Linear-time algorithms for the subpath kernel

Kilho Shin[1]
Graduate School of Applied Informatics, University of Hyogo
Minatojima-Minamimachi, Chuo, Kobe, Japan
yshn@ai.u-hyogo.ac.jp

Taichi Ishikawa
Graduate School of Applied Informatics, University of Hyogo
Minatojima-Minamimachi, Chuo, Kobe, Japan
t.i.tkgw@gmail.com

Abstract

The subpath kernel is a useful positive definite kernel, which takes arbitrary rooted trees as input, no matter whether they are ordered or unordered, We first show that the subpath kernel can exhibit excellent classification performance in combination with SVM through an intensive experiment. Secondly, we develop a theory of irreducible trees, and then, using it as a rigid mathematical basis, reconstruct a bottom-up linear-time algorithm for the subtree kernel, which is a correction of an algorithm well-known in the literature. Thirdly, we show a novel top-down algorithm, with which we can realize a linear-time parallel-computing algorithm to compute the subpath kernel.

2012 ACM Subject Classification Theory of computation → Kernel methods

Keywords and phrases tree, kernel, suffix tree

Digital Object Identifier 10.4230/LIPIcs.CPM.2018.22

1 Introduction

Recently, designing efficient kernel functions for tree-type data has become more important in various fields including bioinformatics, natural language processing (NLP) and so forth. First of all, we have many applications where data are represented in the form of trees. For instance, glycans are attracting wide attention of researchers as the third life molecule that follows DNA and proteins, and their chemical structures are trees in contrast that DNA and proteins are sequences. Also, results of syntactical analysis of natural languages and documents created according to markup languages such as HTML/XML are all represented as trees. In this paper, a tree always means a rooted tree.

To capture features of tree-type data, kernel functions are known useful. The basic nature of kernel functions is a measure to evaluate similarity of data. Furthermore, when used with various methods of multivariate analysis such as PCA and SVM, kernels are significantly useful for the purposes of classification, clustering, regression and so forth.

Kernel functions applicable to tree-type data have been intensively studied in the literature. In fact, since Haussler first introduced a generic class of positive definite kernels for semi-structured data, named the *convolution kernel* [6], a variety of tree kernels have been proposed: for example, Collins and Duffy designed the first tree kernel for the study of parse trees of natural languages [3]; Kashima and Koyanagi relaxed application-specific constraints of the

[1] This work was supported by JSPS KAKENHI Grant Number JP17H007623 and JP16K12491.

29th Annual Symposium on Combinatorial Pattern Matching (CPM 2018).
Editors: Gonzalo Navarro, David Sankoff, and Binhai Zhu; Article No. 22; pp. 22:1–22:13
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

parse tree kernel by Collins and Duffy and introduced the *elastic tree kernel* [8]. The idea that underlies these kernels is to count shared sub-structures.

In parallel, Shin and Kuboyama [14] showed a method to derive kernel functions from various tree edit distances such as Taï distance [16] and the constrained distance [18]. In fact, these counting-up-based and distance-based tree kernels can be discussed within the common generalized framework of the *mapping kernel* [14]. In [15], a wide variety of tree kernels designed within the mapping kernel framework are investigated from the accuracy performance point of view.

This paper focuses on the *subpath kernel*, which extends and generalizes the *spectrum kernel* [11] and the all-sequences kernel for strings, and the spectrum kernel for trees [10]. We see that the subpath kernel outperforms the benchmark tree kernels in prediction performance, and its superiority is statistically significant. Furthermore, we present linear-time fast algorithms to compute it with mathematical proof for their correctness.

2 The Subpath Kernel (SPK) for Trees

The *subpath kernel* takes two rooted labeled trees T_1 and T_2 as an input and returns a real value.

The idea of the subpath kernel dates back to the *spectrum kernel* for strings that Leslie et al. proposed [11]. Leslie's spectrum kernel counts up all the pairs of congruent substrings of a fixed length such that one substring appears in the first input string, while the other does in the second.

Kuboyama et al. [10] have extended Leslie's idea to trees and introduced the spectrum tree kernel, which counts up congruent *subpaths* instead of substrings.

- Starting from an arbitrary vertex v, a subpath of length q is the sequence of vertices $\pi = (v, p(v), p^2(v), \dots, p^{q-1}(v))$, where $p(w)$ denotes the parent of a vertex v.
- From π, we obtain a string $\ell(\pi) = \ell(v)\ell(p(v)) \dots \ell(p^{q-1}(v)) \in \Sigma^q$, where Σ is an alphabet of labels and $\ell(w)$ is the label of a vertex w.
- For a tree T and $s \in \Sigma^q$, we let $c(s; T)$ denote the number of subpaths π with $\ell(\pi) = s$.
- Finally, the q-spectrum kernel K_q is define by

$$K_q(T_1, T_2) = \sum_{s \in \Sigma^q} c(s; T_1) \cdot c(s; T_2).$$

▶ **Definition 1.** With a *decay factor* $\lambda \in (0, 1)$ and spectrum tree kernels K_q, the subpath kernel is defined by $\mathrm{SPK}(T_1, T_2) = \sum_{q \in \mathbb{N}} \lambda^q K_q(T_1, T_2)$.

The subpath kernel is positive definite and the yields an inner product function in the reproducing kernel Hilbert space [1].

3 High accuracy performance as a similarity measure.

We first see that the subpath kernel has prediction accuracy superior to major tree kernels known in the literature through an intensive experiment.

3.1 Datasets

In the experiment, we use ten datasets, which cover three different areas of applications: bioinformatics (three), natural language processing (six) and web access analysis (one). Three (Colon, Cystic and Leukemia) are retrieved from the KEGG/GLYCAN database ([5]) and contain glycan structures annotated relating to colon cancer, cystic fibrosis and leukemia

Table 1 Datasets: Number of examples, averaged sizes and averaged heights of trees.

Dataset	AIMed	BioInfer	Colon	Cystic	Hprd50	Iepa	Leukemia	Lll	Syntactic	Web
Examples	100	70	134	160	100	100	442	100	225	500
Size	94.4	116.4	8.4	8.3	84.4	105.2	13.5	106.4	19.7	12.0
Height	13.5	14.1	5.6	5.0	12.7	13.6	7.4	14.3	6.5	4.3

Table 2 Accuracy scores, averaged ranks, and p-values in Hommel test

Kernel	AIMed	BioInf.	Colon	Cystic	Hprd50	Iepa	Leuk.	Lll	Syn.	Web	Av. Rnk.	p-Val.
Spk	**0.75**	**0.84**	**0.91**	**0.79**	**0.70**	**0.71**	**0.90**	**0.65**	**0.87**	**0.82**	1.1	–
Prs	**0.75**	0.81	0.77	0.60	0.64	0.60	0.89	0.63	0.65	0.77	3.75	0.0031
Els	0.74	0.81	0.82	0.67	0.60	0.64	0.88	0.60	0.68	0.77	3.95	0.0020
Cdk	**0.75**	0.81	0.83	0.66	0.57	0.59	0.87	0.59	0.68	0.77	4.65	0.0001
Ctk	0.73	0.78	0.88	0.72	0.60	0.60	0.88	0.59	0.76	0.78	4.0	0.0016
Stk	0.72	0.79	0.90	0.73	0.61	0.59	0.88	0.60	0.82	0.78	3.55	0.0034

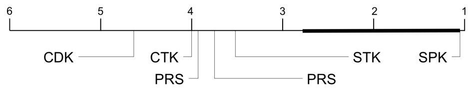


Figure 1 Hommel test: $p < 0.01$.

cells. One (Syntactic) is the dataset PropBank provided in [12]. This dataset includes parse trees labeled with two syntactic role classes for modeling the syntactic/semantic relation between a predicate and the semantic roles of its arguments in a sentence. Five (AIMed, BioInfer, HPRD50 IEPA and LLL) are the corpora that include parse trees obtained by analyzing documents regarding protein-protein interaction (PPI) extraction ([13]). PPI is an intensively studied problem of the BioNLP field. The remaining one (Web), used in [17], consists of trees representing web-page accesses by users, and the annotation is based on whether the user is from a .edu site or not. Table 1 describes the basic features of these datasets.

3.2 Kernels to compare

The benchmark kernels to compare with are the parse tree kernel (Prs) [3], the elastic tree kernel (Els) [8], the sparse path kernel (STK) and the contiguous kernel (Crs). The Stk and Ctk are two kernels that performed the best in an intensive experiment in [15]. Each kernel includes two adjustable parameters α and β with $1 \geq \alpha \geq \beta > 0$.

3.3 Experimental results

Table 2 shows the results of the experiments. We run ten-fold cross validation with a libSVM classifier [2] and measure accuracy scores by the accuracy index, determined by $\text{Acc} = \frac{\text{TP+TN}}{\text{TP+TN+FP+TN}}$. The accuracy values in Table 2 are the best values obtained through grid search changing the parameters. The subpath kernel includes one adjustable parameter to tune, a decay factor λ, while the others include two, α and β.

Remarkably, for all of the datasets tested, the subtree kernel is ranked top. Also, Table 2 specifies the p-values obtained when we perform the Hommel multiple comparison test as recommended by [4]. With a significance level 0.01, we can conclude that the exhibited superiority of the subpath kernel is statistically significant (Figure 1).

4 Linear-time algorithms for the subpath kernel

The subpath kernel "is known" to be one of a few tree kernels that have linear-time complexity in the size of the input trees. In fact, [9] presented a linear-time algorithm but at the same time reported not so good accuracy performance. For example, the accuracy scores the subpath kernel with LEUKEMIA was the lowest of the five tree kernels tested. This could not help raising a question with us, and we have found the reason for this. The algorithm proposed in [9] was wrong.

In this section, we reconstruct the algorithm based on the mathematically rigid ground, a theory of irreducible trees (Section 4.1) and further, introduce a novel algorithm for the subpath kernel, which realizes parallel computation of the subpath kernel in combination with the corrected algorithm.

4.1 A theory of irreducible trees

An irreducible tree is rooted, ordered and labeled. A rooted tree T is a partially ordered set (poset) with respect to an *generation order*: $v < w$ means that a vertex v is an ancestor of another vertex w, and hence, the root r_T of T is the unique minimum vertex. Furthermore, we let $p(v)$ denote the parent of a vertex v, and $p^k(v)$ does the ancestor of v for $k > 0$ such that there are exactly $k-1$ intermediate vertices between v and $p^k(v)$. If a vertex is not the parent of any other vertices, we call it a *leaf*. From the generation order, the *nearest common ancestor* of a pair vertices (v, w) can be naturally introduced.

▶ **Definition 2.** For any $\{v, w\} \subseteq T$, $v \smile w = \max_{\leq}\{u \in T \mid u \leq v, u \leq w\}$ is the *nearest common ancestor* of v and w.

To define an *ordered* tree T, it is common to introduce a sibling order, but we deploy the following definition, since we are only interested in a numbering of the leaves of T.

▶ **Definition 3.** When the entire leaves of a rooted tree T is numbered as $(l_1, \dots, l_n)$, T is said to be *ordered*, if, and only if, $l_i \smile l_k = l_i \smile l_j \smile l_k$ holds for any $1 \leq i < j < k \leq n$.

▶ **Proposition 4.** *For a vertex v of an ordered tree, $\{i \mid l_i \geq v\} = [a, b]$ holds for some a and b in $\{1, \dots, n\}$. We say that $[a, b]$ is the* span *of v.*

Proof. We let $a = \min\{i \mid l_i \geq v\}$ and $b = \max\{i \mid l_i \geq v\}$. For any $i \in (a, b)$, $l_i \smile l_a \geq l_a \smile l_b \geq v$ holds. In particular, we have $l_i \geq v$. ◀

Finally, we define an *irreducible* tree in Definition 5.

▶ **Definition 5.** A rooted and ordered tree is *irreducible*, iff no vertex has only one child.

For study of irreducible trees, α_i defined below plays a crucial role.

▶ **Definition 6.** For $i \in \{1, 2, \dots, n-1\}$, α_i denotes $l_i \smile l_{i+1}$.

The rightmost (leftmost) leaf of a vertex v is l_b (l_a), when v spans $[a, b]$. The rightmost leaf of v can be characterized by α_i as follows.

▶ **Proposition 7.** *We assume $v < l_i$. l_i is the rightmost leaf of v, if, and only if, $v > \alpha_i$.*

Proof. If l_i is the rightmost leaf, $\alpha_i \ngeq v$ holds, since $l_{i+1} \geq v$ holds, otherwise; If $\alpha_i < v$, $l_i \smile l_k \leq \alpha_i < v$, and therefore, $l_k \ngeq v$ holds for any $k > i$. ◀

Any non-leaf vertex v has at least one i such that $v = \alpha_i$. We have

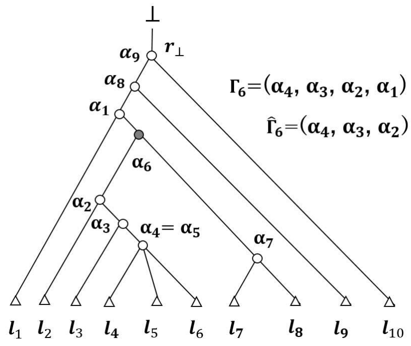


Figure 2 An irreducible tree.

▶ **Proposition 8.** *For a non-leaf vertex v, we let w be the leftmost child of v and l_i be the rightmost leaf of w. Then, $i = \min\{j \mid \alpha_j = v\}$ holds.*

Proof. By Proposition 7, $\alpha_i < w$ holds. On the other hand, since l_i is not the rightmost leaf of v, $\alpha_i \geq v$ holds. $\alpha_i = v$ immediately follows. ◀

▶ **Definition 9.** For an intermediate vertex v, $\gamma(v)$ denotes $\min\{i \mid \alpha_i = v\}$.

For example, in Figure 2, α_4 and α_5 are identical, and $\gamma(\alpha_5) = 4$ holds. Corollary 10 will play a central role when we introduce a top-down algorithm for the subpath kernel in Section 4.5. For the convenience of explanation, without loss of generality, we add an imaginary root $\perp$ on top of r_T and let $\alpha_n =\perp$.

▶ **Corollary 10.** *If a non-leaf vertex v that spans $[a, b]$ has children $w_1, \ldots, w_t$, their rightmost leaves are $l_{i_1}, \ldots, l_{i_t}$ with $\{i_1, \ldots, i_t\} = \{j \mid j \in [a, b], \alpha_j \leq v\}$.*

Proof. We assume $i_1 < \cdots < i_t$. $i_t = b$ follows from Proposition 7. l_{i_1} is the rightmost leaf of w_1 by Proposition 8. To verify that l_{i_i} is the rightmost leaf of w_i for $1 < i < t$, we have only to eliminate $w_1, \ldots, w_{i-1}$ and their subordinates and then to apply Proposition 8. ◀

Theorem 14 and 16 stated below will be a theoretical basis to justify the correctness of the bottom-up traversal algorithm introduced in [7] and to correct errors of the algorithm to compute the subpath kernel proposed in [9]. We start with defining Γ_i and $\widehat{\Gamma}_i$.

▶ **Definition 11.** Γ_i and $\widehat{\Gamma}_i$ are the subsequences of the subpath $(p^1(l_i), p^2(l_i), \ldots, p^{\ell_i}(l_i) = r_T)$ consisting of the vertices $p^j(l_i)$ such that $\gamma(p^j(l_i)) < i$ and $p^j(l_i) > \alpha_i$, respectively.

▶ **Example 12.** In Figure 2, Γ_i and $\widehat{\Gamma}_i$ for $i = 1, \ldots, 10$ are determined as follows.

i	Γ_i	$\widehat{\Gamma}_i$	i	Γ_i	$\widehat{\Gamma}_i$
1	$()$	$()$	6	$(\alpha_4, \alpha_3, \alpha_2, \alpha_1)$	$(\alpha_4, \alpha_3, \alpha_2)$
2	(α_1)	$()$	7	(α_6, α_1)	$()$
3	(α_2, α_1)	$()$	8	$(\alpha_7, \alpha_6, \alpha_1)$	$(\alpha_7, \alpha_6, \alpha_1)$
4	$(\alpha_3, \alpha_2, \alpha_1)$	$()$	9	(α_8)	(α_8)
5	$(\alpha_4, \alpha_3, \alpha_2, \alpha_1)$	$()$	10	(α_9)	(α_9)

▶ **Proposition 13.** *Any $v \in \widehat{\Gamma}_i$ has $\gamma(v) < i$. Hence, $\widehat{\Gamma}_i \subseteq \Gamma_i$ holds.*

Proof. $l_i \smile l_k \leq \alpha_i < v$ holds for $k > i$, and hence, $l_k \ngeq v$ holds. ◀

▶ **Theorem 14.** *The sequence* $\prod_{i=1}^{n} \left[(l_i) \cdot \widehat{\Gamma}_i\right]$ *yields the bottom-up traversal of the vertices of* T*. Given two sequences* s *and* t*,* $s \cdot t$ *denotes their concatenation.*

Proof. Since every vertex v has a unique leftmost leaf, it appears in the sequence exactly once. On the other hand, for a vertex w with $w > v$, the span of w is a subset of the span of v, and hence, w appears before v in the sequence. ◀

▶ **Example 15.** In Figure 2, $(l_i) \cdot \widehat{\Gamma}_i$ for $i = 1, \ldots, 10$ is determined as follows.

i	$\widehat{\Gamma}_i$	$(l_i) \cdot \widehat{\Gamma}_i$	i	$\widehat{\Gamma}_i$	$(l_i) \cdot \widehat{\Gamma}_i$
1	$()$	(l_1)	6	$(\alpha_4, \alpha_3, \alpha_2)$	$(l_6, \alpha_4, \alpha_3, \alpha_2)$
2	$()$	(l_2)	7	$()$	(l_7)
3	$()$	(l_3)	8	$(\alpha_7, \alpha_6, \alpha_1)$	$(l_8, \alpha_7, \alpha_6, \alpha_1)$
4	$()$	(l_4)	9	(α_8)	(l_9, α_8)
5	$()$	(l_5)	10	(α_9)	(l_{10}, α_9)

In fact, their concatination $(l_1, l_2, l_3, l_4, l_5, l_6, \alpha_4, \alpha_3, \alpha_2, l_7, l_8, \alpha_7, \alpha_6, \alpha_1, l_9, \alpha_8, l_{10}, \alpha_9)$ gives the bottom-up traversal of the vertices of the tree.

▶ **Theorem 16.** *For* $i = 1, \ldots, n-1$*, the following hold.*
1. *If* $\alpha_i \in \Gamma_i$*,* $\Gamma_{i+1} = \Gamma_i \setminus \widehat{\Gamma}_i$*.*
2. *If* $\alpha_i \notin \Gamma_i$*,* $\Gamma_{i+1} = (\alpha_i) \cdot \left(\Gamma_i \setminus \widehat{\Gamma}_i\right)$*.*

Proof. If j with $j < i$ meets $\alpha_j < l_{i+1}$, $\alpha_j \leq \alpha_i$ holds. In fact, since $\alpha_j \geq l_j \smile l_{i+1}$, we have $\alpha_j = l_j \smile l_{i+1} \leq \alpha_i$, and hence, $\alpha_j \in \Gamma_i \setminus \widehat{\Gamma}_i$. If $\alpha_i \in \Gamma_i \setminus \widehat{\Gamma}_i$, $\Gamma_{i+1} = \Gamma_i \setminus \widehat{\Gamma}_i$ holds. Otherwise, we prepend α_i to $\Gamma_i \setminus \widehat{\Gamma}_i$ to obtain Γ_{i+1}. ◀

▶ **Example 17.** In Figure 2, $\alpha_i \in \Gamma_i$ holds only for $i = 5$. In fact, $\Gamma_6 = (\alpha_4, \alpha_3, \alpha_2, \alpha_1)$ is identical to $\Gamma_5 \setminus \widehat{\Gamma}_5 = (\alpha_4, \alpha_3, \alpha_2, \alpha_1) \setminus ()$. For the other i, $\Gamma_{i+1} = (\alpha_i) \cdot \left(\Gamma_i \setminus \widehat{\Gamma}_i\right)$ holds. For example, $\widehat{\Gamma}_5 = (\alpha_4, \alpha_3, \alpha_2)$ and $(\alpha_6) \cdot \left(\Gamma_5 \setminus \widehat{\Gamma}_5\right) = (\alpha_6, \alpha_1) = \Gamma_7$ hold.

▶ **Definition 18.** $h : T \to \mathbb{N}$ is a *height function*, if $h(v) > h(w)$ holds for any $(v, w) \in T^2$ with $v > w$, and if $h(r_T) = 0$.

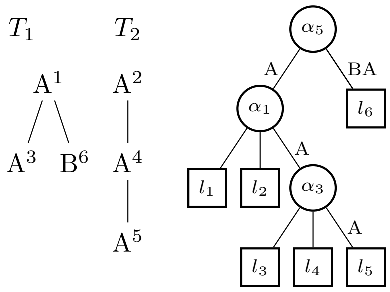
A height function can be defined for an arbitrary rooted tree, which is not necessarily irreducible.

▶ **Example 19.** For a rooted tree T and a vertex v in T, we let h_v denote the number of ancestors of v: that is, $h_v = |\{w \in T \mid w < v\}|$. Evidently, h_v is a height function.

4.2 Suffix arrays and suffix trees

The well known suffix tree is an example of irreducible trees.

Consider two rooted labeled trees T_1 and T_2, which are not necessarily ordered. For each vertex $v \in T_i$, its entire path is the sequence of vertices $\left(v, p(v), p^2(v), \ldots, p^{h_v}(v) = r_T\right)$, and the *suffix* of v is the string "$L(v)L(p(v)) \ldots L(p^{h_v}(v))$", where $L(v)$ denotes the label of a vertex v. To determine the suffix array for T_1 and T_2, we collect all the suffices across all the vertices of T_1 and T_2, and then sort them in the lexicographical order as strings. The suffix array includes $n = |T_1| + |T_2|$ entries. In Figure 3, the first column of the right table describes the suffix array for T_1 and T_2 depicted by the same figure.



Suffix	LCP (h)	c_1	c_2
A^1	1	1	0
A^2	1	0	1
A^3A^1	2	1	0
A^4A^2	2	0	1
$A^5A^4A^2$	0	0	1
B^6A^1	-1	1	0

Figure 3 A suffix array (right) and the associated suffix tree (middle).

The *suffix tree* ST for T_1 and T_2 is derived from the suffix array. The leaf vertices $l_1, l_2, \ldots, l_{|T_1|+|T_2|}$ of the suffix tree uniquely correspond to the entries of the suffix array in the order in which they appear in the array: the leaf l_i represents the suffix s_i, which is the entry of the suffix array at position i. Because there is a one-to-one correspondence between the entries of the suffix array and the vertices of T_1 and T_2, each leaf of the suffix tree also uniquely represents a vertex in T_1 or T_2. Furthermore, each edge of ST is labeled with a string of vertex labels so that the following conditions are met:

1. The concatenation of the edge labels of the path from the root r_{ST} to l_i is identical to s_i.
2. The labels of two downward edges from the same vertex of the suffix tree have no common prefix.

Combined with the condition that the suffix tree is irreducible, these conditions uniquely determine the suffix tree ST.

The center tree displayed in Figure 3 describes the suffix tree derived from T_1 and T_2 depicted in the same figure. Note that an edge label is omitted, if it is an empty string. For example, l_5 corresponds to the fifth entry of the suffix array, and therefore, represents the vertex A^5 in T_2. In fact, the downward concatenation of the labels for the entire path of l_5 is identical to $s_5 = \mathrm{AAA}$.

An LCP value $h(i)$ for an entry at the position i in a suffix array gives the length of the longest common prefix between s_i and s_{i+1}. For example, in Figure 3, we have $s_2 = \mathrm{A}$ and $s_3 = \mathrm{AA}$, and therefore, the LCP value $h(2)$ turns out to be 1. For the last entry of the suffix array, we define its LCP value to be -1 for convenience of computation. In the corresponding suffix tree, $h(i)$ determines a height of the intermediate vertex α_i.

Finally, we introduce two arrays c_1 and c_2 in addition to the LCP array h. $c_1(i)$ and $c_2(i)$ for the entry at position i of a suffix array describes to which the suffix s_i belongs, T_1 or T_2: $c_1(i) = 1$, if s_i is a subpath of T_1, and $c_2(i) = 1$, if s_i is a subpath of T_2.

To compute the subpath kernel, we have only to input these three arrays h, c_1 and c_2 into algorithms.

In [9], an algorithm to generate suffix arrays and suffix trees whose time complexity is linear to the size of trees is proposed.

4.3 Reconstruction of the bottom-up traversal algorithm of [7]

We first reconstruct a linear-time bottom-up traversal algorithm based on the theory shown in Section 4.1, which is equivalent to the one introduced in [7]. Algorithm 1 shows the algorithm Theorem 14 and 16 clearly explain the algorithm and at the same time give a mathematical justification for its correctness.

Algorithm 1 A bottom-up traversal algorithm of an irreducible tree.

Require: $(h(\alpha_1), \dots, h(\alpha_n)) \in \mathbb{N}^n$ ▷ $h(\alpha_i)$: the height of α_i

Ensure: A sequence $(v_1, \dots, v_{|T|})$ of vertices of T in the bottom-up traversal order.

Clear a stack Γ ▷ $\mathrm{pop}_\Gamma, \mathrm{push}_\Gamma(\cdot), \mathrm{top}_\Gamma$ are operations on Γ
for $i = 1, 2, \dots, n$ **do**
  Write l_i
  while $\Gamma \neq \emptyset \wedge h(\mathrm{top}_\Gamma) > h(\alpha_i)$ **do** ▷ $\mathrm{top}_\Gamma > \alpha_i \Leftrightarrow h(\mathrm{top}_\Gamma) > h(\alpha_i)$
    Write top_Γ
    Do pop_Γ
  end while
  if $\Gamma = \emptyset \vee h(\mathrm{top}_\Gamma) \neq h(\alpha_i)$ **then** ▷ $\alpha_i \in \Gamma \Leftrightarrow \alpha_i = \mathrm{top}_\Gamma$
    Do $\mathrm{push}_\Gamma(\alpha_i)$
  end if
end for

1. The first-in-last-out stack Γ holds Γ_i for each i of the **for** loop. If $\Gamma_i = (v_1, \dots, v_k)$ with $v_1 > \dots > v_k$, v_1 is stored at the top, and v_k is stored at the bottom of Γ.
2. Note that, if α_i and α_j are comparable with respect to the generation order, we have $\alpha_i < \alpha_j \Leftrightarrow h(\alpha_i) < h(\alpha_j)$. Therefore, the exit condition of the **while** loop is for $h(\mathrm{top}_\Gamma) \leq h(\alpha_i)$ to hold.
3. The **while** loop outputs the elements of $\widehat{\Gamma}_i$ in the decreasing direction of the generation order. Hence, Theorem 14 asserts that the algorithm outputs the vertices of T in the bottom-up traverse order.
4. The **while** loop also eliminates $\widehat{\Gamma}_i$ from Γ_i in the stack Γ. This is done by performing pop_Γ. By Theorem 16, this updates Γ_i to Γ_{i+1}, if $\alpha_i \in \Gamma_i$.
5. If $\alpha_i \notin \Gamma_i$, by Theorem 16, α_i is to be prepended to $\Gamma_i \setminus \widehat{\Gamma}_i$ to obtain Γ_{i+1}. In fact, this is done by performing $\mathrm{push}_\Gamma(\alpha_i)$.
6. To know whether $\alpha_i \in \Gamma_i$, we have only to examine whether $\mathrm{top}_\Gamma = \alpha_i$, equivalently, whether $h(\mathrm{top}_\Gamma) = h(\alpha_i)$.

4.4 A linear-time bottom-up algorithm for the subpath kernel

In [9], the key formula to compute $\mathrm{SPK}(T_1, T_2)$ is given as

$$\mathrm{SPK}(T_1, T_2) = \sum_{v \in ST} (w(h(v)) - w(h(p(v)))) \cdot c_1(v) \cdot c_2(v). \tag{1}$$

The function w is determined by $w(h) = \sum_{i=1}^{h} \lambda^i$ and $c_i(v)$ is the number of leaves below v that belong to T_i. Algorithm 2 computes $\mathrm{SPK}(T_1, T_2)$ by Eq. (1) and is also a correction to the algorithm exhibited in [9].

The steps commented with "▷ For bottom-up traversal" are to perform bottom-up traversal of vertices of the suffix tree ST derived from T_1 and T_2, the following are added to Algorithm 1.

- The stack Γ stores a triplet (α_i, c_1, c_2) (Step 13) instead of α_i. The second and third components store intermediate values to compute $c_1(v)$ and $c_2(v)$.
- The value $(w(h(v)) - w(h(p(v)))) \cdot c_1(v) \cdot c_2(v)$ computed for each vertex v is accumulated in the variable *kernel* (Step 9).
- When $\alpha_i \in \Gamma_i$ (Step 14), the triplet (v, c_1', c_2') is updated so that leaves found during eliminating $\widehat{\Gamma}_i$ from Γ_i are counted (Step 16).

Algorithm 2 A bottom-up algorithm for SPK (correction to [9]).

Require: $(h(\alpha_1),\dots,h(\alpha_n)) \in \mathbb{N}^n$; $(c_1(l_1),\dots,c_1(l_n)) \in \mathbb{Z}_2^n$; $(c_2(l_1),\dots,c_2(l_n)) \in \mathbb{Z}_2^n$ ▷ $h(\alpha_i)$: the height of α_i; $c_i(l_j)$: belonging of l_j to T_i
Ensure: $\mathrm{SPK}(T_1, T_2)$

1: **procedure** $\mathrm{SPK}_{\mathrm{BU}}(h(\alpha_1),\dots,h(\alpha_n); c_1(l_1),\dots,c_1(l_n); c_2(l_1),\dots,c_2(l_n))$
2:     Clear a stack Γ ▷ For bottom-up traversal
3:     Let $kernel = 0$
4:     **for** $i = 1, 2, \dots, n$ **do** ▷ For bottom-up traversal
5:         Let $c_1 = c_1(l_i)$ and $c_2 = c_2(l_i)$
6:         **while** $\Gamma \neq \emptyset \wedge h(\mathrm{top}_\Gamma[0]) > h(\alpha_i)$ **do** ▷ For bottom-up traversal
7:             Let $(v, c_1', c_2') = \mathrm{top}_\Gamma$
8:             Do pop_Γ ▷ For bottom-up traversal
9:             Let $c_1 = c_1 + c_1'$ and $c_2 = c_2 + c_2'$ ▷ $c_i = c_i(v)$
10:             **if** $h(v) \neq 0$ **then**
11:                 Let $kernel = kernel + (w(h(v)) - w(h(p(v)))) \cdot c_1 \cdot c_2$
12:             **end if** ▷ Eq. (1)
13:         **end while** ▷ For bottom-up traversal
14:         **if** $\Gamma = \emptyset \vee h(\mathrm{top}_\Gamma[0]) \neq h(\alpha_i)$ **then** ▷ For bottom-up traversal
15:             Do $\mathrm{push}_\Gamma(\alpha_i, c_1, c_2)$ ▷ For bottom-up traversal
16:         **else**
17:             Let $(v, c_1', c_2') = \mathrm{top}_\Gamma$
18:             Let $\mathrm{top}_\Gamma = (v, c_1' + c_1, c_2' + c_2)$
19:         **end if** ▷ For bottom-up traversal
20:     **end for** ▷ For bottom-up traversal
21: **end procedure**

Algorithm 3 Computation of $p(v)$.

Require: $(h(\alpha_1),\dots,h(\alpha_n)) \in \mathbb{N}^n$; $\Gamma = \Gamma_i \setminus \{v_1,\dots,v_j\}$; $v = v_j$ ▷ $\{v_1,\dots,v_j\} \subseteq \widehat{\Gamma}_i$
Ensure: $p(v)$

1: **if** $\Gamma = \emptyset \vee h(\mathrm{top}_\Gamma[0]) < h(\alpha_i)$ **then** ▷ $\mathrm{top}_\Gamma[0] < \alpha_i \Leftrightarrow h(\mathrm{top}_\Gamma[0]) < h(\alpha_i)$
2:     **return** α_i
3: **else**
4:     **return** $\mathrm{top}_\Gamma[0]$
5: **end if**

We should be careful when computing $p(v)$ in Step 10. Proposition 13 asserts that, when (v, c_1', c_2') is the last element eliminated from Γ, $p(v) = \mathrm{top}_\Gamma[0]$ holds, if $\mathrm{top}_\Gamma[0] > \alpha_i$, and $p(v) = \alpha_i$ holds, otherwise.

The most important error of the algorithm of [9] was that it wrongly assumed $p(v) = \mathrm{top}_\Gamma[0]$ unconditionally. For example, for two trees T_1 and T_2 and the suffix tree derived from them depicted by Figure 4, the subpath kernel value for T_1 and T_2 turns out to be $\lambda^2 + 3\lambda$, because the subpaths of T_1 are $\{\mathrm{A}^1, \mathrm{B}^3, \mathrm{B}^3\mathrm{A}^1\}$, while the subpaths of T_2 are $\{\mathrm{A}^2, \mathrm{B}^4, \mathrm{B}^5, \mathrm{B}^4\mathrm{A}^2, \mathrm{B}^5\mathrm{B}^4, \mathrm{B}^5\mathrm{B}^4\mathrm{A}^2\}$.

In Algorithm 1, the bottom-up traversal visits α_3, when $i = 4$. Since $p(\alpha_3)$ is α_4, the value $(w(h(\alpha_3)) - w(h(\alpha_4))) \cdot 1 \cdot 1 = \lambda^2 + \lambda - \lambda = \lambda^2$ is added to the variable *kernel* at Step 10. On the other hand, since $\Gamma_4 = (\alpha_3, \alpha_2)$ holds, the algorithm of [9] adds $(w(h(\alpha_3)) - w(h(\alpha_2))) \cdot 1 \cdot 1 = \lambda^2 + \lambda$, instead. By this, the kernel value that the algorithm of [9] computes becomes $\lambda^2 + 4\lambda$.

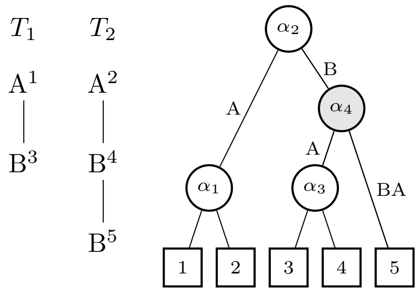


Figure 4 A counter example.

Algorithm 4 Decomposition into child trees.

Require: a; $h(v)$; $\{h(\alpha_1), \dots, h(\alpha_n)\} \subset \mathbb{N}^n$ ▷ l_a: the leftmost leaf of v

Ensure: $((i_1, h_1), \dots, (i_t, h_t))$

▷ $(w_1, \dots, w_t)$: the children of v; l_{i_j} is the rightmost leaf of w_j; $h_j = h(w_j)$

Let $i = a$

while true **do**

  $min_h = h(\alpha_i)$

  while $h(\alpha_i) > h(v)$ **do** ▷ $\alpha_i > v \Leftrightarrow h(\alpha_i) > h(v)$

    $min_h = \min\{h(\alpha_i), min_h\}$

    Let $i = i + 1$

  end while

  Write (i, min_h)

  if $h(\alpha_i) < h(v)$ **then** ▷ $\alpha_i < v \Leftrightarrow h(\alpha_i) < h(v)$

    return

  end if

  Let $i = i + 1$

end while

4.5 A Top-Down Algorithm for the subpath kernel

We introduce a novel algorithm that computes the subpath kernel leveraging recursive function calls. Algorithm 4 below is the key component of the algorithm, which decomposes a tree into a sequence of child trees. Corollary 10 guarantees the correctness of Algorithm 4.

Algorithm 5 defines the function $\mathrm{SPK_{TD}}$ that computes the subpath kernel. For convenience of explanation, we simply assume that we call the function $\mathrm{SPK_{TD}}$ specifying an interval of leaves as an input to obtain three values: the number of leaves that belong to T_1 in the interval; the number of leaves that belong to T_2 in the interval; and the kernel value computed for the interval. To be specific, $\mathrm{SPK_{TD}}(I)$ is formulated by

$$\mathrm{SPK_{TD}}(I) = \left(|STL_1 \cap I|, |STL_2 \cap I|, \sum_{i \in STL_1 \cap I} \sum_{j \in STL_2 \cap I} w(h(l_i \smile l_j)) \right), \tag{2}$$

where $STL_i = \{j \mid l_j \in T_i\}$ for $i = 1, 2$ and $I = [a, b]$ for $1 \le a \le b \le n$. Evidently, $\mathrm{SPK_{TD}}([1, n]) = \mathrm{SPK}(T_1, T_2)$ holds.

The function first performs Algorithm 4 to decompose the input interval of leaves, spanned by an intermediate vertex v in ST, into more than one intervals, each of which is spanned by a child of v (Step 5). Then, the function recursively applies itself to each interval obtained (Step 10).

The time complexity of computing $\mathrm{SPK_{TD}}(I)$ can be estimated to be $O\big(|I| \cdot \mathrm{dp}(v)\big)$, where the depth function $\mathrm{dp}(v)$ gives the longest length of downward paths in the suffix tree

Algorithm 5 A top-down algorithm for SPK

Require: a; b; $h(v)$; $(h(\alpha_1), \dots, h(\alpha_n)) \in \mathbb{N}^n$; $(c_1(l_1), \dots, c_1(l_n)) \in \mathbb{Z}_2^n$; $(c_2(l_1), \dots, c_2(l_n)) \in \mathbb{Z}_2^n$ ▷ v: a vertex that spans $(l_a, \dots, l_b)$ in ST

Ensure: $c'_1; c'_2; \mathit{kernel}'$ ▷ c'_i: the number of leaves of T_i in $[a, b]$; kernel': the kernel value for $[a, b]$

procedure $\mathrm{SPK}_{\mathrm{TD}}(a;\ b;\ h(v);\ h(\alpha_a), \dots, h(\alpha_b);\ c_1(l_a), \dots, c_1(l_b);\ c_2(l_a), \dots, c_2(l_b))$
  if $a = b$ **then**
    return $(c_1(l_a), c_2(l_b), 0.0)$
  end if
  Compute $((i_1, h_1), \dots, (i_t, h_t))$ by Algorithm 4
    ▷ $(w_1, \dots, w_t)$: the children of v; l_{i_j}: the leftmost leaf of w_j; $h_j = h(w_j)$
  Let $i_0 = a - 1$
  Let $c_1, c_2, \mathit{kernel} = 0, 0, 0.0$
  for $j = 1, \dots, t$ **do**
    Let $(c'_1, c'_2, \mathit{kernel}') = \mathrm{SPK}_{\mathrm{TD}}(i_{j-1} + 1; i_j; h_j; h(\alpha_{i_{j-1}+1}), \dots, h(\alpha_{i_j});$
      $c_1(l_{i_{j-1}+1}), \dots, c_1(l_{i_j}); c_2(l_{i_{j-1}+1}), \dots, c_2(l_{i_j}))$
    Let $\mathit{kernel} = \mathit{kernel} + w(h(v)) \cdot (c_1 \cdot c'_2 + c_2 \cdot c'_1) + \mathit{kernel}'$
      ▷ $w(h) = \lambda + \lambda^2 + \cdots + \lambda^h$, where λ is a decay factor
    Let $c_1, c_2 = c_1 + c'_1, c_2 + c'_2$
  end for
  return $(c_1, c_2, \mathit{kernel})$
end procedure

that start at the vertex v. This can be proven by mathematical induction as follows. Since Algorithm 4 scans all the leaves in I exactly one time for each, its time complexity is $O(|I|)$. As a result of running Algorithm 4, I is partitioned to intervals $I_1, \dots, I_t$. Since a suffix tree is irreducible, $t > 1$ holds. By the hypothesis of mathematical induction, we suppose that the time complexity to execute Algorithm 5 for I_i is $O\big(|I_i| \cdot \mathrm{dp}(w_i)\big)$, where w_i is a child of v in the suffix tree and spans I_i. Hence, the time complexity to execute Algorithm 5 for I is bounded above by

$$O(|I|) + \sum_{i=1}^{t} O\big(|I_i| \cdot \mathrm{dp}(w_i)\big) \le O(|I|) + \sum_{i=1}^{t} O\big(|I_i| \cdot (\mathrm{dp}(v) - 1)\big) = O\big(|I| \cdot \mathrm{dp}(v)\big)$$

In particular, the time complexity of Algorithm 5 for two trees T_1 and T_2 is bounded above by $O\big((|T_1| + |T_2|) \cdot \max\{\mathrm{dp}(T_1), \mathrm{dp}(T_2)\}\big)$, where $\mathrm{dp}(T_i)$ is the depth of the root of T_i in T_i.

Although this top-down algorithm is not linear with respect to the size of trees, it leads us to a hybrid parallel-computing linear-time algorithm as shown in the next section.

4.6 A hybrid parallel-computing linear-time algorithm

The top-down algorithm (Algorithm 5) enables us to compute the subpath kernel within the parallel computing framework. The idea is:

1. Apply the decomposition algorithm of Algorithm 4 until the entire tree is decomposed into an appropriate number of subtrees;
2. Use the bottom-up subpath kernel algorithm of Algorithm 2 to compute the kernel values for the subtrees obtained in Step 1;
3. Call the $\mathrm{SPK}_{\mathrm{TD}}$ function of Algorithm 5 recursively until reaching the subtrees precomputed in Step 2.

Since the time complexity of Step 1 and Step 3 is linear to $|T_1| + |T_2|$, since the parallelism is a constant number. On the other hand, the time complexity of Step 2 is evidently linear, and hence, the total time complexity of the hybrid algorithm is linear.

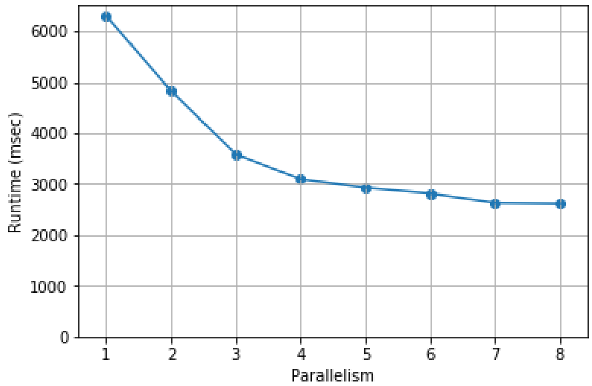


Figure 5 Runtime to compute 20 kernel values.

We conducted an experiment to compare the run-time of

For the experiment, we used a Mac Book Pro with 2.9GHz Quad Core Intel Core™ i7 CPU and ran the program written in Scala on macOS High Sierra 10.13.4. For parallel computation, we used the `ParArray` collection class.

The dataset used in the experiment consists of 20 pairs of randomly generated synthetic trees, each of which consists of $\frac{10^7-1}{9} = 1,111,111$ vertices and uniformly has degree 10 and height 7. The size of the alphabet of vertex lables is 100.

Figure 5 shows the run-time sores in milliseconds to compute the 20 kernel values, when we change the parallelism from 1 to 8. Since the CPU includes four cores, the runtime rapidly decreases until the parallelism reaches three. For the parallelism greater than three, although the gradient of the curve becomes gentler, the runtime steadily decreases.

5 Conclusion

We have shown superiority of the subpath kernel to other benchmark tree kernels in classification performance. The superiority has proven to be statistically significant through Hommel multiple comparison test with the significance level 0.01. In addition, we presented a linear-time bottom-up algorithm for the subpath kernel as well as a top-down algorithm. We have given mathematical proofs for the correctness of these algorithms based on a theory that we have developed. By combining the bottom-up and top-down algorithms, we can build hybrid linear-time parallel-computing algorithm, which has proven to improve the run-time performance through experiments. Considering all the above, we conclude that the subpath kernel should be the best kernel for analyzing tree data. As future studies, we will investigate their performance for other purposes of data analysis such as clussification and regression.

References

1. Christensen Berg, C. and R. J. P. R., Ressel. Harmonic analysis on semigroups. theory of positive definite and related functions. *Springer*, 1984.
2. C. C. Chang and C. J. Lin. Libsvm: a library for support vector machines, 2001. URL: `http://www.csie.ntu.edu.tw/~cjlin/libsvm/`.
3. M. Collins and N. Duffy. Convolution kernels for natural language. *Neural Information Processing Systems*, 2001.

4 J. Demšar. Statistical comparisons of classifiers over multiple data sets. *Journal of Machine Learning Theory*, 7:1–30, 2006.

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

6 D. Haussler. Convolution kernels on discrete structures. *UCSC-CRL 99-10*, 1999.

7 T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. *the 12th Annual Symposium on Combinatorial Pattern Matching. pp.*, 2001.

8 H. Kashima and T. Koyanagi. Kernels for semi-structured data. in: the 9th international conference on machine learning. *ICML*, 2002.

9 D. Kimura and H. Kashima. Fast computation of subpath kernel for trees. *ICML*, 2012.

10 T. Kuboyama, K. Hirata, H. Kashima, K.F. Aoki-Kinoshita, and H. Yasuda. A spectrum tree kernel. *JSAI*, 2007.

11 C. S. Leslie, E. Eskin, and W. Stafford Noble. The spectrum kernel: A string kernel for SVM protein classification. *Pacific Symposium on Biocomputing*, 2002.

12 Alessandro Moschitti. Example data for TREE KERNELS IN SVM-LIGHT. URL: `http://disi.unitn.it/moschitti/Tree-Kernel.htm`.

13 S. Pyysalo, A. Airola, J. Heimonen, J. Bjorne, F. Ginter, and T. Salakoski. Comparative analysis of five protein-protein interaction corpora. *BMC Bioinformatics*, 9(S-3), 2008.

14 K. Shin and T. Kuboyama. A generalization of Haussler's convolution kernel - mapping kernel. *ICML*, 2008.

15 K. Shin and T. Kuboyama. A comprehensive study of tree kernels. in: Jsai-isai post-workshop proceedings. *Lecture Notes in Articial Intelligence*, 2014.

16 K. C. Taï. The tree-to-tree correction problem. *journal of the ACM*, 1979.

17 M. J. Zaki and C. C. Aggarwal. XRules: An effective algorithm for structural classification of XML data. *Machine Learning*, 62:137–170, 2006.

18 K. Zhang. Algorithms for the constrained editing distance between ordered labeled trees and related problems. *Pattern Recognition*, 1995.