Explicit versus Implicit Graph Feature Maps: A Computational Phase Transition for Walk Kernels

Nils Kriege[†], Marion Neumann[*], Kristian Kersting[†], Petra Mutzel[†]
[†] Dept. of Computer Science, TU Dortmund, Germany
Email: {nils.kriege,kristian.kersting,petra.mutzel}@tu-dortmund.de
[*] BIT, University of Bonn, Germany
Email: marion.neumann@uni-bonn.de

***Abstract*—As many real-world data can elegantly be represented as graphs, various graph kernels and methods for computing them have been proposed. Surprisingly, many of the recent graph kernels do not employ the kernel trick anymore but rather compute an explicit feature map and report higher efficiency. So, is there really no benefit of the kernel trick when it comes to graphs? Triggered by this question, we investigate under which conditions it is possible to compute a graph kernel explicitly and for which graph properties this computation is actually more efficient. We give a sufficient condition for R-convolution kernels that enables kernel computation by explicit mapping. We theoretically and experimentally analyze efficiency and flexibility of implicit kernel functions and dot products of explicitly computed feature maps for widely used graph kernels such as random walk kernels, subgraph matching kernels, and shortest-path kernels. For walk kernels we observe a phase transition when comparing runtime with respect to label diversity and walk lengths leading to the conclusion that explicit computations are only favorable for smaller label sets and walk lengths whereas implicit computation is superior for longer walk lengths and data sets with larger label diversity.**

I. Introduction

Analyzing complex data is becoming more and more important. In numerous application domains, e.g., chem- and bioinformatics or image and social network analysis, the data is structured and hence can naturally be represented as graphs. To achieve successful learning we need to exploit the rich information inherent in the graph structure and the annotations of nodes and edges. A popular approach to mining structured data is to design graph kernels measuring the similarity between graphs. The graph kernel can then be plugged into a kernel machine, such as support vector machines or Gaussian processes, for efficient learning and prediction.

The kernel-based approach to predictive graph mining simply requires a positive semidefinite (p.s.d.) kernel function between graphs. Graphs, composed of labeled vertices and edges possibly enriched with continuous attributes, however, are not fix-length vectors but rather complicated data structures, and thus standard kernels cannot be used. Instead, the general strategy to design graph kernels is to decompose the graph into small substructures among which kernels are defined. The graph kernel itself is then a combination, for example, a sum or a product, of the kernels between the possibly overlapping parts. As the graphs of interest have a finite number of nodes and edges, any substructure of a graph comprises a countable set and hence graph kernels are usually convolution kernels and therefore p.s.d. [1]. So, the various graph kernels proposed in the literature mainly differ in the way the parts are constructed and in the similarity measure used to compare them. Further, existing graph kernels also differ in their ability to exploit annotations such as labels and attributes on the nodes and edges.

It is well-known that every kernel $K(G, H)$ can be expressed as a dot product $\langle \phi(G), \phi(H) \rangle$, where ϕ is a *feature map*. Hence, there are two strategies to actually compute the kernel value: (1) One way is functional computation, e.g., by computing kernel values for the individual parts of the decomposition that are then combined. Some kernels employ product graphs, which store intermediate results, to speed up computation. In this case the feature maps not necessarily are known and may be of infinite dimension. Therefore we refer to this approach as *implicit* computation. (2) The other strategy is to compute the feature map $\phi(G)$ for each graph *explicitly* to obtain the kernel values from the dot product between two feature maps. These feature maps commonly count how often certain substructures occur in a graph.

So far, previous work introducing novel graph kernels followed one of the strategies for computation. In contrast, we are interested in analyzing and comparing the computation schemes. To the best of our knowledge such a comparison does not exist, although – as we will show – it is of high value for practitioners and researchers who want to gain a deeper understanding of what it means to measure graph similarity.

To achieve our goal, we first assess under which conditions the computation of an explicit mapping from graphs to vector-valued feature spaces is possible. In particular, we give a sufficient condition for convolution kernels. We theoretically analyze both methods of computation regarding runtime and flexibility. Then, we derive explicit computation schemes for random walk kernels [2], [3], subgraph matching kernels [4], and shortest-path kernels [5]. We compare efficient algorithms for the explicit and implicit computation of these kernels experimentally. Our product graph based computation of the walk kernel fully supports arbitrary node and edge kernels and exploits their sparsity. Further, we present the first explicit computation scheme for walk-based kernels. Given this, we are finally able to experimentally compare the runtimes of both computation strategies systematically with respect to the label

diversity, data set size, and substructure size, i.e., walk length and subgraph size. As it turns out, there exists a computational phase transition for walk kernels. Already for rather small walk lengths, implicit graph kernel computations are considerably faster than explicit graph feature maps.

II. Related Work – Graph Kernels

In the following we review existing graph kernels based on explicit or implicit computation. For random walk kernels implicit computation schemes based on product graphs have been proposed. The product graph $G_\times$ has a node for each pair of nodes in the original graphs. Two nodes in the product graph are neighbors only if the corresponding nodes in the original graphs were both neighbors as well. Product graphs have some nice properties making them suitable for the computation of graph kernels. First, the adjacency matrix of a product graph is the Kronecker product of the adjacency matrices of the original graphs $A_\times = A \otimes A'$, same holds for the weight matrix $W_\times$ and the transition matrix $T_\times$. $W_\times$ can, however, in general be used to capture any kind of edge similarity. Further, performing random walks on the product graph is equivalent to performing simultaneous random walks on the original graphs [2]. That is, $\Pr(X_t = (u,v) \mid X_0 = (w,z)) = (T_\times^t)_{ab}$ with $a = (u-1)n' + v$ and $b = (w-1)n' + z$ corresponds to the probability that random walks of length t on G and H, where the walk on G starts in w and ends in u and the walk on H starts in z and ends in v, are equal. The *random walk kernel* introduced in [3] is now given by $K(G,H) = \sum_{t=0}^{\infty} \mu_t T_\times^t \boldsymbol{\pi}_\times$, where μ_t is a weight guaranteeing the convergence of the infinite sum and $\boldsymbol{\pi}_\times = \boldsymbol{\pi}_G \otimes \boldsymbol{\pi}_H$ is the initial probability distribution among nodes in G and H. Several variations of the random walk kernel have been introduced in the literature. Instead of adding the probabilities of two walks being the same the geometric random walk kernel introduced in [2] counts the number of matching walks. Other variants of random walk kernels have been proposed, cf. [6], [7], [8], [9]. Also computed via a product graph, the *subgraph matching kernel* compares subgraphs of small sizes [4].

Avoiding the construction of potentially huge product graphs, explicit feature maps for graph kernels can be computed more memory efficient and often also faster. The features are either some kind of similarity measure among substructures or counts or indicators of occurrences of substructures of particular sizes. The *graphlet kernel*, for example, counts induced subgraphs of size $k \in \{3,4,5\}$ of unlabeled graphs according to $K(G,H) = \mathbf{f}_G^\top \mathbf{f}_H$, where $\mathbf{f}_G$ and $\mathbf{f}_H$ are the count features of G and H, respectively [10]. The *cyclic pattern kernel* measures the occurrence of cyclic and tree patterns and maps the graphs to pattern indicator features which are independent of the pattern frequency [11]. The *Weisfeiler-Lehman subtree kernel* counts label-based subtree-patterns [12] according to $K_d(G,H) = \sum_{h=1}^{d} K(G_h, H_h)$, where $K(G_h, H_h) = \langle \mathbf{f}_G^{(h)}, \mathbf{f}_H^{(h)} \rangle$ and $\mathbf{f}_G^{(h)}$ is a feature vector counting subtree-patterns in G of depth h. A subtree-pattern is a tree rooted at a particular node where each level contains the neighbors of its parent node; the same nodes can appear repeatedly. Other graph kernels on subtree-patterns have been proposed in the literature [13], [9]. In a similar spirit, the *propagation kernel* iteratively counts similar label or attribute distributions to create an explicit feature map for efficient kernel computation [14]. Another substructure used to measure the similarity among graphs are shortest paths. The *shortest-paths kernel* [5] is a kernel comparing all shortest-paths in two graphs according to their lengths and the node information:

$$K(G,H) = \sum_{u,v \in G} \sum_{w,z \in H} \kappa_V(u,w)\kappa_d(d_{uv}, d_{wz})\kappa_V(v,z), \quad (1)$$

where κ_V is a kernel comparing node labels of the respective starting and end nodes of the paths, d_{uv} denotes the length of a shortest-path from u to v and κ_d is a kernel comparing path lengths. Another recently introduced graph kernel based on paths is the *GraphHopper kernel* which compares the nodes encountered while hopping along shortest paths [15].

The large amount of recently introduced graph kernels indicates that machine learning on structured data is both considerably difficult and extremely important. Surprisingly, none of the above introduced kernels is flexible enough to consider any kind of node and edge information while still being fast and memory efficient across arbitrary graph databases.

III. Kernel Matrix Computation

In order to apply kernel methods a symmetric p.s.d. matrix is computed that stores the kernel values between pairs of data objects. Using implicit computation this matrix is generated directly by computing the kernel functions. When explicit mapping into a feature space is applied, first a vector embedding of each object of the data set is generated and then the matrix is computed by taking the dot product between these vectors. Both approaches differ in terms of runtime, which depends on the complexity of the individual functions that must be computed in the course of the algorithms.

Theorem 1. *Implicit computation of an $N \times N$ kernel matrix requires time $\mathcal{O}(N^2 T_k)$, where T_k is the time to compute a single kernel value.*

Theorem 2. *Explicit computation of an $N \times N$ kernel matrix requires time $\mathcal{O}(NT_\phi + N^2 T_{dot})$, where T_ϕ is the time to compute the feature vector for a single graph and T_{dot} the time for computing the dot product.*

The feature map of a graph G can be considered as vector, where the i-th element counts the occurrences of the i-th feature in G. The ordering of the features must be consistent over all feature vectors, but in principle is arbitrary. In addition, feature vectors are typically sparse and many of the theoretically possible features do not occur at all in a specific data set. Therefore, we may adequately represent a feature map by a set of pairs $\phi(G) = \{(i,c) \mid \text{Feature } i \text{ occurs } c > 0 \text{ times in } G\}$, where i is a unique identifier of a feature. An appropriate implementation of this dictionary data structure is a hash table which yields $T_{dot} = \mathcal{O}(n^\star)$ in the average case, where $n^\star$ is the number of different features stored in one of the feature maps.

IV. Explicit Mapping of R-convolution Kernels

Graph kernels typically compare pairs of substructures and many of them are instances of R-convolution kernels [1], [3]. We propose an explicit computation scheme applicable to R-convolution kernels that fulfill a clearly defined condition. We then apply this idea to several concrete graph kernels. Suppose $\vec{x} \in \mathcal{X}$ is a decomposition of $G \in \mathcal{G}$ and let $R \subseteq \mathcal{X} \times \mathcal{G}$ be a relation with $R^{-1}(G) := \{\vec{x} \mid (\vec{x}, G) \in R\}$ finite for all $\vec{x} \in \mathcal{X}$. The function

$$K(G,H) = \sum_{\vec{x} \in R^{-1}(G)} \sum_{\vec{y} \in R^{-1}(H)} \kappa(\vec{x}, \vec{y}) \tag{2}$$

is a valid kernel provided that κ is a valid kernel on $\mathcal{X} \times \mathcal{X}$. The definition of R-convolution kernels as in [1] is obtained for $\mathcal{X} = \mathcal{X}_1 \times \cdots \times \mathcal{X}_d$ and $\kappa(\vec{x}, \vec{y}) := \prod_{i=0}^{d} \kappa_i(x_i, y_i)$ with $\vec{x} = (x_0, \ldots, x_d)$ and $\vec{y} = (y_0, \ldots, y_d)$. Note that κ yields a valid kernel provided that all κ_i are valid kernels.

Assume $\kappa(\vec{x}, \vec{y})$ is either 0 or 1 and induces an equivalence relation on $\mathcal{X}$, i.e., $\vec{x} \sim \vec{y} \Leftrightarrow \kappa(\vec{x}, \vec{y}) = 1$. Let $C_\kappa : \mathcal{X} \to \mathbb{N}$ be a function that assigns an element to the equivalence class of the relation induced by κ. Let $F(G,i) = \{\vec{x} \in R^{-1}(G) \mid C_\kappa(\vec{x}) = i\}$. We define the feature map of a graph G as $\phi(G) = \{(i,c) \in \mathbb{N}^2 \mid c = |F(G,i)| > 0\}$.

Theorem 3. *Let $K(G,H)$ be an R-convolution kernel, such that κ induces an equivalence relation on $\mathcal{X}$. Let ϕ be defined as above, then $K_\phi(G,H) := \langle \phi(G), \phi(H) \rangle = K(G,H)$.*

Proof. Consider an equivalence class i of the relation induced by κ. Clearly, only two elements from the same equivalence class contribute with one to the sum in Eq. (2). Assume there are k elements of $R^{-1}(G)$ in the equivalence class i and l elements of $R^{-1}(H)$. Then pairs of elements from these classes contribute a total of lk to the sum in Eq. (2). Note that (i,l) and (i,k) are contained in the feature maps $\phi(G)$ and $\phi(H)$, respectively. Therefore, the feature associated with class i by the function C exactly contributes lk to the dot product. The result follows, since the observation holds for all equivalence classes. □

Note that the sufficient condition of Theorem 3 not necessarily requires that κ corresponds to a Dirac kernel. One may, for example, also think of kernel functions for continuous attributes that perform some kind of binning and are 1 if and only if two values are in the same bin and 0 otherwise [14].

V. Subgraph & Matching Kernels

We consider a natural instance of an R-convolution kernel. Given the graphs $G, H \in \mathcal{G}$, the *subgraph kernel* is defined as

$$K_s(G,H) = \sum_{G' \subseteq G} \sum_{H' \subseteq H} k_{\simeq}(G', H'),$$

where $k_{\simeq} : \mathcal{G} \times \mathcal{G} \to \{0,1\}$ is the isomorphism kernel, i.e., $k_{\simeq}(G', H') = 1$ if and only if G' and H' are isomorphic. A similar kernel was defined in [2] and its computation was shown to be $\mathcal{NP}$-hard. However, it is polynomial time computable when considering only subgraphs up to a fixed size. Assume there is a function C with $C(G) = C(H) \iff k_{\simeq}(G,H) = 1$. Let $F(G,i) = \{G' \subseteq G \mid C(G') = i\}$. Then the corresponding feature map $\phi(G)$ distinguishes subgraphs up to isomorphism and counts their number of occurrences in G. As a matter of fact computing the function C corresponds to the graph canonization problem that is well-studied. By solving the graph canonization problem instead of graph isomorphism we obtain an explicit feature map for the subgraph kernel. Although graph canonization clearly is at least as hard as graph isomorphism, the number of canonizations required is linear in the number of subgraphs, while a quadratic number of isomorphism tests would be required for a single naive computation of the kernel. The gap in terms of runtime even increases when computing a whole kernel matrix, cf. Sec. III.

Indeed, the observations above are a key to several graph kernels recently proposed. The graphlet kernel, see. Sec. (II), is an instance of the subgraph kernel and computed by an explicit mapping scheme. However, only unlabeled graphs of small size are considered by the graphlet kernel, such that the canonizing function can be computed easily. The same approach was taken in [16] considering larger connected subgraphs of labeled graphs derived from molecular structures. On the contrary, for attributed graphs with continuous node labels, a function $k_{\simeq}$ is not sufficient to compare subgraphs adequately. Therefore, subgraph matching kernels were proposed [4], which allow to specify arbitrary kernel functions to compare node and edge attributes. Essentially, this kernel considers all mappings between subgraphs and scores each mapping by the product of node and edge kernel values of the node and edge pairs involved in the mapping. When the specified node and edge kernels are Dirac kernels, the subgraph matching kernel is closely related to the subgraph kernel [4]. Based on these observations, explicit computations of subgraph matching kernels are – if applicable – more efficient. However, the flexibility of specifying arbitrary node and edge kernels comes at the cost of making explicit computation schemes difficult, since the condition of Theorem 3 is not necessarily fulfilled.

VI. Fixed Length Walk Kernels

Walk-based kernels count "common" walks of two graphs. A walk of length k in a graph G is a sequence $(v_0, e_0, v_1, \ldots, e_{k-1}, v_k)$ of vertices and edges such that $e_i = (v_i, v_{i+1}) \in E$ for $i \in \{0, \ldots, k-1\}$.

Definition 1 (k-walk kernel)**.** *Let $\mathcal{W}_k(G)$ denote the set of walks of length k of a graph G. The* k-walk kernel *between two graphs $G, H \in \mathcal{G}$ is given by*

$$K_k^{=}(G,H) = \sum_{w \in \mathcal{W}_k(G)} \sum_{w' \in \mathcal{W}_k(H)} \kappa_W(w, w'),$$

where κ_W is a kernel between walks.

Definition 1 is very general and does not specify how to compare walks. An obvious choice is to decompose walks and define κ_W in terms of node and edge kernel functions,

denoted by κ_V and κ_E, respectively. Given two walks $w = (v_0, e_0, \ldots, v_k)$ and $w' = (v'_0, e'_0, \ldots, v'_k)$, we consider

$$\kappa_W(w, w') = \prod_{i=0}^{k} \kappa_V(v_i, v'_i) \prod_{i=0}^{k-1} \kappa_E(e_i, e'_i). \tag{3}$$

Assume the graphs in a data set have a finite set of discrete node and edge labels $l : V \cup E \to \Sigma$. An appropriate choice then is to use a Dirac kernel for both, node and edge kernels, between the the associated labels, i.e., $\kappa_V(v, v') = \delta(l(v), l(v'))$ (and κ_E analogously). We refer to the resulting kernel between walks by κ_W^δ, which is 1 if and only if the labels of all vertices and edges are equal and 0 otherwise.

The k-walk kernel clearly is an instance of an R-convolution kernel. In case of applying κ_W^δ we can give an explicit mapping into an associated feature space. A walk w of length k is then associated with a label sequence $l(w) = (l(v_0), l(e_0), \ldots, l(v_k)) \in \Sigma^{2k+1}$. In this case the parts of the decomposition of two graphs correspond to walks and two walks w and w' are considered equivalent if and only if $l(w) = l(w')$. We can associate the feature map $\phi_k^=(G) = \{(i, c) \mid c = |\{w \in \mathcal{W}_k(G) \mid l(w) = i\}| > 0\}$. Let $\phi_k^=$ be the mapping into this feature space, then $K_k^=(G, H) = \langle \phi_k^=(G), \phi_k^=(H) \rangle$ according to Theorem 3.

A. Kernel Computation

We propose an explicit and implicit computation scheme for the fixed-length walk kernel. To obtain authoritative experimental results we carefully implemented and engineered algorithms for both approaches.

1) Implicit mapping: An essential part of the implicit computation scheme is the generation of the product graph that is then used to compute the k-walk kernel. Node and edge kernels κ_V and κ_E are considered as part of the input and their values for pairs of nodes and edges are represented by the weights of the product graph. We avoid to create vertices and edges that would represent incompatible pairs with kernel value zero to obtain a compact representation.

Definition 2 (Weighted Direct Product Graph)**.** *Given two graphs $G = (V, E)$, $H = (V', E')$ and node and edge kernels κ_V and κ_E, the* weighted direct product graph *(WDPG) $G \times H = (V_P, E_P, w)$ of G and H is defined by*

$$\begin{aligned} V_P =& \{(v, v') \in V \times V' \mid \kappa_V(v, v') > 0\} \\ E_P =& \{(u, u')(v, v') \in V_P \times V_P \mid uv \in E \wedge u'v' \in E' \\ & \wedge \kappa_E(uv, u'v') > 0\} \\ w(v) =& \kappa_V(v, v') \quad \forall v = (v, v') \in V_P \\ w(e) =& \kappa_E(uv, u'v') \quad \forall e \in E_P, \text{where } e = (u, u')(v, v'). \end{aligned}$$

Let $n = |V|$, $n' = |V'|$ and $m = |E|$, $m' = |E'|$. The WDPG can be computed in time $\mathcal{O}(nn' T_{\kappa_V} + mm' T_{\kappa_E})$, where T_{κ_V} and T_{κ_E} is the runtime to compute node and edge kernels, respectively. Note that in case of a sparse node kernel $|V_P| \ll nn'$. Our algorithm compares two edges by κ_E only in case of matching end vertices, therefore in practice the runtime to compare edges might be considerably less than suggested by the analysis above. We show this empirically in Sec. VII.

Algorithm 1: k-walk kernel with implicit mapping

Input : Graphs G, H; kernels κ_V, κ_E, length k.
Output: Value $K_k^=(G, H)$ of the k-walk kernel.

```
(V_P, E_P, w) ← WDPG(G, H, κ_V, κ_E)
forall the v ∈ V_P do
    r^0(v) ← w(v)                          ▷ Initialization
for i ← 1 to k do
    forall the v ∈ V_P do
        r^i(v) ← 0
        forall the e = vu ∈ E_P do
            r^i(v) ← r^i(v) + w(v) · w(e) · r^{i-1}(u)
return Σ_{v∈V_P} r^k(v)
```

In order to compute the k-walk kernel we want sum up the weights of each walk, i.e., the product of node and edge weights. Let κ_W be defined according to Eq. (3), then we can formulate the k-walk kernel recursively according to

$$\begin{aligned} K_k^=(G, H) &= \sum_{w \in \mathcal{W}_k(G)} \sum_{w' \in \mathcal{W}_k(H)} \kappa_W(w, w') = \sum_{v \in V(G \times H)} r^k(v) \\ r^i(v) &= \sum_{uv \in E(G \times H)} w(v) \cdot w(vu) \cdot r^{i-1}(u) \\ r^0(v) &= w(v) \quad \forall v \in V(G \times H). \end{aligned}$$

Note that r^i can easily be formulated as matrix-vector product. Since the direct product graph tends to be sparse we present a graph-based approach for computation, see Algorithm 1.

Theorem 4. *Let $n_P = |V_P|$, $m_P = |E_P|$. Algorithm 1 computes the k-walk kernel in time $\mathcal{O}(n_P + k(n_P + m_P) + T_{\text{WDPG}})$, where T_{WDPG} is the time to compute the WDPG.*

2) Explicit mapping: The explicit approach to compute the k-walk kernel is only applicable for the kernel κ_W^δ between walks. A straightforward approach would require to enumerate all walks in order to count the label sequences associated with them. We propose a more elaborated approach that exploits the simple composition of walks similar to the approach for weighted walks, cf. Sec. VI-A1. A walk of length i can be decomposed into a walk of length $i - 1$ with an additional edge and node added at the front. This allows to obtain the number of walks of length i with a given label sequence starting at a fixed node v by concatenating $(l(v), l(vu))$ with all label sequences for walks staring from a neighbor u of v. Algorithm 2 provides the pseudo code of this computation.

Theorem 5. *Let $n = |V(G)|$ and $m = |E(G)|$. Algorithm 2 computes $\phi_k^=(G)$ in time $\mathcal{O}(n + k(n + m)l)$, where l is the maximum number of different label sequences of $(k-1)$-walks staring at a node of G.*

Assume Algorithm 2 is applied to unlabeled sparse graphs, i.e., $|E(G)| = \mathcal{O}(|V(G)|)$, then $l = 1$ and the feature map can be computed in time $\mathcal{O}(n + kn)$.

Algorithm 2: Explicit computation of ϕ for k-walk kernel

Input : Graph G, length k.
Output: Feature map $\phi(G)$ of label sequences and counts associated with length k walks in G.
Data : Feature counts $\phi_v^i : \Sigma^{2i+1} \to \mathbb{N}$ of label sequences associated with i-walks starting at v.

forall the $v \in V(G)$ **do**
  $\phi_v^0(l(v)) \leftarrow 1$ ▷ Initialization
for $i \leftarrow 1$ **to** k **do**
  forall the $v \in V(G)$ **do**
    forall the $e = vu \in E(G)$ **do**
      forall the w *with* $\phi_u^{i-1}(w) > 0$ **do**
        $w' \leftarrow (l(v), l(e)) \cdot w$ ▷ concatenate
        $\phi_v^i(w') \leftarrow \phi_v^i(w') + \phi_u^{i-1}(w)$
return $\bigcup_{v \in V(G)} \phi_v^k$ ▷ combine vectors

B. Application to Shortest-Path Kernels

To compute the shortest-path (SP) kernel, cf. Eq. (1), essentially two steps are performed. For each graph G of the data set the complete graph G' with node set $V(G)$ is generated, where an edge uv is annotated with the length of a shortest-path between u and v. Originally, the shortest-path kernel is then defined as the 1-walk kernel between these complete graphs [5]. Note that κ_E then compares shortest-path length and can, for example, be realized by a Brownian Bridge kernel [5] or a Dirac kernel. Vertices may be annotated with discrete label or continuous attributes and are compared by κ_V. Therefore, our two approaches to compute walk kernels directly yield efficient explicit and implicit computation schemes for the SP kernel. Clearly, the explicit version is more efficient as it avoids the construction of the product graph, however, it is only applicable when shortest-path lengths and node labels are compared by a Dirac kernel.

VII. Experimental Evaluation

We compare implicit and explicit computation schemes experimentally with focus on runtimes. Both approaches consistently produced the same kernel matrix in all experiments. All algorithms were implemented in Java and the default Java HashMap implementation was used to store feature maps, see Sec. III. Experiments were conducted using Java OpenJDK v1.7.0 on an Intel Core i7-3770 CPU at 3.40GHz with 16GB of RAM using a single processor only. The reported runtimes are average values over 5 runs.

For shortest-path and subgraph kernels with size 3 we found that explicit mapping clearly outperforms implicit computation by several orders of magnitude with respect to runtime. This is in accordance with our theoretical analysis and our results suggest to always use explicit computation schemes for these kernels whenever applicable. Due to space limitations we only report on the results for walk kernels in detail, where we observe a computational phase transition. The discussion of runtimes for walk kernels in Sec. VI-A suggested that (1) implicit computation benefits from sparse node and edge kernels, (2) explicit computation is promising for graphs with a uniform label structure, which exhibit few different features, and then scales to larger data sets.

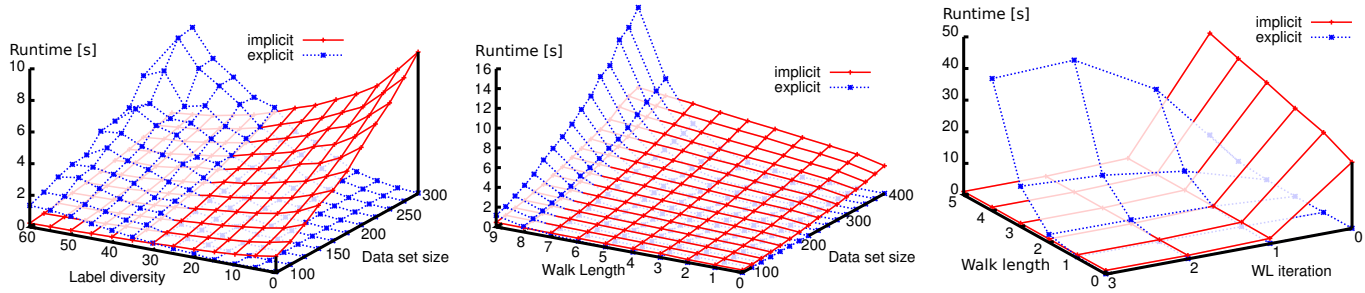
Synthetic data sets: In order to systematically vary the label diversity we generated synthetic graphs by the following procedure: The number of vertices was determined by Poisson distribution with mean 20. Edges were inserted between a pair of vertices with probability 0.1. The label diversity depends on the parameter p_V. Edges were uniformly labeled; a node obtained the label 0 with probability $1 - p_V$. Otherwise the labels 1 or 2 were assigned with equal probability. In addition, we vary the data set size N between 100 and 300 adding 20 randomly generated graphs in each step. We analyze the runtime to compute the $N \times N$ kernel matrix. For the product graph based computation κ_V and κ_E are Dirac kernels.

The results are depicted in Fig. 1(a), were a label diversity of 50 means that $p_V = 0.5$. The runtime for implicit computation increases with the data set size and decreases with the label diversity. This observation is in accordance with our hypotheses. When the label diversity increases, there are less compatible pairs of vertices and the product graph becomes smaller. Consequently, computation of the product graph and the counting of weighted walks require less runtime. For explicit computation we observe a different trend: While the runtime increases with the size of the data set, the approach is extremely efficient for graphs with uniform labels ($p_V = 0$) and becomes slower when the label diversity increases. Both approaches yield the same runtime for a label diversity of $p_V \approx 0.3$, while for higher values of p_V implicit computation is preferable and explicit otherwise.

Molecular data sets: We use a data set of graphs derived from small molecules (NCI Open Database, GI50, U251; http://cactus.nci.nih.gov) to analyze the runtime on a real-world data set with fixed label diversity. Vertex labels correspond to the atom types and edge labels represent single, double, triple and aromatic bonds, respectively. This time we vary the walk length and the data set size.

Figure 1(b) shows that the runtime of implicit computation heavily depends on the size of the data set. The increase with the walk length is less considerable. This can be explained by the time T_{WDPG} required to compute the product graph that is always needed independent of the walk length. For short walks explicit computation is very efficient, even for larger data sets. However, when a certain walk length is reached the runtime increases drastically. This can be explained by the growing number of different label sequence. Notably for walks of length 8 and 9 the runtime increases significantly with the data set size. This indicates that the time T_{dot} has a considerable influence on the runtime. For walk length up to 7 explicit computation beats implicit computation on the molecular data set.

Weisfeiler-Lehman label refinement: Walk kernels have been successfully combined with label refinement techniques [7]. Weisfeiler-Lehman label refinement (WL) was proposed as a heuristic for the graph isomorphism problem, but has recently been applied to develop graph kernels [12].



(a) Synthetic data set, walk length 7 (b) Molecular data set (c) Enzyme with Weisfeiler-Lehman refinement

Fig. 1. Runtime to generate kernel matrix by implicit and explicit computation of walk kernels.

To further analyze the sensitivity w.r.t. label diversity, we use the Enzyme data set [8], which consists of graphs with three initial node labels, with 0 to 3 iterations of WL, see Fig. 1(c).

If no refinement is applied, the explicit mapping approach beats the product graph based algorithm for the used walk lengths. However, as soon as a single iteration of label refinement is performed, the product graph based algorithm becomes competitive for walk length 0 and 1 and outperforms the explicit mapping approach for higher walk lengths. The runtimes do not change significantly for more iterations of refinement. This indicates that a single iteration of WL results in a high label diversity that does not increase significantly for more iterations on the Enzyme data set. When using our walk-based kernel as base kernel of a WL graph kernel [12], our observation suggests to start with explicit computation and switch to implicit after few iterations of refinement.

VIII. Conclusion

The breadth of problems requiring to deal with graph data is growing rapidly and graph kernels have become an efficient and widely-used method for measuring similarity between graphs. Highly scalable graph kernels have recently been proposed for graphs with thousands and millions of nodes, both for graphs with and without node attributes, based on explicit graph feature maps. These recent successes suggests that there is no benefit of the kernel trick – computing feature maps implicitly – when it comes to graphs. In this paper, we have empirically shown that this is not the case. In contrast, already for rather small walk lengths, implicit kernel computation is considerably faster than explicit computation.

To set the stage for the experimental comparison, we actually made several contributions to the theory and algorithmic of graph kernels. We presented a first step towards a unified view on implicit and explicit graph features. More precisely, we gave a sufficient condition for when an explicit mapping from the implicit feature space of convolution kernels to a vector-valued feature space is possible. Using this results, we developed explicit computation schemes for random walk kernels [2], [3], subgraph matching kernels [2], [4], and shortest-path kernels [5]. This finally set the stage for our experimental comparison and its main result, a computational phase transition for walk kernels.

For future work, one should develop explicit computation schemes for other implicit graph kernels and vice versa.

Acknowledgment

This work grew out of discussions within the DFG Collaborative Research Center SFB 876.

References

[1] D. Haussler, "Convolution kernels on discrete structures," University of California, Santa Cruz, CA, USA, Tech. Rep. UCSC-CRL-99-10, 1999.
[2] T. Gärtner, P. Flach, and S. Wrobel, "On graph kernels: Hardness results and efficient alternatives," in *Learning Theory and Kernel Machines*, ser. Lecture Notes in Computer Science, B. Schölkopf and M. Warmuth, Eds. Springer Berlin / Heidelberg, 2003, vol. 2777, pp. 129–143.
[3] S. V. N. Vishwanathan, N. N. Schraudolph, R. I. Kondor, and K. M. Borgwardt, "Graph kernels," *JMLR*, vol. 11, pp. 1201–1242, 2010.
[4] N. Kriege and P. Mutzel, "Subgraph matching kernels for attributed graphs," in *Int. Conf. on Machine Learning (ICML)*, 2012.
[5] K. Borgwardt and H.-P. Kriegel, "Shortest-path kernels on graphs," in *Int. Conf. on Data Mining (ICDM)*, 2005, pp. 74–81.
[6] H. Kashima, K. Tsuda, and A. Inokuchi, "Marginalized kernels between labeled graphs," in *Int. Conf. on Machine Learning (ICML)*, 2003.
[7] P. Mahé, N. Ueda, T. Akutsu, J.-L. Perret, and J.-P. Vert, "Extensions of marginalized graph kernels," in *Int. Conf. on Machine Learning (ICML)*, 2004.
[8] K. M. Borgwardt, C. S. Ong, S. Schönauer, S. V. N. Vishwanathan, A. J. Smola, and H.-P. Kriegel, "Protein function prediction via graph kernels." *Bioinformatics*, vol. 21 Suppl 1, pp. i47–i56, 2005.
[9] Z. Harchaoui and F. Bach, "Image classification with segmentation graph kernels," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2007, pp. 1–8.
[10] N. Shervashidze, S. Vishwanathan, T. Petri, K. Mehlhorn, and K. Borgwardt, "Efficient graphlet kernels for large graph comparison," *Journal of Machine Learning Research - Proceedings*, vol. 5, pp. 488–495, 2009.
[11] T. Horváth, T. Gärtner, and S. Wrobel, "Cyclic pattern kernels for predictive graph mining," in *Proceedings of Knowledge Discovery in Databases (KDD)*, 2004, pp. 158–167.
[12] N. Shervashidze, P. Schweitzer, E. van Leeuwen, K. Mehlhorn, and K. Borgwardt, "Weisfeiler-lehman graph kernels," *JMLR*, vol. 12, pp. 2539–2561, 2011.
[13] J. Ramon and T. Gärtner, "Expressivity versus efficiency of graph kernels," in *First International Workshop on Mining Graphs, Trees and Sequences*, 2003.
[14] M. Neumann, N. Patricia, R. Garnett, and K. Kersting, "Efficient graph kernels by randomization," in *Machine Learning and Knowledge Discovery in Databases (ECML/PKDD)*, 2012, pp. 378–393, long version `arXiv:1410.3314 [stat.ML]`.
[15] A. Feragen, N. Kasenburg, J. Petersen, M. de Bruijne, and K. Borgwardt, "Scalable kernels for graphs with continuous attributes," in *Advances in Neural Information Processing Systems 26*. Curran Associates, Inc., 2013, pp. 216–224.
[16] N. Wale, I. A. Watson, and G. Karypis, "Comparison of descriptor spaces for chemical compound retrieval and classification," *Knowl. Inf. Syst.*, vol. 14, no. 3, pp. 347–375, 2008.