Mapping Kernels for Trees

Kilho Shin YSHIN@AI.U-HYOGO.AC.JP
Graduate School of Applied Informatics, Hyogo University, Kobe

Marco Cuturi MCUTURI@I.KYOTO-U.AC.JP
Graduate School of Informatics, Kyoto University

Tetsuji Kuboyama KUBOYAMA@GAKUSHUIN.AC.JP
Gakushuin University, Tokyo

Abstract

We propose a comprehensive survey of tree kernels through the lens of the *mapping kernels* framework. We argue that most existing tree kernels, as well as many more that are presented for the first time in this paper, fall into a typology of kernels whose seemingly intricate computation can be efficiently factorized to yield polynomial time algorithms. Despite this fact, we argue that a naive implementation of such kernels remains prohibitively expensive to compute. We propose an approach whereby some computations for smaller trees are cached, which speeds up considerably the computation of all these tree kernels. We provide experimental evidence of this fact as well as preliminary results on the performance of these kernels.

1. Introduction

For about a decade now, kernel methods (Hofmann et al., 2008) have been recognized as one of the most efficient and generic approaches to deal with structured data. Kernels on images (Harchaoui & Bach, 2007), probability densities (Martins et al., 2009), graphs (Shervashidze & Borgwardt, 2009), sequences (Sonnenburg et al., 2007) to quote a few recent references have all been the subject of extensive research.

The computation of large Gram matrices is known to be one of the bottlenecks of kernel methods. When dealing with simple vectors of reasonable size, this problem only materializes with very large databases. For very complex structures such as texts, dense graphs or images, computational limits are hit much earlier. Hence, defining a kernel that can leverage the wealth of information provided by such structures while being still computationally tractable remains a major challenge. Between the two ends of this spectrum lie structures such as sequences or trees, which, although challenging from a combinatorial perspective, can be handled with relative efficiency through Haussler's (1999) convolution kernels framework. Using such ideas, Collins & Duffy (2001) proposed a kernel guided by the simple principle that trees are similar when they share many substructures. A brute force enumeration whereby all substructures of a given tree are parsed can be avoided through clever factorizations (Kashima et al., 2003) but it is widely admitted that these kernels remain too expensive to be used on large scale databases.

We show in this paper that the wealth of data available in all these substructures need not be sacrificed to obtain fast tree kernels. In order to do so, we use the framework of mapping kernels (Shin & Kuboyama, 2008) which was proposed recently as a generalization of Haussler's work. Our contribution in this paper is two-fold. We propose first an exhaustive typology of tree kernels that can be efficiently factorized. We follow by showing that the computation of kernels that fall in this typology can be dramatically sped-up by caching some intermediate computations.

This paper is organized as follows. Section 2 details the comprehensive typology of tree kernels that we highlight using mapping kernels. Section 3 describes the algorithm which is paired with this typology and which is shown to in Section 4 to yield significant speed-ups.

Appearing in *Proceedings of the 25th International Conference on Machine Learning*, Helsinki, Finland, 2008.

2. Mapping Trees onto Substructures

2.1. Notations: Ordered Trees and Forests

We consider in this paper directed labeled graphs. Each of these graphs is defined by a set of vertices $\mathcal{V}$ and directed edges $\mathcal{E} \subset \mathcal{V}^2$, with a label taken in a set $\mathcal{L}$ associated to each vertex. **A directed path** is a set of vertices $(x_1, \cdots, x_p)$ such that either all edges (x_i, x_{i+1}) or (x_{i+1}, x_i) belong to $\mathcal{E}$. **A labeled rooted tree** is a connected graph such that it is acyclic as an undirected graph and each vertex has at most one incoming edge. A partial order on the vertices of the tree, called the *generation* order, can be defined as follows: $x > y$ when there exists a directed path from x to y. In such a case, x (resp. y) is called an ancestor of y (resp. a descendant of x). When the edge $(x, y) \in \mathcal{E}$, x is called the *parent* of y and y the *child* of x. **An ordered tree** X is a rooted tree in which all children of a given vertex are ordered according to another partial order $\succ$. This order among siblings can be extended as follows. If two ancestors x' and y' of two vertices x and y share the same parent, and x' is greater than y' according to the sibling order, namely $x' \succ y'$, then $x \succ y$. Under this extended order, all descendants of a given vertex x' will be older than the descendants of any of its younger siblings y'. We call this extended sibling ordering the *family* order. **An ordered forest** is a family of ordered trees $(X_1, \ldots, X_n)$. The order of the forest extends to the family order by defining that $x \succ y$ for all $i < j, x \in X_i, y \in X_j$. A **substructure** of a forest is a subgraph of that forest that inherits both generation and family orders. The size $|X|$ of a forest X is equal to the cardinal of its vertex set. Note that we only consider **labeled and ordered {trees, forests, substructures}** in this paper.

2.2. Fundamental Approaches

We highlight two fundamental approaches to build tree kernels in this section, with interesting analogies with the spectrum (Leslie et al., 2002) and local alignment kernels (Vert et al., 2004) for strings.

Substructures Spectrum In this approach, two trees are considered similar when they share substructures. For example, the parse tree kernel proposed by Collins & Duffy (2001) counts shared *corooted* subtrees; the elastic tree kernel (Kashima et al., 2003) counts *agreement* subtrees (these terms are defined in Section 2.4.1). More formally, let X and Y be two trees, and define a family $\mathcal{T}$ of substructures. The set

$$M^{\mathcal{T}}_{X,Y} = \{(X', Y') \mid X' \subset X, Y' \subset Y, X', Y' \in \mathcal{T}, X' \cong Y'\},$$

is called the mapping set corresponding to $\mathcal{T}$, where $X' \cong Y'$ means that the substructures X' and Y' are isomorphic, that is, there exists a bijection between the vertices of X' and Y' that preserves both generation and family orders. Obviously, $X' \cong Y'$ implies that $|X'| = |Y'|$. When every two vertices of a vertex pair in this bijection $(x_i, y_i)_{1 \leq i \leq |X'|}$ share the same label, X' and Y' are said to be congruent and we write $X' \equiv Y'$. Substructure spectrum kernels define the similarity between two trees X and Y based upon the number of congruent substructures they share, *e.g.*

$$K(X, Y) = \sum_{(X', Y') \in M^{\mathcal{T}}_{X,Y}} \prod_{i=1}^{|X'|} \lambda \delta_{\mathcal{L}}(x_i, y_i),$$

where λ is a decay factor and $\delta_{\mathcal{L}}(x, y)$ is Kronecker's symbol for the labels associated with vertices x and y.

Tree Edit Distances. Taï (1979) defined an edit-distance between two trees X and Y in order to measure their (dis)similarity as follows: an edit script σ that converts X into Y is a finite sequence of edit operations, where each edit operation can either be the

1. deletion of a vertex x from X, written as $\langle x \to \bullet \rangle$,
2. insertion of a vertex y of Y into X, $\langle \bullet \to y \rangle$,
3. substitution of a vertex y of Y for x of X: $\langle x \to y \rangle$.

To each operation $\langle a \to b \rangle$, is associated a cost $\gamma(\langle a \to b \rangle)$ which is symmetric, *i.e.* $\gamma(\langle a \to b \rangle) = \gamma(\langle b \to a \rangle)$. The cost $\gamma(\sigma)$ of a script σ is the sum of the costs of all edit operations in σ. Taï defines the distance between X and Y as the minimum over the costs of all scripts that convert X into Y. Building upon this idea, Shin & Kuboyama (2008) show that the costs of *all* possible scripts that convert X into Y can be considered,

$$k(X, Y) = \sum_{\sigma : X \to Y} e^{-\lambda \gamma(\sigma)}.$$

To compute this kernel, Shin & Kuboyama show that the *alignment* of a script plays a crucial role. The alignment of a script σ that converts X into Y is the set of all substitutions $\{\langle x_i \to y_i \rangle \mid i = 1, \ldots, n\}$ described in σ, namely all edit-operations excluding insertions and deletions. The alignment of σ can be used to factor its edit-cost as

$$e^{-\gamma(\sigma)} = g(X) \cdot g(Y) \cdot \prod_{i=1}^{n} \frac{e^{-\gamma(\langle x_i \to y_i \rangle)}}{e^{-\gamma(\langle x_i \to \bullet \rangle)} e^{-\gamma(\langle \bullet \to y_i \rangle)}}, \quad (1)$$

where $g(X) = \prod_{x \in X} e^{-\gamma(\langle x \to \bullet \rangle)}$. One can prove that there exist two substructures $X' \subset X$ and $Y' \subset Y$ that are isomorphic under the alignment of an edit script σ if and only if σ transforms X into Y.

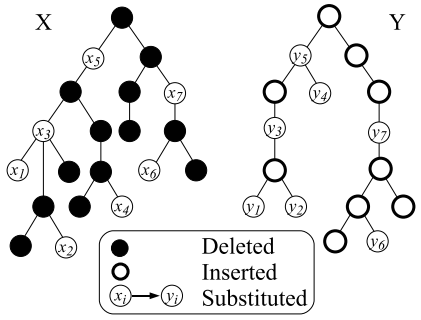

Figure 1. Illustration of an edit script that converts X into Y: the script proceeds first by deleting vertices (filled circles), then substitutes the vertices labeled x_i by vertices labeled y_i and finally inserts additional vertices (hollow circles). The set of substitutions maps the substructure X' composed of the connected vertices $x_1, \cdots, x_7$ unto Y' formed by $y_1, \cdots, y_7$.

Theorem 1 (Taï (1979)) *Given two substructures $X' \subset X$ and $Y' \subset Y$ and given a mapping μ from the vertex set of X' to that of Y', μ is the alignment of an edit script that converts X into Y if and only if μ is an isomorphism between X' and Y'.*

Hence, by normalizing the kernel k described in Eq. (1) by g to define $K(X,Y) = \frac{k(X,Y)}{g(X)\cdot g(Y)}$, Theorem 1 proves

$$K(X,Y) = \sum_{(X',Y') \in M^{\mathsf{All}}_{X,Y}} \prod_{i=1}^{n} \frac{e^{-\gamma(\langle x_i \to y_i \rangle)}}{e^{-\gamma(\langle x_i \to \bullet \rangle)} e^{-\gamma(\langle \bullet \to y_i \rangle)}},$$

where the sum over all edit-scripts has been replaced by a sum over all isomorphic substructures of X, Y

$$M^{\mathsf{All}}_{X,Y} = \{(X',Y') \mid X' \subset X, Y' \subset Y, X' \cong Y'\}.$$

2.3. Mapping Kernels

Shin & Kuboyama (2008) have introduced the *mapping kernel* template to define kernels between structured objects. Mapping kernels are kernels of the form

$$K(\mathbf{X},\mathbf{Y}) = \sum_{(\mathbf{x},\mathbf{y}) \in M_{\mathbf{X},\mathbf{Y}}} k(\mathbf{x},\mathbf{y}). \qquad (2)$$

This template defines kernels over large structures that can be described through smaller structures enumerated in a set $\mathcal{X}$. The kernel is defined by the mapping set $M_{\mathbf{X},\mathbf{Y}} \subset \mathcal{X}^2$ paired with a *base* kernel $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$.

Theorem 2 *The kernel K given in Eq.* (2) *is positive definite (p.d.) for* all *p.d. base kernels k if and only if the mapping $M_{X,Y}$ is* transitive*, that is,*

$$(\mathbf{x},\mathbf{y}) \in M_{\mathbf{X},\mathbf{Y}}, (\mathbf{y},\mathbf{z}) \in M_{\mathbf{Y},\mathbf{Z}} \Rightarrow (\mathbf{x},\mathbf{z}) \in M_{\mathbf{X},\mathbf{Z}}.$$

Both kernels introduced in Section 2.2 can be cast as mapping kernels. In fact, Shin & Kuboyama (2010) report that 18 of 19 tree kernels found in the literature can be described as mapping kernels, and Theorem 2 provides a systematic way to prove that they are p.d.

2.4. Designing New Kernels

Eq. (2) provides a generic approach to *build* kernels but certainly not to *compute* them. Yet, mappings $M_{X,Y}$ and base kernels k only have an interest if the kernels they define are computationally tractable. Having this goal in mind, we introduce first a typology of families of substructures $\mathcal{T}$ to define the mapping set $M^{\mathcal{T}}_{X,Y}$. We then study a class of base kernels k which ensure that the summation over all isomorphic substructures of X and Y can be factorized efficiently. To simplify the presentation, we consider that X, Y will now denote forests, and not necessarily single trees.

2.4.1. Typology of Substructures

A substructure can be first characterized by its *shape*. A substructure can either be constrained to be a *path*, a *tree* or not constrained at all and be a *forest*. These shapes are ranked by increasing complexity.

It is also possible to constraint substructures in a forest so that they either replicate exactly the *parent-child relations* of the original forest, share more loosely *nearest-common-ancestor relations*, or have no constraints at all in which case they are simply called *elastic* as seen in the typology below,

Contiguous	If a parent and child vertices in a substructure are always in a parent-child relation in the parent tree, the substructure is said to be contiguous.
Agreement	If a structure is not contiguous, and if the nearest-common-ancestor of two vertices in a substructure is always the nearest-common-ancestor in the original structure, the substructure is said to be an agreement substructure.
Elastic	If a structure is not contiguous or agreement, it is called elastic.

These two characterizations provide a grid of 3×3 substructures that is indexed by two letters. The first letter stands for the **C**ontiguous, **A**greement or **E**lastic characterization of the substructure while the second letter is either **F**, **T** or **P**. For example, EF stands for elastic forests substructures. Let us follow with a few remarks:

- EF structures cover all substructures, as introduced in Section 2.

- A path always satisfies the agreement condition.
- An agreement forest always has a single root within a tree.
- Some of these types have been mentioned in the kernel literature: AT by Kashima et al. (2003), EF by Shin & Kuboyama (2008) and EP by Culotta & Sorensen (2004).
- The substructures considered in the parse tree kernel of Collins & Duffy (2001), namely *co-rooted subtrees*, are not contained in this typology. For sake of completeness we also consider them under the abbreviation CoT.

2.4.2. Base Kernels

Mapping kernels for trees are computationally tractable when they can be efficiently factorized. The algorithms that make it possible to compute the elastic tree kernel (Kashima et al., 2003) and Taï's distribution kernel are inherently recursive. They both have in commonly the application of a technique proposed by Taï (1979) to derive recursive formulas to compute tree edit-distances. Both kernels in Section 2.2 use base kernels that are products of simple kernels on vertices. Let (X', Y') be an isomorphic pair of substructures, such that (x_i, y_i) for $i = 1, \ldots, |X'| = |Y'|$ is the isomorphism that associates X' and Y'. Given a kernel ℓ defined over vertices, the corresponding product kernel factorizes as

$$k(X', Y') = \prod_{i=1}^{|X'|} \ell(x_i, y_i).$$

We illustrate the basic technique to derive recursive formulas by considering first substructures $\mathcal{T} = \mathsf{EF}$ as an example. Some additional notations are given in Table 1 and Figure 2.

Table 1. Notations

${}^{\bullet}X$	The root of the leftmost (oldest) subtree of X.
${}^{\circ}X$	$= X \setminus \{{}^{\bullet}X\}$.
${}^{\blacktriangle}X$	The leftmost (oldest) subtree of X.
${}^{\triangle}X$	$= X \setminus {}^{\blacktriangle}X$.
$X[x]$	The *complete* subtree $\{y \in X \mid y \leq x\}$.
$X[x, y)$	$= \{z \in X \mid z < y, z \not\prec x\} \cup X[x]$.
$\hat{x}$	The parent of a vertex x.

For two forests X and Y, we define

$$M_1 = \{(X', Y') \in M^{\mathsf{EF}}_{X,Y} \mid {}^{\bullet}X \in X', {}^{\bullet}Y' \in Y'\}$$
$$M_2 = M^{\mathsf{EF}}_{X,Y} \setminus M_1$$

To obtain $K_{\mathsf{EF}}(X, Y)$, we consider the sums over M_1 and M_2 separately. For M_2, it is easy to see

$$\sum_{(X',Y') \in M_2} k(X', Y') = K_{\mathsf{EF}}({}^{\circ}X, Y) + K_{\mathsf{EF}}(X, {}^{\circ}Y) - K_{\mathsf{EF}}({}^{\circ}X, {}^{\circ}Y).$$

For the calculation over M_1, we decompose $X' \in M_1$ into $X' = \{{}^{\bullet}X\} \cup \mathsf{b}(X') \cup \mathsf{r}(X')$, where $\mathsf{b}(X') = \{x \in X' \mid x < {}^{\bullet}X\}$ and $\mathsf{r}(X') = \{x \in X' \mid x \prec {}^{\bullet}X\}$. Then, we have

$$\begin{aligned}
&\sum_{(X',Y') \in M_1} k(X', Y') \\
&= \sum_{(X',Y') \in M_1} \ell({}^{\bullet}X, {}^{\bullet}Y) k(\mathsf{b}(X'), \mathsf{b}(Y')) k(\mathsf{r}(X'), \mathsf{r}(Y')) \\
&= \ell({}^{\bullet}X, {}^{\bullet}Y) K_{\mathsf{EF}}({}^{\circ\blacktriangle}X, {}^{\circ\blacktriangle}Y) K_{\mathsf{EF}}({}^{\triangle}X, {}^{\triangle}Y).
\end{aligned}$$

This yields the recursive formula

$$\begin{aligned}
K_{\mathsf{EF}}(X, Y) = {} & \ell({}^{\bullet}X, {}^{\bullet}Y) K_{\mathsf{EF}}({}^{\circ\blacktriangle}X, {}^{\circ\blacktriangle}Y) K_{\mathsf{EF}}({}^{\circ\triangle}X, {}^{\circ\triangle}Y) \\
& + K_{\mathsf{EF}}({}^{\circ}X, Y) + K_{\mathsf{EF}}(X, {}^{\circ}Y) - K_{\mathsf{EF}}({}^{\circ}X, {}^{\circ}Y).
\end{aligned}$$

Applying the very same techniques to all other substructure types, we obtain the long list of formulas shown in Table 2. K_1, K_2 and K_3 are auxiliary kernels used to compute some of the kernels we consider.

Taking a closer look at Table 2, we observe that these recursive formulas can be classified into two groups: Those which contain the operator $^{\circ}$, namely ${}^{\circ}X$ or ${}^{\circ}Y$, and the rest. Indeed, only formulas for substructures of type EF, ET and CF contain these operators. This seemingly minor detail has an important impact on their computational complexity. The computational complexity for formulas that contain the $^{\circ}$ operator scales cubicly in the size of these trees, whereas it is quadratic for those that do not use it. Hence, we call these groups ***C**(ubic)* and ***Q**(uadratic)*.

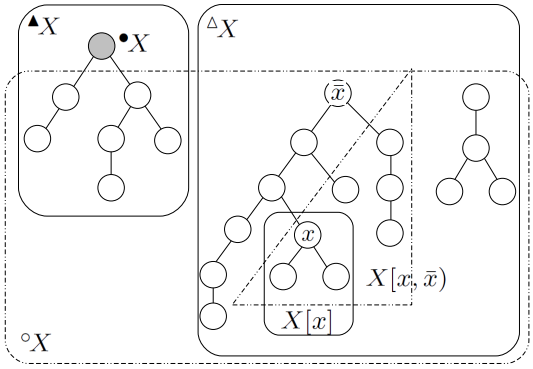

Figure 2. Illustration of ${}^{\bullet}X, {}^{\circ}X, {}^{\blacktriangle}X, {}^{\triangle}X, X[x]$ and $X[x, \bar{x})$

Table 2. Recursive formulas

Tree Kernel	Recursive Formula
$K_{\mathsf{CT}}(X,Y) =$	$k(^{\bullet}X, ^{\bullet}Y)\Big(1 + K_2(^{\circ\blacktriangle}X, ^{\circ\blacktriangle}Y)\Big) + K_{\mathsf{CT}}(^{\blacktriangle}X, ^{\circ\blacktriangle}Y) + K_{\mathsf{CT}}(^{\circ\blacktriangle}X, ^{\blacktriangle}Y) - K_{\mathsf{CT}}(^{\circ\blacktriangle}X, ^{\circ\blacktriangle}Y)$ $+K_{\mathsf{CT}}(X, ^{\triangle}Y) + K_{\mathsf{CT}}(^{\triangle}X, Y) - K_{\mathsf{CT}}(^{\triangle}X, ^{\triangle}Y)$
$K_{\mathsf{CF}}(X,Y) =$	$k(^{\bullet}X, ^{\bullet}Y)\Big(1 + K_2(^{\circ\blacktriangle}X, ^{\circ\blacktriangle}Y)\Big)\Big(1 + K_{\mathsf{CF}}(^{\triangle}X, ^{\triangle}Y)\Big) + K_{\mathsf{CF}}(X, ^{\circ}Y) + K_{\mathsf{CF}}(^{\circ}X, Y) - K_{\mathsf{CF}}(^{\circ}X, ^{\circ}Y)$
$K_2(X,Y) =$	$k(^{\bullet}X, ^{\bullet}Y)(1 + K_2(^{\circ\blacktriangle}X, ^{\circ\blacktriangle}Y))(1 + K_2(^{\triangle}X, ^{\triangle}Y)) + K_2(X, ^{\triangle}Y) + K_2(^{\triangle}X, Y) - K_2(^{\triangle}X, ^{\triangle}Y)$
$K_{\mathsf{CP}}(X,Y) =$	$k(^{\bullet}X, ^{\bullet}Y)\Big(1 + \sum_{x \in \mathsf{children}(^{\bullet}X)} \sum_{x \in \mathsf{children}(^{\bullet}X)} K_{\mathsf{CPR}}(X[x], Y[y])\Big)$ $+K_{\mathsf{CP}}(^{\blacktriangle}X, ^{\circ\blacktriangle}Y) + K_{\mathsf{CP}}(^{\circ\blacktriangle}X, ^{\blacktriangle}Y) - K_{\mathsf{CP}}(^{\circ\blacktriangle}X, ^{\circ\blacktriangle}Y)$ $+K_{\mathsf{CP}}(X, ^{\triangle}Y) + K_{\mathsf{CP}}(^{\triangle}X, Y) - K_{\mathsf{CP}}(^{\triangle}X, ^{\triangle}Y)$
$K_{\mathsf{AT}}(X,Y) =$	$k(^{\bullet}X, ^{\bullet}Y)\Big(1 + K_1(^{\circ\blacktriangle}X, ^{\circ\blacktriangle}Y)\Big) + K_{\mathsf{AT}}(^{\blacktriangle}X, ^{\circ\blacktriangle}Y) + K_{\mathsf{AT}}(^{\circ\blacktriangle}X, ^{\blacktriangle}Y) - K_{\mathsf{AT}}(^{\circ\blacktriangle}X, ^{\circ\blacktriangle}Y)$ $+K_{\mathsf{AT}}(X, ^{\triangle}Y) + K_{\mathsf{AT}}(^{\triangle}X, Y) - K_{\mathsf{AT}}(^{\triangle}X, ^{\triangle}Y)$
$K_1(X,Y) =$	$k(^{\bullet}X, ^{\bullet}Y)\Big(1 + K_1(^{\circ\blacktriangle}X, ^{\circ\blacktriangle}Y)\Big)\Big(1 + K_1(^{\triangle}X, ^{\triangle}Y)\Big)$ $+\Big(1 + K_1(^{\triangle}X, ^{\triangle}Y)\Big)\Big(K_{\mathsf{AT}}(^{\blacktriangle}X, ^{\circ\blacktriangle}Y) + K_{\mathsf{AT}}(^{\circ\blacktriangle}X, ^{\blacktriangle}Y) - K_{\mathsf{AT}}(^{\circ\blacktriangle}X, ^{\circ\blacktriangle}Y)\Big)$ $+K_1(X, ^{\triangle}Y) + K_1(^{\triangle}X, Y) - K_1(^{\triangle}X, ^{\triangle}Y)$
$K_{\mathsf{ET}}(X,Y) =$	$k(^{\bullet}X, ^{\bullet}Y)\Big(1 + K_{\mathsf{EF}}(^{\circ\blacktriangle}X, ^{\circ\blacktriangle}Y)\Big) + K_{\mathsf{ET}}(^{\blacktriangle}X, ^{\circ\blacktriangle}Y) + K_{\mathsf{ET}}(^{\circ\blacktriangle}X, ^{\blacktriangle}Y) - K_{\mathsf{ET}}(^{\circ\blacktriangle}X, ^{\circ\blacktriangle}Y)$ $+K_{\mathsf{ET}}(X, ^{\triangle}Y) + K_{\mathsf{ET}}(^{\triangle}X, Y) - K_{\mathsf{ET}}(^{\triangle}X, ^{\triangle}Y)$
$K_{\mathsf{EF}}(X,Y) =$	$k(^{\bullet}X, ^{\bullet}Y)\Big(1 + K_{\mathsf{EF}}(^{\circ\blacktriangle}X, ^{\circ\blacktriangle}Y)\Big)\Big(1 + K_{\mathsf{EF}}(^{\triangle}X, ^{\triangle}Y)\Big) + K_{\mathsf{EF}}(X, ^{\circ}Y) + K_{\mathsf{EF}}(^{\circ}X, Y) - K_{\mathsf{EF}}(^{\circ}X, ^{\circ}Y)$
$K_{\mathsf{EP}}(X,Y) =$	$k(^{\bullet}X, ^{\bullet}Y)\Big(1 + K_{\mathsf{EP}}(^{\circ\blacktriangle}X, ^{\circ\blacktriangle}Y)\Big) + K_{\mathsf{EP}}(^{\blacktriangle}X, ^{\circ\blacktriangle}Y) + K_{\mathsf{EP}}(^{\circ\blacktriangle}X, ^{\blacktriangle}Y) - K_{\mathsf{EP}}(^{\circ\blacktriangle}X, ^{\circ\blacktriangle}Y)$ $+K_{\mathsf{EP}}(X, ^{\triangle}Y) + K_{\mathsf{EP}}(^{\triangle}X, Y) - K_{\mathsf{EP}}(^{\triangle}X, ^{\triangle}Y)$
$K_{\mathsf{CoT}}(X,Y) =$	$k(^{\bullet}X, ^{\bullet}Y)\Big(1 + K_3(^{\circ\blacktriangle}X, ^{\circ\blacktriangle}Y)\Big) + K_{\mathsf{CoT}}(^{\blacktriangle}X, ^{\circ\blacktriangle}Y) + K_{\mathsf{CoT}}(^{\circ\blacktriangle}X, ^{\blacktriangle}Y) - K_{\mathsf{CoT}}(^{\circ\blacktriangle}X, ^{\circ\blacktriangle}Y)$ $+K_{\mathsf{CoT}}(X, ^{\triangle}Y) + K_{\mathsf{CoT}}(^{\triangle}X, Y) - K_{\mathsf{CoT}}(^{\triangle}X, ^{\triangle}Y)$
$K_3(X,Y) =$	$k(^{\bullet}X, ^{\bullet}Y)\Big(1 + K_3(^{\circ\blacktriangle}X, ^{\circ\blacktriangle}Y)\Big)$, if $^{\triangle}X = \emptyset$ and $^{\triangle}Y = \emptyset$. $k(^{\bullet}X, ^{\bullet}Y)\Big(1 + K_3(^{\circ\blacktriangle}X, ^{\circ\blacktriangle}Y)\Big)K_3(^{\triangle}X, ^{\triangle}Y)$, if $^{\triangle}X \neq \emptyset$ or $^{\triangle}Y \neq \emptyset$.

3. Extracting Substructures

Despite the existence of recursive formulas, the computation of all tree kernels listed in Table 2 is known to be time-consuming. As shown in Section 3.1, the complexity of the dynamic programming algorithms which can be derived from these recursive formulas scale in $O(n^4)$ for kernels of the group C and $O(n^2)$ for group Q, where n denotes the size of the compared trees. For kernels in the group C, techniques such as the one proposed by Demaine et al. (2007) in a Taï distance context can be applied, but their complexity still scales in $O(n^3)$.

In this section, we formulate the hypothesis that the larger the dataset of trees, the more substructures they will share. In that sense, the recursions described in Table 2 would go much faster if a cache of kernel values for frequent substructures was available. This would avoid duplicated computations and expensive recursive function calls. We implement this approach through an algorithm that extracts efficiently all congruent substructures in a dataset.

3.1. Preliminary Result

For a tree X, we write $\mathcal{Q}(X)$ and $\mathcal{C}(X)$ for the sets of substructures of X that are visited during the iterative evaluations of formulas of group Q and group C respectively. The following lemma is given without proof but will be useful to prove Theorem 3 below,

Lemma 1 *For arbitrary $x \leq y$,*

1. $^{\circ}X[x,y) = X[^{\bullet\circ}X[x,y), y)$, *unless* $^{\circ}X[x,y) = \emptyset$.
2. $^{\triangle}X[x,y) = X[^{\bullet\triangle}X[x,y), y)$, *unless* $^{\triangle}X[x,y) = \emptyset$,
3. $^{\blacktriangle}X[x,y) = X[x,x) = X[x]$

Theorem 3

$$\mathcal{C}(X) = \{X[x,y) \mid x \in X, y \in X, x \leq y\} \cup \{\emptyset\}$$
$$\mathcal{Q}(X) = \{X[x] \mid x \in X\} \cup \{X[x,\hat{x}) \mid x \in X\} \cup \{\emptyset\}$$

Proof. (Sketch) To prove the inclusion $\subset$, we show that the set of the right hand side is closed under the operators $^{\circ}$, $^{\triangle}$ and $^{\blacktriangle}$ for $\mathcal{C}(X)$; $^{\circ\blacktriangle}$, $^{\triangle}$ and $^{\blacktriangle}$ for $\mathcal{Q}(X)$. To prove $\supset$, we show any structure in $\mathcal{C}(X)$ or $\mathcal{Q}(X)$ is reachable from X by iteratively applying the corresponding operators. □

Since $|\mathcal{C}(X)| = |X|^2 + 1$ and $|\mathcal{Q}(X)| = 2|X|$ hold, the time complexities of the algorithm needed to calculate $K_{\mathcal{T}}(X,Y)$ using directly recursive formulas are $O(|X|^2|Y|^2)$ and $O(|X||Y|)$ for Group C and Q, respectively.

3.2. Algorithm

Let $\mathcal{E}$ denote a dataset consisting of one or more trees. We define $\tilde{\mathcal{Q}}(\mathcal{E})$ and $\tilde{\mathcal{C}}(\mathcal{E})$ as sets of congruence classes.

$$\tilde{\mathcal{Q}}(\mathcal{E}) = \bigcup_{X \in \mathcal{E}} \mathcal{Q}(X)/\equiv \quad \text{and} \quad \tilde{\mathcal{C}}(\mathcal{E}) = \bigcup_{X \in \mathcal{E}} \mathcal{C}(X)/\equiv .$$

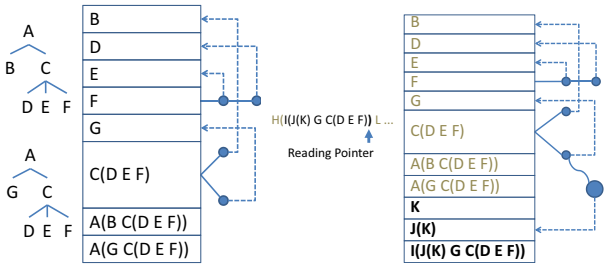

Figure 3. Extraction table

The algorithm we introduce here is designed to to calculate $\tilde{\mathcal{Q}}(\mathcal{E})$ and $\tilde{\mathcal{C}}(\mathcal{E})$ using $\mathcal{E}$. For illustration purposes, we first focus on the simpler case $\tilde{\mathcal{Q}}(\mathcal{E})$, and then show how to extend the algorithm to compute $\tilde{\mathcal{C}}(\mathcal{E})$. The algorithm consists of two phases: the *parse phase* and the *extraction phase*.

3.2.1. Parse Phase

The algorithm maintains a table of subtrees as follows.

1. Every complete subtree $X[x]$ for some $x \in X \in \mathcal{E}$ has an entry in the table.
2. Any two entries of the table are not congruent with each other.
3. If an entry is the youngest (rightmost) sibling vertex in some $X \in \mathcal{E}$, an *extraction tree* rooted at the entry is associated. Each vertex of the extraction tree references one of the entries of the table, and every path from a leaf to the root yields a *sibling subtree sequence* $(X[x_1], \ldots, X[x_n])$, where $(x_1, x_2, \ldots, x_n)$ is the entire sequence of children of some $x \in X \in \mathcal{E}$.

For convenience, we call this table an *extraction table*. The left chart of Figure 3 depicts a snapshot of the extraction table, when the algorithm has completed the parsing of two trees `A(B C(D E F))` and `A(G C(D E F))`. Since these two trees include 8 complete subtrees up to congruence, the extraction table includes 8 entries. The sibling subtree sequence `(D E F)` appears in the input trees, and hence the extraction tree rooted at entry `F` has two vertices that reference the entries for `E` and `D`. Furthermore, since the two sibling subtree sequences `(B C(D E F))` and `(G C(D E F))` appear in the input trees, the entry for `C(D E F)` has two child vertices that reference the entries for `B` and `G` in the extraction tree rooted at the entry.

Next, we assume that the algorithm is parsing the tree `H(I(J(K) G C(D E F)) L(M N) O)`. The algorithm reads the characters in the representing string sequentially from head to tail. When the algorithm reads a parenthesis `)` that immediately follows `F`, the algorithm recognizes the subtree `C(D E F)`. Since the algorithm can find the same subtree in the extraction table, it moves to the next character in the string without updating the extraction table. When reading the next parenthesis `)`, the algorithm recognizes the subtree `I(J(K) G C(D E F))`. Since the algorithm cannot find an entry for `I(J(K) G C(D E F))`, it generates a new entry for `I(J(K) G C(D E F))`, and adds it to the extraction table.

At the same time, the algorithm recognizes a sibling subtree sequence `(J(K) G C(D E F))`, and examines whether the sequence already exists in the extraction tree rooted at the vertex for `C(D E F)`. As a result, it will find out that the path `(G C(D E F))` is the longest match, and hence, generate a new vertex referencing the entry for `J(K)` to place it as a child of the vertex referencing `G` in the extraction tree. The right chart of Figure 3 depicts the snapshot of this operation. The completed extraction table and the associated extraction trees are the outputs of the parse phase.

3.2.2. Extraction Phase

In the extraction phase, the algorithm scans every extraction tree obtained, and collects all of the contiguous paths ending at the root. By replacing the vertices of the collected paths with their referenced subtrees, the algorithm obtains a set of subtree sequences. For example, from the extraction tree rooted at `C(D E F)`, the algorithm extracts four substructures `(B C(D E F))`, `(J(K) G C(D E F))`, `(G C(D E F))` and `C(D E F)`. The union of these sets of substructures is the final output of the algorithm.

3.3. Validity of the Algorithm

We first prove that the output of the algorithm includes $\tilde{\mathcal{Q}}(\mathcal{E})$. By Theorem 3, an element of $\mathcal{Q}(X)$ is either $X[x]$ or $X[x, \hat{x})$. It is clear that $X[x]$ appears in the extraction table. On the other hand, when we let $(x_1, \ldots, x_n)$ be the entire sequece of children of $\hat{x}$, the sibling subtree sequence $(X[x_1], \ldots, X[x_n])$ appears as a path in the extraction tree rooted at the entry for $X[x_n]$. Since we have $X[x, \hat{x}) = (X[x_i], \ldots, X[x_n])$ for some $i \in [1, n]$, $X[x, \hat{x})$ is extracted from the extraction tree in the extraction phase.

To prove that $\tilde{\mathcal{Q}}(\mathcal{E})$ includes the output of the algorithm, we only need to notice that $(X[x_i], \ldots, X[x_n])$ in the output is a partial path of $(X[x_1], \ldots, X[x_n])$, which is a path from a leaf to the root in some extraction tree. By definition, $(X[x_1], \ldots, X[x_n])$ represents some sibling subtree sequence of $x \in X \in \mathcal{E}$, and hence, $X[x_i, x) = (X[x_i], \ldots, X[x_n])$ holds.

Table 3. Notation for complexity evaluation

Symbol	Definition
N	total number of vertices in $\mathcal{E}$
δ	average number of complete subtrees in $\mathcal{E}$ that are mutually congruent.
d	average degree (number of children) for trees in $\mathcal{E}$.
D	average degree for extraction trees.

3.4. Computational Complexity

We provide a rough estimation of the average computational complexity of the algorithm presented so far. Table 3 describes the notations used in this section.

In the parse phase, each time that the algorithm has to recognize $X[x]$ for $x \in X \in \mathcal{E}$, the algorithm will have to scan the extraction table. If the extraction table is implemented as a binary search tree, the number of congruence tests required for this table scan is $\log_2(N/\delta)$ on average. Moreover, since the corresponding entry is known for every child of x, the congruence test can be reduced to a series of identity tests between entries. Hence, the number of steps for the table scan is $d\log_2(N/\delta)$ on average. Also, it is straightforward to see that the number of steps necessary to find the largest matching path to the sibling subtree sequence of x in the extraction tree is at most dD. In the extraction phase, since $1+D+D^2+\cdots+D^{d-1} \leq dD^{d-1}$ the latter numbers is an upperbound on the number of operations needed to extract a single extraction tree.

On the other hand, the inequality $D^{d-1} \leq \delta$ can be derived as follows. Assume that an entry in the extraction table corresponds to $X[x]$. The extraction tree rooted at this entry has D^{d-1} leaves on average. hence, there exist at least D^{d-1} subtrees congruent to $X[x]$ in $\mathcal{E}$. Applying this inequality, we can estimate the average number of operations of the algorithm by $Nd(\log_2(N/\delta) + \sqrt[d-1]{\delta} + 1)$.

Note that we can naturally assume that d is not affected by N. Hence, if we can further assume that δ converges asymptotically to a constant, the time complexity of the algorithm scales as $O(N \log N)$. On the other hand, if we assume that δ is asymptotically linear w.r.t. N, the time complexity becomes $O(N^{d/d-1})$. It is easy to see that the space complexity of the algorithm can be upperbounded by $O(Nd)$.

3.5. Extension to Group C

If used on kernels from group C, the algorithm will add more paths to the extraction trees in the parse phase. To illustrate this, let $y \leq x$, and define a sequence of vertices $\tau(y \to x)$ by the following recursive formulas:

$$\tau(y \to x) = \emptyset \quad \text{if } y = x,$$
$$\tau(y \to x) = (y_i, \ldots, y_n) \cdot \tau(\hat{y} \to x) \quad \text{if } y < x,$$

where $\cdot$ denotes the simple concatenation, $(y_1, \ldots, y_n)$ is the entire sequence of children of $\hat{y}$ where $y = y_i$.

Whenever the algorithm recognizes $X[x]$, it registers to the corresponding extraction tree all paths $(X[z_1], \ldots, X[z_m], X[\bar{y}_1], \ldots, X[\bar{y}_\ell])$ such that $\bar{y}_1 \leq x$, $\tau(\bar{y}_1 \to x) = (\bar{y}_1, \ldots, \bar{y}_\ell)$ and $(z_1, \ldots, z_m)$ is the entire sequence of children of $\bar{y}_1$. In the case of Group Q, the algorithm registers only the path for $\bar{y}_1 = x$. Indeed, this additional registration of paths increases the computational complexity of the algorithm. When h denotes the average height of trees in the dataset, the time and space complexities become $O(Nd(\log(N/\delta) + h^2 \cdot \sqrt[d-1]{\delta} + h^2))$ and $O(Ndh^2)$.

4. Experimental Results

We have implemented the algorithm described in Section 3.2 and the substructure spectrum tree kernels discussed in Section 2.2 for types $\mathcal{T}$ defined in Section 2.4.1. Our original source code in Scala was compiled as a jar file and will be shared on the authors' webpage. We plant to re-implement this prototype using a faster language for improved speed. The dataset used in our experiments includes 442 trees of glycan structures from the KEGG/GLYCAN database (Hashimoto et al., 2006) whose size and height average to 13.5 and 7.4.

4.1. Runtime

Figure 4 provides the computational times required to compute the kernel $K_{\mathsf{EF}}(X, Y)$ over the considered database. The `Top-Down` mode takes a brute force approach and evaluates recursive formulas with recursive function calls. On the other hand, `Bottom-Up` and `Extract` modes take advantage of our extraction algorithm. The `Bottom-Up` uses it independently to each pair of trees in the dataset, while the `Extract` mode digs first 2994 substructures using the entire dataset. The curves in Figure 4 can be interpolated quadratically as $t = 17.2n^2$, $t = 11.2n^2$ and $t = 0.48n^2$, respectively. Hence, the program runs 36 and 23 times faster in the `Extract` mode than in the `Top-Down` and `Bottom-Up` modes.The runtime for our algorithm to extract the 2994 substructures was 951 msec.

4.2. Performance

We run experiments to test the predictive performance of these kernels. We split the dataset into a training

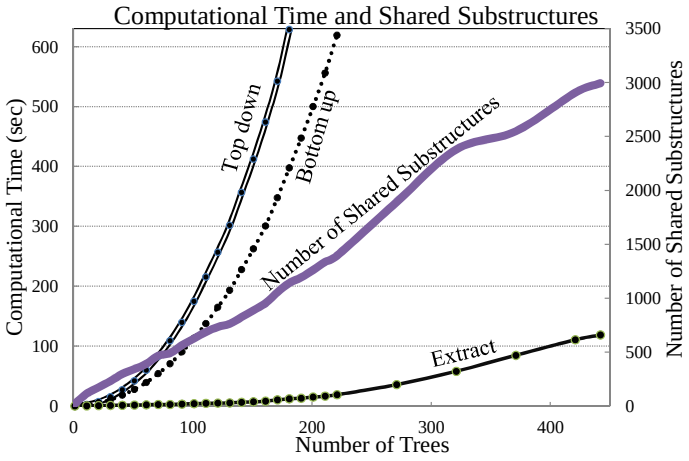

Figure 4. Comparison of Runtime

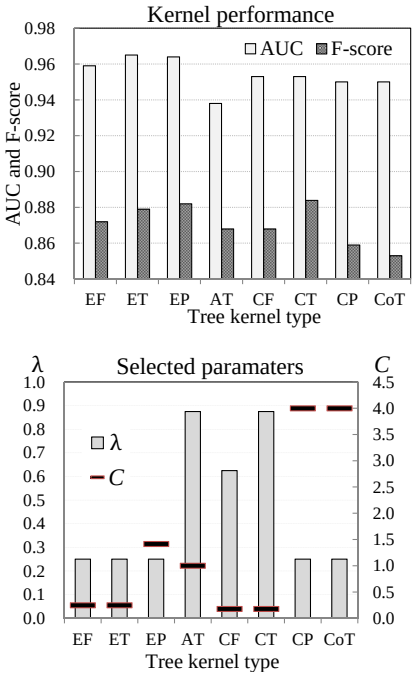

Figure 5. AUC of ROC curve and F-score above, selected parameters in the training phase displayed below

and test fold of 220 and 221 trees respectively. For each kernel, we select the decay factor λ and the regularization parameter C of the SVM that give the best mean AUC over a 5 fold cross validation on the training fold. These parameters are then used when testing the kernel on the test fold. No kernel seems to perform significantly worse than all others, as shown in Figure 5. The kernel that considers substructures of type EP, namely paths that ignores the family order in forests, is the one that performed the best based on both AUC and F-scores. Further experiments are needed to compare these kernels more accurately, but we argue that our platform is an efficient way to do so.

Acknowledgements This work was supported by JSPS grant 23300061. We thank anonymous reviewers for helpful comments.

References

Collins, M. and Duffy, N. Convolution Kernels for Natural Language. In *Advances in NIPS*, pp. 625–632, 2001.

Culotta, A. and Sorensen, J. Dependency tree kernels for relation extraction. In *ACL 2004*, pp. 423 – 429, 2004.

Demaine, E. D., Mozes, S., Rossman, B., and Weimann, O. An optimal decomposition algorithm for tree edit distance. In *The 34th International Colloquium on Automata, languages and Programming (ICALP)*, 2007.

Harchaoui, Z. and Bach, F. Image classification with segmentation graph kernels. In *CVPR*, 2007.

Hashimoto, K., Goto, S., Kawano, S., Aoki-Kinoshita, K. F., and Ueda, N. Kegg as a glycome informatics resource. *Glycobiology*, 16:63R – 70R, 2006.

Haussler, D. Convolution kernels on discrete structures. Technical report, UCSC, 1999. USCS-CRL-99-10.

Hofmann, T., Scholkopf, B., and Smola, A.J. Kernel methods in machine learning. *Annals of Statistics*, 36(3):1171, 2008.

Kashima, H., Tsuda, K., and Inokuchi, A. Marginalized kernels between labeled graphs. In *Proceedings of the 20th ICML*, pp. 321–328, 2003.

Leslie, C., Eskin, E., and Noble, W. S. The spectrum kernel: a string kernel for svm protein classific ation. In *Proc. of PSB 2002*, pp. 564–575, 2002.

Martins, A.F.T., Smith, N.A., Xing, E.P., Aguiar, P.M.Q., and Figueiredo, M.A.T. Nonextensive information theoretic kernels on measures. *The Journal of Machine Learning Research*, 10:935–975, 2009. ISSN 1532-4435.

Shervashidze, N. and Borgwardt, K.M. Fast subtree kernels on graphs. *Advances in NIPS 22*, 2009.

Shin, K. and Kuboyama, T. A generalization of Haussler's convolution kernel: mapping kernel. In *Proceedings of the 25th ICML*, pp. 944–951, 2008.

Shin, K. and Kuboyama, T. Generalization of haussler's convolution kernel - mapping kernel and its application to tree kernels. *J. Comput. Sci. Technol*, 25(5)::1040–1054, 2010.

Sonnenburg, S., Rieck, K., and Rätsch, G. Large scale learning with string kernels. In *Large Scale Kernel Machines*, pp. 73–103. MIT Press, 2007.

Taï, K. C. The tree-to-tree correction problem. *JACM*, 26 (3):422–433, July 1979.

Vert, J.-P., Saigo, H., and Akutsu, T. Local alignment kernels for protein sequences. In Schölkopf, Bernhard, Tsuda, Koji, and Vert, Jean-Philippe (eds.), *Kernel Methods in Computational Biology*. MIT Press, 2004.