A Tree-Based Kernel for Graphs[*]

Giovanni Da San Martino[†] Nicolò Navarin[†] Alessandro Sperduti[†]

Abstract

This paper proposes a new tree-based kernel for graphs. Graphs are decomposed into multisets of ordered Directed Acyclic Graphs (DAGs) and a family of kernels computed by application of tree kernels extended to the DAG domain. We focus our attention on the efficient development of one member of this family. A technique for speeding up the computation is given, as well as theoretical bounds and practical evidence of its feasibility. State of the art results on various benchmark datasets prove the effectiveness of our approach.

1 Introduction

The increasing availability of data in structured form (strings, trees or graphs) has triggered the development of machine learning techniques able to deal directly with such types of data. Among these, kernel methods have become very popular due to their generalization ability and state of the art performances on many tasks. They require the definition of symmetric, positive semidefinite, similarity functions on the input domain: the kernel functions.

A popular strategy for defining kernel functions for structured data is to decompose them into their constituent parts, and then, for each pair of parts, apply a local kernel [9]. While this strategy has been proved successful for strings and trees [4, 16, 21], it is not directly applicable to graphs because of the computational complexity issues which arise: representing a graph in terms of its subgraphs is not feasible since subgraph isomorphism should be checked for each pair of subgraphs. In [8] it has been demonstrated that, any kernel whose feature space mapping is injective, is as hard to compute as graph isomorphism. Due to this limitation, the available strategies for building kernels are: *i*) restricting the input domain to a class of graphs for which isomorphism can be checked quickly [19]; *ii*) select a priori a set of features, usually corresponding to a specific type of substructure, such as walks [8], paths [2, 5], subtree patterns [15, 20]. The former approach can be applied to a limited type of graphs, the latter tends to have a limited flexibility: when the available kernels are not relevant to the task, a new one has to be designed, but the definition of an efficient symmetric positive semidefinite kernel, corresponding to the desired feature space, can be a hard task. All the above approaches discard information about the original graph and are effective only when the selected features are relevant for the current problem.

In this paper we propose to transform the graphs into simpler structures, i.e. multisets of DAGs, and then extend the definition of a large class of already available kernels for trees to DAGs. Our approach allows the application of the vast literature on kernels for trees, which consists of fast and/or very expressive kernels, to the graph domain. Among all the possibilities, we focus our attention on a class of convolution kernels for trees: we discuss techniques for speeding up the computation of the kernel matrix providing theoretical as well as practical evidence of their effectiveness.

2 Notation

A graph is a triplet $G = (V, E, L)$, where V (alternatively V_G) is the set of nodes ($|V|$ is the number of nodes), E the set of edges and $L()$ a function returning the label of a node. A graph is undirected if $(v_i, v_j) \in E \Leftrightarrow (v_j, v_i) \in E$, otherwise it is directed. A path in a graph is a sequence of nodes $v_1, \ldots, v_n$ such that $v_i \in V$, $1 \leq i < n$ and $(v_i, v_i + 1) \in E$. A cycle is a path for which $v_1 = v_n$; a cycle is even/odd if its number of nodes is even/odd, respectively. A graph is connected if there exists a path connecting each pair of nodes. A connected graph is rooted if exactly one node has no incoming edges. A graph is ordered if the set of neighbours of each node is ordered. A tree is a rooted connected directed acyclic graph where each node has at most one incoming edge. A subtree of a tree T is a connected subset of nodes of T. A proper subtree is a subtree composed by a node and all of its descendants. A subset tree is a subtree with the constraint that, for each node, either all or none of its children are included.

[*]This work was supported by the Italian Ministry of Education, University, and Research (MIUR) under Project PRIN 2009 2009LNP494_005.

[†]Dept. of Mathematics, University of Padova, Italy; {dasan,nnavarin,sperduti}@math.unipd.it

3 Roadmap

The framework we propose for the definition of a kernel for graphs is based on the following steps:

1. Given a graph G, generate one unordered rooted DAG, say DD_v, for each $v \in V_G$. G is then represented by the multiset of unordered rooted DAGs (see Section 4 for details). The generation process must guarantee that isomorphic graphs are represented exactly by the same multiset of DAGs.

2. The kernel for graphs is defined as the sum of the computation of a local kernel for DAGs, over all pairs of DAGs in the multiset. Since we are not aware of any kernel specifically designed for DAGs, our strategy is to extend the definition of tree kernels to the rooted DAG domain.

   (a) While there is a vast literature on kernels for ordered trees, just a few kernel functions for unordered trees are defined. In order to broaden the applicability of our approach we define an ordering between DAG nodes (Section 5.2) such that kernels for ordered trees can be applied, too.

   (b) The definition of the most popular class of kernels for ordered trees, the convolution kernels (Section 5.1), is extended to the DAG domain (Section 5.3). Furthermore, in Section 5.5, we introduce a variant of the Subtree Kernel extended to graphs which is based on limiting the depth of the visits during the Graph-to-Dag decompositions.

   One nice aspect of the dag-kernels proposed in this paper is that, by representing the DAG multiset generated by a graph G via an annotated DAG (BigDAG), where sub-structures shared by DAGs in the multiset are represented only once and their frequency of occurrence is annotated, the computation of the kernel can be sped up for common classes of graphs. We formally characterize the computational gain by providing a general upper bound on the number of nodes of the BigDAG and verify, on popular graph datasets, that the computational gain practically reduce the complexity of our kernels.

4 Decomposition of a Graph into DAGs and Derived Graph Kernels

We start by describing our proposal for decomposing a graph G into $|V_G|$ DAGs, one for each $v \in V_G$. The decomposition is based on a simplification of the graph isomorphism testing proposed by [7]. Given a starting node v_i, the basic idea is to keep those edges belonging to the shortest paths between v_i and any $v_j \in G$. This can be achieved efficiently by performing a breadth-first visit on the graph starting from node v_i together with the following operations:

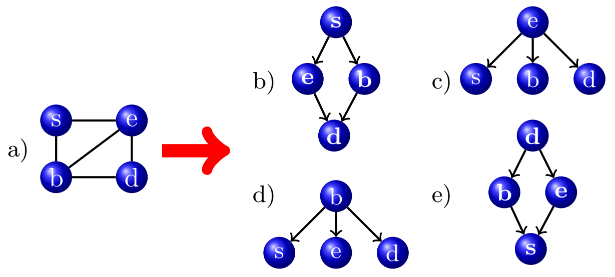

Figure 1: Example of decomposition of a graph a) into its 4 DDs b-e).

1. during the visit a direction is given to each edge;

2. edges connecting nodes visited at level l to nodes visited at level $g < l$ are removed.

The resulting DAG, which is referred to as *Decompositional DAG*, is denoted as $DD_G^{v_i}$. Note that, for every choice of G and v_i, a single *Decompositional Dag* is generated. By repeating the procedure for each node of the graph, $|V|$ DAGs are obtained. Figure 1 gives an example of the application of the algorithm. Note that when the same node is reached multiple times by the visit at the same level then all the edges involved are preserved. For example, when considering the visit at level 2 starting from node **s**, the node **d** is reached simultaneously by edges $(\mathbf{b}, \mathbf{d})$ and $(\mathbf{e}, \mathbf{d})$, and both of them are preserved in the corresponding Decompositional DAG (see Figure 1-b)). On the contrary, edge $(\mathbf{b}, \mathbf{e})$ is not inserted into the corresponding Decompositional DAG: when the visit at level 2 attempts to traverse it (either starting from node **b** or **e**) a node already visited at level 1 (i.e., **e** or **b**, respectively) is reached.

It must be pointed out that, even if some edges are deleted during the decomposition, the fact that the visit is repeated for each node of the graph ensures that every edge e_{ij}, which is not a self-loop, appears at least in one DD_G, i.e. $DD_G^{v_i}$.

THEOREM 4.1. *Given two isomorphic graphs G_1 and G_2, the multisets of Decompositional DAGs associated to G_1 and G_2 is identical.*

Proof. Since G_1 and G_2 are isomorphic with respect to a function h, we analyse the case in which an edge $e_1 \in E_{G_1}$ is added to $DD_{G_1}^{v_1}$ and not to $DD_{G_2}^{h(v_1)}$ (or vice-versa). Other cases are trivial and thus omitted.

The only possible scenario is when $e_{ij} \in E_{DD^{v_1}_{G_1}}$ and $h(e_{ij}) \notin E_{DD^{h(v_1)}_{G_2}}$ (or vice-versa). We prove the theorem by contradicting the thesis. If $h(e_{ij})$ has been deleted there must be a shorter path connecting $h(v_1)$ to $h(v_j)$. Since G_1 and G_2 are isomorphic the corresponding path from $h^{-1}(v_1)$ to $h^{-1}(v_j)$ must be shorter than the one comprising e_{ij}. This contradicts the fact that $e_{ij} \in E_{DD^{v_1}_{G_1}}$, i.e. it belongs to the shorter path connecting v_1 and v_j. $\square$

Note that the DAGs resulting from the decomposition are not ordered.

Let assume a positive definite (PD) kernel K_{DAG} for DAG is available. Then, we can define a kernel for graphs as follows:

$$K_{K_{DAG}}(G_1, G_2) = \sum_{\substack{D_1 \in DD(G_1)\\ D_2 \in DD(G_2)}} K_{DAG}(D_1, D_2) \tag{4.1}$$

where $DD(G)$ is the multiset defined as $\{DD^{v_i}_G | v_i \in V_G\}$. The above kernel is positive semidefinite since it is an instance of the convolution kernel framework [9], where the relation R is defined on $X_1 \times \ldots \times X_{|V|} \times G$, with $X_1 \times \ldots \times X_{|V|}$ being the multiset of Decompositional DAGs obtained from G.

In the following, we define a family of PD kernels for DAGs.

5 Extending Tree Kernels to DAGs

This section shows how to define kernels for DAGs. The proposed strategy is to preprocess the DAGs in order to apply an extended version of a kernel for trees. The motivation for such a choice is twofold: we are not aware of any kernel specifically designed for DAGs and we would like to allow the application of a large class of tree kernels to graphs. Section 5.1 introduces some of the kernels for trees which can be extended to the DAG domain. Since they require the children of the nodes to be ordered, we define an appropriate ordering function in Section 5.2. Finally the extension of tree kernels to the DAG domain is described in Section 5.3, where its consistency is proved.

5.1 Kernels for Trees This section introduces the kernels for trees used in the remainder of the paper. All of them can be reconducted to the convolution kernel framework [9], thus they are instances of the following formula:

$$K(T_1, T_2) = \sum_{v_1 \in T_1} \sum_{v_2 \in T_2} C(v_1, v_2),$$

where $C()$ is a function counting the number of matching substructures in the proper subtrees rooted at nodes v_1 and v_2. By changing the definition of $C()$ many tree kernels can be obtained. As an example, the $C()$ function of the Partial tree (PT) kernel [16], is described in the following. It counts the number of matching subtrees. Its worst-case complexity is $O(p^3|T|^2)$, where p is the maximum out-degree of the two trees. The $C()$ function computing the PT kernel has three cases: *i*) if $L(v_1) \neq L(v_2)$ then $C(v_1, v_2) = 0$; *ii*) if $L(v_1) = L(v_2)$ and both v_1 and v_2 are preterminal nodes, i.e. parent of a leaf node, then $C(v_1, v_2) = \lambda$; *iii*) if $L(v_1) = L(v_2)$ and either v_1 or v_2 is not a preterminal node, then

$$C(v_1, v_2) = \lambda\Big(\mu^2 + \sum_{J_1, J_2, |J_1|=|J_2|} \mu^{d(J_1)+d(J_2)} \cdot \prod_{i=1}^{|J_1|} (1 + C_{PT}(chs_{v_1}[J_{1i}], chs_{v_2}[J_{2i}]))\Big),$$

where $J_{11}, J_{12}, J_{13}, \ldots$ $J_{21}, J_{22}, J_{23}, \ldots$ are sequences of indexes associated with the ordered sequences of children chs_{v_1} and chs_{v_2} respectively, J_{1i} and J_{2i} point to the i-th child in the two sequences and $|J_1|$ denotes the length of the sequence J_1. Finally, $d(J_1) = J_{1|J_1|} - J_{11}$ and $d(J_2) = J_{2|J_2|} - J_{21}$. The parameter λ penalizes matchings between large trees (if a large tree has a match, then all its subtrees have a match) and μ penalizes subtrees built on subsequences of children that contain gaps. Other popular kernels for trees are the Subtree (ST) kernel [21] and the Subset tree (SST) kernel [4]. The ST kernel counts the number of matching proper subtrees between the input trees T_1 and T_2. Its worst-case computational complexity is $O(|T| \log |T|)$. The SST kernel counts the number of matching subset trees. Its complexity is $O(|T|^2)$. Both kernels are instances of the PT kernel.

Two observation are noteworthy: *i*) the formulas in this section, although defined for trees, can be applied to DAGs with no modification; *ii*) all the kernels described in this section require the children of a node to be ordered. Since the DAGs resulting from the decomposition described in Section 4 do not meet this requirement, in the next section we show how to define a canonical ordering for DAG nodes.

5.2 Ordering Dag vertices Decompositional DAGs, obtained according to the procedure described in Section 4 are not ordered. Inspite of the fact that it has been demonstrated that the computation of any tree kernel for unordered trees with subtrees as their substructures is $\sharp P$-complete [11], some kernels for unordered trees have been defined in literature [14, 21]. Such kernels, in order to be tractable, tend to restrict the type of substructures used as features, thus potentially limiting their expressiveness. In order to broaden the applicability of our approach, in this section we define a *strict partial order* among vertices of a DAG which allows us to employ the kernels defined in Section 5.1.

Definition 1. ***Strict partial order relation $\dot{<}$ between DAG vertices***
Let us assume to have a total order between vertex labels (e.g. the lexicographic order). Then the relation $v_i \dot{<} v_j$ holds if one of the following conditions is met:

1. $L(v_i) < L(v_j)$;
2. $[L(v_i) = L(v_j)] \ \wedge\ [outdeg(v_i) < outdeg(v_j)]$;
3. *Assume* $[L(v_i) = L(v_j)] \wedge [outdeg(v_i) = outdeg(v_j)]$ *and the sons* $ch_l[v_i]$ *of* v_i *and the sons* $ch_l[v_j]$ *of* v_j *are partially ordered by the relation we are defining, i.e. they form two partially ordered sets.*
   Then we can define two sequences
   $ch_1[v_i], ch_2[v_i], \dots, ch_m[v_i]$,
   $ch_1[v_j], ch_2[v_j], \dots, ch_m[v_j]$,
   where $m = outdeg(v_i) = outdeg(v_j)$, *and every* $ch_k[v_i]$ *is a (not necessarily unique) minimal element of the set defined as* $\{ch_l[v_i] | l \in \{k, \dots, m\}\}$.
   Then the relation holds if:
   $[\exists l. ch_l[v_i] \dot{<} ch_l[v_j]] \ \wedge$
   $[\neg \exists k < l. (ch_k[v_i] \dot{<} ch_k[v_j] \vee ch_k[v_j] \dot{<} ch_k[v_i])]$;

From the above definition it is not difficult to show that

Lemma 5.1. $\neg(v_j \dot{<} v_k) \wedge \neg(v_k \dot{<} v_j)$ *if and only if the following conditions are simultaneously verified*

i) $L(v_j) = L(v_k)$ *(otherwise 1. applies);*

ii) $outdeg(v_j) = outdeg(v_k)$ *(otherwise 2. applies);*

iii) $\forall l, \neg(ch_l[v_j] \dot{<} ch_l[v_k]) \wedge \neg(ch_l[v_k] \dot{<} ch_l[v_j])$ *(otherwise 3. applies).*

Let us now define a mapping from vertices to alphanumeric strings as follows:

Definition 2. *Given a vertex* v, *the following alphanumeric string* $S(v)$ *is associated to it:*

$$S(v) = L(v) \cdot outdeg(v) \cdot \left(\prod_{i=1}^{outdeg(v)} S(ch_i[v]) \right)$$

where '$\cdot$' is the concatenation operator and '$\prod$' represents the application of '$\cdot$' to an indexed sequence of arguments; moreover, the order adopted for the children of a node is the same order defined in Definition 1.

By exploiting Definition 1, Definition 2, and the recursive application of Lemma 5.1, we can now prove the following lemma

Lemma 5.2. *Let $>$ and $=$ be the natural relations of order and equivalence between alphanumeric strings, then the following statements are true*

i) $\neg(v_j \dot{<} v_k) \wedge \neg(v_k \dot{<} v_j)$ *if and only if* $S(v_j) = S(v_k)$.

ii) $v_j \dot{<} v_k$ *if and only if* $S(v_j) < S(v_k)$;

Proof. i): if $\neg(v_j \dot{<} v_k) \wedge \neg(v_k \dot{<} v_j)$ then $S(v_j) = S(v_k)$ by recursive application of Lemma 5.1; if $S(v_j) = S(v_k)$ then neither $v_j \dot{<} v_k$ nor $v_k \dot{<} v_j$ can hold, otherwise there will exist a position where $S(v_j)$ and $S(v_k)$ differ, which is in contradiction with the hypothesis.
ii): Let us assume that $v_j \dot{<} v_k$ then if 1. or 2. applies, we have $[L(v_j) < L(v_k)] \vee [outdeg(v_j) < outdeg(v_k)]$ which, by Definition 2, entails that $S(v_j) < S(v_k)$; if 3. must be applied, then $S(v_j) = L \cdot m \cdot (\prod_{i=1}^{m} S(ch_i[v_j]))$ and $S(v_k) = L \cdot m \cdot (\prod_{i=1}^{m} S(ch_i[v_k]))$, where $L = L(v_j) = L(v_k)$ and $m = outdeg(v_j) = outdeg(v_k)$. Moreover, let l be the value of the index for which 3. applies. Then, by *i)*, we have $\prod_{i=1}^{l-1} S(ch_i[v_j]) = \prod_{i=1}^{l-1} S(ch_i[v_k])$ and the above arguments can be recursively applied to $ch_l[v_j] \dot{<} ch_l[v_k]$, eventually till only vertices with 0 out-degree are reached. For those vertices, only 1. or 2. may apply, which leads to the desired result.
Viceversa, if $S(v_j) < S(v_k)$, let q be the first position where the two strings differ, i.e. $pos(q, S(v_j)) < pos(q, S(v_k))$, then it is easy to see that that position corresponds to the first application of 1. (i.e. $pos(q, S(v_j))$ and $pos(q, S(v_k))$ are labels) or 2. (i.e. $pos(q, S(v_j))$ and $pos(q, S(v_k))$ are out-degrees) after having only applied 3. (possibly 0 times if $q < 3$), which entails that $v_j \dot{<} v_k$. □

Theorem 5.1. *The relation just defined is a strict partial order between DAG vertices.*

Proof. A strict partial order is a relation on a set P that is irreflexive, asymmetric and transitive. In the following each property is proven.

- *Irreflexivity*: $\neg(v_i \dot{<} v_i)$, i.e. no condition in Definition. 1 is satisfied by the pair (v_i, v_i): since v_i is mapped into $S(v_i)$, Lemma 5.2.*ii)* applies and thus $\neg(v_i \dot{<} v_i)$;
- *Asymmetry*: $(v_i \dot{<} v_j \Rightarrow \neg v_j \dot{<} v_i)$. By Lemma 5.2, if $v_i \dot{<} v_j$ then $S(v_i) < S(v_j)$, which is not consistent with the condition $S(v_j) < S(v_i)$ required by $v_j \dot{<} v_i$.
- *Transitivity* $(v_i \dot{<} v_j \ \wedge\ v_j \dot{<} v_k \ \Rightarrow\ v_i \dot{<} v_k)$. By Lemma 5.2, if $v_i \dot{<} v_j \wedge v_j \dot{<} v_k \Rightarrow S(v_i) < S(v_j) \wedge S(v_j) < S(v_k)$, which implies that $S(v_i) < S(v_k) \Rightarrow v_i \dot{<} v_k$. □

In the following, we will denote the Decompositional Dag ordered according to the relation defined in this section as $ODD_G^{v_i}$. The time complexity of the ordering phase is $O(|V|p\log p)$ since, for every node, we have to order its p children.

The reason for defining the ordering according to Definition 1 is that it must be efficient to compute and must ensure that the swapping of non comparable nodes does not affect the feature space representation of the DAG. After introducing the extension of tree kernels to DAGs in Section 5.3, we will show that our ordering meets both constraints. The remainder of this section introduces concepts and prove results needed in Section 5.3.

Let a tree visit be a function $T(v_i)$ that, given a node v of a $ODD_G^{v_i}$, returns the tree resulting from the visit of the DAG starting in v_i. Figure 2 gives an example of tree visits. Note that, in Figure 2-a), the tree resulting from the visit starting in node **s** has two leaf nodes **d**, while the $T()$ related to the DAG of Figure 2-b) has associated twice the tree composed of only the leaf node **d**. Moreover, note that the set of tree visits would be different even if the nodes labelled with **d** were not leaves. The concept of tree visit is used in the following:

THEOREM 5.2. *If $\neg(v_i \dot{<} v_j) \wedge \neg(v_j \dot{<} v_i)$ according to Definition 1, then the trees $T(v_i)$ and $T(v_j)$, obtained as visits of the subdags rooted at v_i and v_j, are identical.*

Proof. if $\neg(v_i \dot{<} v_j) \wedge \neg(v_j \dot{<} v_i)$, by LEMMA 5.2 $S(v_i) = S(v_j)$. Moreover, it is not difficult to see that for any vertex v, $S(v)$ is an unambiguous representation of $T(v)$ where the labels of the visited vertices are presented according to a depth visit of $T(v)$ (remember that vertices are ordered according to the same relation $\dot{<}$) along with the out-degree of each vertex, which allows the correct reconstruction of $T(v)$. Because of that, and since $S(v_i) = S(v_j)$, we deduce that $T(v_i)$ and $T(v_j)$ are identical. □

5.3 Tree-based Kernels for ordered DAGs and Graphs Here we define a family of kernels for ordered DAGs based on tree-kernels. The basic idea is to use tree-visits to project subdags to a tree space and then apply tree kernels on the visits. If we consider the tree kernels defined in Section 5.1, then we can write

(5.2)
$$K_{DAG}(D_1, D_2) = \sum_{\substack{v_1 \in V_{D_1} \\ v_2 \in V_{D_2}}} C(root(T(v_1)), root(T(v_2))),$$

where $T()$ are tree-visits as defined in Section 5.2 and $C()$ is any of the $C()$ functions described in 5.1. Theorem 5.2 ensures that the mapping in feature space

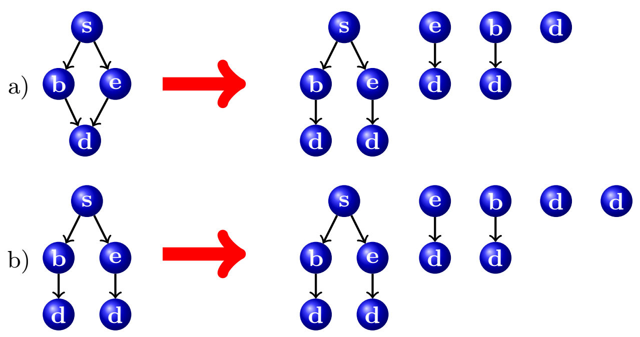

Figure 2: Two DAGs (left) and their associated tree visits $T()$ starting from each node.

of each tree visit is well defined since the swapping of any two non comparable children of a node does not change the resulting tree visit. This, in turn, ensures that: *i)* $\forall i,j$ $C(root(T(ch_i[v_1])), root(T(ch_j[v_2]))) = C(ch_i[root(T(v_1))], ch_j[root(T(v_2))])$, which allows us to significantly reduce the computational burden; *ii)* $C()$ remains a valid local kernel. Finally, the kernel of eq. (5.2) is positive semidefinite since it is an instance of convolution kernel framework [9], where the relation R is defined on $X_1 \times \ldots \times X_{|V|} \times ODD$, with $X_1 \times \ldots \times X_{|V|}$ being the set of tree-visits obtained from ODD. The feature space, i.e. the set of features associated with the kernels of eq. (5.2), coincides with the one of the tree kernel K_T. However, note that a DAG with a "diamond" shape, such as the DAG in Figure 2-a), has a different representation with respect to the DAG of Figure 2-b): while the non-zero features are the same, the feature related to the leaf node **d** occurs once in Figure 2-a), while it occurs twice in Figure 2-b).

After introducing the kernels used in this paper, we can now motivate why the ordering in Section 5.2 have been employed. We recall that the requirements for the ordering are to be efficiently computable and to ensure that the swapping of non comparable nodes does not affect the feature space representation of the DAG. More "standard" orderings would not meet both constraints. For example, a total order between DAG vertices would require an algorithm with GI-complete computational complexity [17]. On the contrary, simple partial orderings, such as the one based on reachability of the nodes, would not allow us to consistently order structurally different subdags. This makes impossible to have a consistent mapping in feature space for any kernel whose features are tree structures, which basically includes the vast majority of tree kernels.

On the basis of the above definition, given a tree-kernel K_T, we can define a kernel K_{K_T} between two graphs G_1 and G_2 as follows:

$$(5.3) \qquad K_{K_T}(G_1,G_2) = \sum_{\substack{D_1 \in ODD(G_1)\\ D_2 \in ODD(G_2)}} K_{DAG}(D_1,D_2)$$

where $ODD(G)$ is the multiset $\{ODD_G^{v_i} | v_i \in V_G\}$, K_{DAG} is defined according to eq. (5.2) and depends on a kernel for trees K_T. In the following, we refer to this class of kernels as ODD kernels.

From a computational point of view, let $Q(n)$ denote the worst-case complexity of K_{K_T}, where $n = \max_{\substack{D_1 \in ODD(G_1)\\ D_2 \in ODD(G_2)}} \{|V_{D_1}|, |V_{D_2}|\}$, then the kernel of eq. (5.3) has time complexity $O(|V_{G_1}||V_{G_2}| \cdot Q(n))$. Thus, in the worst case where $n = \max\{|V_{G_1}|, |V_{G_2}|\}$, using ST, SST, and PT as tree-kernels, leads to a time complexity of $O(n^3 \log n)$, $O(n^4)$ and $O(p^3 n^4)$, respectively.

In Section 5.4 we show how the computation of $K_{K_T}(G_1,G_2)$ can be optimized.

5.4 Speeding up the kernel This section shows how to speed up the computation of eq. (5.3) when K_T is a convolution kernel where the input tree is decomposed into proper subtrees. All the kernels described in Section 5.1 and the vast majority of tree kernels defined in literature meet this constraint. The strategy for speeding up kernel computation is based on avoiding to recompute $C()$ values for identical proper subtrees appearing in different DAGs. The $|V_G|$ Decompositional DAGs generated according to the procedure described in Section 4 can be represented by a single Annotated DAG, that we call BigDAG, where each node is annotated with the frequency of appearance of the substructure rooted at v in all the ODDs of the graph. An example of BigDAG construction is shown in Figure 3. The algorithm for computing the BigDAG can be found in [1]. Its complexity is $O(|V_G|^2 \log |V_G|)$, thus it does not affect the worst-case complexity of any of the kernels considered in the paper. The kernel of eq. (5.3) can be rewritten as

$$(5.4) \qquad K_{BigDAG}(G_1,G_2) = \sum_{\substack{u_1 \in V(BigDAG(G_1))\\ u_2 \in V(BigDAG(G_2))}} f_{u_1} f_{u_2} C(u_1,u_2),$$

where f_u is the frequency of the ordered DAG rooted at u in $ODD(G)$. We want to stress the fact that the BigDAG does not loose any information about the ODDs, thus eq. (5.4) and eq. (5.3) are equivalent. The speed up due to the BigDAG depends on the number of identical substructures found in the ODDs. We now turn our attention to the derivation of an upper bound on the number of nodes of the BigDAG in terms of the in-degree of the nodes and the size of the cycles in the original graph G. We first derive the bound for a specific class of graphs and then extend it to general graphs.

Without loss of generality, we will assume in the following to deal with connected and undirected graphs. Let's consider a polytree P, i.e. a graph for which at most one undirected path exists between any two nodes. Let us consider any node $v_i \in P$ with degree p_{v_i}. The procedure for obtaining a Decompositional DAG, when applied to all the nodes of P, generates exactly $p_{v_i}+1$ different patterns of connectivity for v_i. In fact, since P is a polytree, only 1 edge at a time can be entering v_i while all the remaining edges are outgoing. Moreover, we have to consider the case where v_i is the starting node for the visit and all the edges are outgoing. These considerations are valid for all nodes belonging to P, thus BigDAG will exactly have $\sum_{i=1}^{|P|}(p_{v_i}+1)$ nodes.

Let's now turn our attention to general graphs. Given a graph G we can represent it as a polytree $P(G)$ by iteratively grouping the nodes forming local cycles into a single node $v \in P(G)$ until no more cycles can be grouped (see Figure 4 for an example). Let us define n_{v_i} as the number of nodes of G represented by v_i in $P(G)$ and $q = |P|$. The values p_{v_i} represent now the sum of the incoming edges of all the nodes represented by v_i. Then, the bound can be computed by applying the same reasoning for polytrees, the only difference being that v_i contains n_{v_i} choices for starting the visit from a node "inside" v_i, thus generating n_{v_i} different substructures with, in the worst case of a single cycle involving all the nodes of G represented by v_i, a total of n_i^2 nodes added to the BigDAG. The number of nodes added to the BigDAG is thus bounded by:

$$(5.5) \qquad |BigDAG(G)| \leq \sum_{i=1}^{q}(p_{v_i} + n_{v_i}^2).$$

Note that $2|E(G)| + |V_G| \leq |BigDAG(G)| \leq |V_G|^2$, being the worst case when there's only one cycle of length $|V_G|$. Let us define the length of the longest cycle

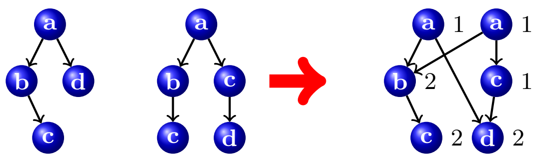

Figure 3: Two DAGs (left) and their associated *BigDAG*. Numbers on the right of nodes represent the frequencies.

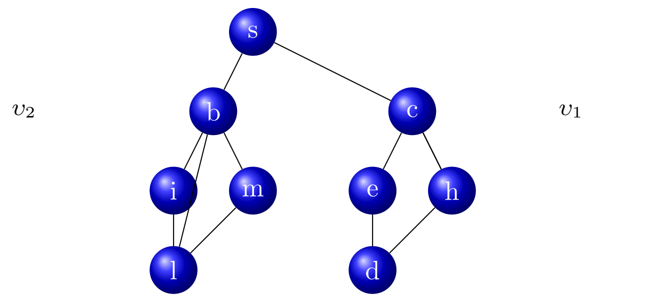

Figure 4: Example of a polytree representing a graph with two "complex" polytree nodes v_1 and v_2 representing local cycles.

in G to be o. The bound then becomes $o|V_G| + \sum_i p_{v_i}$, since $\sum_i n_{v_i} = |V_G|$. Note that $\sum_i p_{v_i} \le 2q$ otherwise $P(G)$ wouldn't be a polytree. Just to give an example, if $o = |V_G|^{\frac{1}{2}}$, then $|BigDAG(G)| \le 2q + \sum_i n_{v_i} n_{v_i} \le 2q + |V_G|^{\frac{1}{2}} \sum_i n_{v_i} = 2q + |V_G|^{\frac{3}{2}}$.

Note that if the number of nodes in the BigDAG is $O(|V_G|^{\frac{3}{2}})$, then the complexity of the kernels reduces significantly, for example the application of the subset tree kernel would have a complexity of $O(|V_G|^3)$, thus reducing the complexity of a factor $|V_G|$. Section 10 will discuss the reduction due to the BigDAG on benchmark datasets.

5.5 Limiting the Depth of the Visits It is a common practice in many graph kernels (see Section 8 and the references therein) to somehow limit the features that are generated. We apply this approach by limiting the depth of the visits during the generation of the multiset of DAGs when the tree kernel employed is the *subtree kernel*. Our aim is to

1. further reduce the computational complexity of the kernel when large graphs are involved;

2. add features that otherwise would be discarded if only unlimited visits were performed.

The method is implemented as follows. Given a graph G, for any $j \in \{0..h\}$ and for any $v_i \in V_G$, an ODD is generated with the additional constraint that the maximum depth is at most j (we will refer to it as $ODD_G^{v_i,j}$). All the $ODD_G^{v_i,j}$ are merged together to form the $BigDAG(G)$ (see Section 5.4). The kernel definition remains the same as in eq. (5.3), the only difference being that $ODD(G)$ is now replaced by $\{ODD_G^{v_i,j} | v_i \in V_G, j \in \{0 \dots h\}\}$. This kernel will be referred to as K_{ODD-ST_h} in the following.

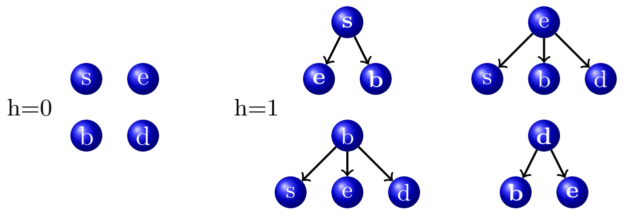

Figure 5: Example of decomposition of the graph shown in Figure 1-a) into its DDs with visits up to depth $h = 1$.

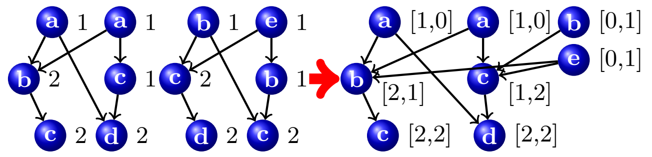

Figure 6: Two *BigDAGs* (left) and their associated *Big*^{2}*DAG*. Arrays on the right of nodes represent the frequencies associated to each example.

While new features are introduced when the *subtree kernel* is employed and the visits are limited, all DAGs generated by visits of depth $j > h$ are lost (see Figure 5).

It is not difficult to see that, given a node, in the worst case no more than $O(p^h)$ nodes are generated by all the visits up to depth h. In fact, the total number of generated nodes is

$$H = \sum_{j=0}^{h} (h+1-j)p^j = \frac{(h+1) - (h+2)p + p^{h+2}}{(1-p)^2}.$$

So the *BigDAG* of a graph cannot have more than nH nodes[1]. The creation of a *BigDAG* is thus dominated by an $O(nH \log(nH))$ term. Since H can be considered constant, the complexity is $O(n \log n)$, which matches the complexity of the kernel when using the *subtree kernel*.

6 Speeding up the kernel matrix computation

When using a kernel method, such as the SVM, the tuning of the hyperparameters through a validation set or through cross-validation is common practice. If the kernel function depends on multiple parameters, then it is often more efficient to precompute the kernel matrix of the dataset than computing the kernel values on demand. Other kernel methods, e.g. KPCA, require the full kernel matrix to be computed.

[1]It is worth to notice that the bound is not tight, since the same leaf nodes are generated in turn by visits of neighbours nodes with depth 0 and 1.

In this section, we discuss how the computation of the full kernel matrix, especially when using the *subtree kernel*, can be optimized. Specifically, we observe that:

1. the BigDAG idea used to avoid multiple calculations of kernel values for the same structures in different ODDs, can be extended to the whole training set avoiding multiple calculations of the kernel values for the same structures in different examples.

2. If the *subtree kernel* is used, the method we propose yields an explicit (and compact) feature space representation of the kernel; this fact leads to a very efficient implementation.

Let's assume that the training set contains M graphs. Following the same idea introduced in Section 5.4, we can get a compact representation of all the $BigDAGs$ coming from the M graphs. This way, we can avoid to compute multiple times the contribution to the kernel of substructures shared by several graphs. Note that these substructures *should* exist, otherwise the kernel would be extremely sparse for the considered training set, and not worth to be used for learning.

The idea is implemented as follows. An annotated DAG, we call it Big^2DAG, is created starting from all the $BigDAGs$ generated by the training set, where each different structure is represented only once. A (sparse) vector F_{v_i}, representing the frequency of the structure rooted in that node in all graphs, is associated to each node of the Big^2DAG. For example, $F_{v_i}[j]$ represents the frequency of structure rooted in node v_i in $BigDAG(G_j)$. Figure 6 shows an example of Big^2DAG construction.

As mentioned above, the Big^2DAG is an explicit representation of the feature space of the *subtree kernel* described in Section 5.1. In fact, we can notice that the frequency associated to each node in $BigDag(G_i)$ is the frequency of the proper subtree rooted at that node in all the ODDs related to the graph G_i. In the same way, the vector F_{u_i} represents the frequencies of the proper subtree rooted in node u_i in the various training graphs. So with this new representation, eq. (5.4) can be rewritten as:

(6.6)
$$K_{Big^2DAG}(G_i, G_j) = \sum_{u_1,u_2 \in V(Big^2DAG)} F_{u_1}[i] * F_{u_2}[j] C(u_1, u_2)$$

If the *subtree kernel* is used, there is a match only between identical subtrees, that is $C(u_1, u_2) \neq 0 \iff T(u_1) = T(u_2)$, and eq. (6.6) can be rewritten as:

$$K_{Big^2DAG}(G_i, G_j) = \sum_{u \in V(Big^2DAG)} F_u[i] * F_u[j] C(u, u) \tag{6.7}$$

Exploiting eq. (6.7) leads to a very efficient implementation. In fact, the whole kernel matrix can be computed with a single scan of the Big^2DAG nodes.

7 An efficient implementation using Hash tables

In the previous section we discussed how the Big^2DAG can improve the speed of the kernel matrix computation. In this section we discuss an hash-based implementation that allows us to not explicitly store in memory the dag-structure of the Big^2DAG. Not surprisingly, we can do that when the Big^2DAG constitutes an explicit representation of the *subtree kernel* feature space. In that case, each node u of the Big^2DAG represents a specific feature (tree) occurring at least once in the dataset, i.e. u is the root of a (limited) tree visit of some vertex of a graph of the dataset. Associated to u is the frequency vector $F_u[\cdot]$ reporting for each graph of the dataset how frequent the feature is, and the size of the tree rooted in u to be used in conjunction with the *lambda* parameter. This information is sufficient to compute the kernel.

The basic idea is to use an hash table to efficiently store and retrieve this information for all the features. The frequency values and the sizes are updated incrementally during the generation of the tree visits. Specifically, each vertex u is mapped into a tuple $< D_u, F_u[\cdot], ID_u >$ where D_u is the size of the subtree rooted at vertex u, $F_u[\cdot]$ is the (sparse) vector of frequencies in the dataset, and ID_u is a unique id associated to each tuple. The mapping is created by using as access key for u the string $L(u)\left(ID_{ch_1[u]}, \ldots, ID_{ch_{outdeg(u)}[u]}\right)$. Of course, if $outdeg(u) = 0$, the key will only be constituted by the associated label $L(u)$, while if $outdeg(u) > 0$ the key will also include the ids of the children of u. Using an inverse topological order for the vertices does guarantee that each feature gets a unique id. Moreover, if all the visits up to depth h must be generated, it is sufficient to perform only the visit at depth h and for each vertex u at level $k \leq h$ all the required $h-k+1$ features can be generated by using the features already generated for the children of u. For each generated feature, it is checked if it is already present into the hash table. If it is not, a new feature is inserted, otherwise the $F_u[\cdot]$ field is updated accordingly. It must be noticed that all the needed information is collected at once by performing (single) visits at depth h for each vertex of each graph in the dataset.

8 Related Work

This section introduces those kernels for graphs which are used for comparison in the experimental section or may look related to the ones proposed in the paper. In the latter case, the differences are discussed. Given the generally high-dimensionality of graph data, they have been used in learning tasks either after a preprocessing phase aimed at selecting possibly relevant features, or, in the context of kernel methods, by using tractable kernel functions.

Generally speaking, the methods following the first approach extract frequent patterns, build a vectorial representation of the graphs according to such patterns and then apply a kernel method. When the kernel method is an SVM, the approach is referred to as SVM with frequent pattern mining (freqSVM). The techniques for extracting the features include Gaston [13], Correlated Pattern Mining (CPM) [3], MOLFEA [10]. Saigo et al. [18] proposed gBoost, a mathematical programming boosting method that progressively collects informative (according to the target output) patterns.

The second approach includes a set of kernel functions for graphs. The Marginalized Graph Kernel (MGK) considers common walks as features [12]. The Shortest Path Kernel associates a feature to each pair of nodes of one graph. The value of the feature is the length of the shortest path between the corresponding nodes in the graph [2]. The complexity of the kernel is $O(|V|^4)$. Being the Shortest Path Kernel based on paths, it can be represented as an instance of eq. (4.1). The Fast Subtree Kernel counts the number of identical subtree patterns obtained by breadth-first visits where each node can appear multiple times [20]. The complexity of the kernel is $O(|E|h)$, where h is the, a priori selected, depth of the visit. While being fast to compute, the kernel may lack of expressiveness for some tasks given that the number of non-zero features generated by one graph is at most $|V|h$. Note that, during the visit, any node can be traversed more than once, which makes the kernel not reproducible from eq. (4.1). Costa and De Grave [5] extended the Fast Subtree Kernel by computing exact matches between pairs of subgraphs with controlled size and distance. The complexity of the kernel is $O(|V||V_h||E_h|\log|E_h|)$, where $|V_h|$ and $|E_h|$ are the number of nodes and the number of edges of the subgraph obtained by a breadth-fist visit of depth h. The authors state that, for small values of the subgraph size and distance, the complexity of the kernel becomes practically linear. Mahé and Vert [15] described a graph kernel based on extracting tree patterns from the graph. The difference with the approach of this paper is that the tree patterns are obtained as result of walks on the graph, i.e. the same node can appear more than once in the same tree pattern. The complexity of the kernel is $O(|V_1||V_2|hp^{2p})$, where h is the depth of the visit.

Table 1: Statistics of MUTAG, CAS, CPDB, AIDS and NCI1 datasets: number of graphs, percentage of positive examples, average number of atoms, average number of edges.

Dataset	graphs	pos(%)	avg atoms	avg edges
Mutag	188	66.48	45.1	47.1
CAS	4337	55.36	29.9	30.9
CPDB	684	49.85	14.1	14.6
AIDS	1503	28.07	58.9	61.4
NCI1	4110	50.04	29.87	32.3

9 Experimental results

While eq. (5.3) can be instantiated with any tree kernel, in order to reduce the time required for the experimentation, only one kernel of this class is considered in the following: the subtree kernel with visits of limited depth, i.e. K_{ODD-ST_h}. The reason for such choice lies in the fact that previous sections heavily focused on K_{ODD-ST_h}, especially from the computational point of view.

The kernel has been tested on five datasets: Mutag [6], CAS[2], CPDB [10], AIDS [23] and NCI1 [22]. All datasets involve chemical compounds and represent binary classification problems. Mutag, CAS, CPDB are mutagenicity datasets, while AIDS and NCI1 are, respectively, antiviral and anti-cancer screen datasets. Table 1 summarizes the statistics of the datasets.

The proposed kernel, as each of the kernel functions in the following, was employed together with a Support Vector Machines. $ODD-ST_h$ has been compared against the Fast Subtree Kernel (FS) and gBoost. In addition, results from the following algorithms, when available, are provided from [18] and [20]: Gaston, Correlated Pattern Mining (CPM), MOLFEA, Marginalized Graph Kernel (MGK) and SVM with frequent pattern mining (freqSVM). For the sake of comparison with the above mentioned results, the accuracy of each algorithm was measured by selecting a posteriori the best parameters among the results of a 10−fold cross validation. The values of the parameters of the $ODD-ST_h$ kernel have been restricted to: $\lambda = \{0.1, 0.2, \ldots, 2.0\}$, $h = \{1, 2, \ldots, 10\}$. The parameter selection procedure for the Fast Subtree Kernel replicates the one described in [20], i.e. only the parameter $h = \{1, 2, \ldots, 10\}$ is optimized. gBoost has many parameters, but following the parameter selection process exposed in [18], only μ (that controls training accuracy) has been optimized among the values $\{0.01, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6\}$. For

[2]http://www.cheminformatics.org/datasets/bursi

Table 2: Average accuracy results (when available) for Gaston, MOLFEA, Correlated Pattern Mining, Marginalized Graph Kernel, SVM with Frequent Mining, gBoost, the Fast Subtree and the ODD-ST kernels obtained on MUTAG, CAS, CPDB, AIDS and NCI1 datasets. The executions performed by us report between brackets the optimal values of the parameters.

Kernel	Mutag		CAS		CPDB		AIDS		NCI1	
	Acc	Rank	Acc	Rank	Acc	Rank	Acc	Rank	Acc	Rank
Gaston	-	-	0.79	5	-	-	-	-	-	-
MOLFEA	-	-	-	-	-	-	0.785	5	-	-
CPM	-	-	0.801	4	0.760	6	0.832	2	-	-
MGK	0.808	4	0.771	7	0.765	4	0.762	7	-	-
freqSVM	0.808	4	0.773	6	0.778	3	0.782	6	-	-
gBoost	0.852	3	0.825	2	0.788	2	0.802	3	0.708	3
									(μ=0.01)	
FS	0.893	1	0.816	3	0.763	5	0.791	4	0.863	1
	(h=1, c=10000)		(h=3, c=0.1)		(h=1, c=1)		(h=9, c=0.01)		(h=8, c=0.01)	
$ODD-ST_h$	0.878	2	0.842	1	0.804	1	0.835	1	0.853	2
	(h=2, λ=0.5, c=1000)		(h=4, λ=1.2, c=100)		(h=8, λ=1.2, c=10)		(h=5, λ=1.8, c=10)		(h=5, λ=1.4, c=100)	

the parameters of the other methods, please refer to the corresponding papers. Table 2 reports the average accuracy and the ranking obtained by the considered methods on the five datasets. From the experimental results, it seems that gBoost tends to have a stable ranking, i.e. between the second and third position. On the contrary, the Fast Subtree, while placing first on two datasets, has a lower ranking, i.e. third, fourth and fifth, on the other datasets. $ODD-ST_h$ has best accuracy in three out of five datasets. On the remaining datasets it always places in second position.

We next report the average computational time for a single fold with the optimal parameters on CAS, since it is the largest dataset. The $ODD-ST_h$ kernel took 86 seconds to compute the kernel matrix and 2.4 seconds for the SVM to converge. FS kernel took 282 seconds for the computation of the kernel matrix, and 4.2 seconds for the training of the SVM. gBoost took approximately 4000 seconds. All the experiments are performed on a PC with two Quad-Core AMD Opteron(tm) 2378 Processors and 64GB of RAM.

10 Discussion

The generally good results of the $ODD-ST_h$ kernel reported in Section 9 may be attributed to the fact that it has associated a large feature space, which improves its adaptability to different datasets. We now analyse when the mapping in feature space is not injective. This is not a requirement for a learning algorithm, but can give insights on the limitations of each method: for example, two graphs having the same representation in feature space but different output targets, cannot be discriminated. In the following, the analysis will focus on gBoost, the Fast Subtree and the proposed $ODD-ST_h$ kernel. Specifically, we are going to present some cases in which $\phi(G_1) = \phi(G_2)$ having $G_1 \neq G_2$. Since gBoost extracts only a set of informative subgraphs, it is sufficient to consider $G_1 = \{g\} \cup \{g_1\}$ and $G_2 = \{g\} \cup \{g_2\}$, where g is an informative subgraph and g_1, g_2 are not. Clearly $\phi(G_1) = \phi(G_2) = \phi(g)$.

When considering the Fast Subtree Kernel, it can be observed that, in order for two graphs to get the same representation in feature space, they need to have the same number of vertices and the same statistics on the out-degree of vertices. However, K_{ODD-ST_h} also requires the same distribution on the length of all the pairwise shortest paths. This is not the case for the Fast Subtree Kernel. Figure 7 shows an example of two graphs having the same feature space representation. Note that K_{ODD-ST_h} is able to discriminate those two graphs: the bottom one has a shorter path of length 4, while the longest one for the upper graph has length 3. This implies that there are at least two different tree visits.

Let us turn our attention to the $ODD-ST_h$ kernel. Clearly, two graphs differing just in self-loops, yield identical multisets of DAGs. However, the information about the presence of self-loops could be transferred into the representation of the corresponding node, for example by adding a field to the representation of the label. A less trivial example of non-injective mapping for the $ODD-ST$ kernel is shown in Figure 8.

Let us now turn our attention to the efficiency of the proposed $ODD-ST_h$ kernel. In Section 5.4 we have

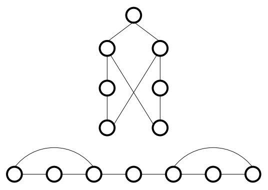
Figure 7: An example of two different graphs having the same representation in feature space according to the Fast Subtree kernel. All the nodes have the same label.

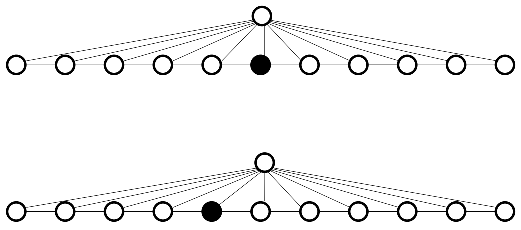
Figure 8: An example of two different graphs having the same representation in feature space according to the ODD kernels. All the nodes have the same label except the black one.

proposed a technique for speeding up the kernel computation by compactly representing the ODDs by means of an Annotated DAG (BigDAG). The complexity of the DAG kernels, for example the one of K_{ODD-ST} and K_{ODD-ST_h}, depends on the number of nodes of the BigDAG. In order to evaluate the actual complexities we have computed, for the K_{ODD-ST_h} kernel, the number of nodes in each BigDAG on the NCI1 dataset. The obtained figures are plotted in Figure 9 as a function of the nodes of the graphs. We have also plotted the polynomial function interpolating these points. In order to have a comparison with the values of the bounds discussed in Section 5.4, the functions $n^{\frac{3}{2}}$ and n^2 are plotted as well. The number of nodes inserted in the BigDAG are for the vast majority of points under the curve $n^{1.5}$. The size of the BigDAG tends to an asymptotic value with the number of nodes contained in the input graphs (see Section 5.5). In this case, the "actual" complexity of the kernel is proportional (via H) to $n \log n$. Similar results are obtained for the other datasets. Figure 10 reports the time needed to compute the kernel matrix, as a function of h, for the K_{ODD-ST_h} and the FS kernels on NCI1 and CAS. Given that the available implementations of the two kernels are in different programming languages, the plots of the FS kernel were added just for a qualitative comparison. Notice that, as h increases, the time required for computing the kernel matrix seems to increase almost linearly. This can be explained with the fact that smaller graphs do not allow too deep visits, so the increase in complexity due to the increase of the depth visit is compensated by the smaller number of graphs where visits of that depth can be actually performed.

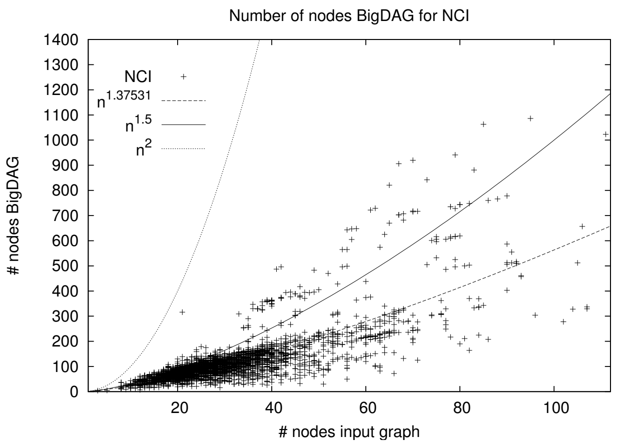

Figure 9: Number of nodes inserted into the BigDAG as a function of the nodes of the graphs. The plots refer to NCI1 with limitation to the depth of the visits ($h = 6$).

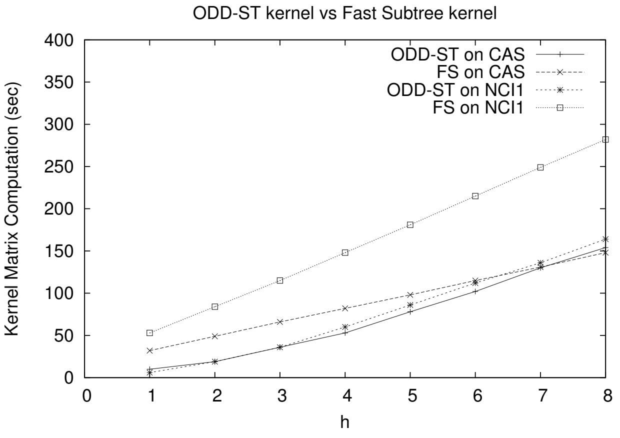

Figure 10: Time needed to compute the kernel matrix, as a function of h, for the K_{ODD-ST_h} and the FS kernels on NCI1 and CAS datasets.

11 Conclusions and future work

We have proposed a framework for graph kernels based on the decomposition of a graph into a multiset of unordered DAGs. By extending the definition of convolution tree kernels to DAGs and by defining an ordering of the nodes of the DAGs, we favoured the application of a vast class of tree kernels to graph data. We have described a technique for speeding up the kernel computation for a large category of DAG kernels and proved its effectiveness both from the theoretical and the practical point of view. The experimental results on 5 benchmark datasets show that the kernel employed in the experiments is able to reach optimal or competitive results on practically all the datasets considered. As future work, a systematic assessment of DAG kernels will be performed.

References

[1] F. Aiolli, G. Da San Martino, A. Sperduti, and A. Moschitti, *Fast on-line kernel learning for trees*, in Proceedings of the 2006 IEEE Conference on Data Mining, Los Alamitos, CA, USA, 18 - 22 December 2006, IEEE Computer Society, pp. 787–791.

[2] K. M. Borgwardt and H.-P. Kriegel, *Shortest-path kernels on graphs*, in Proceedings of the Fifth IEEE International Conference on Data Mining, IEEE Computer Society, 2005, pp. 74–81.

[3] B. Bringmann, A. Zimmermann, L. D. Raedt, and S. Nijssen, *Don't be afraid of simpler patterns*, in PKDD, J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, eds., vol. 4213 of Lecture Notes in Computer Science, Springer, 2006, pp. 55–66.

[4] M. Collins and N. Duffy, *New ranking algorithms for parsing and tagging: kernels over discrete structures, and the voted perceptron*, in Proceedings of the 40th Annual Meeting on Association for Computational Linguistics, Philadelphia, Pennsylvania, 2002, Association for Computational Linguistics, pp. 263–270.

[5] F. Costa and K. De Grave, *Fast neighborhood subgraph pairwise distance kernel*, in Proceedings of the 26th International Conference on Machine Learning, 2010.

[6] A. K. Debnath, R. L. Lopez de Compadre, G. Debnath, A. J. Shusterman, and C. Hansch, *Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. correlation with molecular orbital energies and hydrophobicity*, Journal of Medicinal Chemistry, 34 (1991), pp. 786–797.

[7] O. Flint, *Private communications*, 2010.

[8] T. Gärtner, P. Flach, and S. Wrobel, *On graph kernels: Hardness results and efficient alternatives*, Lecture notes in computer science, (2003), pp. 129—143.

[9] D. Haussler, *Convolution kernels on discrete structures*, tech. rep., Department of Computer Science, University of California at Santa Cruz, 1999.

[10] C. Helma, T. Cramer, S. Kramer, and L. D. Raedt, *Data mining and machine learning techniques for the identification of mutagenicity inducing substructures and structure activity relationships of noncongeneric compounds*, Journal of Chemical Information and Computer Sciences, 44 (2004), pp. 1402–1411.

[11] H. Kashima, *Machine Learning Approaches for Structured Data*, PhD thesis, Graduate School of Informatics, Kyoto University, Japan, 2007.

[12] H. Kashima, K. Tsuda, and A. Inokuchi, *Marginalized kernels between labeled graphs.*, in ICML, T. Fawcett and N. Mishra, eds., AAAI Press, 2003, pp. 321–328.

[13] J. Kazius, S. Nijssen, J. Kok, T. Back, and A. P. Ijzerman, *Substructure mining using elaborate chemical representation*, J. Chem. Inf. Model., 46 (2006), pp. 597–605.

[14] D. Kimura, T. Kuboyama, T. Shibuya, and H. Kashima, *A subpath kernel for rooted unordered trees*, in In Proceedings of the 15th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD), Shenzeng, China, 2011.

[15] P. Mahé and J. Vert, *Graph kernels based on tree patterns for molecules*, Machine Learning, 75 (2009), pp. 3–35.

[16] A. Moschitti, *Efficient convolution kernels for dependency and constituent syntactic trees*, in ECML, J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, eds., vol. 4212 of Lecture Notes in Computer Science, Machine Learning: ECML 2006, 17th European Conference on Machine Learning, Proceedings, Berlin, Germany, September 2006, pp. 318–329.

[17] R. C. Read and D. G. Corneil, *The graph isomorphism disease*, Journal of Graph Theory, 1 (1977), pp. 339–363.

[18] H. Saigo, S. Nowozin, T. Kadowaki, T. Kudo, and K. Tsuda, *gboost: a mathematical programming approach to graph classification and regression.*, Machine Learning, (2009), pp. 69–89.

[19] L. Schietgat, F. Costa, J. Ramon, and L. De Raedt, *Maximum common subgraph mining: a fast and effective approach towards feature generation*, in 7th International Workshop on Mining and Learning with Graphs, 2009, pp. 1–3.

[20] N. Shervashidze and K. M. Borgwardt, *Fast subtree kernels on graphs*, in NIPS, Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I. Williams, and A. Culotta, eds., Curran Associates, Inc., 2009, pp. 1660–1668.

[21] S. V. N. Vishwanathan and A. J. Smola, *Fast kernels for string and tree matching*, in Advances in Neural Information Processing Systems 15, MIT Press, 2003, pp. 569–576.

[22] N. Wale, I. Watson, and G. Karypis, *Comparison of descriptor spaces for chemical compound retrieval and classification*, Knowledge and Information Systems, 14 (2008), pp. 347–375. 10.1007/s10115-007-0103-5.

[23] O. S. Weislow, R. Kiser, D. L. Fine, J. Bader, R. H. Shoemaker, and M. R. Boyd, *New soluble-formazan assay for hiv-1 cytopathic effects: application to high-flux screening of synthetic and natural products for aids-antiviral activity.*, Journal of the National Cancer Institute, 81 (1989), pp. 577–86.