

# Efficient Geometric-based Computation of the String Subsequence Kernel\*

Slimane Bellaouar<sup>1</sup>, Hadda Cherroun<sup>1</sup>, and Djelloul Ziadi<sup>2</sup>

<sup>1</sup> Laboratoire LIM, Université Amar Telidji, Laghouat, Algérie  
`{s.bellaouar,hadda.cherroun}@mail.lagh-univ.dz`

<sup>2</sup> Laboratoire LITIS - EA 4108, Université de Rouen, Rouen, France  
`djelloul.ziadi@univ-rouen.fr`

**Abstract.** Kernel methods are powerful tools in machine learning. They have to be computationally efficient. In this paper, we present a novel Geometric-based approach to compute efficiently the string subsequence kernel (SSK). Our main idea is that the SSK computation reduces to range query problem. We started by the construction of a *match list*  $L(s, t) = \{(i, j) : s_i = t_j\}$  where  $s$  and  $t$  are the strings to be compared; such match list contains only the required data that contribute to the result. To compute efficiently the SSK, we extended the *layered range tree* data structure to a *layered range sum tree*, a range-aggregation data structure. The whole process takes  $O(p|L|\log|L|)$  time and  $O(|L|\log|L|)$  space, where  $|L|$  is the size of the match list and  $p$  is the length of the SSK. We present empiric evaluations of our approach against the dynamic and the sparse programming approaches both on synthetically generated data and on newswire article data. Such experiments show the efficiency of our approach for large alphabet size except for very short strings. Moreover, compared to the sparse dynamic approach, the proposed approach outperforms absolutely for long strings.

**Keywords:** string subsequence kernel, computational geometry, layered range tree, range query, range sum

## 1 Introduction

Kernel methods [4] offer an alternative solution to the limitation of traditional machine learning algorithms, applied solely on linear separable problems. They map data into a high dimensional feature space where we can apply linear learning machines based on algebra, geometry and statistics. Hence, we may discover non-linear relations. Moreover, kernel methods enable other data type processings (biosequences, images, graphs, ...).

Strings are among the important data types. Therefore, machine learning community devotes a great effort of research to string kernels, which are widely used in the fields of bioinformatics and natural language processing. The philosophy of all string kernels can be reduced to different ways to count common

---

\* This work is supported by the MESRS - Algeria under Project 8/U03/7015.

substrings or subsequences that occur in both strings to be compared, say  $s$  and  $t$ .

In the literature, there are two main approaches to improve the computation of the SSK. The first one is based on dynamic programming; Lodhi et al. [6] apply dynamic programming paradigm to the suffix version of the SSK. They achieve a complexity of  $O(p|s||t|)$ , where  $p$  is the length of the SSK. Later, Rousu and Shawe-Taylor [7] propose an improvement to the dynamic programming approach based on the observation that most entries of the dynamic programming matrix (DP) do not really contribute to the result. They use a set of match lists combined with a sum range tree. They achieve a complexity of  $O(p|L| \log \min(|s|, |t|))$ , where  $L$  is the set of matches of characters in the two strings. Beyond the dynamic programming paradigm, the trie-based approach [5,7,8] is based on depth first traversal on an implicit trie data structure. The idea is that each node in the trie corresponds to a co-occurrence between strings. But the number of gaps is restricted, so the computation is approximate.

Motivated by the efficiency of the computation, a key property of kernel methods, in this paper we focus on improving the SSK computation. Our main idea consists to map a machine learning problem on a computational geometry one. Precisely, our geometric-based SSK computation reduces to 2-dimensional range queries on a layered range sum tree (a layered range tree that we have extended to a range-aggregate data structure). We started by the construction of a *match list*  $L(s, t) = \{(i, j) : s_i = t_j\}$  where  $s$  and  $t$  are the strings to be compared; such match list contains only the required data that contribute to the result. To compute efficiently the SSK, we constructed a *layered range sum tree* and applied the corresponding computational geometry algorithms. The overall time complexity is  $O(p|L| \log |L|)$ , where  $|L|$  is the size of the match list.

The rest of this paper is organized as follows. Section 2 deals with some concept definitions and introduces the layered range tree data structure. In section 3, we recall formally the SSK computation. We also review three efficient computations of the SSK, namely, dynamic programming, trie-based and sparse dynamic programming approaches. Our contribution is addressed in Section 4. Section 5 presents the conducted experiments and discusses the associated results, demonstrating the practicality of our approach for large alphabet sizes. Section 6 presents conclusions and further work.

## 2 Preliminaries

We first deal with concepts of string, substring, subsequence and kernel. We then present the layered range tree data structure.

### 2.1 String

Let  $\Sigma$  be an alphabet of a finite set of symbols. We denote the number of symbols in  $\Sigma$  by  $|\Sigma|$ . A string  $s = s_1 \dots s_{|s|}$  is a finite sequence of symbols of length  $|s|$  where  $s_i$  marks the  $i^{th}$  element of  $s$ . The symbol  $\epsilon$  denotes the empty string. We

use  $\Sigma^n$  to denote the set of all finite strings of length  $n$  and  $\Sigma^*$  the set of all strings. The notation  $[s = t]$  is a boolean function that returns

$$\begin{cases} 1 & \text{if } s \text{ and } t \text{ are identical;} \\ 0 & \text{otherwise.} \end{cases}$$

## 2.2 Substring

For  $1 \leq i \leq j \leq |s|$ , the string  $s(i : j)$  denotes the substring  $s_i s_{i+1} \dots s_j$  of  $s$ . Accordingly, a string  $t$  is a substring of a string  $s$  if there are strings  $u$  and  $v$  such that  $s = utv$  ( $u$  and  $v$  can be empty). The substrings of length  $n$  are referred to as  $n$ -grams (or  $n$ -mers).

## 2.3 Subsequence

The string  $t$  is a subsequence of  $s$  if there exists an increasing sequence of indices  $I = (i_1, \dots, i_{|t|})$  in  $s$ , ( $1 \leq i_1 < \dots < i_{|t|} \leq |s|$ ) such that  $t_j = s_{i_j}$ , for  $j = 1, \dots, |t|$ . In the literature, we use  $t = s(I)$  if  $t$  is a subsequence of  $s$  in the positions given by  $I$ . The empty string  $\epsilon$  is indexed by the empty tuple. The absolute value  $|t|$  denotes the length of the subsequence  $t$  which is the number of indices  $|I|$ , while  $l(I) = i_{|t|} - i_1 + 1$  refers to the number of characters of  $s$  covered by the subsequence  $t$ .

## 2.4 Kernel methods

Traditional machine learning and statistic algorithms have been focused on linearly separable problems (i.e. detecting linear relations between data). It is the case where data can be represented by a single row of table. However, real world data analysis requires non linear methods. In this case, the target concept cannot be expressed as simple linear combinations of the given attributes [4]. This was highlighted in 1960 by Minsky and Papert.

Kernel methods were proposed as a solution by embedding the data in a high dimensional feature space where linear learning machines based on algebra, geometry and statistics can be applied. This embedding is called kernel. It arises as a similarity measure (inner product) in a high dimension space so-called feature description.

The trick is to be able to compute this inner product directly from the original data space using the kernel function. This can be formally clarified as follows: the kernel function  $K$  corresponds to the inner product in a feature space  $F$  via a map  $\phi$ .

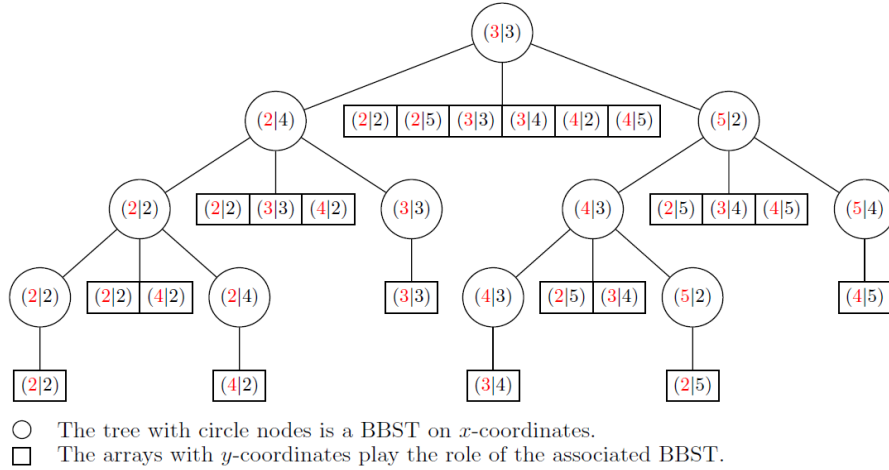
$$\begin{aligned} \phi : X &\rightarrow F \\ x &\mapsto \phi(x) \\ K(x, x') &= \langle \phi(x), \phi(x') \rangle. \end{aligned}$$

## 2.5 Layered Range Tree

A Layered Range Tree (LRT) is a spatial data structure that supports orthogonal range queries. It is judicious to describe a 2-dimensional range tree in order to understand LRT. Consider a set  $S$  of points in  $\mathcal{R}^2$ . A range tree is primarily a balanced binary search tree (BBST) built on the  $x$ -coordinate of the points of  $S$ . Data are stored in the leaves only. Every node  $v$  in the BBST is augmented by an associated data structure ( $\mathcal{T}_{assoc}(v)$ ) which is a 1-dimensional range tree, it can be a BBST or a sorted array, of a canonical subset  $P(v)$  on  $y$ -coordinates. The subset  $P(v)$  is the points stored in the leaves of the sub tree rooted at the node  $v$ . Figure 1 depicts a 2-dimensional range tree for a set of points  $S = \{(2, 2), (5, 2), (3, 3), (4, 3), (2, 4), (5, 4)\}$ . In the case where two points have the same  $x$  or  $y$ -coordinate, we have to define a total order by using a lexicographic one. It consists to replace the real number by a composite-number space [2]. The composite number of two reals  $x$  and  $y$  is denoted by  $(x|y)$ , so for two points, we have:

$$(x|y) < (x'|y') \Leftrightarrow x < x' \vee (x = x' \wedge y < y').$$

Based on the analysis of computational geometry algorithms, our 2-dimensional



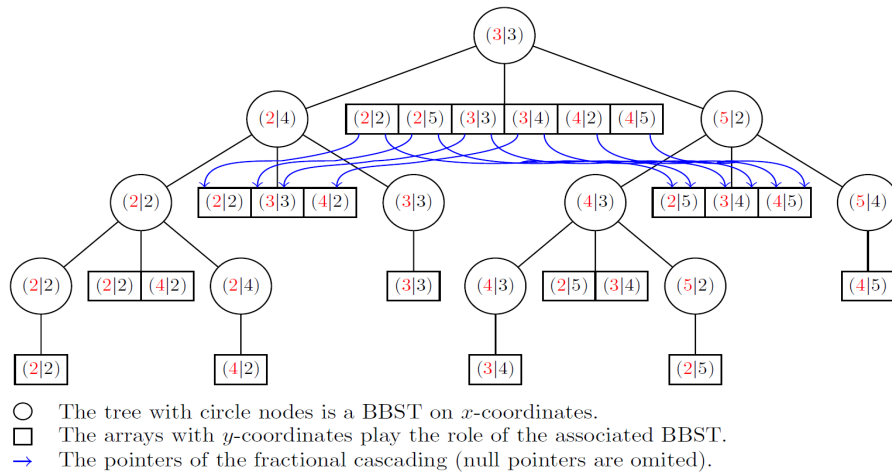
**Fig. 1.** A 2-dimensional range tree.

range tree for a set  $S$  of  $n$  points requires  $O(n \log n)$  storage and can be constructed in  $O(n \log n)$  time.

The range search problem consists to find all the points of  $S$  that satisfy a range query. A useful idea, in terms of efficiency, consists on treating a rectangular range query as a two nested 1-dimensional queries. In other words, let  $[x_1 : x_2] \times [y_1 : y_2]$  be a 2-dimensional range query, we first ask for the points with  $x$ -coordinates in the given 1-dimensional range query  $[x_1 : x_2]$ . Consequently, we select a

The total task of a range query can be performed in  $O(\log^2 n + k)$  time, where  $k$  is the number of points that are in the range. We can improve it by enhancing the 2-dimensional range tree with the fractional cascading technique which is described in the following paragraph.

The application of the fractional cascading technique introduced by [3] on a range tree creates a new data structure so called *layered range tree*. The technique consists to add pointers from the entries of an associated data structure  $\mathcal{T}_{assoc}$  of some level to the entries of an associated data structure below, say  $\mathcal{T}'_{assoc}$  as follows: If  $\mathcal{T}_{assoc}[i]$  stores a value with the key  $y_i$ , then we store a pointer to the entry in  $\mathcal{T}'_{assoc}$  with the smallest key larger than or equal  $y_i$ . We illustrate such technique in Fig. 2 for the same set represented in Fig. 1. Using this technique, the rectangular search query time becomes  $O(\log n + k)$ ,  $O(\log n)$  for the first binary search and  $O(k)$  for browsing the  $k$  reported points.



**Fig. 2.** A layered range tree, an illustration of the fractional cascading (only between two levels).

### 3 String Subsequence Kernels

The philosophy of all string kernel approaches can be reduced to different ways to count common substrings or subsequences that occur in the two strings to compare. This philosophy is manifested in two steps:

- Project the strings over an alphabet  $\Sigma$  to a high dimension vector space  $F$ , where the coordinates are indexed by a subset of the input space.
- Compute the distance (inner product) between strings in  $F$ . Such distance reflects their similarity.

For the String Subsequence Kernel (SSK) [6], the main idea is to compare strings depending on common subsequences they contain. Hence, the more similar strings are ones that have the more common subsequences. However, a new weighting method is adopted. It reflects the degree of contiguity of the subsequence in the string. In order to measure the distance of non contiguous elements of the subsequence, a gap penalty  $\lambda \in ]0, 1]$  is introduced. Formally, the mapping function  $\phi^p(s)$  in the feature space  $F$  can be defined as follows:

$$\phi_u^p(s) = \sum_{I: u=s(I)} \lambda^{l(I)}, \quad u \in \Sigma^p.$$

The associated kernel can be written as:

$$\begin{aligned} K_p(s, t) &= \langle \phi^p(s), \phi^p(t) \rangle \\ &= \sum_{u \in \Sigma^p} \phi_u^p(s) \cdot \phi_u^p(t) \\ &= \sum_{u \in \Sigma^p} \sum_{I: u=s(I)} \sum_{J: u=t(J)} \lambda^{l(I)+l(J)}. \end{aligned}$$

In order to clarify the idea of the SSK, we present a widespread example in the literature. Consider the strings *bar*, *bat*, *car* and *cat*, for a subsequence length  $p = 2$ , the mapping to the feature space is as follows:

$\phi_u^2$	ar	at	ba	br	bt	ca	cr	ct
bar	$\lambda^2$	0	$\lambda^2$	$\lambda^3$	0	0	0	0
bat	0	$\lambda^2$	$\lambda^2$	0	$\lambda^3$	0	0	0
car	$\lambda^2$	0	0	0	0	$\lambda^2$	$\lambda^3$	0
cat	0	$\lambda^2$	0	0	0	$\lambda^2$	0	$\lambda^3$

The unnormalized kernel between *bar* and *bat* is  $K_2(\text{bar}, \text{bat}) = \lambda^4$ , while the normalized version is obtained by :

$$\widehat{K}_2(\text{bar}, \text{bat}) = K_2(\text{bar}, \text{bat}) / \sqrt{K_2(\text{bar}, \text{bar}) \cdot K_2(\text{bat}, \text{bat})} = \lambda^4 / (2\lambda^4 + \lambda^6) = 1 / (2 + \lambda^2).$$

A direct implementation of this kernel leads to  $O(|\Sigma^p|)$  time and space complexity. Since this is the dimension of the feature space. To assist the computation of the SSK a Suffix Kernel is defined through the embedding given by:

$$\phi_u^{p,S}(s) = \sum_{I \in I_p^{|s|}: u=s(I)} \lambda^{l(I)}, u \in \Sigma^p,$$

where  $I_p^k$  denotes the set of  $p$ -tuples of indices  $I$  with  $i_p = k$ . In other words, we consider only the subsequences of length  $p$  that the last symbol is identical to the last one of the string  $s$ . The associated kernel can be defined as follows:

$$\begin{aligned} K_p^S(s, t) &= \langle \phi^{p,S}(s), \phi^{p,S}(t) \rangle \\ &= \sum_{u \in \Sigma^p} \phi_u^{p,S}(s) \cdot \phi_u^{p,S}(t). \end{aligned}$$

To illustrate this kernel counting trick, we take back the precedent example where the mapping is as follows:

$\phi_u^{2,S}$	ar	at	ba	br	bt	ca	cr	ct
bar	$\lambda^2$	0	0	$\lambda^3$	0	0	0	0
bat	0	$\lambda^2$	0	0	$\lambda^3$	0	0	0
car	$\lambda^2$	0	0	0	0	0	$\lambda^3$	0
cat	0	$\lambda^2$	0	0	0	0	0	$\lambda^3$

for example  $K_2^S(bar, bat) = 0$  and  $K_2^S(bat, cat) = \lambda^4$ . The SSK can be expressed in terms of its suffix version as:

$$K_p(s, t) = \sum_{i=1}^{|s|} \sum_{j=1}^{|t|} K_p^S(s(1:i), t(1:j)). \quad (1)$$

with  $K_1^S(s, t) = [s_{|s|} = t_{|t|}] \lambda^2$ .

### 3.1 Naive Implementation

The computation of the similarity of two strings ( $sa$  and  $tb$ ) is conditioned by their final symbols. In the case where  $a = b$ , we have to sum kernels of all prefixes of  $s$  and  $t$ . Hence, a recursion has to be devised:

$$K_p^S(sa, tb) = [a = b] \sum_{i=1}^{|s|} \sum_{j=1}^{|t|} \lambda^{2+|s|-i+|t|-j} K_{p-1}^S(s(1:i), t(1:j)). \quad (2)$$

This computation leads to a complexity of  $O(p(|s|^2|t|^2))$ .

### 3.2 Efficient Implementations

We present three methods that compute the SSK efficiently, namely the dynamic programming [6], the trie-based [5,7,8] and the sparse dynamic programming approaches [7].

To describe such approaches, we use two strings  $s = \text{gatta}$  and  $t = \text{cata}$  as a running example.

**Dynamic Programming Approach.** The starting point of the dynamic programming approach is the suffix recursion given by equation (2). From this equation, we can consider a separate dynamic programming table  $DP_p$  for storing the double sum:

$$DP_p(k, l) = \sum_{i=1}^k \sum_{j=1}^l \lambda^{k-i+l-j} K_{p-1}^S(s(1:i), t(1:j)). \quad (3)$$

It is easy to see that:  $K_p^S(sa, tb) = [a = b] \lambda^2 DP_p(|s|, |t|)$ .

Computing ordinary  $DP_p$  for each  $(k, l)$  would be inefficient. So we can devise a recursive version of equation (3) with a simple counting device:

$$DP_p(k, l) = K_{p-1}^S(s(1:k), t(1:l)) + \lambda DP_p(k-1, l) + \lambda DP_p(k, l-1) - \lambda^2 DP_p(k-1, l-1).$$

Consequently, using the dynamic programming approach (Algorithm 1), the complexity of the SSK becomes  $O(p|s||t|)$ .

Table 1 illustrates the computation of the dynamic programming tables for the running example for  $p = 1, 2$ . The evaluation of the kernel is given by the sum of entries of the suffix table KPS:

$$\begin{aligned} K_1(s, t) &= 6\lambda^2. \\ K_2(s, t) &= 2\lambda^4 + 2\lambda^5 + \lambda^7. \end{aligned}$$

**Trie-based Approach.** This approach is based on search trees known as tries, introduced by E. Fredkin in 1960. The key idea of the trie-based approach is that leaves play the role of the feature space indexed by the set  $\Sigma^p$  of strings of length  $p$ . In the literature, there are variants of trie-based string subsequence kernels. For instance the  $(p, m)$ -mismatch string kernel [5] and restricted SSK [8]. In the present section, we try to describe a trie-based SSK presented in [7] that slightly differ from those cited above [5,8]. Figure 3 illustrates the trie data structure for the running example. Each node in the trie corresponds to a co-occurrence between strings. The algorithm maintains for all matches  $u = s(I) = u_1 \cdots u_q$ ,  $I = i_1 \cdots i_q$  a list of alive matches  $A_s(u, g)$  as presented in Table 2 that records



**Algorithm 1:** Dynamic SSK computation

---

**Input:** Strings  $s$  and  $t$ , the length of the subsequence  $p$ , and the decay penalty  $\lambda$   
**Output:** Kernel values  $K_q(s, t) = K(q) : q = 1, \dots, p$

```

1  $m \leftarrow \text{length}(s)$ 
2  $n \leftarrow \text{length}(t)$ 
3  $K[1:p] \leftarrow 0$ 
  /* Computation of  $K_1(s, t)$  */
4 for  $i = 1:m$  do
5   for  $j = 1:n$  do
6     if  $s[i] = t[j]$  then
7        $KPS[i, j] \leftarrow \lambda^2$ 
8        $K[1] \leftarrow K[1] + KPS[i, j]$ 
  /* Computation of  $K_q(s, t) : q = 2, \dots, p$  */
9 for  $q = 2:p$  do
10  for  $i = 1:m$  do
11    for  $j = 1:n$  do
12       $DP[i, j] \leftarrow KPS[i, j] + \lambda DP[i-1, j] + \lambda DP[i, j-1] - \lambda^2 DP[i-1, j-1]$ 
13      if  $s[i] = t[j]$  then
14         $KPS[i, j] \leftarrow \lambda^2 DP[i-1, j-1]$ 
15         $K[q] \leftarrow K[q] + KPS[i, j]$ 

```

---

the last index  $i_q$  where  $g = l(I) - |I|$  is the number of gaps in the match. Notice that in the same list we are able to record many occurrences with different gaps. Alive lists for longer matches  $uc, c \in \Sigma$ , can be constructed incrementally by extending the alive list corresponding to  $u$ . Similarly, the algorithm is applied to the string  $t$ . The process will continue until achieving the depth  $p$ . For efficiency reasons, we need to restrict the number of gaps to a given integer  $g_{max}$ , so the computation is approximate. The kernel is evaluated as follows:

$$K_p(s, t) = \sum_{u \in \Sigma^p} \phi_u^p(s) \phi_u^p(t) = \sum_{g_s, g_t} \lambda^{g_s+p} |L_s(u, g_s)| \cdot \lambda^{g_t+p} |L_t(u, g_t)|.$$

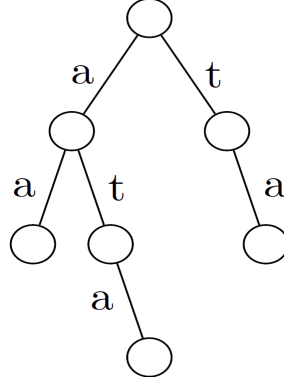
Given that, there are  $\binom{p+g_{max}}{g_{max}}$  possible combinations to assign  $p$  letters and  $g_{max}$  gaps in a window of length  $p + g_{max}$ , the worst-case time complexity of the algorithm is  $O\left(\binom{p+g_{max}}{g_{max}} (|s| + |t|)\right)$ .

The string subsequence kernel for the running example for  $p = 1$  is:

$$\begin{aligned} K_1(s, t) &= \lambda^{0+1} |A_s('a', 0)| \cdot \lambda^{0+1} |A_t('a', 0)| + \lambda^{0+1} |A_s('t', 0)| \cdot \lambda^{0+1} |A_t('t', 0)| \\ &= 4 \cdot \lambda^2 + 2 \cdot \lambda^2 = 6 \cdot \lambda^2. \end{aligned}$$

**Table 1.** Suffix tables and dynamic programming tables to compute the SSK for  $p = 1, 2$ .

$KPS_1$	g	a	t	t	a
c					
a		$\lambda^2$			$\lambda^2$
t			$\lambda^2$	$\lambda^2$	
a		$\lambda^2$			$\lambda^2$
$DP_2$	g	a	t	t	a
c	0	0	0	0	
a	0	$\lambda^2$	$\lambda^3$	$\lambda^4$	
t	0	$\lambda^3$	$\lambda^2 + \lambda^4$	$\lambda^2 + \lambda^3 + \lambda^5$	
a					
$KPS_2$	g	a	t	t	a
c					
a					
t		$\lambda^4$		$\lambda^5$	
a					$\lambda^4 + \lambda^5 + \lambda^7$

**Fig. 3.** The trie data structure for the running example  $s = gatta, t = cata$ 

Similar computation is performed for  $K_2$  and  $K_3$ :

$$\begin{aligned}
 K_2(s, t) &= (1 \cdot \lambda^{2+2}) \cdot (1 \cdot \lambda^{1+2}) + (1 \cdot \lambda^{0+2} + 1 \cdot \lambda^{1+2}) \cdot (1 \cdot \lambda^{0+2}) + (1 \cdot \lambda^{0+2} + 1 \cdot \lambda^{1+2}) \cdot (1 \cdot \lambda^{0+2}) \\
 &= \lambda^7 + 2 \cdot \lambda^5 + 2 \cdot \lambda^4.
 \end{aligned}$$

and

$$K_3(s, t) = (2 \cdot \lambda^{1+3}) \cdot (1 \cdot \lambda^{0+3}) = 2 \cdot \lambda^7.$$

**Table 2.** The alive indices for all subsequences of the running example for  $p = 1, 2, 3$  with the number of gaps from 0 to 3.

$g$	$A_s('a', g)$	$A_t('a', g)$	$A_s('t', g)$	$A_t('t', g)$	$A_s('aa', g)$	$A_t('aa', g)$
0	{2, 5}	{2, 4}	{3, 4}	{3}		
1						{4}
2					{5}	
3						
$g$	$A_s('at', g)$	$A_t('at', g)$	$A_s('ta', g)$	$A_t('ta', g)$	$A_s('ata', g)$	$A_t('ata', g)$
0	{3}	{3}	{5}	{4}		{4}
1	{4}		{5}		{5, 5}	
2						
3						

**Sparse Dynamic Programming Approach.** It is built on the fact that in many cases, most of the entries of the  $DP$  matrix are zero and do not contribute to the result. Rousu and Shawe-Taylor [7] have proposed a solution using the sparse dynamic programming technique to avoid unnecessary computations. To do so, two data structures were proposed: the first one is a range sum tree, which is a B-tree, that replaces the  $DP_p$  matrix. It is used to return the sum of  $n$  values within an interval in  $O(\log n)$  time. The second one is a set of match lists instead of  $K_p^S$  matrix.  $L_q(i) = \{(j_1, \overline{K_p^S}(s(1:i), t(1:j_1))), (j_2, \overline{K_p^S}(s(1:i), t(1:j_2))), \dots\}$  where  $\overline{K_p^S}(s(1:i), t(1:j)) = \lambda^{m-i+n-j} K_p^S(s(1:i), t(1:j))$ . This dummy gap weight  $\lambda^{m-i+n-j}$  allows to address the problem of scaling the kernel values as the computation progress. Consequently the recursion (2) becomes:

$$\overline{K_p^S}(sa, tb) = [a = b] \lambda^2 \sum_{i \leq |s|} \sum_{j \leq |t|} \overline{K_{p-1}^S}(s(1:i), t(1:j)). \quad (4)$$

and the separate dynamic programming table (3) can be expressed as follows:

$$\overline{DP_p}(k, l) = \sum_{i \leq k} \sum_{j \leq l} \overline{K_{p-1}^S}(s(1:i), t(1:j)). \quad (5)$$

Thereafter, the authors devise a recursive version of (5):

$$\overline{DP_p}(k, l) = \overline{DP_p}(k-1, l) + \sum_{j \leq l} \overline{K_{p-1}^S}(s(1:k), t(1:j)). \quad (6)$$

This can be interpreted as reducing the evaluation of an orthogonal range query (5) to an evaluation of a simple range query multiple times as much as the number of lines of the  $K_p^S$  matrix.

To evaluate efficiently a range query, the authors use a range-sum tree to store a set  $S = \{(j, v_j)\} \subset \{1, \dots, n\} \times \mathbf{R}$  of key-value pairs. A range-sum tree is a binary tree of height  $h = \lceil \log n \rceil$  where each node in depth  $d = 0, 1, \dots, h-1$  contains a key  $j$  with a sum of values in a sub range  $[j - 2^{h-d} + 1, j]$ . The root

is labeled with  $2^h$ , the left child of a node  $j$  is  $j - j/2$  and the right child if it exists is  $j + j/2$ . Odd keys label the leaves of the tree.

To compute the range sum of values within an interval  $[1, j]$  it suffices to browse the path from the node  $j$  to the root and sum over the left subtrees as follows:

$$Rangsum([1, j]) = v_j + \sum_{h \in \text{Ancestors}(j)/h < j} v_h.$$

Moreover to update the value of a node  $j$ , we need to update all the values of parents that contain  $j$  in their subtree ( $h \in \text{Ancestors}(j)/h > j$ ). These two operations are performed in  $O(\log n)$  time because we traverse in the worst case the height of the tree.

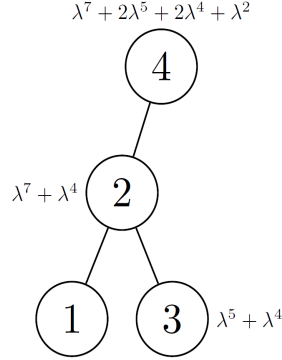
For the sparse dynamic programming algorithm (Algorithm 2) the range-sum tree is used incrementally when computing (6). So that when processing the match list  $L_p(k)$  the tree will contain the values  $v_j$  that satisfy  $\sum_{i=1}^k \overline{K_{p-1}^S}(s(1:i), t(1:j)), 1 \leq j \leq l$ . Hence the evaluation of (6) is performed by involving a one dimensional range query:

$$\begin{aligned} Rangsum([1, j]) &= \sum_{j=1}^l v_j \\ &= \sum_{i=1}^k \sum_{j=1}^l \overline{K_{p-1}^S}(s(1:i), t(1:j)) \\ &= \overline{DP}_p(k, l). \end{aligned}$$

Concerning the cost of computation of this approach, the set of match lists is created on  $O(m + n + |\Sigma| + |L_1|)$  time and space, while the kernel computation time is  $O(p|L_1| \log n)$ , knowing that  $|L_1| \geq |L_2| \geq \dots \geq |L_p|$ .

To illustrate the mechanism of the sparse dynamic programming algorithm, Figure 4 depicts the state of the range-sum tree when computing  $K_2^S(s, t)$ . Initially the set of match lists is created as follows:

$$\begin{aligned} L_1(1) &= () \\ L_1(2) &= ((2, \lambda^7), (4, \lambda^5)) \\ L_1(3) &= ((3, \lambda^5)) \\ L_1(4) &= ((3, \lambda^4)) \\ L_1(5) &= ((2, \lambda^4), (4, \lambda^7)). \end{aligned}$$



**Fig. 4.** The state of the range-sum tree when computing  $K_2^S(s, t)$

Meanwhile maintaining the range-sum tree, the algorithm update the set of match lists as presented below:

$$\begin{aligned}
 L_2(1) &= () \\
 L_2(2) &= () \\
 L_2(3) &= ((3, \lambda^7)) \\
 L_2(4) &= ((3, \lambda^7)) \\
 L_2(5) &= ((4, \lambda^7 + \lambda^5 + \lambda^4)).
 \end{aligned}$$

Finally, summing the values of the updated match list after discarding the dummy weight gives the kernel value  $K_2(s, t)$ :

$$\begin{aligned}
 K_2(s, t) &= \lambda^7 \cdot \lambda^{-9+3+3} + \lambda^7 \cdot \lambda^{-9+4+3} + (\lambda^7 + \lambda^5 + \lambda^4) \cdot \lambda^{-9+5+4} \\
 &= \lambda^7 + 2\lambda^5 + 2\lambda^4.
 \end{aligned}$$

## 4 Geometric based Approach

Looking forward to improving the complexity of SSK, our approach is based on two observations. The first one concerns the computation of  $K_p^S(s, t)$  that is required only when  $s_{|s|} = t_{|t|}$ . Hence, we have kept only a list of index pairs of these entries rather than the entire suffix table,  $L(s, t) = \{(i, j) : s_i = t_j\}$ .

In the rest of the paper, while measuring the complexity of different computations, we will consider,  $|L|$ , the size of the match list  $L(s, t)$  as the parameter indicating the size of the input data.

The complexity of the naive implementation of the list version is  $O(p|L|^2)$ , and it seems not obvious to compute  $K_p^S(s, t)$  efficiently on a list data structure.

**Algorithm 2:** Sparse Dynamic SSK computation

---

**Input:** Strings  $s$  and  $t$ , the length of the subsequence  $p$ , and the decay penalty  $\lambda$   
**Output:** Kernel value  $K_p(s, t) = K$

```

1  $m \leftarrow \text{length}(s)$ 
2  $n \leftarrow \text{length}(t)$ 
3 Creation of the set of match lists  $L_1$ 
4 for  $q = 2:p$  do
5    $\text{Rangesum}(1 : n) \leftarrow 0$  (Initialization of the range-sum tree)
6   for  $i = 1:m$  do
7     foreach  $(j_h, v_h) \in L_{q-1}(i)$  do
8        $S \leftarrow \text{Rangesum}[1, j_h - 1]$ 
9       if  $S > 0$  then
10         $\text{appendlist}(L_q(i), (j_h, S))$ 
11        /* Update of the range-sum tree */
12        foreach  $(j_h, v_h) \in L_{q-1}(i)$  do
13           $\text{update}(\text{Rangesum}, (j_h, v_h))$ 
14        /* Computation of the kernel value for the final level */
15   $K \leftarrow 0$ 
16  for  $i = 1:m$  do
17    foreach  $(j_h, v_h) \in L_p(i)$  do
18       $K \leftarrow K + v_h \lambda^{-m-n+i+j_h}$ 

```

---

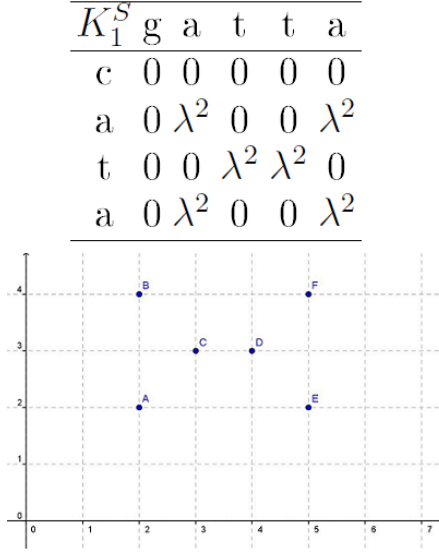
In order to address this problem, we have made a second observation that the suffix table can be represented as a 2-dimensional space (plane) and the entries where  $s_{|i|} = t_{|j|}$  as points in this plane as depicted in Fig. 5. In this case, the match list generated is

$$L(s, t) = \{A, B, C, D, E, F\} = \{(2, 2), (2, 4), (3, 3), (4, 3), (5, 2), (5, 4)\}.$$

With a view to improving the computation of the SSK, it is easy to perceive from Fig. 5 that the computation of (2) can be interpreted as orthogonal range queries. In this respect, we have used a layered range tree (LRT) in [1]. But the LRT data structure reports all points that lie in a specific range query. However, for the SSK computation we require only the sum of values of the reported points.

To achieve this goal, we extend the LRT with the aggregate operations, in particular the summation one. Hence, a novel data structure was created, for instance a Layered Range Sum Tree (LRST). A LRST is a LRT with two substantial extensions to reduce the range sum query time from  $O(\log |L| + k)$  to  $O(\log |L|)$  where  $k$  is the number of reported points in the range.

The first extension consists to substitute the associated data structures  $\mathcal{T}_{assoc}$  in the LRT with new associated data structures  $\mathcal{T}'_{assoc}$  where each entry  $i$  contains a key-value pair  $(j, ps_j)$ ,  $ps_j = \sum_{k=1}^i v_k$  is the partial sum of  $\mathcal{T}_{assoc}$  in the position  $i$ . This extension is made to compute the range sum of  $\mathcal{T}_{assoc}$  within  $[i, j]$  in  $O(1)$  time as follows :  $\text{Rangesum}[i, j] = \mathcal{T}'_{assoc}[j] - \mathcal{T}'_{assoc}[i - 1]$ .



**Fig. 5.** Representation of the suffix table as a 2-dimensional space

The second extension involves the fractional cascading technique. Let  $\mathcal{T}'_{assoc1}$  and  $\mathcal{T}'_{assoc2}$  be two sorted arrays that store partial sums of  $\mathcal{T}_{assoc1}$  and  $\mathcal{T}_{assoc2}$  respectively. Suppose that we want to compute the range sum within a query  $q = [y_1, y_2]$  in  $\mathcal{T}_{assoc1}$  and  $\mathcal{T}_{assoc2}$ . We start with a binary search with  $y_1$  in  $\mathcal{T}'_{assoc1}$  to find the smallest key larger than or equal  $y_1$ . We make also an other binary search with  $y_2$  in  $\mathcal{T}'_{assoc1}$  to find the largest key smaller than or equal  $y_2$ . If an entry  $\mathcal{T}'_{assoc1}[i]$  stores a key  $y_i$  then we store a pointer to the entry in  $\mathcal{T}'_{assoc2}$  with the smallest key larger than or equal to  $y_i$ , say *small pointer*, and a second pointer to the entry in  $\mathcal{T}'_{assoc2}$  with the largest key smaller than or equal to  $y_i$ , say *large pointer*. If there is no such key(s) then the pointer(s) is (are) set to **nil**.

It is easy to see that our extensions does not affect neither the space nor the time complexities of the LRT construction. So according to the analysis of of computational geometry algorithms, our LRST requires  $O(|L| \log |L|)$  storage and can be constructed in  $O(|L| \log |L|)$  time. This leads to the following lemma.

**Lemma 1.** *Let  $s$  and  $t$  be two strings and  $L(s, t) = \{(i, j) : s_i = t_j\}$  the match list associated to the suffix version of the SSK. A Layered range sum tree (LRST) for  $L(s, t)$  requires  $O(|L| \log |L|)$  storage and takes  $O(|L| \log |L|)$  construction time.*

We can now exploit these extensions to compute efficiently the range sum inherent to  $q = [y_1, y_2]$  in  $\mathcal{T}_{assoc1}$  and  $\mathcal{T}_{assoc2}$ . Let  $\mathcal{T}'_{assoc1}[i_1]$  and  $\mathcal{T}'_{assoc1}[i_2]$  be the results of the binary search with  $y_1$  and  $y_2$  respectively in  $\mathcal{T}'_{assoc1}$ . So the  $Rangesum(y_1, y_2) = \mathcal{T}'_{assoc1}[i_2] - \mathcal{T}'_{assoc1}[i_1 - 1]$  in  $\mathcal{T}_{assoc1}$  takes  $O(\log |L|)$

time. To compute the range sum in  $\mathcal{T}_{assoc2}$  we avoid the binary searches. We consider first the entry  $\mathcal{T}'_{assoc2}[j_1]$  pointed by the *small pointer* of  $\mathcal{T}'_{assoc1}[i_1]$ , it contains the smallest key from  $\mathcal{T}'_{assoc2}$  larger than or equal to  $y_1$ . The second entry is  $\mathcal{T}'_{assoc2}[j_2]$  pointed by the *large pointer* of  $\mathcal{T}'_{assoc1}[i_2]$ , it contains the largest key from  $\mathcal{T}'_{assoc2}$  smaller than or equal to  $y_2$ . Finally the range sum within  $[y_1, y_2]$  in  $\mathcal{T}_{assoc2}$  is given by  $Rangesum(y_1, y_2) = \mathcal{T}'_{assoc2}[j_2] - \mathcal{T}'_{assoc2}[j_1 - 1]$  and it takes  $O(1)$  time.

---

**Algorithm 3:** Geometric SSK computation

---

**Input:** Strings  $s$  and  $t$ , the length of the subsequence  $p$ , and the decay penalty  $\lambda$   
**Output:** Kernel values  $K_q(s, t) = K(q) : q = 1, \dots, p$

```

1  $m \leftarrow \text{length}(s)$ 
2  $n \leftarrow \text{length}(t)$ 
3 Creation of the initial match list  $L$ 
  /* Computation of  $K_1(s, t)$  */
4 foreach  $((i, j), v) \in L$  do
5    $K[1] \leftarrow K[1] + v \cdot \lambda^{i+j}$ 
  /* Computation of  $K_q(s, t) : q = 2, \dots, p$  */
6 for  $q = 2:p$  do
7   Building of the LRST corresponding to the match list  $L$ 
8   foreach  $((i, j), v) \in L$  do
9     /* Preparing the range query for the entry  $(i, j)$  */
10     $rq \leftarrow [(0| - \infty) : (i - 1| + \infty)] \times [(0| - \infty) : (j - 1| + \infty)]$ 
11     $result \leftarrow Rangesum[rq]$ 
12    if  $result > 0$  then
13       $K[q] = K[q] + result \cdot \lambda^{i+j}$ 
14       $appendlist(newL, ((i, j), result))$ 
15    $L \leftarrow newL$ 
```

---

For our geometric approach (Algorithm 3) we will use the LRST to evaluate the SSK. We start by the creation of the match list  $L(s, t) = \{((i, j), \widetilde{K}_p^S(s(1 : i), t(1 : j))) : s_i = t_j\}$  where  $\widetilde{K}_p^S(s(1 : i), t(1 : j)) = \lambda^{2-i-j} K_p^S(s(1 : i), t(1 : j))$ . This trick is inspired from [7] to make the range sum results correct. Thus the recursion (2) becomes as follows:

$$\widetilde{K}_p^S(sa, tb) = [a = b] \sum_{i \leq |s|} \sum_{j \leq |t|} \lambda^{i+j} \widetilde{K}_{p-1}^S(s(1 : i), t(1 : j)). \quad (7)$$

In order to construct efficiently the match list we have to create for each character  $c \in \Sigma$  a list  $I(c)$  of occurrence positions ( $c = s_i$ ) in the string  $s$ . Thereafter, for each character  $t_j \in t$  we insert key-value pairs  $((i, j), \widetilde{K}_p^S(s(1 : i), t(1 : j)))$  in the match list  $L(s, t)$  corresponding to  $I(t_j)$ . This process takes  $O(|s| + |t| + |\Sigma| + |L|)$  space and  $O(|s| + |\Sigma| + |L|)$  time. For example, the match list for our running example is  $L(s, t) = \{((2, 2), \lambda^7), ((2, 4), \lambda^5), ((3, 3), \lambda^5), ((4, 3), \lambda^4), ((5, 2), \lambda^4), ((5, 4), \lambda^2)\}$ .



Once the initial match list created, we start computing the SSK for the subsequence length  $p = 1$ . This computation doesn't require the LRST ; it suffices to traverse the match list and sum over its values. For length subsequence  $q > 1$  we will first create the LRST corresponding to the match list, afterward for each item  $((k, l), \widetilde{K_p^S}(s(1 : k), t(1 : l)))$  we invoke the LRST with the query  $rq = [0, k - 1] \times [0, l - 1]$ . This latter return the range sum within  $rq$ :  $Rangesum(rq) = \sum_{i < k} \sum_{j < l} (\widetilde{K_p^S}(s(1 : i), t(1 : j)))$ . If  $Rangesum(rq)$  is positive then insert the key-value in a new match list for the level  $q + 1$  and summing the  $Rangesum(rq)$  to compute the SSK at the level  $q$ . At each iteration, we have to create a new LRST corresponding to the new match list until achieving the request subsequence length  $p$ .

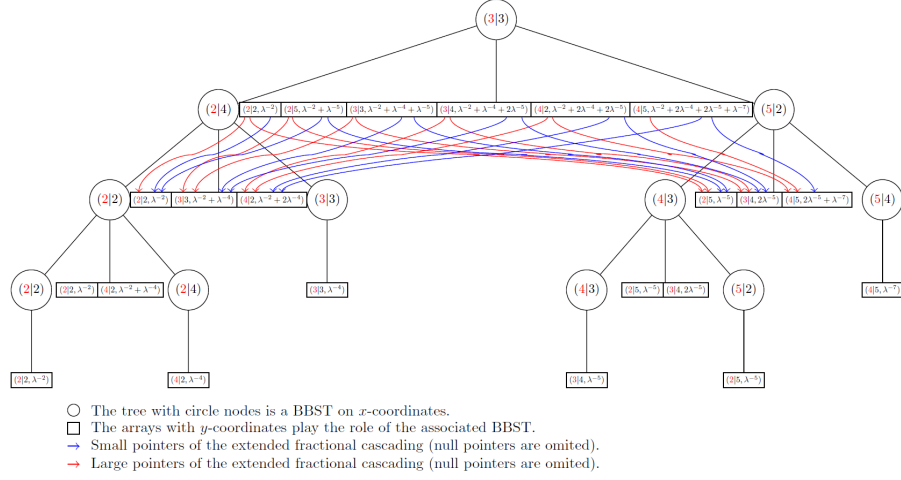
We recall that in our case, we use composite numbers instead of real numbers, see section (2.5). In such situation, we have to transform the range query  $[x_1 : x_2] \times [y_1 : y_2]$  related to a set of points in the plane to the range query  $[(x_1 | -\infty) : (x_2 | +\infty)] \times [(y_1 | -\infty) : (y_2 | +\infty)]$  related to the composite space.

Using our geometric approach, the range sum query time becomes  $O(\log |L|)$ . For the computation of  $K_p^S(s, t)$  we have to consider  $|L|$  entries of the match list. The process iterates  $p$  times, therefore, we get a time complexity of  $O(p|L| \log |L|)$  for evaluating the SSK. This result combined to that of Lemma. 1 lead to the following theorem that summarizes the result of our proposed approach to compute SSK.

**Theorem 2.** *Let  $s$  and  $t$  be two strings and  $L(s, t) = \{(i, j) : s_i = t_j\}$  the match list associated to the suffix version of the SSK. A layered range sum tree requires  $O(|L| \log |L|)$  storage and it can be constructed in  $O(|L| \log |L|)$  time. With these data structures, the SSK of length  $p$  can be computed in  $O(p|L| \log |L|)$ .*

To compute  $K_2(s, t)$ , for our running example, we have to invoke the range sum on the LRST at the step  $p = 2$  represented by Fig.6. The SSK computation is performed by summing over all the range sums corresponding to the entries of the match list as follows:  $K_2(s, t) = Rangesum[(0 | -\infty) : (1 | +\infty)] \times [(0 | -\infty) : (1 | +\infty)] + Rangesum[(0 | -\infty) : (1 | +\infty)] \times [(0 | -\infty) : (3 | +\infty)] + Rangesum[(0 | -\infty) : (2 | +\infty)] \times [(0 | -\infty) : (2 | +\infty)] + Rangesum[(0 | -\infty) : (3 | +\infty)] \times [(0 | -\infty) : (2 | +\infty)] + Rangesum[(0 | -\infty) : (4 | +\infty)] \times [(0 | -\infty) : (1 | +\infty)] + Rangesum[(0 | -\infty) : (4 | +\infty)] \times [(0 | -\infty) : (3 | +\infty)]$ .

To describe how this can be processed, we deal by the range sum of the query  $[(0 | -\infty) : (4 | +\infty)] \times [(0 | -\infty) : (3 | +\infty)]$ . At the associate data structure corresponding to the split node (3|3) of Fig.6 we find the entries (2|2) and (3|4) whose  $y$ -coordinates are the smallest one larger than or equal to  $(0 | -\infty)$  and the largest one smaller or equal to  $(3 | +\infty)$  respectively. This can be done by binary search. Next, we look for the nodes that are below the split node (3|3) and that are the right child of a node on the search path to  $(0 | -\infty)$  where the path go left, or the left child of a node on the search path to  $(4 | +\infty)$  where the path go right. The collected nodes are (3|3), (2|2), (4|3) and the result returned from the associated data structures is  $\lambda^{-5} + \lambda^{-4} + \lambda^{-2}$ . This is done on a constant time by following the small and large pointers from the associated data structure



**Fig. 6.** The state of the layered range sum tree for the running example at the step  $p = 2$  with an illustration of the extended fractional cascading (only between two levels).

of the split node. By the same process we obtain the following results of the invoked range sums:

$$\text{Rangesum}[(0| - \infty) : (1| + \infty)] \times [(0| - \infty) : (1| + \infty)] = 0$$

$$\text{Rangesum}[(0| - \infty) : (1| + \infty)] \times [(0| - \infty) : (3| + \infty)] = 0$$

$$\text{Rangesum}[(0| - \infty) : (2| + \infty)] \times [(0| - \infty) : (2| + \infty)] = \lambda^{-2}$$

$$\text{Rangesum}[(0| - \infty) : (3| + \infty)] \times [(0| - \infty) : (2| + \infty)] = \lambda^{-2}$$

$$\text{Rangesum}[(0| - \infty) : (4| + \infty)] \times [(0| - \infty) : (1| + \infty)] = 0$$

After rescaling the returned values by the factor  $\lambda^{i+j}$  we obtain the value of  $K_2(s, t) = \lambda^{-2} \cdot \lambda^{3+3} + \lambda^{-2} \cdot \lambda^{4+3} + (\lambda^{-5} + \lambda^{-4} + \lambda^{-2}) \cdot \lambda^{5+4} = 2\lambda^4 + 2\lambda^5 + \lambda^7$ . While invoking the range sums we will prepare the new match list for the next step. In our case the new match list contains the following matches :  $\{((3, 3), \lambda^{-2}), ((4, 3), \lambda^{-2}), ((5, 2), \lambda^{-5} + \lambda^{-4} + \lambda^{-2})\}$ .

## 5 Experimentation

In this section we describe the experiments that focus on the evaluation of our geometric algorithm against the dynamic and the sparse dynamic ones. Thereafter, these algorithms are referenced as Geometric, Dynamic and Sparse respectively. We have discarded the trie-based algorithm from this comparison because it is an approximate algorithm on the one hand, on the other hand in the preliminary experiments conducted in [7] it was significantly slower than Dynamic and Sparse.

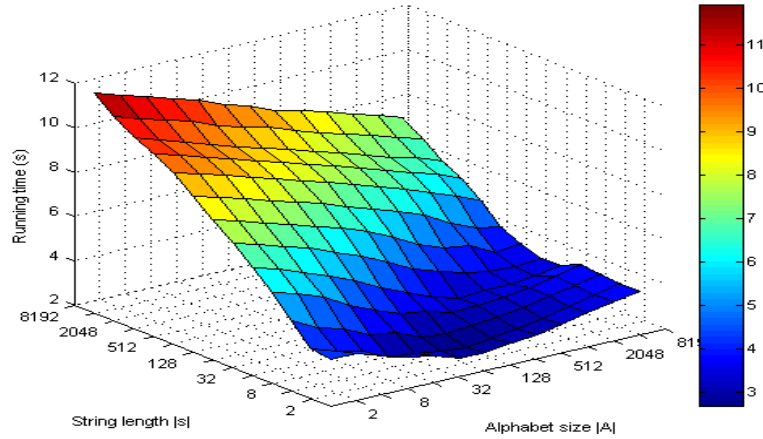
To benefit from the empiric evaluation conducted in [7], we tried to keep the same conditions of their experiments. For this reason, we have conducted a series of experiments on both synthetically generated and on newswire article data on Reuter's news articles.

We ran the tests on Intel Core i7 at 2.40 GHZ processor with 16 GB RAM under Windows 8.1 64 bit. We implemented all the tested algorithms in Java. For the LRST implementation, we have extended the LRT implementation available on the page <https://github.com/epsilony/>.

### 5.1 Experiments with synthetic data

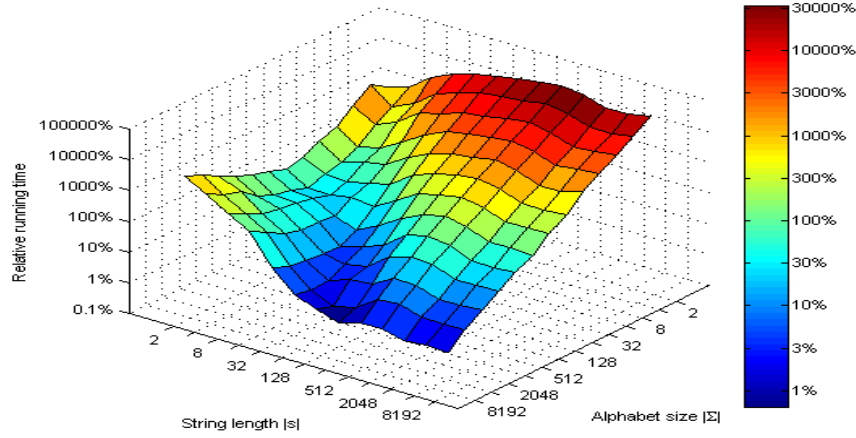
These experiments concern the effects of the string length and the alphabet size on the efficiency of the different approaches and to determine under which conditions our approach outperforms.

We randomly generated string pairs with different lengths ( $2, 4, \dots, 8192$ ) over alphabets of different sizes ( $2, 4, \dots, 8192$ ). To simplify the string generation, we considered string symbols as integer in  $[1, \text{alphabet size}]$ . For convenience of data visualization, we have used the logarithmic scale on all axes. To perform accurate experiments, we have generated multiple pairs for the same string length and alphabet size and for each pair we have took multiple measures of the running time with a subsequence length  $p = 10$  and a decay parameter  $\lambda = 0.5$ . This being



**Fig. 7.** Running Time of the Geometric algorithm on synthetic data.

said, Fig. 7 reveals, for our geometric approach, an inverse dependency of the running time with the alphabet size. However, for an alphabet size the running time is proportional to the string length. Figure 8 shows experimental comparison of the performance of the proposed approach against Dynamic. Note that the rate of 100% indicates that the two algorithms deliver the same performances. For the rates less than 100% our approach outperforms, it is the case for strings based on medium and large alphabets excepting those having short length (say alphabet size great than or equal 256, where the string length exceeds 128 characters). For



**Fig. 8.** Relative running Time on synthetic data: Geometric/Dynamic.

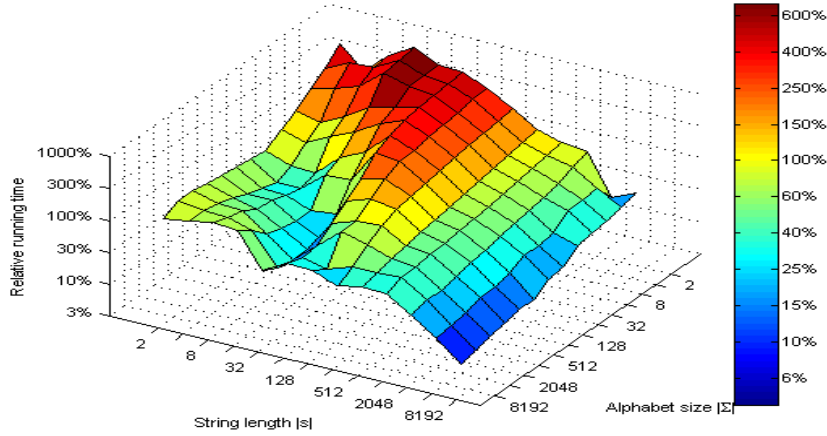
short strings and also for long strings based on small alphabets, Dynamic excels. It remains to present results of the comparison experiment with Sparse which share the same motivations with our approach. Rousu and Shawe-Taylor [7] state that with long strings based on large alphabets their approach is faster. Figure 9 shows that in these conditions our approach dominates. Moreover, our approach is faster than the Sparse one for long strings and for large alphabets absolutely, but gets slower than Sparse for short strings based on small alphabets.

## 5.2 Experiments with newswire article data

Our second experiments use the Reuters-21578 collection to evaluate the speed of Geometric against Dynamic and Sparse on English articles. We created a dataset represented as sequences of syllables by transferring all the XML articles on to text documents. Thereafter, the text documents are preprocessed by removing stop words, punctuation marks, special symbols and finally word syllabifying. We have generated 22260 distinct syllables. As in the first experiment, each syllable alphabet is assigned an integer. To treat the documents randomly, we have shuffled this preliminary dataset.

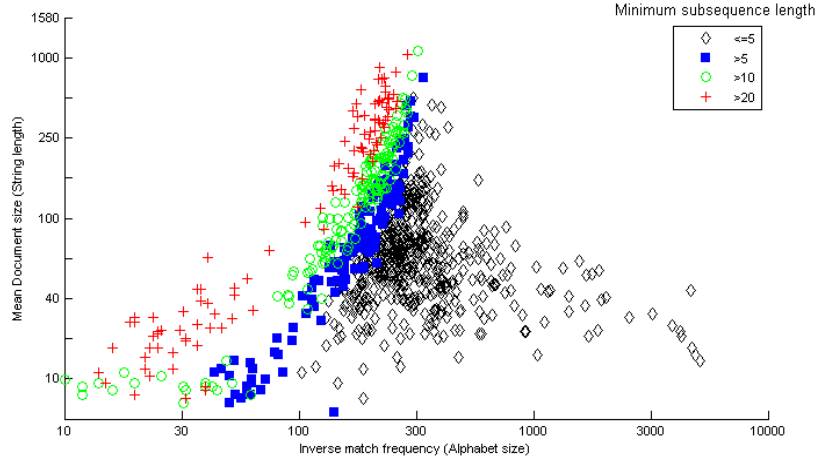
For visualization convenience, while creating document pairs, we have ensured that their lengths are close. Under this condition, we have collected 916 pair documents as final dataset.

To compare the candidate algorithms, we have computed the SSK for each document pair of the data set by varying the subsequence length from 2 to 20. Figure 10 and Figure 11 depict the clusters of documents where Geometric is faster than Dynamic and Sparse respectively. A document pair  $(s, t)$  is plotted according to the inverse match frequency (X-axis) and the document size (Y-axis).



**Fig. 9.** Relative running Time on synthetic data: Geometric/Sparse.

The inverse match frequency is given by:  $|s||t|/|L|$ , it plays the role of the alphabet size  $|\Sigma|$  inherent to the documents  $s$  and  $t$ . The document size is calculated as the arithmetic mean of the document pair sizes, it plays the role of the string length. Each cluster is distinguished by a special marker that corresponds to the necessary minimum subsequence length to make Geometric faster than Dynamic or Sparse. For the cluster marked by black diamonds,  $p \leq 5$  is sufficient. The length  $5 < p \leq 10$  is required for the cluster marked by blue filled squares. For the cluster marked by green circles  $10 < p \leq 20$  is required and the last cluster marked by plus signs  $p \geq 20$  is needed. We can distinguish three cases in Fig. 10. The first one arises when the inverse match frequency is weak (small alphabet size), that is to say for dense matrix. In this case, we require important values of the subsequence length ( $p > 10$  for small documents and  $p > 20$  for larger ones) to make Geometric faster than Dynamic. The second case concerns good inverse match frequencies (large alphabet size) corresponding to sparse matrix. In this case, small values of the subsequence length ( $p \leq 5$ ) suffice to make Geometric faster than Dynamic. The third case appear for moderate inverse match frequency (medium alphabet size), the values of  $p$  that makes Geometric faster than Dynamic depend on the document size. The large document size the large  $p$  is required. The results of the comparison between Geometric and Sparse on newswire article data are depicted in Fig. 11. We can discuss 3 cases: The first case emerge when the document size becomes large and also for good inverse match frequency. In this case small values of the subsequence length ( $p \leq 5$ ) suffice to make Geometric faster than Sparse. The second case appear for small documents and bad inverse match frequencies. The necessary subsequence length must be important ( $p > 10$  for very small documents and  $p > 20$  for the small ones). The third case concerns moderate inverse frequencies. In this case



**Fig. 10.** Clusters of document pairs where Geometric is faster than Dynamic according to the subsequence length  $p$ .

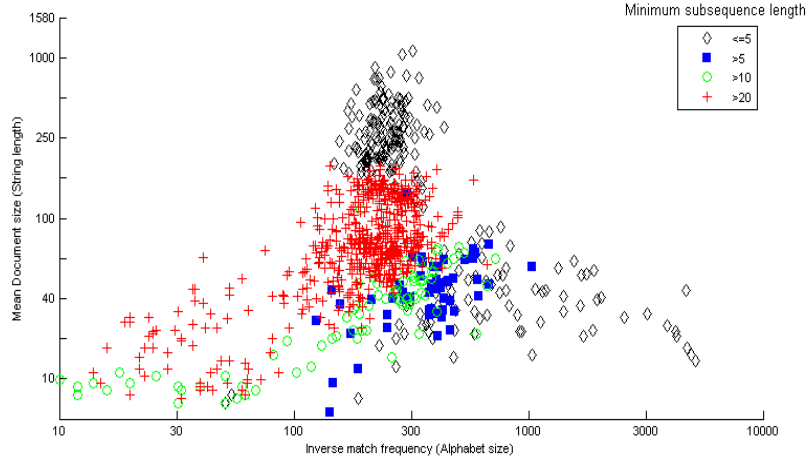
the value of the subsequence length that makes Geometric faster than Sparse depends on the document size except large sizes which fall in the first case.

### 5.3 Discussion of the experiment results

In step with the results of the two experiments, it is easy to see that the algorithms behave essentially in the same way both on synthetically generated data and newswire article data. These results reveal that our approach outperforms for large alphabet size except for very small strings. Moreover, regarding to the Sparse, Geometric is competitive for long strings.

We can argue this as follows: first, the alphabet size and the string length affect substantially the kernel matrix form. Large alphabets can reduce potentially the partially matching subsequences especially on long strings, giving rise to sparse matrix form. Consequently, great number of data stored in the kernel matrix do not contribute to the result. In the other cases, for dense matrix, our approach can be worse than Dynamic by at most  $\text{Log}|L|$  factor.

On the other hand, The complexities of Geometric and Sparse differ only by the factors  $\text{Log}|L|$  and  $\text{Log } n$ . The inverse dependency of  $|L|$  and  $|\Sigma|$  goes in favor of our approach. Also, the comparisons conducted on our datasets give evidence that for long strings  $|L| \ll n$ , remembering that the size of the match list decrease while the SSK execution progress. Moreover, to answer orthogonal range queries, Sparse invoke one dimensional range query multiple times. Whereas, Geometric mark good scores by using orthogonal range queries in conjunction with the fractional cascading and exploit our extension of the LRT data structure to get directly the sum within a range.



**Fig. 11.** Clusters of document pairs where Geometric is faster than Sparse according to the subsequence length  $p$ .

## 6 Conclusions and further work

We have presented a novel algorithm that efficiently computes the string subsequence kernel (SSK). Our approach is refined over two phases. We started by the construction of a match list  $L(s, t)$  that contains, only, the information that contributes in the result. Thereafter, in order to compute, efficiently, the sum within a range for each entry of the match list, we have extended a layered range tree to be a layered range sum tree. The Whole task takes  $O(p|L|\log|L|)$  time and  $O(|L|\log|L|)$  space, where  $p$  is the length of the SSK and  $|L|$  is the initial size of the match list.

The reached result gives evidence of an asymptotic complexity improvement compared to that of a naive implementation of the list version  $O(p|L|^2)$ . The experiments conducted both on synthetic data and newswire article data attest that the dynamic programming approach is faster when the kernel matrix is dense. This case is achieved on long strings based on small alphabets and on short strings. Furthermore, recall that our approach and the sparse dynamic programming one are proposed in the context where the most of the entries of the kernel matrix are zero, i.e. for large-sized alphabets. In such case our approach outperforms. For long strings our approach behave better than the sparse one.

This well scaling of the proposed approach with document size and alphabet size could be useful in very tasks of machine learning on long documents as full-length research articles.

A noteworthy advantage is that our approach can be favorable if we assume that the problem is multi-dimensional. In terms of complexity, this can have influence the storage and the running time, only, by a logarithmic factor. Indeed,

the layered range sum tree needs  $O(|L| \log^{d-1} |L|)$  storage and can compute the sum within a rectangular range in  $O(\log^{d-1} |L|)$ , in a  $d$ -dimensional space.

At the implementation level, great programming effort is supported by well-studied and ready to use computational geometry algorithms. Hence, the emphasis is shifted to a variant of string kernel computations that can be easily adapted.

Finally, it would be very interesting if the LRST can be extended to be a dynamic data structure. This can relieve us to create a new LRST at each evolution of the subsequence length. An other interesting axis consists to combine the LRST with the dynamic programming paradigm. We believe that using rectangular intersection techniques seems to be a good track, though this seems to be a non trivial task.

## References

1. Bellaouar, S., Cherroun, H., Ziadi, D.: Efficient list-based computation of the string subsequence kernel. In: LATA'14. pp. 138–148 (2014)
2. Berg, M.d., Cheong, O., Kreveld, M.v., Overmars, M.: Computational Geometry: Algorithms and Applications. Springer-Verlag TELOS, Santa Clara, CA, USA, 3rd ed. edn. (2008)
3. Chazelle, B., Guibas, L.J.: Fractional cascading: I. a data structuring technique. *Algorithmica* 1(2), 133–162 (1986)
4. Cristianini, N., Shawe-Taylor, J.: An introduction to support Vector Machines: and other kernel-based learning methods. Cambridge University Press, New York, NY, USA (2000)
5. Leslie, C., Eskin, E., Noble, W.: Mismatch String Kernels for SVM Protein Classification. In: Neural Information Processing Systems 15. pp. 1441–1448 (2003), <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.58.4737>
6. Lodhi, H., Saunders, C., Shawe-Taylor, J., Cristianini, N., Watkins, C.: Text classification using string kernels. *J. Mach. Learn. Res.* 2, 419–444 (Mar 2002), <http://dx.doi.org/10.1162/153244302760200687>
7. Rousu, J., Shawe-Taylor, J.: Efficient computation of gapped substring kernels on large alphabets. *J. Mach. Learn. Res.* 6, 1323–1344 (Dec 2005), <http://dl.acm.org/citation.cfm?id=1046920.1088717>
8. Shawe-Taylor, J., Cristianini, N.: Kernel Methods for Pattern Analysis. Cambridge University Press, New York, NY, USA (2004)