_i, $i \in \{1, 2\}$. □

The computation of this kernel, however, is NP-hard, as we will prove in the following.

Lemma 2 *Computing the all-paths kernel is NP-hard.*

Proof: We show this result by proving that finding all paths in a graph is NP-hard. If determining the set of all paths in a graph was not NP-hard, one could determine whether a graph has a Hamilton path or not by checking whether a path exists with length $n - 1$. This problem, however, is known to be NP-complete [12]. Consequently, determining the set of all paths is NP-hard and therefore the computation of the all-paths kernel is NP-hard. □

In [9] it is shown that computing kernels based on subgraphs is NP-hard. Although we are restricting ourselves to a small subset of all subgraphs, namely to paths, kernel computation is still NP-hard in our case.

4 Graph kernels on shortest paths

While determining all paths is NP-hard, finding special subsets of paths is not necessarily. Determining longest paths in a graph is again NP-hard, as it would allow to decide whether a graph contains a Hamilton path or not. Computing shortest paths in a graph, however, is a problem solvable in polynomial time. Prominent algorithms such as Dijkstra (for shortest paths from one source node) [4] or Floyd-Warshall [6, 22] (for all pairs of nodes) allow to determine shortest distances in $O(m+n*\log n)$ [7] and $O(n^3)$ time, respectively.

4.1 Floyd-transformation

The essential first step in our shortest-path kernel is to transform the original graphs into shortest-paths graphs. A shortest-paths graph S contains the same set of nodes as the input graph I. Unlike in the input graph, there exists an edge between all nodes in S which are connected by a walk in I. Every edge in S between nodes v_i and v_j is labeled by the shortest distance between these two nodes.

Any algorithm which solves the all-pairs-shortest-paths problem can be applied to determine all shortest distance edge labels in S. We propose to use Floyd's algorithm (see Table 1). This algorithm has a runtime of O(n^3), is applicable to graphs with negative edge weights, but must not contain negative-weighted cycles. Furthermore, it is easy to implement. In the following, we will refer to the process of transforming a graph I into S via Floyd's algorithm as *Floyd-transformation*.

4.2 Shortest-path graph kernel

After Floyd-transformation of our input graphs, we can now define a shortest-path kernel.

Definition 3 (Shortest-path graph kernel) *Let G_1 and G_2 be two graphs that are Floyd-transformed into S_1 and S_2. We can then define our shortest-path graph kernel on $S_1 = (V_1, E_1)$ and $S_2 = (V_2, E_2)$ as*

$$k_{shortest\ paths}(S_1, S_2) = \sum_{e_1 \in E_1} \sum_{e_2 \in E_2} k^{(1)}_{walk}(e_1, e_2),$$

where $k^{(1)}_{walk}$ is a positive definite kernel on edge walks of length 1.

In the following, we will prove the validity of our shortest-path kernel.

Lemma 3 *The shortest-path graph kernel is positive definite.*

Floyd-Warshall(Graph G with n nodes
and adjacency matrix A)
for i := 1 to n
  for j := 1 to n
    if ((A(i,j) == 1) and i $\neq$ j
      cost[i,j] = distance(i,j);
    else
      if (i = j)
        cost[i,j]=0;
      else
        cost[i,j]=∞;
      end
    end
  end
end
for k := 1 to n
  for i := 1 to n
    for j := 1 to n
      if (cost[i, k] + cost[k, j] < cost[i, j])
        cost[i, j] := cost[i, k] + cost[k, j];
      end
    end
  end
end

Table 1. Pseudocode for Floyd-Warshall's algorithm [6] for determining all-pairs shortest paths.

Proof: The shortest-path kernel is simply a walk kernel run on a Floyd-transformed graph considering walks of length 1 only. We follow the proofs in [13] and [3]. First, we choose a positive definite kernel on nodes and a positive definite kernel on edges. We then define a kernel on pairs of walks of length 1, $k^{(1)}_{walk}$, as the product of kernels on nodes and edges encountered along the walk. As a tensor product of node and edge kernels [18], $k^{(1)}_{walk}$ is positive definite. We then zero-extend $k^{(1)}_{walk}$ to the whole set of pairs of walks, setting kernel values for all walks with length $\neq$ 1 to zero. This zero-extension preserves positive definiteness [10]. The positive definiteness of the shortest-path kernel follows directly from its definition as a convolution kernel, proven to be positive definite by [10]. □

Run time complexity The shortest-path kernel avoids tottering, yet it remains an interesting question how it compares to the known random walk kernel (measuring partial similarity) in terms of runtime complexity.

W. l. o. g. let us assume that we are dealing with two graphs with n nodes and m edges each. To compute the walk kernel, we first have to determine the direct product

graph whose number of nodes is obviously upper bounded by n^2. We then have to invert the adjacency matrix of this direct product graph; standard algorithms for inversion of an $x * x$ matrix require $O(x^3)$ time. As $x \leq n^2$ in our case, the random walk graph kernel has a total runtime complexity of $O(n^6)$.

The shortest-path kernel requires a Floyd-transformation which can be done in $O(n^3)$ when using the Floyd-Warshall algorithm. The number of edges in the transformed graph is n^2, if the original graph is connected. Pairwise comparison of all edges in both transformed graphs is then necessary to determine the kernel value. We have to consider $n^2 * n^2$ pairs of edges, resulting in a total runtime of $O(n^4)$.

In favor of the walk kernel, one may argue that the number of nodes in the product graph is only n^2 if all nodes in both factor graphs are equally labeled. This is true for the random walk based on total similarity, whereas the random walk based on partial similarity always leads to a product graph with n^2 nodes. Thus, only a random walk kernel which measures total similarity and which creates product graphs with $n^{\frac{4}{3}}$ nodes is of the same runtime complexity as our graph kernel. One method to reach this is label enrichment as used in [16].

4.3 Equal length shortest-path kernel

Label enrichment can also be applied to our Floyd-transformed graphs to speed up kernel computation. Both edges and nodes can be enriched by additional attributes. When performing the Floyd-Warshall algorithm, one is usually interested in the shortest distance between all nodes. However, if we store information about the shortest paths, i.e. the number of edges or the average length of an edge in these shortest paths, then we can exploit this extra information to reduce computational cost. This is done by setting kernels to zero for all pairs of shortest paths where e.g. the number of edges in the shortest paths is not identical, i.e.

$$k_{steps}(w, w') = \begin{cases} 1 & \text{if steps}(w) = \text{steps}(w'), \\ 0 & \text{otherwise} \end{cases}$$

where w and w' are walks and $steps(x)$ is the number of edges in walk x. If the steps kernel is zero for a pair of walks, we do not have to evaluate the node and edge kernel.

4.4 K shortest-path kernel

Even more valuable information for our kernel could be not only to know the shortest path between two nodes, but the k shortest paths. For each of the k shortest paths, one edge could then be created in the Floyd-transformed graph.

Finding k shortest walks and paths in a graph is a well-studied topic in graph theory and applied sciences [23, 15]. Many of the algorithms proposed for solving this problem, however, determine k shortest walks, not k shortest paths. Applying these algorithms would reintroduce the problem of tottering into our path-based kernel. It is therefore essential to chose an algorithm for finding "k loopless shortest paths" in a network. Such algorithms have been proposed over 30 years ago [23, 15] and any of those can be run on our input graphs, as long as there are no cycles in our graphs with negative weights. The setback of this method is the increased runtime complexity for determining k shortest loopless paths. Yen's algorithm in [23] requires $O(kn(m + n * \log n))$ time complexity for finding k shortest loopless paths between a pair of nodes. Consequently, theoretical complexity would be $O(kn^5)$ for determining k shortest loopless paths for all pairs of nodes in a fully connected graph and pairwise comparison of all k shortest paths in two graphs would be of complexity $O((k * n^2) * (k * n^2)) = O(k^2 * n^4)$. As a result, the preprocessing step has a higher runtime complexity than the kernel computation in this case.

A simple way to determine k shortest disjunct paths between two nodes, where no pair of paths shares any identical edge, is to iteratively apply Dijkstra's algorithm to the same graph and to remove all edges that belong to the currently shortest path. Still, this procedure would be of runtime complexity $O(n^2 * k * (m + n * \log n))$, which could become $O(k * n^4)$ in a fully connected graph.

5 Experiments

5.1 Experimental setting

To evaluate the practical performance of our shortest-path graph kernel, we chose a classification task from bioinformatics [3]. 540 proteins, 90 per class, should be classified into 6 distinct functional classes in 10-fold cross-validation, solely based on protein structure information.

We obtained the protein structures from the Protein Data Bank [2] and their corresponding enzyme class labels from the BRENDA enzyme database [20]. We randomly choose 90 proteins from each of the 6 enzyme EC hierarchy top level classes. We translated these protein structures into graph models in which the secondary structure elements of a protein represent the nodes.

Every node is connected to its three nearest neighbors in space. As a simplification, distances between secondary structure elements are calculated as distances between their spatial centers. Edges are labeled by the distance they represent in angstroms. Node bear labels representing their type, namely helix, sheet or loop, and their length in amino acids.

On these graph models of proteins, we ran walk kernels and shortest-paths kernels. As calculating the walk kernel for walks up to infinity results in memory problems, we

kernel type	accuracy	st. dev.
2 shortest paths	94.44	2.52
e.l. shortest paths	93.52	2.93
shortest paths	93.33	3.22
walks up to length 4	89.63	2.30
walks up to length 5	88.89	1.99
walks up to length 6	88.15	1.67
walks up to length 7	87.96	1.78

Table 2. Walk kernel vs. shortest-path kernel. Prediction accuracy on 540 proteins from 6 EC classes in 10-fold cross-validation. (st. dev. = standard deviation, e.l. = equal length)

approximate it by walks of up to length k, setting kernel values for longer walks to zero. We performed tests for k in the range from 4 to 7. We also employed our shortest-path kernel and the equal length shortest-path kernel on the same data. Furthermore, we ran a 2 shortest-path kernel determining the 2 shortest disjunct paths between nodes via Dijkstra's algorithm.

All graph kernels use the same set of node and edge kernels. Node types are compared via a delta kernel, i.e.

$$k_{type}(x, x') = \begin{cases} 1 & \text{if } type(x) = type(x'), \\ 0 & \text{otherwise} \end{cases}$$

Node lengths are compared via a Brownian bridge kernel, i.e.

$$k_{length}(x, x') = max(0, c - |length(x) - length(x')|).$$

The same Brownian bridge kernel is applied to edges to measure their difference in length. c is set to 3 for nodes and to 2 for edges via cross-validation as in [3].

After calculating all graph kernel matrices mentioned above, we predicted enzyme class membership in 10-fold cross-validation for 540 proteins. We performed "one-class vs. rest" Support Vector Machine classification and repeated this for all six EC top level classes. We report results as averages across all EC classes in Table 2.[1]

5.2 Results

The shortest-path kernels outperform all walk kernels with an accuracy of at least 93.33%. The accuracy level of the worst shortest-path kernel on 540 proteins is significantly higher than that of the best kernel using walks up to length 4 (Yates-χ^2 = 4.8, P = 0.028). As a result, considering shortest paths instead of walks increases classification accuracy significantly.

Among the walk kernels, classification is decreasing with the length of the walks under study. This is an indicator that the longer the walks we examine, the more numerous walks created by tottering get. With an increasing number of tottering walks, classification accuracy decreases.

Among the shortest-path kernels, the 2 shortest-path kernels perform slightly better than the equal length shortest-path kernel and the standard shortest-path kernel. However, the accuracy differences between the different types of shortest-path kernels are not significant on our test set.

6 Discussion and conclusions

We have defined graph kernels based on shortest paths, which are polynomial to compute, positive definite and retain expressivity while avoiding the phenomenon of "tottering". In experiments on classifying graphs model of proteins into functional classes, they outperformed kernels based on random walks significantly.

The shortest-path kernels prevent tottering. It is not possible that the same edge appears twice in the same shortest path, as this would violate the definition of a path. Subsequently, artifically high similarity scores caused by repeated visiting of the same cycle of nodes are prohibited in our graph kernel.

The shortest-path kernel as described in this article is applicable to all graphs on which Floyd-Warshall can be performed. Floyd-Warshall requires that cycles with negative weight do not exist. If edge labels represent distances, which is the case in most molecular classification tasks, this condition generally holds.

As all and longest paths are NP-hard to compute, our graph kernel uses shortest paths. As shown in our experiments, shortest distances between nodes are a characteristic of graphs which is essential for graph classification in many applications such as molecular graphs or telecommunication networks. However, there might be areas of application where longest paths or average path or walk lengths within a graph are more adequate and important for classifying graphs than shortest paths. Designing graph kernels for specific tasks therefore remains a necessity.

In our experiments, we reached the highest accuracy using a kernel looking at 2 disjunct shortest paths between all nodes. Both are represented by an edge of their own in the Floyd-transformed graphs we ran our graph kernel on. An alternative approach would to model both as one joint edge which bears one label for the length of the shortest path and one for the length of the 2nd shortest path. Defining a proper kernel for combining these two attributes, maybe even giving different weights to the shortest path and the

[1] Our graph kernel was implemented in MATLAB, release 13. We used a Linux Debian workstation with 3 GHz Intel CPUs for our experiments. We employed the SVM package SVLAB.

2nd shortest path could lead to even higher classification accuracy.

An open question concerning our shortest graphs kernels is how to deal with graphs that bear more than one edge label, i.e. in which edges do not represent distances only. As long as one edge attribute reflects some kind of distance between nodes, our algorithm remains applicable. The difficulty is then to include the other edge attributes into the kernel computation. One could think of many different alternatives, either taking an average over numerical edge attributes along the shortest path or to apply an intersection kernel on sets of edge attributes along two paths. Analogously, one might think of a Floyd-transformation which preserves information about intermediate nodes in shortest paths, as in the shortest-path kernels we defined, only attributes of start and end node are considered. How to optimally include this information into a graph kernel is a question of kernel engineering. Domain knowledge from the area of application will be most beneficial in this search for a good kernel on a particular type of graphs.

Our shortest-path graph kernel is computable in polynomial time. In theory, its runtime complexity for fully connected graphs is two powers lower than that of a random walk kernel. In practice, efficient computations for matrix inversion and label enrichment which creates a sparse direct product graph matrix will lead to comparable run-time performances of shortest-path kernel and walk kernel.

In future studies, we will look at further connections between graph theory and kernel methods. Graphs in bioinformatics promise to be an interesting area of application, as these graphs have particular characteristics (e.g. scale-free networks) that could be exploited in special kernel functions.

Acknowledgements

This work was supported in part by the German Ministry for Education, Science, Research and Technology (BMBF) under grant no. 031U112F within the BFAM (Bioinformatics for the Functional Analysis of Mammalian Genomes) project which is part of the German Genome Analysis Network (NGFN).

References

[1] A. Ben-Hur, D. Horn, H. Siegelmann, and V. Vapnik. A support vector method for hierarchical clustering. In T. K. Leen, T. G. Dietterich, and V. Tresp, editors, *Advances in Neural Information Processing Systems 13*, pages 367 – 373. MIT Press, 2001.

[2] H. Berman, J. Westbrook, Z. Feng, G. Gilliland, T. Bhat, H. Weissig, I. Shindyalov, and P. Bourne. The protein data bank. *Nucleic Acids Research*, 28:235–242, 2000.

[3] K. M. Borgwardt, C. S. Ong, S. Schoenauer, S. Vishwanathan, A. Smola, and H.-P. Kriegel. Protein function prediction via graph kernel. *Bioinformatics*, 2005. in press.

[4] E. W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematics*, 1:269–271, 1959.

[5] H. Drucker, C. J. C. Burges, L. Kaufman, A. Smola, and V. Vapnik. Support vector regression machines. In M. C. Mozer, M. I. Jordan, and T. Petsche, editors, *Advances in Neural Information Processing Systems 9*, pages 155 – 161, Cambridge, MA, 1997. MIT Press.

[6] R. Floyd. Algorithm 97, shortest path. *Comm. ACM*, 5:345, 1962.

[7] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *JACM*, 34(3):596–615, 1987.

[8] T. Gärtner. Exponential and geometric kernels for graphs. In *NIPS*02 workshop on unreal data*, volume Principles of modeling nonvectorial data, 2002.

[9] T. Gärtner, P. Flach, and S. Wrobel. On graph kernels: Hardness results and efficient alternatives. In B. Schölkopf and M. Warmuth, editors, *Sixteenth Annual Conference on Computational Learning Theory and Seventh Kernel Workshop, COLT*. Springer, 2003.

[10] D. Haussler. Convolutional kernels on discrete structures. Technical Report UCSC-CRL-99 - 10, Computer Science Department, UC Santa Cruz, 1999.

[11] T. Horvath, T. Gärtner, and S. Wrobel. Cyclic pattern kernels for predictive graph mining. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining*, 2004.

[12] D. Jungnickel. *Graphen, Netzwerke und Algorithmen*. BI-Wiss.-Verlag, Mannheim, Germany, 1994.

[13] H. Kashima, K. Tsuda, and A. Inokuchi. Marginalized kernels between labeled graphs. In *Proceedings of the* 20^{th} *International Conference on Machine Learning (ICML)*, Washington, DC, United States, 2003.

[14] R. S. Kondor and J. Lafferty. Diffusion kernels on graphs and other discrete structures. In *Proceedings of the ICML*, 2002.

[15] E. Lawler. A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. *Management Science*, 18:401–405, 1972.

[16] P. Maha, N. Ueda, T. Akutsu, J.-L. Perret, and J.-P. Vert. Extensions of marginalized graph kernels. In *Proceedings of the Twenty-First International Conference on Machine Learning*, 2004.

[17] J. Ramon and T. Gärtner. Expressivity versus efficiency of graph kernels. Technical report, First International Workshop on Mining Graphs, Trees and Sequences (held with ECML/PKDD'03), 2003.

[18] B. Schölkopf and A. J. Smola. *Learning with Kernels*. MIT Press, 2002.

[19] B. Schölkopf, A. J. Smola, and K.-R. Müller. Kernel principal component analysis. In B. Schölkopf, C. J. C. Burges, and A. J. Smola, editors, *Advances in Kernel Methods - - Support Vector Learning*, pages 327 – 352. MIT Press, Cambridge, MA, 1999.

[20] I. Schomburg, A. Chang, C. Ebeling, M. Gremse, C. Heldt, G. Huhn, and D. Schomburg. Brenda, the enzyme database: updates and major new developments. *Nucleic Acids Res*, 32 Database issue:D431–D433, Jan 2004.

[21] V. Vapnik. *Statistical Learning Theory*. Wiley, New York, 1998.

[22] S. Warshall. A theorem on boolean matrices. *J. ACM*, 9:11–12, 1962.

[23] J. Y. Yen. Finding the k shortest loopless paths in a network. *Management Sciences*, 17:712–716, 1971.