Efficient computation of gap-weighted string kernels on large alphabets

Juho Rousu* and John Shawe-Taylor†

Abstract

We present a sparse dynamic programming algorithm that, given two strings s, t, a gap penalty λ, and an integer p, computes the value of the gap-weighted length-p subsequences kernel. The algorithm works in time $O(p|M| \log \min(|s|, |t|))$, where $M = \{(i, j)|s_i = t_j\}$ is the set of matches of characters in the two sequences.

The new algorithm is empirically evaluated against a full dynamic programming approach and a trie-based algorithm on synthetic data. Based on the experiments, the full dynamic programming approach is the fastest on short strings, and on long strings if the alphabet is small. On large alphabets, the new sparse dynamic programming algorithm is the most efficient. On medium-sized alphabets the trie-based approach is best if the maximum number of allowed gaps is strongly restricted.

1 Introduction

Machine learning algorithms working on sequence data are needed both in bioinformatics and text understanding and mining. In contrast, standard machine learning algorithms work on feature vector representation, thus requiring a feature extraction phase to map sequence data into feature vectors.

Representing these feature vectors explicitly is often problematic due to the potentially high dimensionality. Kernel methods (Vapnik, 1995; Cristianini and Shawe-Taylor, 2000) provide an efficient way of tackling the problem of dimensionality via the use of a kernel function, corresponding to the inner product of two feature vectors. With these precomputed inner products, it is possible to efficiently accomplish a variety of machine learning and data analysis tasks, e.g. classification, regression and clustering.

The family of kernel functions defined on feature vectors computed from strings, are called *string kernels* (Watkins, 2000; Haussler, 1999). These kernels are based on features corresponding to occurrences of certain kinds of subsequences in the string. There is a wide variety of string kernels depending on how the subsequences are defined: they can be contiguous or non-contiguous, they can have bounded or unbounded length, and the mismatches or gaps can be penalized in different ways.

There are three main approaches in computing string kernels efficiently. Dynamic programming approaches (Lodhi et al., 2000) are based on composing the solution from simpler subproblems, in this case, from kernel values of shorter subsequences and prefixes of the two strings. These approaches usually have time complexity of order $\Omega(p|s||t|)$ since one typically needs to compute intermediate results for each character pair s_i, t_j for each subsequence length $1 \leq l \leq p$. However, there is no extra computational cost associated when using gap penalties or mismatch costs between the characters. In trie-based approaches (Leslie and Kuang, 2002) one makes a depth-first traversal to an implicit trie data structure. The search continues along each trie path while in both of the strings there

*Department of Computer Science, Royal Holloway University of London.
†ISIS Group, School of Electronics and Computer Science, University of Southampton.

exist an occurrence of the p-gram corresponding to the trie node. This termination condition prunes the search space very efficiently if the number of gaps is restricted enough. The third approach is to build a suffix tree of one of the strings and then compute matching statistics of the other string by traversing the suffix tree to compute matching statistics (Vishwanathan and Smola, 2002). The computation of the kernel value takes a linear time. However, the approach does not deal with gapped strings.

In this paper, we concetrate on improving the time-efficiency of the dynamic programming approach to gapped string kernel computation. In particular, we present an algorithm that works in time $O(p|M|log\min(|s|,|t|))$ where $M = \{(i,j)|s_i = t_j\}$ is the set of matches of characters in the two sequences. Since the expected size of the match set M is inversely dependent on the alphabet size, the new algorithm works best on large alphabets, such as word or syllable alphabets.

2 Gap-weighted string kernel

Let $s, t \in \Sigma^*$ be two strings from a finite alphabet Σ. Denote the lengths of the strings by $m = |s|, n = |t|$ and assume without loss of generality that $n \leq m$. Given a gap penalty λ, and an integer p, the gap-weighted subsequences (GWS) kernel is

$$\kappa_{GWS}(s,t) = \sum_{u \in \Sigma^p} \phi_u^p(s)\phi_u^p(t),$$

where

$$\phi_u^p(s) = \sum_{\mathbf{i}:u=s(\mathbf{i})} \lambda^{l(\mathbf{i})}, u \in \Sigma^p$$

is the embedding to the feature space of subsequences of length p.

The GWS kernel can be efficiently computed by a dynamic programming algorithm in $O(pmn)$ time. The algorithm solves the recurrence

$$\kappa_p(k,l) = \begin{cases} \sum_{i<k,j<l} \lambda^{k-i+l-j}\kappa_{p-1}(i,j), & \text{if } s_i = t_j \\ 0, & \text{otherwise,} \end{cases}$$

for $1 \leq k \leq m, 1 \leq l \leq n$, where we use the shorthand $\kappa_p(i,j) = \kappa_p(s(1:i), t(1:j))$. The linear time-complexity is due to efficiently maintaining the sum

$$s_{q-1}(k,l) = \sum_{i<k,j<l} \lambda^{k-i+l-j}\kappa_{q-1}(i,j)$$

of the gap-weighted kernel values of matching length $q-1$ subsequences in the prefixes $s_1 \ldots s_i, i < k$ and $t_1 \ldots t_j, j < l$. This is accomplished via the relationship

$$s_p(k,l) = s_p(k,l-1)\lambda + s_p(k-1,l)\lambda - s_p(k-1,l-1)\lambda^2 + \kappa_p(k,l), \qquad (1)$$

Note that it is necessary to substract the term $s_p(k-1,l-1)\lambda^2$, otherwise it will be counted twice by the first two terms.

Despite its relatively low time-complexity, the dynamic programming algorithm makes unnecessary computations: the value $s_{p-1}(k,l)$ is required only when $s_k = t_l$, but using (1) requires all values $s_{p-1}(i,j), i \leq k, j \leq l$ to be computed.

In the following we present an algorithm that avoids these unnecessary computations. The algorithm works in time $O(p|M|\log n)$, where $M = \{(i,j)|s_i = t_j\}$ is the set of matches of characters in the two sequences, or the *match set*.

Our algorithm belongs to the family of *sparse dynamic programming* algorithms (Eppstein et al. 1992). These algorithms utilize the fact that most entries in the dynamic progamming matrix do not actually contribute to the results. (In our case: since they have value

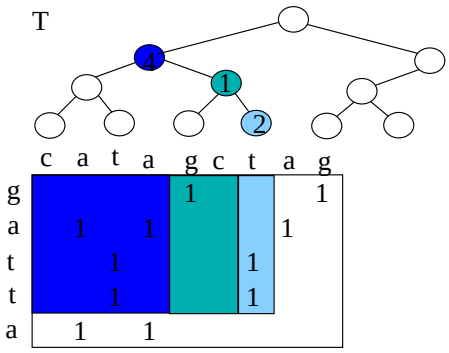

Figure 1: The value $\kappa_q(5,8)$ is obtained by making a query $T(]-\infty,7])$ to the range-sum tree containing the values $\kappa_{q-1}(i,j)$ for $1 \leq i \leq 4$. Answering the query requires summing along path from a leaf to the root. The gap penalty $\lambda = 1$ is used in the figure.

zero). The technique has been previously used, for example, to speed up transposition invariant string matching (Mäkinen, 2003) and, more close to our problem, in computing the longest-common subsequence of two strings given a fixed set of basis fragments (Baker and Ciancarlo, 1998).

As a preprocessing step to the GWS kernel computation, a match set is created out of the two strings. It is stored as a sequence of lists $M_1(1), \ldots, M_1(m)$, each list $M_1(i) = (j, \lambda^{m-i+n-j}\kappa_1(i,j))|s_i = t_j$ consisting of matching indices together with the kernel value (corresponding to the one character subsequence match at (i,j)) weighted by a (dummy) gap weight $\lambda^{m-i+n-j}$. The use of such dummy gap weighting relieves us from repeatedly scaling the kernel values as the search progresses. The structure can be computed in $O(n+m)$ time and $O(|\Sigma| + |M_1| + n)$ space. The computation involves creating in time $O(n)$, for each character $c \in \Sigma$ a list $I(c) = \{j|t_j = c\}$ of indices in the string t that contain the character c. To create a match list $M_1(i)$ then involves copying the indices in $I(s_i)$ to $M_1(i)$ and storing the corresponding weighted kernel values with the the indices.

The main algorithm (Figure 2) computes the GWS kernel κ_p by incrementally working out the kernels $\kappa_2, \ldots, \kappa_p$ of shorter subsequences. The processing of subsequence length q entails making one pass through the match sets $M_{q-1}(i)$ of the previous level in an increasing order of i. For each match (k,l) in the match set, the value

$$\lambda^{m-k+n-l}\kappa_q(k,l) = \lambda^2 \sum_{i<k, j<l} \lambda^{m-i+n-j}\kappa_{q-1}(i,j) \qquad (2)$$

is computed by making an $O(\log n)$ time query to a dynamically maintained range-sum tree data structure (see below) to retrieve the sum on the right-hand side. The computed kernel value is then inserted to the match set $M_q(k)$ for use in the subsequent computation. When processing the final level p, the values are rescaled to obtain the correct kernel values.

2.1 Range-sum tree

For a set $S = \{(j, value_j)\}$ of key-value pairs, the range-sum tree (Figure 1) is a binary tree where the nodes contain (j, sum_j) pairs where $sum_j = \sum_{j'<h\leq j} value_h$ where j' is the nearest ancestor node of j that contains j in its right subtree. The root of the tree is labeled with $2^{\log\lceil n\rceil}$. The left child of the node j is the node $j - j/2$ and the right child is the node $j + j/2$. The leaves of the tree are labeled with the odd indices $j = 1, 3, \ldots$. Note that the height of the tree by this construction will be $O(\log n)$.

Input: A matchset $M_1 = (M_1(1), \ldots, M_1(m))$ in lists $M_1(i) = (j_1, \lambda^{K-i+L-j_1}\kappa_1(i, j_1)), \ldots, (j_{r_i}, \lambda^{-i-j_{r_i}}\kappa_1(i, j_{r_i}))$, a gap weight $0 < \lambda \leq 1$, and subsequence length p.

Output: Kernel value $\kappa_p(s, t)$

Method:

function κ = compute-GWS(M, p, λ)
for $q = 2 : (p-1)$
  $Tree = \{\}$; % initially the range-sum tree is empty
  for $i = p : m$
    % compute the kappa values for the next level
    $M_q(i) = \{\}$;
    for $h = 1 : |M_{q-1}(i, :)|$
      $(j_h, \kappa_h) = M_{q-1}(i, h)$;
      $\kappa = Tree(]-\infty, j_h - 1])$; % make range query
      if $\kappa > 0$
        $appendlist(M_q(i), (j_h, \kappa))$;
      end
    end
    % update the range sum tree
    for $h = 1 : |M_{q-1}(i)|$
      $(j_h, \kappa_h) = \kappa_{q-1}(i, j)$;
      $updateTree(Tree, (j_h, \kappa_h))$;
    end
  end
end
% compute the kappa values for the final level
$\kappa = 0$;
for $i = p : (m-1)$
  for $h = 1 : |M_{p-1}(i, :)|$
    $(j_h, \kappa_h) = M_{p-1}(i, h)$;
    if $j_h < n$
      $\kappa = \kappa + \kappa_h \lambda^{i+j_h}$; % rescale
    end
  end
end

Figure 2: The algorithm for computing the gap-weighted subsequences kernel for two strings s and t.

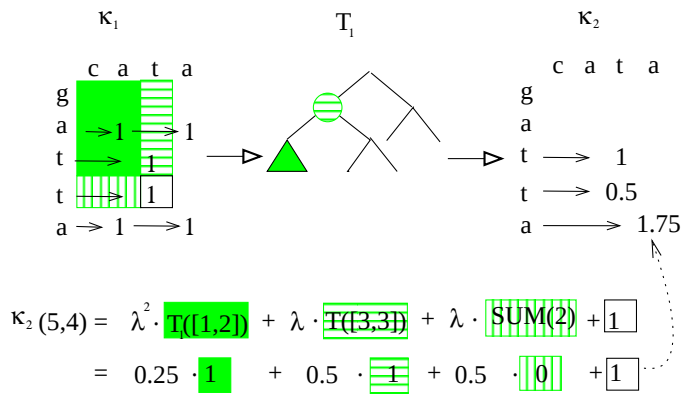

Figure 3: Computation of the gap-number weighted subsequence kernel using the range-sum tree. The solid colored and shaded rectangles illustrate the decomposition in the recurrence.

The range sum can be used to efficiently return the sum of values within an interval $[\min_h\{key_h\}, key]$ in $O(\log n)$ time: it suffices to traverse the path from the node containing key to the root and sum over the left subtrees passed. Updating the value of a node requires time $O(\log n)$ as well, since the value fields of parents that contain key in their left subtree need to be updated.

In our sparse dynamic programming algorithm, a range-sum tree is maintained so that, when computing (2) at (k, l), the values in the tree satisfy $value_j = \sum_{i<k} \lambda^{m-i+n-j} \kappa_{q-1}(i, j)$ and thus (2) is simply answered by making a single interval query $Tree([1, l-1])$ to the range-sum tree (Figure 1)).

After constructing the list $M_q(k)$, the values in the list $M_{q-1}(k)$ are inserted to the range-sum tree, so that when processing the next list $M_{q-1}(k+1)$, the ranges correctly include all values in the already processed lists.

The queries and updates amount to $O(\log n)$ per item in the match list so the time complexity to process level q is $O(|M_q| \log n)$. Since we have $|M_1| \geq |M_2| \geq \cdots \geq |M_p|$, the total time complexity will be $O(p|M_1| \log n)$. On random strings $|M_1| \approx |s||t|/|\Sigma|$. Hence, the sparse algorithm is likely to excel when $\log n/|\Sigma|$ is small, which we verify in the experiments presented in Section 3.

2.2 Weighting by the number of gaps

In some applications the actual length of the gap may not be important. It is straightforward to modify the algorithm to penalize the number of gaps in the subsequence, instead of the total gap length. The kernel to be computed is

$$\kappa_{Gap\#}(s, t) = \sum_{u \in \Sigma^p} \phi_u^p(s) \phi_u^p(t),$$

where

$$\phi_u^p(s) = \sum_{\mathbf{i}: u = s(\mathbf{i})} \lambda^{\sum_j [i_{j+1} - i_j > 1]}, u \in \Sigma^p$$

is again an embedding to the feature space of subsequences of length p. The kernel is computed via the recurrence

$$\kappa_q(k, l) = [s_i == t_j]\Big(\sum_{i<k-1, j<l-1} \lambda^2 \kappa_{q-1}(i, j) + \sum_{i<k-1} \lambda \kappa_{q-1}(i, l-1) + \sum_{j<l-1} \lambda \kappa_{q-1}(k-1, j) + \kappa_{q-1}(k-1, l-1)\Big) \quad (3)$$

again starting from matching length one subsequences contained in the match set M_1. Figure 2.2 illustrates the decomposition in the recurrence. The first term corresponds matching occurences of the form $s_{i_1} \cdots s_{i_{q-1}} * s_k$ in s and $t_{j_1} \cdots t_{j_{q-1}} * t_l$, thus adding one gap to both the matched subsequences. The term is computed by making a a query $[1, l-2]$ to the range-sum tree containing entries in lists $M_q(1), \ldots, M_q(k-2)$, that is. The second term, corresponding to matches where a gap is inserted to occurrences in s only, is computed by making the query $[l-1, l-1]$, that is, retrieving the range sum in the $l-1$'th column. The two final terms are incrementally computed by traversing the lists $M_{q-1}(k-1)$ side by side with the list $M_{q-1}(k)$. The last term will be non-zero only if $s_{k-1} = t_{l-1}$.

After computing $\kappa_q(k, l)$ for all l, the range-sum tree is updated with the values in the list $M_p(k-1)$. Thus, as compared to the gap-weighted kernel, the tree update is delayed by one iteration.

2.3 Implementation details

Range sum tree storage. In our MATLAB implementation, the range sum tree is implictly stored in a weight vector w storing the sum of the left subtree of each node $1 \leq j \leq n$. To speed up computations we also precompute in separate tables the nodes that need to be visited when querying or updating the range sum tree. For example, in the situation depicted in Figure (1) the precomputed *query path* for will include the nodes 7, 6 and 4. The corresponding *update path* will contain the nodes 7 and 8 only.

Avoiding numerical overflow. The algorithm in Figure 2 stores the items in the form $\lambda^{m-i+n-j}\kappa_{q-1}(i, j)$ and rescales them when computing the level p. This approach suffers from the potential of numerical overflow when handling long strings. In order to avoid that, we divide the index plane into reactangles of height and width sufficiently small (depending on the value λ) such that within a rectangle $[x', x''] \times [y', y'']$ the values are stored in the form $\lambda^{x''+y''-i-j}$. The handling of the boundaries of the rectangles causes a small additive overhead to the time complexity.

3 Empirical evaluation

3.1 Experimental setup

We compared the time consumption of the following gapped string kernel algorithms, all implemented in Matlab:

DYNPROG The full dynamic programming approach.

SPARSE The sparse dynamic programming approach presented in the previous section.

TRIE Trie-based computation of the *restricted* gap weighted subsequence kernel. The algorithm composes an implicit trie structure of matching subsequences much like Leslie and Kuang (2003) but with some important differences. In a trie-node corresponding to subsequence $u_1 \cdots u_q$ the algorithm keeps track of all *alive* indices $A_s(u, g) = \{j | \exists \mathbf{i} : s(\mathbf{i}) = u, i_q = j, i_q - i_1 + 1 - q = g\}$ where $s_i = u_q$ is the last character of an occurrence of u in s with g gaps. Similarly for t the set $A_t(u, g)$ is maintained.

To expand the node u, for an alive index $i \in A_s(u, g)$ and all $g' \leq MAXGAP - g$ the algorithm puts the indices $i + 1 + g'$ into $A_s(us_{i+1+g'}, g + g')$. The sets $A_t(ut_{i+1+g'}, g + g')$ are constructed the same way. The search is continued only for nodes uc that have non-empty $\cup_g A_s(u_c, g) \cap \cup_g A_t(uc, g)$ that is, there is at least one occurrence of uc in both s and t, with some number of gaps. When a node u in

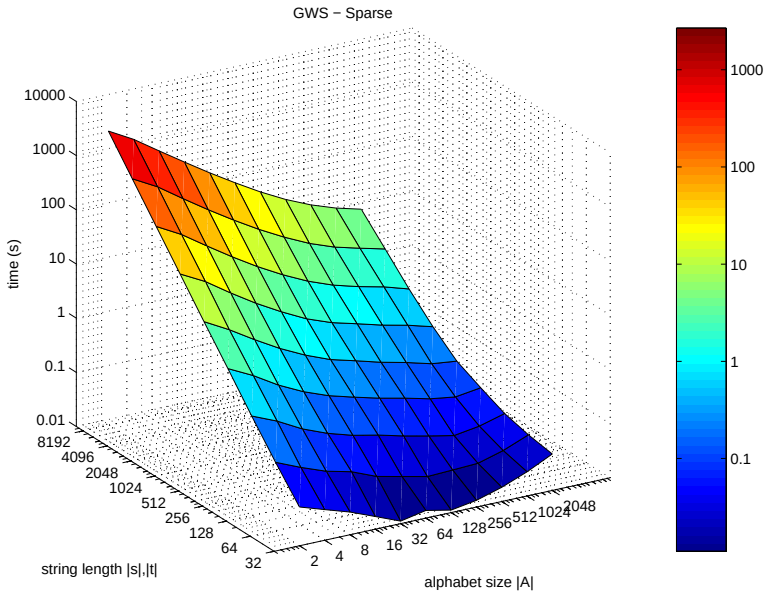

Figure 4: The time consumption of the sparse dynamic programming algorithm as a function of alphabet size and string length. The subsequence length $p = 10$ was used. Note the logarithmic scale.

depth p is encountered one easily obtains the relevant terms in the kernel via the sum $\sum_{g_s,g_t} \lambda^{g_s+p}|A_s(u,g_s)| \cdot \lambda^{g_t+p}|A_t(u,g_t)|$.

Note that the above approach differs from the algorithm by Leslie and Kuang (2003) in two respects: First, the strings s and t are not broken into frames before the search but the algorithm maintains pointers to the strings to keep track of the subsequence occurences. Second, the algorithm keeps track of the number of gaps in the occurrences during the search which relieves us from embarking on dynamic programming search in the leaves.

Note that, differently from TRIE, DYNPROG and SPARSE place no hard restriction on the gap length but softly penalize the increase in gap length.

The data was randomly generated strings, with varying length and alphabet sizes. The tests were run on a 3GHZ Pentium 4 processor with 1.5GB main memory.

3.2 Results

In our first test we tested the time-comsumption of the algorithm SPARSE. In Figure 4 the time-consumption of SPARSE algorithm is depicted. The inverse dependency of the time-consumption on the alphabet size is clearly visible. Also, the larger the alphabet, the slower the time-consumption increases when the string length is increased.

In Figure 5 the relative time-consumption of the sparse approach to the full dynamic programming approach is shown. With small alphabets and short strings DYNPROG is faster than SPARSE. With long strings and large alphabets the roles reverse.

In our second test we compare the speed of TRIE algorithm to the SPARSE algorithm. Figure 6 depicts the relative time consumption as a function of alphabet size and gap length. Subsequence length of 10 and string length of 512 were used. Since SPARSE does not place any restriction to the gap length, in the comparison only the time taken by TRIE actually varied when the maximum number of gaps was varied.

The figure shows that TRIE algorithm gets very significantly slower then SPARSE when more gaps are allowed especially so on small alphabets. TRIE is faster than SPARSE

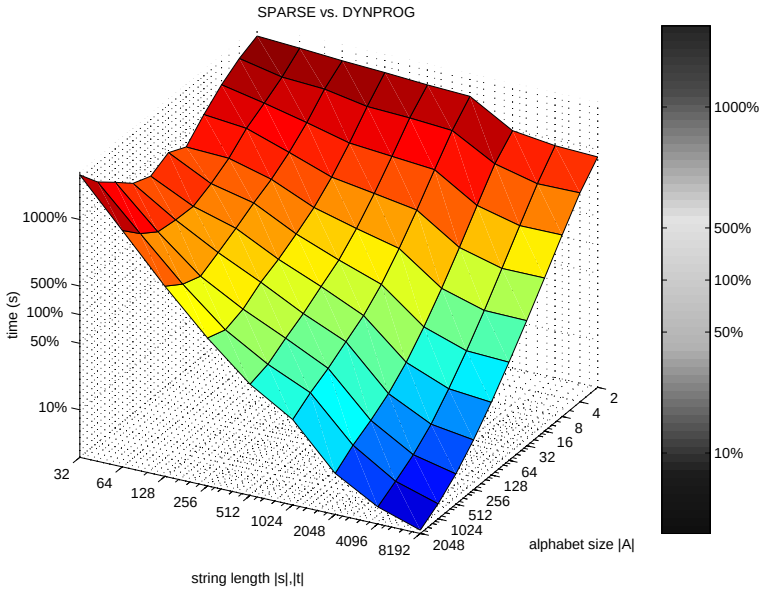

Figure 5: The relative time consumption of the sparse dynamic programming algorithm as compared to the full dynamic programming algorithm, as a function of alphabet size and string length. The subsequence length $p = 10$ was used. Note the logarithmic scale.

only when the number of gaps is restricted to 2 or below. On very large alphabets even disallowing gaps does not bring TRIE below the time consumption of SPARSE.

The fastest algorithm as a function of string length and alphabet size is depicted in Figure 7, with different settings for the subsequence length (p) and the maximum number of gaps allowed in the TRIE algorithm. DYNPROG is the fastest method on short strings independent on the alphabet size and the subsequence length (a-d). If no gaps are allowed, TRIE is competitive on small to medium-size alphabets and long strings (a). When the subsequence length is increased, TRIE is faster than SPARSE even on large alphabets (c). The situation changes when gaps are allowed in TRIE algorithm: then SPARSE is the best method on large alphabets, and DYNPROG on small alphabets, TRIE excelling on medium-sized alphabets if long subsequences are searched for (d).

4 Discussion

For algorithm development, an open question is whether the time-complexity of the sparse dynamic programming approach could still be reduced from $O(p|M|\log n)$. The literature on geometric range searching does not a direct route forward: no index structures are known for one or two-dimensional range queries that can be maintained in less than amortized $O(\log n)$ time (Agarwal and Erickson, 1999), even when taking advantage of the fact that the points are situated on an integer grid (Overmars, 1988, Alstrup et al. 2000). On the other hand the best lower bounds are of order $\Omega(\log\log n)$ per query (Chazelle, 1995).

Based on the presented experiments, the full dynamic programming approach (DYNPROG) is the fastest method on short strings. On longer strings, the best algorithm depends on other parameters: if the alphabet is large (> 1000) the new sparse dynamic programming approach (SPARSE) is the fastest method, if the alphabet is small (< 10) DYNPROG is the best method. On medium-sized alphabets, the trie-based approach is competitive if the number of gaps can be strongly restricted.

The observed relative performance can be explained as follows. When the alphabet size is small, allowing more gaps rapidly expands the number of partially matching sub-

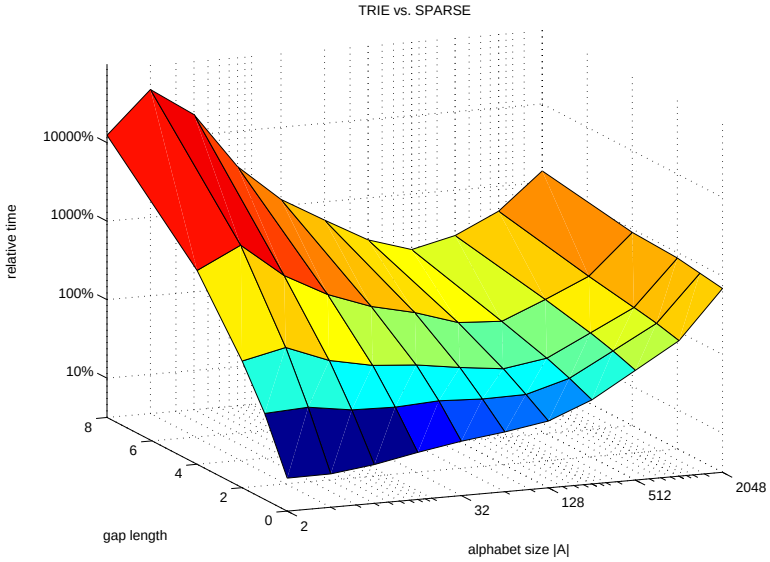

Figure 6: The relative time consumption of the TRIE algorithm as compared to the SPARSE algorithm, as a function of alphabet size and gap length. The subsequence length $p = 10$ and string length $|s|, |t| = 512$ was used. Note the logarithmic scale.

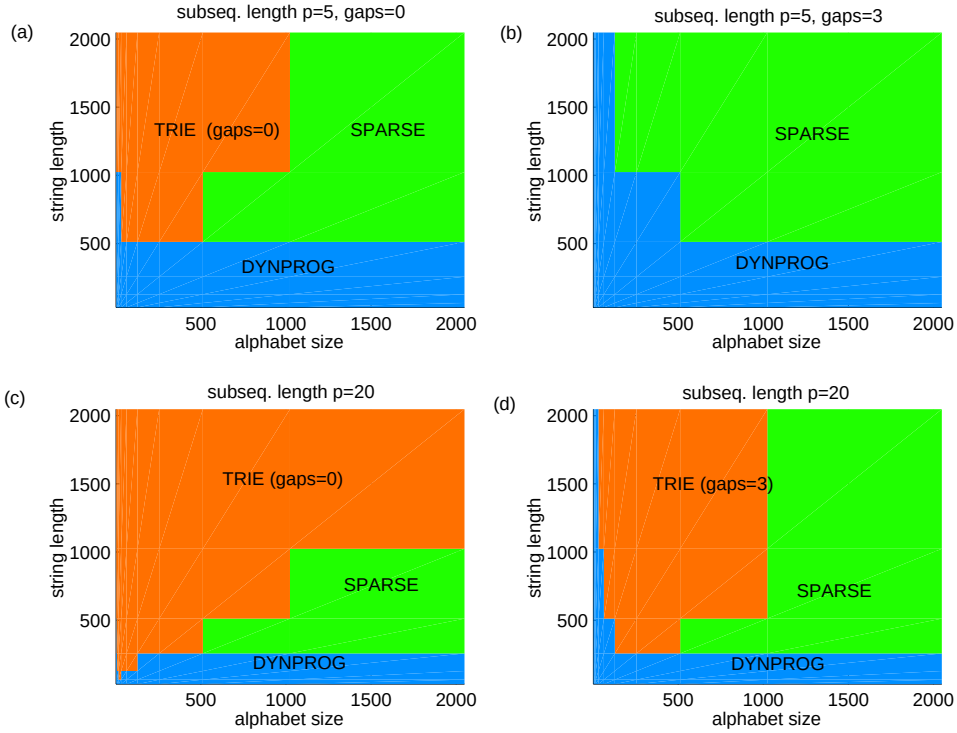

Figure 7: The fastest algorithm as a function of string length and alphabet size, with different subsequence lengths (p) and upper bounds for the number of gaps in the TRIE algorithm.

sequences. Since TRIE explicitly keeps track of them, its time-consumption increases. SPARSE also suffers on small alphabets. However, it can never be worse than DYNPROG by more than a $\log n$ factor. On large alphabets, TRIE has an overhead of keeping track of all subtrees that may or may not need to be expanded. The improving performance of TRIE by increasing subsequence length is also easy to explain: the trie becomes the sparser the deeper the search level. Thus deepening the search is relatively cheap.

The result suggest that the sparse dynamic programming approach could be useful in text mining applications when using syllable or word alphabets, that easily have size over a thousand. Such alphabets have shown to be useful in document classification tasks (Saunders et al. 2002). A future research topic is to evaluate the performance of the new algorithm in such tasks.

5 References

P. Agarwal and J. Erickson. Geometric range searching and its relatives. Contemporary Mathematics 23: Advances in Discrete and Computational Geometry, 1999, pp. 1–56,

S. Alstrup, G.S. Brodal, T. Rauhe. New Data Structures for Orthogonal Range Searching. Proc. of 41st Annual Symposium on Foundations of Computer Science (FOCS 2000), 2000, pp. 198-207.

B. S. Baker and R. Giancarlo. Longest common subsequence from fragments via sparse dynamic programming. In European Symposium on Algorithms, Venice, Italy, 1998, pp. 79–90

B. Chazelle. Lower bounds for off-line range searching, Proc. 27th Annual ACM Symposium on Theory of Computing, 1995, pp. 733–740.

N. Cristianini and J. Shawe-Taylor. *An introduction to Support Vector Machines*. Cambridge University Press, CAmbridge, UK, 2000

D. Eppstein, Z. Galil, R. Giancarlo, and G. F. Italiano. Sparse dynamic programming I: linear cost functions. J. of the ACM 39, 3 (1992), pp. 519–545

D. Haussler. Convolution kernels on discrete structures. Technical report, UC Santa Cruz, 1999

C. Leslie and R. Kuang. Fast Kernels for Inexact String Matching. Proc. 16th Conference on Computational Learning Theory and 7th Kernel Workshop, COLT/Kernel'2003. Lecture Notes in Computer Science 2777 (2003), pp. 114 - 128

H. Lodhi, C. Saunders, J. Shawe-Taylor, N. Cristianini, C. Watkins. Text Classification using String Kernels. Journal of Machine Learning Research 2 (2002), pp. 419–444

V. Mäkinen. Parameterized Approximate String Matching and Local-Similarity-Based Point-Pattern Matching. Report A-2003-6. Department of Computer Science, University of Helsinki, 2003

M. Overmars. Efficient data structures for range searching on a grid. J. Algorithms 9 (1988), pp. 254–275

C. Saunders, H. Tschach, J. Shawe-Taylor. Syllables and other String Kernel Extensions. Proc. 19th International Conference on Machine Learning. Morgan Kaufmann, 2002, pp. 530–537

V. Vapnik. The Nature of Statistical Learning Theory. Springer Verlag, New York, 1995.

S. Vishwanathan and A. Smola. Fast Kernels for String and Tree Matching. Advances in Neural Information Processing Systems 15, 2003, pp. 569–576

C. Watkins. Dynamic alignment kernels. In A.J. Smola, Bartlett, P.L., Schlkopf, B., and D. Schuurmans, eds., Advances in Large Margin Classifiers, Cambridge, MA, 2000, MIT Press, pp. 39–50