Fast and Space Efficient String Kernels using Suffix Arrays

Choon Hui Teo CHOONHUI.TEO@RSISE.ANU.EDU.AU
S.V. N. Vishwanathan SVN.VISHWANATHAN@NICTA.COM.AU
Statistical Machine Learning, National ICT Australia, Locked Bag 8001, Canberra ACT 2601, Australia
Research School of Information Sciences & Engr., Australian National University, Canberra ACT 0200, Australia

Abstract

String kernels which compare the set of all common substrings between two given strings have recently been proposed by Vishwanathan & Smola (2004). Surprisingly, these kernels can be computed in linear time and linear space using annotated suffix trees. Even though, in theory, the suffix tree based algorithm requires $O(n)$ space for an n length string, in practice at least $40n$ bytes are required – $20n$ bytes for storing the suffix tree, and an additional $20n$ bytes for the annotation. This large memory requirement coupled with poor locality of memory access, inherent due to the use of suffix trees, means that the performance of the suffix tree based algorithm deteriorates on large strings. In this paper, we describe a new linear time yet space efficient and scalable algorithm for computing string kernels, based on suffix arrays. Our algorithm is a) faster and easier to implement, b) on the average requires only $19n$ bytes of storage, and c) exhibits strong locality of memory access. We show that our algorithm can be extended to perform linear time prediction on a test string, and present experiments to validate our claims.

1. Introduction

Many problems in machine learning require a data classification algorithm to work with discrete data. Common examples include biological sequence analysis and Natural Language Processing (NLP). In these applications data is given as a string (Durbin et al., 1998), an annotated sequence, or a combination of a string and its parse tree (Collins & Duffy, 2001).

In order to apply kernel methods, one defines a measure of similarity between discrete structures via a feature map $\phi : \mathcal{X} \to \mathcal{H}$. Here $\mathcal{X}$ is the set of discrete structures (*e.g.* the set of all documents), and $\mathcal{H}$ is a Reproducing Kernel Hilbert Space (RKHS). The kernel associated with $\mathcal{H}$ satisfies

$$k(x, x') = \langle \phi(x), \phi(x') \rangle_{\mathcal{H}}$$

for all $x, x' \in \mathcal{X}$. The success of a kernel method employing k depends both on the *faithful representation* of discrete data and an *efficient means of computing* k (Schölkopf & Smola, 2002).

Recent research has focussed on defining meaningful kernels on strings. Many ideas based on the use of substrings (Herbrich, 2002), gapped substrings (Lodhi et al., 2002), k-length substrings (Leslie et al., 2002a), and mismatch penalties (Leslie et al., 2002b) have been proposed. In the same vein, Vishwanathan & Smola (2004) proposed string kernels which use the set of all substrings of a string as their feature map. In a nutshell, each string is mapped to a feature vector which represents all its possible substrings (and their frequency of occurrence), and the kernel is defined as a dot product between these feature vectors. By using different weights for different substrings a family of kernels can be derived. Even though the number of common substrings could be quadratic in the size of the input strings, Vishwanathan & Smola show that these kernels can be computed in linear time by constructing and annotating a suffix tree.

A suffix tree is a data structure which compactly represents the set of all suffixes of a string. Although, theoretically, suffix trees can be constructed in linear time and require linear storage, the best known practical implementations consume around $20n$ bytes of storage for a n length string (Gusfield, 1997). The algorithm for computing string kernels due to Vishwanathan & Smola (2004) stores annotations which consume up to $20n$ bytes of additional storage. Suffix trees and their annotated counterparts suffer from poor locality of memory access, *i.e.,* navigating the tree requires memory access from different blocks of

Appearing in *Proceedings of the 23rd International Conference on Machine Learning*, Pittsburgh, PA, 2006. Copyright 2006 by the author(s)/owner(s).

secondary storage. If the entire tree does not fit into main memory caching schemes, frequently employed by operating systems to speed up secondary memory access, are rendered ineffective. Consequently, computing string kernels using suffix trees does not scale to large problems involving a few Megabytes (MB) of text.

In this paper, we address the above problem by designing a space efficient and scalable algorithm using suffix arrays. Suffix arrays are alternatives to suffix trees which were introduced independently by Manber & Myers (1993) and Gonnet et al. (1992). The suffix array of a n length string requires only $4n$ bytes of storage. Almost all search operations on a suffix tree can also be performed on a suffix array by incurring an additional $O(\log(n))$ additive or multiplicative penalty. Recently, Abouelhoda et al. (2004) showed that an Enhanced Suffix Array (ESA) – which essentially is a suffix array and a set of auxiliary tables – can be used to replace a suffix tree without incurring the log penalty. The ESA is space efficient, yet almost all operations on a suffix tree, including top-down and bottom-up traversal, can also be performed on the ESA. Furthermore, since the ESA is stored and accessed as a set of arrays (as opposed to a tree) it enjoys strong locality of memory access properties. We extend the ESA in order to compute string kernels in linear time.

After training a Support Vector Machine (SVM), prediction on a test string x involves computing $f(x) = \sum_i \alpha_i y_i k(x_i, x)$. Here, x_i are the so-called support vectors, y_i the corresponding labels, and α_i are the coefficients of expansion. Vishwanathan & Smola (2004) show that if x is of length m, then computing $f(x)$ requires $O(m)$ time (independent of the number of support vectors or their size). All the support vectors are concatenated and inserted into a suffix tree, and the leaves associated with a support vector x_i are weighed with the corresponding α_i. Computing $f(x)$ simply involves parsing the test string through this tree. We show that our algorithm can also be extended similarly to perform linear time prediction.

2. Kernel Definition

We begin by introducing some notation. Let $\mathcal{A}$ be a finite set of *characters* which we call the *alphabet*, *e.g.* $\mathcal{A} = \{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}$. Any $x \in \mathcal{A}^k$ for $k = 0, 1, 2 \ldots$ is called a *string*. $\mathcal{A}^*$ represents the set of all strings defined over the alphabet $\mathcal{A}$. The *sentinel* character $\$ \notin \mathcal{A}$ is lexicographically smaller than all the elements in $\mathcal{A}$. An *enhanced string* is a string with the sentinel character appended at the end.

We use $|x|$ to denote the length of a string x, while $u = x[i : j]$ with $1 \leq i \leq j \leq |x|$ denotes a substring of x between locations i and j (both inclusive). We also write $u \sqsubseteq x$ to denote that u is a substring of x. A substring of the form $x[1 : i]$ is called the i-th *prefix*, while a substring of the form $x[i : |x|]$ is called the i-th *suffix*. We use the shorthand $x[i]$ to denote the i-th suffix of x. These definitions can also be extended to enhanced strings.

Let $\mathrm{num}_y(x)$ denote the number of occurrences of y in x (that is the number of times y occurs as a substring of x). The family of kernels we will be studying are defined by (Vishwanathan & Smola, 2004):

$$k(x, x') := \sum_{s \sqsubseteq x, s' \sqsubseteq x'} w_s \delta_{s,s'} = \sum_{s \in \mathcal{A}^*} w_s \, \mathrm{num}_s(x) \, \mathrm{num}_s(x'). \quad (1)$$

That is, we count the number of occurrences of every substring s in both x and x' and weigh it by w_s, where the latter may be a weight chosen *a priori* or after seeing data. Furthermore, kernels on trees can also be computed by first reducing them to strings and then computing kernels on the string representation. For details we refer the reader to Vishwanathan & Smola (2004).

3. String Kernels with Suffix Trees

In this section, we briefly describe how string kernels (1) can be computed efficiently using suffix trees. The aim of our discussion here is to point out the major steps involved in kernel computation. In the next section we will show how these operations can be efficiently performed on a suffix array. Before describing the algorithm we need to introduce some notation, and explain a few concepts.

A suffix tree is a compacted trie that stores all suffixes of a given text string (Gusfield, 1997). Well known linear time algorithms for constructing suffix trees exist (see Gusfield, 1997, and references therein).

Given a string x, we construct the suffix tree $S(x)$ of the enhanced string $x\$$. The presence of the sentinel character \$ ensures that every suffix of $x\$$ is unique. $\mathrm{nodes}(S(x))$ denotes the set of all nodes, while $\mathrm{root}(S(x))$ denotes the root of $S(x)$ (see figure 1). Each edge of $S(x)$ is associated with a substring of $x\$$. If w denotes the path from the root to a node we label the node as $\overline{w}$. For a node $\overline{w}$, $T_{\overline{w}}$ denotes the subtree rooted at that node, $\mathrm{lvs}(\overline{w})$ denotes the number of leaves of $T_{\overline{w}}$, and $\mathrm{parent}(\overline{w})$ denotes its parent node. Let a be a character and $\overline{aw}$ be a node in $S(x)$. An *auxiliary* unlabeled edge $\overline{aw} \to \overline{w}$ is called a suffix link (see figure 1). Every internal node in $S(x)$ has a suffix link (Giegerich & Kurtz, 1997, Proposi-

tion 2.9). In fact, many suffix tree building algorithms routinely construct the suffix links as an intermediate step towards constructing the suffix tree. We denote by words($S(x)$) the set of all non-empty strings w such that $\overline{wu} \in$ nodes($S(x)$) for some (possibly empty) string u. This means that words($S(x)$) is the set of all possible substrings of $x\$$ (Giegerich & Kurtz, 1997). For every $t \in$ words($S(x)$) we define ceil(t) as the node

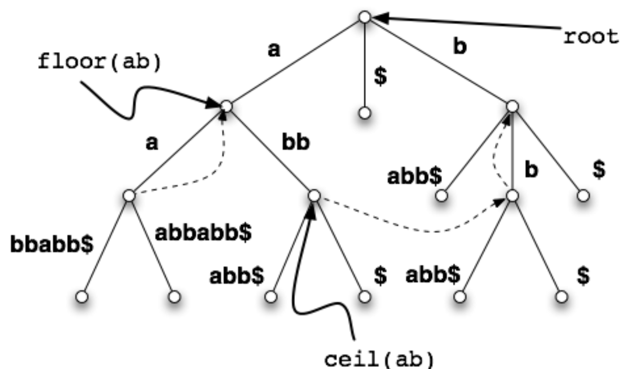

Figure 1. Figure denotes the suffix tree of the string $aaabbabb\$$. The dotted lines (*e.g.* from node $\overline{abb}$ to $\overline{bb}$) represent suffix links. The floor and ceil nodes corresponding to the string ab are also depicted. Observe that bb occurs twice in the string $aaabbabb\$$, and the subtree $T_{\overline{bb}}$ has two leaves.

$\overline{w}$ such that $w = tu$ and u is the shortest (possibly empty) substring such that $\overline{w} \in$ nodes($S(x)$). That is, it is the immediate next node on the path leading up to t in $S(x)$ (see figure 1). Finally, for every $t \in$ words($S(x)$) we define floor(t) as the node $\overline{w}$ such that $t = wu$ and u is the shortest non-empty substring such that $\overline{w} \in$ nodes($S(x)$). That is, it is the last node encountered on the path leading up to t in $S(x)$ (see figure 1). For an internal node $\overline{w}$ it is clear that ceil(w) $= \overline{w}$, and floor(w) is the parent of $\overline{w}$. For any $t \in$ words($S(x)$), lvs(ceil(t)) gives us the number of occurrence of t in $x\$$ (Giegerich & Kurtz, 1997).

Given strings x, y the matching statistics of y with respect to x are given by $v \in \mathbb{N}^{|y|}$ and $c, c' \in \text{nodes}(S(x))^{|y|}$. If we use v_i, c_i, c'_i to denote the i-th component of v, c and c' respectively and define $\widehat{v_i} := i + v_i - 1$, then v_i, is the length of the longest substring of x matching a prefix of $y[i]$ while $c_i = \text{ceil}(y[i : \widehat{v_i}])$, and $c'_i = \text{floor}(y[i : \widehat{v_i}])$ (see table 1). Every substring which occurs in both x and y must be a prefix of $y[i : \widehat{v_i}]$ for some i. To see this consider an arbitrary substring z which occurs in both x and y. This implies that $z = y[i : j]$ for some i and j. But the longest prefix of $y[i]$ which matches a substring in x is $y[i : \widehat{v_i}]$. Therefore, $j \leq \widehat{v_i}$ and z is a prefix of $y[i : \widehat{v_i}]$.

Table 1. The matching statistics of string $aabbb$ with respect to $aaabbabb\$$ is depicted here. The floor and the ceiling nodes are with respect to the suffix tree depicted in figure 1. Observe, for instance, that for $i = 2$, the string abb is the largest prefix of $abbb$ which occurs as a substring of $aaabbabb\$$.

i	1	2	3	4	5
$y[i : \widehat{v_i}]$	$aabb$	abb	bb	bb	b
v_i	4	3	2	2	1
c_i	$\overline{aabbabb\$}$	$\overline{abb}$	$\overline{bb}$	$\overline{bb}$	$\overline{b}$
c'_i	$\overline{aa}$	$\overline{a}$	$\overline{b}$	$\overline{b}$	root

Furthermore, by reading off lvs(ceil(z)) in $S(x)$ we can determine the number of occurrences of the substring z in x. The following theorem exploits both these observations in order to compute string kernels efficiently.

Theorem 1 (Vishwanathan & Smola (2004)) *Let x and y be strings, and v, c, and c' be the matching statistics of y with respect to x. For any $t = uv$ with $\overline{u} = \text{floor}(t)$ in $S(x)$ assume that*

$$W(x, t) = \sum_{z \in \text{prefix}(v)} w_{uz} - w_u \quad (2)$$

can be computed in constant time. Then the kernel (1) *can be computed in $O(|x| + |y|)$ time as*

$$k(x, y) = \sum_{i=1}^{|y|} [\text{val}(c'_i) + \text{lvs}(c_i) \cdot W(x, y[i : \widehat{v_i}])] \quad (3)$$

where

$$\text{val}(\overline{t}) := \text{val}(\text{parent}(t)) + \text{lvs}(\overline{t}) \cdot W(x, t), \quad (4)$$

and val(root) $:= 0$.

A large number of weight functions satisfy the requirement that (2) can be computed in constant time. See section 5.4 of Vishwanathan & Smola (2004) for more details. For our purposes, we simply assume that given any substring of x the function $W(x, \cdot)$ returns the associated weight in constant time.

Efficient computation of string kernels requires us to pre-compute and store val(c'_i) for each internal node of the suffix tree. Note that the val of a node is simply the val of its parent plus the contributions due to all strings which end on the edge connecting the node to its parent. This can be computed in constant time by virtue of Equations (2) and (4). In order to annotate each internal node of $S(x)$ with its val we simply perform a top down traversal of the tree and propagate val from parent to child nodes.

The final task that remains is to compute matching statistics of y with respect to x. Surprisingly, this can be done in linear time by using a suffix tree (Chang & Lawler, 1994). The key observation is that if $y[i : \widehat{v_i}]$ is a substring of x then $y[i+1 : \widehat{v_i}]$ is also a substring of x, and therefore $v_{i+1} \geq v_i - 1$. Besides this, the largest prefix of $y[i+1]$ which matches a substring of x *must* have $y[i+1 : \widehat{v_i}]$ as a prefix. In other words, once we have parsed the string y up to a certain position we never need to revisit the parsed positions again.

Briefly, the matching statistics algorithm of Chang & Lawler (1994) works as follows: Given c'_i the algorithm first finds the intermediate node $p'_{i+1} := \text{floor}(y[i+1 : \widehat{v_i}])$. This is done by walking down the suffix link of c'_i and then walking down the edges corresponding to the remaining portion of $y[i+1 : \widehat{v_i}]$. Now, to find c'_{i+1} the algorithm searches for the longest matching prefix of $y[\widehat{v_i} + 1]$ in $S(x)$, starting from node p'_{i+1}. v_1, c_1, and c'_1 are found by simply walking down $S(x)$ to find the longest prefix of y which matches a substring of x. Given the suffix tree $S(x)$, the total time complexity of the algorithm is $O(|y|)$.

In summary, two main suffix tree operations are required to compute string kernels, a top down traversal for annotation and a suffix link traversal for computing matching statistics. We will show in the next section that both these operations can be performed efficiently on a suffix array.

4. String Kernels with Suffix Arrays

In this section we describe suffix arrays and show how string kernels can be computed efficiently using suffix arrays. Our notation closely follows Abouelhoda et al. (2004).

The suffix array (Manber & Myers, 1993; Gonnet et al., 1992) of a string x of length $n = |x|$, denoted by `suftab`, is an array of integers corresponding to the lexicographically sorted suffixes of x (see table 2). If $n < 2^{32}$ then each entry of `suftab` is an integer between 0 and $2^{32} - 1$, and $4n$ bytes of storage suffice. In the sequel we always assume that $n < 2^{32}$.

In a suffix tree, the leaves which represent suffixes of the string, are lexicographically sorted. Therefore, there is a one-to-one relationship between the leaves of the suffix tree and the suffix array. In fact, until recently, linear time algorithms for suffix array construction constructed a suffix tree, and read off the sorted suffixes from the leaves (Manber & Myers, 1993). Recently, several linear time *direct* algorithms have been proposed, however, in practice, super-linear algorithms generally tend to be faster and more memory efficient (Maniscalco & Puglisi, 2005). *MSufSort*[1] due to Maniscalco & Puglisi (2005), which we use in our experiments, is one of the fastest known practical implementations.

[1] `http://www.michael-maniscalco.com/msufsort.htm`

A suffix array has less structure than a suffix tree, yet, almost all operations on a suffix tree can also be performed on a suffix array by incurring an additional additive or multiplicative $O(\log(n))$ penalty (Manber & Myers, 1993). Recently, Abouelhoda et al. (2004) proposed a data structure which they call the Enhanced Suffix Array (ESA). By storing extra information along with the suffix array almost all suffix tree algorithms can be implemented on the ESA without incurring any additional penalty. In particular, the ESA stores two extra tables, the `lcptab` and `childtab`. Furthermore, we use a `valtab` which stores the annotations required for string kernel computation. All the three tables are explained below.

Table 2. Suffix array and lcp-table of the string *aaabbabb*$

	`suftab`[i]	suffix	`lcptab`[i]
1	9	$	0
2	1	*aaabbabb*$	0
3	2	*aabbabb*$	2
4	6	*abb*$	1
5	3	*abbabb*$	3
6	8	*b*$	0
7	5	*babb*$	1
8	7	*bb*$	1
9	4	*bbabb*$	2

4.1. Longest Common Prefix

The Longest Common Prefix (LCP) of two strings is the longest prefix which occurs in both the strings. For example, given strings *aabccca* and *aabaccb* their LCP is *aab*. The LCP table `lcptab` is an auxiliary table which can be computed in linear time (Manzinni, 2004). It stores the length of the LCP between adjacent suffixes in `suftab` (see table 2). The `lcptab` plays a crucial role in our algorithm.

Since `lcptab` stores integers between 0 and n it can be stored in $4n$ bytes. But, in practice, most of the entries of `lcptab` are less than 255. We exploit this observation to store the LCP table in a two stage data structure. All values less than 255 are stored in a n byte array, and all values greater than or equal to 255 are looked up from a secondary array. This approach consumes only slightly more than n bytes on the average (Abouelhoda et al., 2004).

The LCP of a set of strings is defined as the longest prefix which occurs in every string of the set. An interval in a suffix tree is defined as a tuple (i, j) with $1 \leq i < j \leq n$. The LCP of an interval (i, j), denoted

as LCP(i, j), is defined as the LCP of the set of strings $x[\texttt{suftab}[i]], \ldots x[\texttt{suftab}[j]]$. It is easy to see that the length of LCP(i, j) is given by $\min_{i<k\leq j} \texttt{lcptab}[k]$.

There is a close relation between the internal nodes in a suffix tree, and intervals in a suffix array. To see this, consider an internal node $\overline{w}$ in the suffix tree $S(x)$. Every suffix of x which has w as its prefix is a leaf of the subtree $T_{\overline{w}}$. Furthermore, the LCP of this set of suffixes is exactly given by w. Therefore, the node $\overline{w}$ is uniquely characterized by this set. Since the suffix array is sorted lexicographically, the suffixes in this set occur contiguously in `suftab`. Let (i, j) be the largest interval such that LCP$(i, j) = w$ then, we call (i, j) the w-lcp interval and denote it by w-(i, j) . Since, LCP$(i, j) = |w|$ we have that $\texttt{lcptab}(k) \geq |w|$ for all $i < k \leq j$.

4.2. Child Table

Since each internal node of the suffix tree corresponds to an interval of the suffix array, the parent-child relationship between nodes in the suffix tree is now represented as a parent-child relationship between intervals of the suffix array. The child table is an auxiliary array which compactly represents these relationships (Abouelhoda et al., 2004).

Consider an internal node $\overline{w}$ of the suffix tree $S(x)$. To identify the children of $\overline{w}$ it is sufficient to identify the subtrees hanging off $\overline{w}$. To identify the subtrees we need to identify the left-most leaf, right-most leaf, and all those leaves in $T_{\overline{w}}$ whose LCP with their immediate predecessor is exactly equal to w (see figure 2). These leaves correspond to indices in the interval w-(i, j) whose `lcptab` entry is exactly equal to $|w|$. If $\overline{w}$ has $k + 1$ children, then there are exactly k such indices denoted by $i_1, \ldots, i_k$. The child table conceptually is an array of pointers[2]. The pointer at i points to i_1, while the pointer at i_l points to i_{l+1} for $1 \leq l < k$, and the last pointer i_k is set to NULL. In other words, k non-NULL pointers are sufficient to encode the k children of $\overline{w}$. Since there are at most n internal nodes in $S(x)$ at most $4n$ bytes of storage are required. By storing relative indices, and using a two stage storage scheme analogous to `lcptab` we reduce the average storage requirements to n bytes (Abouelhoda et al., 2004). Given `childtab` enumerating the child intervals can be done via a function *getchildren* which requires at most $O(|\mathcal{A}|)$ time (*i.e.,* constant time, since $|\mathcal{A}|$ is assumed constant). Furthermore, one can also compute the label on the edge connecting $\overline{w}$ to its parent via a function *getedge*. Loosely speaking, this function computes the difference in the LCP values of $\overline{w}$ and its parent (Abouelhoda et al., 2004).

[2]The child table is actually an array of indices. For clarity of exposition, and with some abuse of terminology, we describe the entries as pointers.

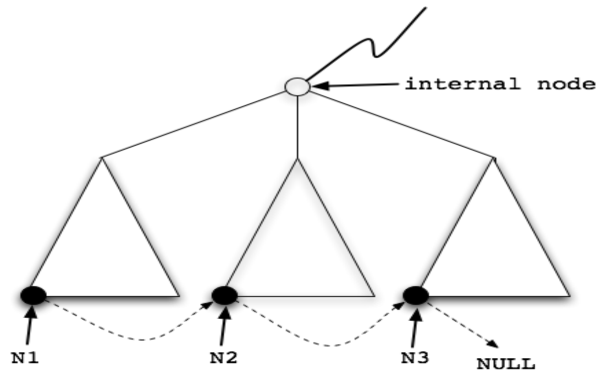

Figure 2. The node depicted as an unfilled circle is an internal node $\overline{w}$ of the suffix tree $S(x)$. The leftmost child of $T_{\overline{w}}$ is labeled $N1$. The child table entry for this node points to the leftmost leaf of next subtree (node $N2$) and so on. The leftmost leaf of the last subtree (node $N3$) points to NULL. Note that the nodes $N2$ and $N3$ are the only leaf nodes in $T_{\overline{w}}$ whose LCP with their immediate predecessor is exactly equal to w.

4.3. Storing Values in Internal Nodes

In order to compute the string kernel we need to compute and store val for each internal node of the tree (see (4)). Since the suffix array does not store the internal nodes of the suffix tree we need to devise a new scheme to compute and store these annotations.

`valtab` is an array of n floats which stores these annotations succinctly. Since a float occupies 4 bytes, `valtab` requires $4n$ bytes of storage. First, we need to associate each internal node of the suffix tree (or the corresponding interval (i, j) of the suffix array) to an unique index. Note that we cannot use either i or j because more than one intervals might have i as their first index or j as their last index. Instead, we associate each internal node with the leftmost leaf of its second child. Such a node always exists and is unique. For example, the internal node in figure 2 is associated with the leaf node $N2$.

In terms of the suffix array, let $\overline{w}$ be the internal node and w-(i, j) be the corresponding w interval. Then, $\overline{w}$ is associated with the first index in w-(i, j) whose `lcptab` entry is $|w|$. Since the child table already stores a pointer to this location, one can write a function *getpos* which takes an interval as input and returns this location in constant time.

The algorithm for computing `valtab` is outlined in Algorithm 1. Essentially, we perform a breadth first search and propagate the val from a parent node to each one of its children. Recall that $W(x, \cdot)$ is a function which given a substring of x returns the associated

weight in constant time.

Algorithm 1 Algorithm for populating `valtab`

```
Input: x, suftab, childtab
Initialize: valtab[i] ← 0 for all i
Initialize: q.push([1, n])
  while q not empty do
    next ← q.head() and q.pop()
    edge ← getedge(next)
    pos ← getpos(next)
    valtab[pos] ← valtab[pos] + W(x, edge)
    childlist ← getchildren(next)
    for child ∈ childlist do
      q.push(child)
      childpos ← getpos(child)
      valtab[childpos] ← valtab[pos]
    end for
  end while
```

4.4. Suffix Link Table

Implementing the matching statistics algorithm of Chang & Lawler (1994) requires suffix links. Recall that suffix links are axillary edges which point from a node labeled $\overline{aw}$ to the node labeled $\overline{w}$. In a suffix array the suffix link of the lcp interval aw-(i, j) is the lcp interval w-(i', j'). A suffix link table `slinktab` is an auxiliary data structure which stores the indices i' and j' for each interval (i, j) corresponding to an internal node of the suffix tree. Since there are at most n internal nodes we need to store at most $2n$ indices which in turn requires $8n$ bytes of storage. By using a variant of the Chang & Lawler (1994) algorithm, and performing a breadth first search one can compute the `slinktab` in $O(n)$ time and space. Looking up indices in this table requires constant time (Abouelhoda et al., 2004).

An alternative method is to do a simple binary search on the suffix array. To compute the suffix link of a node $\overline{aw}$, we begin with the interval $[1, n]$ and successively search for the largest interval whose LCP is exactly equal to w. This can be done by a binary search and requires $O(|w| \log(n))$ time. For large texts we use a bucket table `buctab` to speed up computations. Given a number d, we compute a hash function which lexicographically sorts all the strings of length d or less and associates them to their position in the sorted order. For each substring z of x such that $|z| \leq d$, we compute its hash value, the first index i such that z occurs as a prefix of $x[\texttt{suftab}[i]]$, and the last index j such that z occurs as a prefix of $x[\texttt{suftab}[j]]$ [3] and store this information in `buctab`. d is chosen such that the

Table 3. Average storage requirements of `suftab` and auxiliary arrays.

x	n
`suftab`	$4n$
`lcptab`	n
`childtab`	$4n$
`valtab`	$4n$
`buctab`	n
`lvstab`	$4n$

storage requirements of `buctab` is less than n bytes. If $|w| \leq d$ then we can directly look up the suffix link interval from the bucket table. For $|w| > d$ let u denote the d length prefix of w. We use `buctab` to look up the largest interval (i, j) such that $\mathrm{LCP}(i, j) = u$. Now it is enough to run the binary search on the interval (i, j). Typically, $|w|$ is small and $j - i \ll n$. Therefore, this hybrid scheme works well in practice (Abouelhoda et al., 2004).

4.5. Our Algorithm

Now we are in a position to describe our algorithm for computing string kernels using suffix arrays. Given a pair of strings x and y we first construct its suffix array `suftab` and the auxiliary tables `lcptab`, `childtab`, `valtab`, and `buctab`. The average storage requirements of each of these tables is summarized in table 3. We then run a variant of the (Chang & Lawler, 1994) algorithm on the suffix array (see section 4.7.2 Abouelhoda et al., 2004), and use (3) to compute the kernel.

5. Linear Time Prediction

Let $\widehat{\mathrm{X}} = \{x_1, x_2, \ldots, x_m\}$ denote strings in the training set, X be the *master string* obtained by concatenating the strings in $\widehat{\mathrm{X}}$, and K denote the kernel matrix, *i.e.*, $K_{ij} := k(x_i, x_j)$. Given a coefficient vector $\alpha \in \mathbb{R}^m$, computing the j-th entry of $K\alpha$ entails computing

$$[K\alpha]_j = \sum_{i=1}^{m} \alpha_i k(x_i, x_j). \tag{5}$$

Using (1) we can rewrite $[K\alpha]_j$ as

$$\begin{aligned} [K\alpha]_j &= \sum_{i=1}^{m} \sum_{s \sqsubseteq x_i} \sum_{s' \sqsubseteq x_j} \alpha_i w_s \delta_{s,s'} \\ &= \sum_{s \sqsubseteq x_i} \sum_{s' \sqsubseteq x_j} \left(\sum_{i=1}^{m} \alpha_i w_s \right) \delta_{s,s'}. \end{aligned} \tag{6}$$

In (1) a substring s is assigned a weight w_s whereas in (6) the same substring is assigned a weight $\sum_{i=1}^{m} \alpha_i w_s$.

[3] Strictly speaking, j need not be stored. It can be computed by using the lexicographical neighbor of z which occurs in x.

This suggests the following simple strategy: Every leaf of $S(X)$, which corresponds to a the suffix of x_i is assigned a weight α_i. All we need to do now is define $\text{lvs}(\overline{w})$ of a node $\overline{w}$ as the sum over the weights of the leaves in the subtree $T_{\overline{w}}$. The string kernel algorithm described in Section 3 can be run unchanged to compute $K\alpha$ in linear time.

To implement this algorithm on a suffix array, we need to make two changes: construct the enhanced suffix array of X, and use an auxiliary array `lvstab` for storing the weights on the leaves. This array occupies $4n$ bytes of storage.

6. Experiments

To test the efficacy of our algorithm we perform experiments[4] using data from project Gutenberg (`http://www.gutenberg.org/`). We randomly sampled 40 books, the first 20 are labeled as corpus $\widehat{\mathrm{X}}$ and the next 20 are labeled corpus $\widehat{\mathrm{Y}}$. All experiments were run on a Linux machine with $3.6GHz$ processor and $1GB$ RAM. For all our kernel computations we use a constant weight function (see Eq. (5.9) Vishwanathan & Smola, 2004). In other words, a matching substring of length l is given weight l.

In our first experiment, we study the effect of increasing pattern sizes on kernel evaluation time. To produce our fixed length text we concatenate the strings in $\widehat{\mathrm{X}}$ and use the first 8MB. Similarly, we concatenate strings in $\widehat{\mathrm{Y}}$ and use varying length prefixes as our pattern. Using two variants of our algorithm we compute the kernel between the text and the pattern. The first variant (SA-SL) uses a `slinktab` and hence consumes $8n$ bytes of extra storage, while the second (SA-NSL) uses the hybrid scheme described in Section 4.4. The kernel evaluation time, averaged over 10 runs, is plotted in Figure 3(a), and contrasted with the time required by the suffix tree based algorithm (ST). For small patterns (size $< 10KB$) the suffix array based methods do not exhibit a significant advantage, but as the size of the pattern increases they tend to exhibit better scaling behavior. For a pattern of size $4MB$ they are almost three orders of magnitude faster, and for a $8MB$ pattern the suffix tree based method did not even finish in reasonable time.

In our second experiment we study the effect of increasing text sizes on kernel evaluation time. To produce our fixed length pattern we again concatenate strings in $\widehat{\mathrm{Y}}$ and use the first $128KB$ [5]. Increasing text lengths are produced by concatenating strings in $\widehat{\mathrm{X}}$ and using varying length prefixes. As before, we report the kernel evaluation time averaged over 10 runs for the three algorithms in Figure 3(b). The suffix tree based algorithm suffers from a high overhead and is around 70 times slower than our suffix array based method. While both algorithms seemingly exhibit similar scaling behavior, the constants involved are very different.

Our third experiment studies the effect of the number of support vectors on the prediction algorithm described in Section 5. We use the first 10K verses from the Bible[6](average 145 characters per verse) as our support vectors, and the book *Peter Pan* ($274KB$), from project Gutenberg, as our pattern. Each support vector is uniformly given a weight of $\alpha_i = 1$, and we compute the value of $K\alpha$. Besides the suffix tree and suffix array based methods we also used a direct method. This method explicitly computes $K\alpha$ by first computing the elements of K and then multiplying them with the vector α. The run-times of all the above methods are plotted in Figure 3(c). As expected, the direct method exhibits poor scaling behavior as compared to the suffix tree or suffix array based methods. The suffix array based methods are two to three orders of magnitude faster than the suffix tree based methods. As the number of support vectors increases the time required to populate `lvstab` increases. For each suffix in `suftab` we need to identify the corresponding support vector. The slightly worse scaling behavior, as the number of support vectors increases, can be attributed to this look-up.

7. Outlook and Discussion

We presented a suffix array based algorithm for computing the string kernels. Our algorithm is memory efficient and scalable as demonstrated empirically. In particular, on large strings our algorithm is up to three orders of magnitude faster that the suffix tree based method. We also showed that our algorithm can be used to compute the product of the kernel matrix with an arbitrary coefficient vector in linear time. SimpleSVM (Vishwanathan et al., 2003) is a Quadratic Programming solver which can exploit this property to train a SVM on a large text corpus. Future work will concentrate on studying this link further.

Acknowledgments

National ICT Australia is funded by the Australian Government's Department of Communications, Information Technology and the Arts and the Australian Research Council through Backing Australia's Ability and the ICT Center of Excellence program. This work is supported partly by the IST Program of the

[4] `http://sml.nicta.com.au/code/sask`

[5] The pattern size was chosen to ensure that the suffix tree based algorithm finished in reasonable time.

[6] `http://corpus.canterbury.ac.nz/descriptions/`

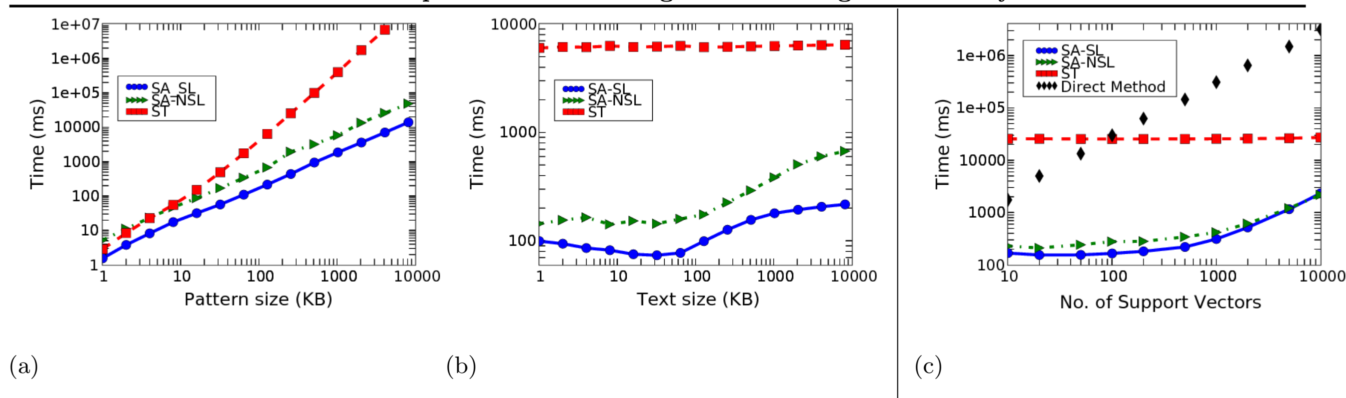

Figure 3. Left/Center: Average kernel evaluation times for a fixed size text and fixed size pattern plotted on a log-log scale. Right: Time required for the linear time prediction algorithm (see Section 5).

European Community, under the Pascal Network of Excellence, IST-2002-506778.

References

Abouelhoda, M. I., Kurtz, S., & Ohlebusch, E. (2004). Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, *2*, 53–86.

Chang, W. I., & Lawler, E. L. (1994). Sublinear approximate sting matching and biological applications. *Algorithmica*, *12*(4/5), 327–344.

Collins, M., & Duffy, N. (2001). Convolution kernels for natural language. In T. G. Dietterich, S. Becker, & Z. Ghahramani, eds., *Advances in Neural Information Processing Systems 14*. Cambridge, MA: MIT Press.

Durbin, R., Eddy, S., Krogh, A., & Mitchison, G. (1998). *Biological Sequence Analysis: Probabilistic models of proteins and nucleic acids*. Cambridge University Press.

Giegerich, R., & Kurtz, S. (1997). From Ukkonen to McCreight and Weiner: A unifying view of linear-time suffix tree construction. *Algorithmica*, *19*(3), 331–353.

Gonnet, G. H., Baeza-Yates, R. A., & Snider, T. (1992). *Information Retrieval: Data Structures and Algorithms*, chap. 5, New indices for text: PAT Trees and PAT arrays, 66–82. Upper Saddle River, NJ, USA: Prentice-Hall, Inc. ISBN 0-13-463837-9.

Gusfield, D. (1997). *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press. ISBN 0 - 521 - 58519 - 8.

Herbrich, R. (2002). *Learning Kernel Classifiers: Theory and Algorithms*. MIT Press.

Leslie, C., Eskin, E., & Noble, W. S. (2002a). The spectrum kernel: A string kernel for SVM protein classification. In *Proceedings of the Pacific Symposium on Biocomputing*, 564–575.

Leslie, C., Eskin, E., Weston, J., & Noble, W. S. (2002b). Mismatch string kernels for SVM protein classification. In *Advances in Neural Information Processing Systems 15*, vol. 15. Cambridge, MA: MIT Press.

Lodhi, H., Saunders, C., Shawe-Taylor, J., Cristianini, N., & Watkins, C. (2002). Text classification using string kernels. *Journal of Machine Learning Research*, *2*, 419–444.

Manber, U., & Myers, G. (1993). Suffix arrays: A new method for on-line string searches. *SIAM journal on computing*, *22*(5), 935–948.

Maniscalco, M. A., & Puglisi, S. J. (2005). An efficient, versatile approach to suffix sorting. *ACM Journal of Experimental Algorithms*. Submitted.

Manzinni, G. (2004). Two space saving tricks for linear time lcp array computation. In *Proc. 9th Scandinavian Workshop on Algorithm Theory (SWAT'04)*, vol. 3111 of *LCNS*, 372–383. Humlebaek, Denmark: Springer-Verlag.

Schölkopf, B., & Smola, A. (2002). *Learning with Kernels*. Cambridge, MA: MIT Press.

Vishwanathan, S. V. N., & Smola, A. J. (2004). Fast kernels for string and tree matching. In K. Tsuda, B. Schölkopf, & J. Vert, eds., *Kernels and Bioinformatics*. Cambridge, MA: MIT Press.

Vishwanathan, S. V. N., Smola, A. J., & Murty, M. (2003). SimpleSVM. In T. Fawcett, & N. Mishra, eds., *Proceedings of the 20th International Conference on Machine Learning (ICML)*. Washington DC: AAAI press.