Faster Filters for Approximate String Matching

Juha Kärkkäinen* Joong Chae Na*

Abstract

We introduce a new filtering method for approximate string matching called the suffix filter. It has some similarity with well-known filtration algorithms, which we call factor filters, and which are among the best practical algorithms for approximate string matching using a text index. Suffix filters are stronger, i.e., produce fewer false matches than factor filters. We demonstrate experimentally that suffix filters are faster in practice, too.

1 Introduction

Given a text string of length n, a pattern string of length m, and a distance k, the *approximate string matching problem* is to find all substrings of the text that are within a distance k of the pattern. The most common distance measure is the *Levenshtein* or *edit distance*, the minimum number of single character insertions, deletions and replacements required to transform one string into the other. This is a widely studied problem with numerous applications in text processing, computational biology, and other areas involving sequential data. An extensive survey is given in [12].

The survey does not cover the *indexed* version of the problem, which allows the use of a precomputed index of the text. This is an active research area (see [3, 5]) and a wide range of solutions have been proposed (see [9] for a brief survey). However, most approaches are not practical for many applications because the index size is superlinear (e.g., [6, 9]) or because the search time is too large for long patterns and high distance limit k (e.g., [4, 3]).

The best practical methods for high n, m and k are based on filtering. A *filter* is an algorithm that quickly discards large parts of the text using a *filter criterion*, leaving the interesting parts, the *potential match areas*, to be checked with a proper (non-indexed) approximate string matching algorithm. These two phases are the *filtration phase* and the *verification phase*. A filter is *lossless* if it never discards a true match. Filter algorithms for indexed string matching are surveyed in [14] though some new filtration methods have been developed since its publication, most notably methods based on gapped q-grams [2] or spaced seeds [8].

A simple but effective family of filters, which we call the *factor filters*, is based on a factorization of the pattern. A *factorization* of a string S is a sequence of strings (factors) whose concatenation is S. In the simplest case, there are $k+1$ factors and the occurrence of one in the text signals a potential match. The losslessness of the filter is shown by the following lemma.

LEMMA 1.1. *Let $A = A_0A_1 \cdots A_k$ be a string that is the concatenation of $k+1$ non-empty factors A_i. If a string B is within edit distance k from A, then at least one of the factors A_i is a factor of B.*

The lemma follows easily from the fact that at least one edit operation is needed for each factor of A that is not a factor of B. An important generalization allows fewer factors with an *approximate* occurrence of a factor signaling a potential match. Some of the best methods for indexed approximate string matching are based on factor filters [10, 13].

We present a new family of filters, the *suffix filters*, that generalizes and strengthens the factor filters. For example, the suffix filter version of Lemma 1.1 is this:

LEMMA 1.2. *Let $A = A_0A_1 \cdots A_k$ be a string that is the concatenation of $k+1$ non-empty factors A_i. If a string B is within edit distance k from A, then there is at least one suffix $A' = A_iA_{i+1} \cdots A_k$ and a suffix B' of B such that, for all $j \in [i,k]$, a prefix of B' is within edit distance $j-i$ of $A_i \cdots A_j$.*

The condition in the lemma allows $k-i$ errors between A' and B' but has tighter limits for their prefixes, i.e., the errors cannot all be in the beginning. We call this a *strong match*. Other variants of suffix filters involve strong matches of suffixes, too.

It is easy to see that Lemma 1.1 is a corollary of Lemma 1.2 (consider the cases where $j = i$). Thus, Lemma 1.2 gives a stronger filter criterion. Other factor filters have corresponding stronger suffix filters, too. A stronger filter criterion leads to fewer potential matches and faster verification phase in approximate string matching. On the other hand, the new filter criteria are more complicated, which could make the

*Department of Computer Science, P.O. Box 68 (Gustaf Hällströmin katu 2b), FI-00014 University of Helsinki, Finland, {Juha.Karkkainen,Joong.Chae.Na}@cs.helsinki.fi

filtration phase slower and more difficult to implement. However, we show that the suffix filters lead to practical and fast algorithms. As the starting point, we took an implementation of an index for approximate string matching based on factor filters described in [13]. Only a relatively straightforward and isolated modification was needed to support suffix filters. In particular, the index structure is not changed at all. We show with experiments that suffix filters lead to significantly faster total search times than factor filters.

2 Filter Criteria

The edit distance (also known as Levenshtein distance) of two strings A and B, denoted by $ed(A, B)$, is the minimum number of single character insertions, deletions and substitutions required to transform A into B. A *filter criterion* for edit distance is a predicate on two strings A and B that correlates with $ed(A, B)$ being small. A *k-lossless filter criterion* is true for every A and B with $ed(A, B) \leq k$.

The *factorization* of a string A is a sequence of strings (factors) $A_i \in \Sigma^*$, $i \in [0, s)$, whose concatenation $A_0A_1 \cdots A_{s-1}$ is A. The following is a well-known property of edit distance.

PROPERTY 2.1. *Let A and B be two strings, and let $A_0A_1 \cdots A_{s-1}$ be a factorization of A. Then, there exists a factorization $B_0B_1 \cdots B_{s-1}$ of B such that $ed(A, B) = \sum_{i\in[0,s)} ed(A_i, B_i)$.*

We call $B_0B_1 \cdots B_{s-1}$ an *optimal factorization* of B. An optimal factorization is not unique in general.

The filters we will describe are defined by a string A, a factorization $A_0A_1 \cdots A_{s-1}$ of A and s non-negative numbers t_i, $i \in [0, s)$. We also define $t = \sum_{i\in[0,s)} t_i$. The filters are based on a condition that a string B has to satisfy to pass the filter and be considered a potential match for A. The filters rely on the existence of an optimal factorization of B but the optimal factorization does not need to be known to test the condition. Here we are interested in the case where A is the pattern and B is a substring of the text but other uses of the filters are possible, see [14].

2.1 Factor filters Before presenting the new suffix filters, we consider the well-known factor filters. The factor filters are based on the following lemma.[1]

LEMMA 2.1. ([14]) *If $ed(A, B) < t$, there exists $i \in [0, s)$ such that $ed(A_i, B_i) < t_i$.*

We get a k-lossless filter criterion if we choose the t_i's so that $t > k$; the optimal choice giving the strongest filter is $t = k + 1$. Based on the lemma, if $ed(A, B) \leq k < t$, B must have a factor whose edit distance to some A_i is less than t_i.

In the filtration phase, the text is searched for approximate occurrences of the factors using a text index. In the verification phase, the area around each occurrence of a factor is searched for an occurrence of the full pattern using a non-indexed algorithm. Searching for factors using an index is generally much faster than searching for the whole pattern because the factors are shorter and the distance limits smaller.

The factorization of A and the t_i's have to be chosen carefully for good performance [13]. In general, the best choice is to make the factor sizes and distance limits as even as possible, i.e., the factor sizes are $\lfloor m/s \rfloor$ and $\lceil m/s \rceil$, and the distance limits t_i are $\lfloor t/s \rfloor$ and $\lceil t/s \rceil$.[2] The remaining free parameter is s, the number of factors. It is a critical parameter, a difference of just one can make a big difference in performance. An extensive analysis is provided in [13], but with the conclusion that the optimal choice can be found only experimentally.

2.2 Suffix filters For $S \in \{A, B\}$ and $0 \leq i \leq j \leq s$, let $S[i, j)$ denote the string $S_iS_{i+1} \cdots S_{j-1}$. In particular, $S[i, s)$ is a suffix of S. We say that A and B *match* on interval $[i, j)$ if $ed(A[i, j), B[i, j)) < \sum_{h\in[i,j)} t_h$, and *strongly match* on $[i, j)$ if they match on every interval $[i, j')$, $j' \in (i, j]$, i.e., on every non-empty prefix of $[i, j)$.

The suffix filters are based on the following result.

LEMMA 2.2. *If $ed(A, B) < t$, there exists $i \in [0, s)$ such that A and B strongly match on $[i, s)$.*

Proof. Let $[0, i)$ be the longest prefix interval, on which A and B do *not* match, i.e., $ed(A[0, i), B[0, i)) \geq \sum_{h\in[0,i)} t_h$. It is well defined as $i = 0$ always satisfies the condition. Then, A and B strongly match on $[i, s)$. To show this, assume the opposite, i.e., that there exists $j \in (i, s]$ such that A and B do not match on $[i, j)$. But then A and B do not match on $[0, j) = [0, i) \cup [i, j)$, which contradicts $[0, i)$ being maximal. □

Example. In the situation of the following table, A and B strongly match on $[1, 5)$ but not on any other suffix $[i, 5)$.

i	0	1	2	3	4
t_i	1	1	2	1	1
$ed(A_i, B_i)$	1	0	1	2	1

[1] Our t_i corresponds to $k_i + 1$ in Lemma 2 of [14].

[2] In [13], the distance limits t_i are all $\lceil t/s \rceil$. Our modification improved running times significantly.

The filter algorithm is identical to factor filters except instead of searching for separate factors, the filtration phase will search for suffixes of the pattern satisfying the strong match condition. That is, it searches for each suffix $A[i, s)$ with less than $\sum_{j\in[i,s)} t_j$ errors with additional restrictions on how the errors are distributed.

To get an intuition on suffix filters, it helps to compare it to factor filters and to searching for the pattern directly using an index. Consider the (sub)problems of using the index to find the occurrences of

1. pattern A with less than t errors,
2. factor A_0 with less than t_0 errors, and
3. suffix $A[0, s)$ under the strong match condition.

In the first case, t errors need to be found to eliminate a candidate. In the other two cases, only t_0 errors within the first factor is sufficient. This can make a huge difference when searching using an index as is explained in Section 3. If those t_0 errors within the first factor are not found, the factor filter lets the candidate pass, but the suffix filter continues the search on the index and has further chances for elimination. Thus, the suffix filter uses the index more effectively.

We have experimentally found that a suffix filter is faster than a factor filter for the same parameters (factorization of A and the t_is) even if those parameters are optimized for the factor filter. However, significant further improvement is achieved with the parameters optimized for suffix filters.

The basic principle in optimizing the parameters is to try to make all the suffixes close to equal in behaviour. This leads to the following three rules:

1. $s = k + 1$ and $t_i = 1$ for all i. For an explanation, see below.
2. The last factor A_k is larger than others. This compensates for the fact that the shortest suffixes produce more potential matches. We determine the last factor size ℓ experimentally.
3. Other factor boundaries are distributed evenly so that each segment of r consecutive factors has the size $\lfloor r(m-\ell)/(s-1)\rfloor$ or $\lceil r(m-\ell)/(s-1)\rceil$. The aim is to make the search times of longer suffixes as even as possible.

As an example, for $m = 40$ and $k = 12$ the rules lead to 13 factors, each with $t_i = 1$, with the following sizes:
3 3 3 3 3 4 3 3 3 3 3 3 6.
This is the best performing filter in our experiments.

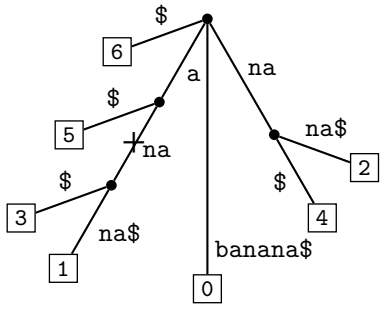
To explain the first rule, let us consider the effect of splitting a factor into two. Let A_i be a factor with error limit $t_i > 1$, which is split into two factors A'_i and A''_i. If we set, $t'_i = t_i$ and $t''_i = 0$, the resulting filter is identical to the original: The suffix starting with A''_i cannot have a strong match because $t''_i = 0$, and other suffixes are not affected. Thus, we can always make $s = k + 1$ by splitting factors or joining factors with distance limit 0 with their predecessor. The principle of making suffixes equal then favors even distribution of limits, i.e, setting $t_i = 1$ for all i.

3 Implementation

We have implemented an indexed approximate string matching algorithm based on suffix filters. The implementation is a modification of the one in [13], which is based on the factor filters. Here we describe only the main ideas and differences to [13].

3.1 Pattern Matching over Suffix Tree/Array

The suffix tree is a widely used full-text index structure that has linear size and can be constructed in linear time [15]. The key feature of a suffix tree is that every substring of the text has a unique position in the suffix tree: the path from the root to that position spells the string and the leaves under that position tell all the occurrences of the substring in the text. The (exact) occurrences of a string P are easily found by following the path starting from the root and spelling P to the position that represents P. Figure 1 gives an example.



6	$
5	a$
3	ana$
1	anana$
0	banana$
4	na$
2	nana$

Figure 1: Suffix tree and suffix array for the text `banana$`. The suffix tree position and the suffix array range representing the string `an` are marked.

A more complex pattern, such as a regular expression, can be searched, too, if there is an automaton that recognizes strings that match the pattern. The automa-

ton should read a string one character at a time and at some point announce a *hit* or a *miss*. A hit means that the string matches the pattern and a miss means that no continuation of the string can lead to a match. The next section describes such automata for approximate string matching. All the occurrences of the pattern in the text are found by following every path in the suffix tree starting from the root until the automaton reports a hit or a miss. By storing intermediate states of the automaton to allow backtracking, the whole search can be performed as a limited depth-first search of the tree.

Typically, a suffix tree is dense near the root but sparse at deeper levels, which is a favorable situation for searching strong matches. Near the root, the strong match condition cuts most branches short. The cumulative distance limit is less restrictive at deeper levels but there are also fewer branches to take there.

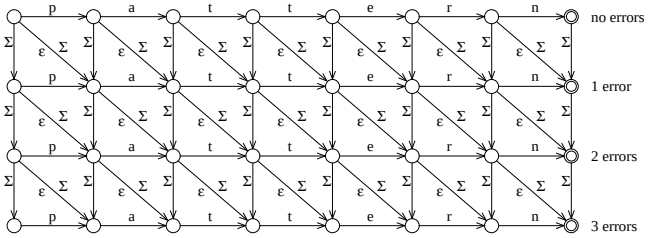
The implementation actually uses a *suffix array*, a sorted array of the suffixes, to simulate the suffix tree. The simulation represents each suffix tree position with a suffix array interval (see Figure 1). The suffix array is substantially smaller than the suffix tree and, while the simulation is theoretically slower by a factor of $\mathcal{O}(\log n)$, it is faster in practice [13].

3.2 Automata To recognize approximate factors, the implementation in [13] uses a non-deterministic finite automaton (NFA), a well-known method for approximate string matching [16, 1]. Figure 2 (a) shows the NFA for recognizing the string "pattern" with at most three errors. Each row denotes the number of errors seen. Each column represents matching a pattern prefix. For example, the state in the second row and the fourth column is active if the string read so far matches "pat" with one error. Initially only the diagonal starting from the top left corner is active. The automaton announces a hit when one of the rightmost states is activated and a miss when no state is active. We refer to [13] for more details.

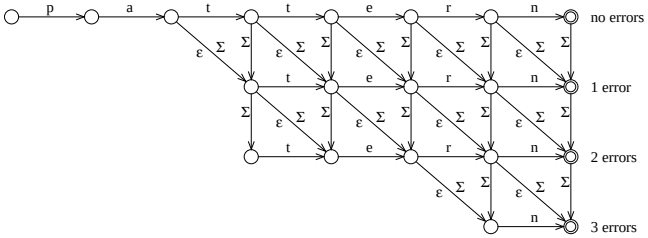
Suffix filters require recognition of strong matches, which do not allow all errors to occur in the beginning. The corresponding NFA, which we call the *staircase NFA*, is obtained from the standard NFA by eliminating states that violate the strong match conditions. For example, consider the following factorization of the string "pattern":

factor	pa	tte	rn
t_i	1	2	1

Figure 2 (b) shows the staircase NFA for the first suffix of this factorization. Because the prefix "pa" allows only exact matching, we eliminate states in the ith row and the jth column for $i \geq 2$ and $j \leq 3$. Because the prefix



(a) An NFA for the pattern "pattern" with 3 errors.



(b) A staircase NFA for suffix filters.

Figure 2: NFAs for recognizing approximate patterns.

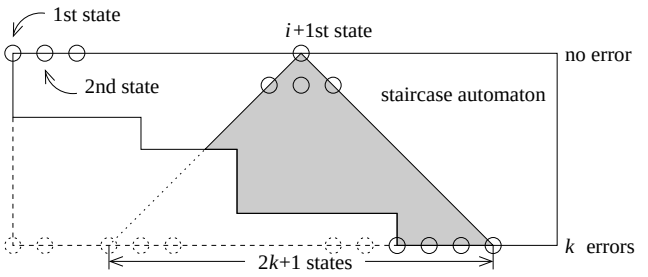


Figure 3: Active area after reading i characters. Only states within the shaded triangle can be active.

"patte" allows at most 2 errors, we eliminate states in the ith row and the jth column for $i \geq 4$ and $j \leq 6$.

We use some heuristics to reduce the time for simulating an NFA. We do not need to update all states during the simulation of the (staircase) NFA. After reading i characters, only states within a triangle whose peak is the $i+1$st state of the first row can be active (see Figure 3). Furthermore, if the top h rows have no active states, they will stay inactive also in the future, and can be omitted. This is a significant optimization during a suffix tree search, because most search branches end in a miss, a situation with no active states.

A practical implementation of an NFA uses "bit-parallelism": the states of the NFA are mapped to bits in a computer word and are updated in parallel by

operations on the words. We (and [13]) use diagonal-wise simulation [1] for the standard NFA but row-wise simulation [16] for the staircase NFA, mainly due to ease of implementation. In the worst case, a diagonal-wise simulation step requires $\mathcal{O}(\lceil k^2/w \rceil)$ time while a row-wise simulation takes $\mathcal{O}(k\lceil k/w \rceil)$ time, where w is the word size. However, the row-wise simulation can easily take advantage of the heuristic of omitting inactive rows but the diagonal-wise simulation can not. We also tried the row-wise simulation for the standard NFA but it made the factor filters slightly slower. We implemented two simulation instances according to how many words are needed for one row of the staircase NFA: one word and more than one word. The NFA in [13] was implemented by five instances according to the lengths of row and column.

3.3 Verification We have described how approximate occurrences of factors or suffixes are found using the index. Each occurrence marks an area around it as a part of the potential match area, and the union of these areas is formed by sorting. The potential match area is then searched sequentially using Myers' bit-parallel simulation of the dynamic programming matrix [11, 7], which is one of the fastest algorithms. We implemented four simulation instances according to how many words are needed for one column of the matrix: one, two, three, and more than three words. The implementation in [13] used a slower algorithm. This change actually benefited factor filters more than suffix filters.

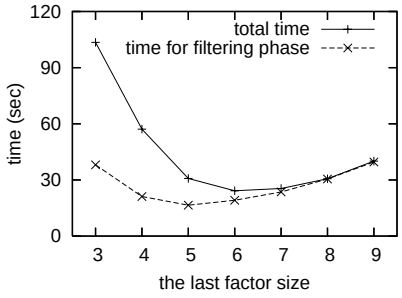
4 Experimental Results

In this section we present results from experiments testing the performance of the suffix filter algorithm described in the previous section and comparing it against the factor filter algorithm.

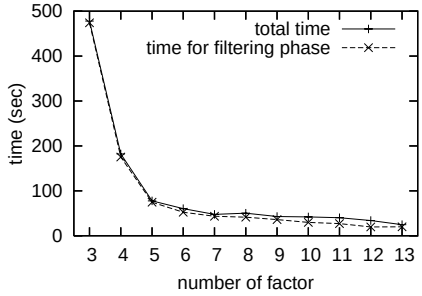
We used three texts: English, DNA, and random data. English and DNA were obtained from Pizza&Chili Corpus web-site (http://pizzachili.dcc.uchile.cl/) and truncated to length 16Mbytes. Random texts are of length 64M with various alphabet sizes. In each test, 100∼10000 patterns of the same length m were selected randomly from text and searched with k errors. We often report the *error level* $\alpha = k/m$ instead of k.

Our machine is a 2.6Ghz Pentium IV with 2GB of RAM, running Linux. We used the gcc compiler version 4.0.2 with option "-O3".

4.1 Suffix filter parameters Section 2.2 describes a method for choosing the suffix filter parameters, which leaves one parameter, the last factor size ℓ, to be determined experimentally. Figure 4 (a) illustrates the effect of ℓ in one case. In this example, the best ℓ is 6



(a) varying the last factor size ℓ ($s = 13$)



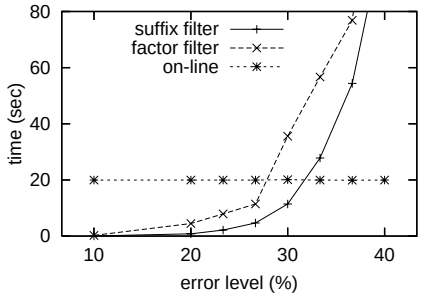
(b) varying the number of factors s ($\ell = 6$)

Figure 4: Performance of suffix filter on DNA data for $m = 40$, $k = 12$, and 200 queries.

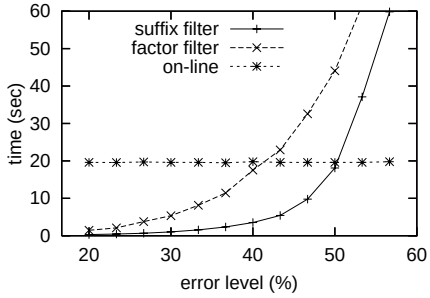
and other factor sizes are 3 and 4. As ℓ gets smaller than optimal, the verification time increases rapidly. With larger than optimal ℓ, filtration time increases but at a more modest rate. The behaviour in other cases is similar. The results indicate that using the absolutely optimal ℓ is not crucial but it is better to err in the too large direction. This is in contrast to factor filters, where having the free parameter s off by one in either direction can have a dramatic effect in the running time [13].

The method of Section 2.2 sets the number of factors s to $k+1$ but we experimented also with smaller values of s (larger values would mean that $t_i = 0$ for some i). Figure 4 (b) shows the results in one case. Here the last factor size ℓ is fixed to 6, its distance limit t_{s-1} to 1, and the other factor sizes and distance limits are as even as possible. The results in this and in other cases, too, show that $s = k + 1$ is the best choice.

4.2 Suffix filter vs. factor filter In this section, we compare suffix filters to factor filters. The parameters are chosen as described in Sections 2.1 and 2.2. For suffix filters the last factor size ℓ, and for factor filters the number of factors s were optimized experimentally.

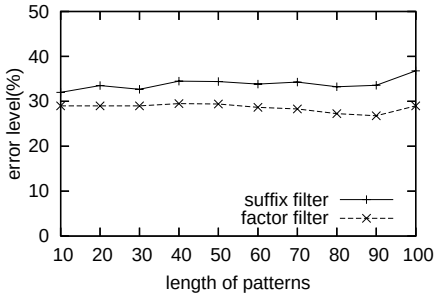


(a) DNA

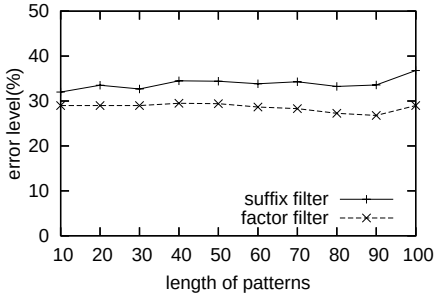


(b) English

Figure 5: The effect of error level for $m = 30$ and 100 queries.



(a) DNA



(b) English

Figure 6: The error levels up to which each indexing method wins upon online search.

First, we show the effect of error level in Figure 5. The curve "on-line" represents the time for verifying the whole text without a filtering step. The figure illustrates that indexing becomes useless when the error level grows too large. Figure 6 shows up to which error level each filtering method wins upon on-line search, as a function of m. For suffix filters this limit of usefulness is substantially higher than for factor filters.

Figure 6 also indicates that the advantage of suffix filters over factor filters increases for longer patterns. More clearly this effect can be seen in Figure 7. In both figures, there is a notable jump when the pattern length goes from 90 to 100. This is caused by the verification stage, which switches from a fast implementation that can handle patterns up to length 96 to a slower implementation that has no limit on the pattern length. As can be seen in Figure 7, suffix filters are less sensitive to the speed of the verification stage than factor filters.

Finally, we give a more comprehensive comparison between the suffix and factor filters. Table 1 shows the ratios of the running times. The suffix filters are significantly faster than the factor filters except in some extreme cases. With small alphabet and short patterns, searching the full pattern directly using an index is competitive with filtering, and this is reflected in the filter parameters (ℓ is large for suffix filters and $s = 1$ for factor filters). With small alphabet and large error level, filtering is no more competitive with on-line searching. With large alphabet and low error level, the filters are highly effective and having a more effective filter does not help anymore.

5 Acknowledgements

We would like to thank Gonzalo Navarro for kindly providing the source code used in [13].

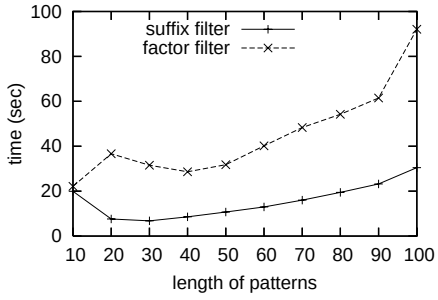
References

[1] R. A. Baeza-Yates and G. Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999.

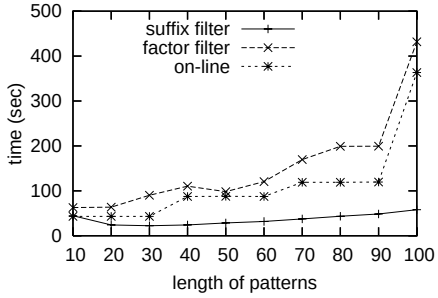
[2] S. Burkhardt and J. Kärkkäinen. Better filtering with gapped q-grams. *Fundam. Informaticae*, 56(1-2):51–70, 2003.

[3] H.-L. Chan, T.-W. Lam, W.-K. Sung, S.-L. Tam, and S.-S. Wong. A linear size index for approximate pattern matching. In *Proc. 17th Annual Symposium on Combinatorial Pattern Matching*, volume 4009, pages 49–59. Springer, 2006.

[4] A. L. Cobbs. Fast approximate matching using suffix trees. In Z. Galil and E. Ukkonen, editors, *Proceedings of the 6th Annual Symposium on Combinatorial*



(a) $\alpha = 10\%$ (10000 queries)



(b) $\alpha = 30\%$ (200 queries)

Figure 7: The effect of pattern length for DNA data.

	m			
	10	20	30	40
$\alpha = 10\%$	1.1	4.8	4.7	3.3
20%	1.2	4.6	7.0	4.2
30%	1.4	2.6	4.0	4.6
40%	1.6	2.0	1.5	1.5

(a) DNA (n=16M)

	m			
	10	20	30	40
$\alpha = 10\%$	2.4	2.4	2.0	2.1
20%	3.6	3.9	4.3	4.8
30%	2.9	4.6	4.8	7.4
40%	2.6	3.7	5.1	7.5

(b) English (n=16M)

	m			
	10	20	30	40
$\alpha = 10\%$	1.1	4.6	4.0	3.7
20%	1.1	3.9	6.9	4.2
30%	1.1	2.8	3.8	5.1
40%	1.2	1.7	1.6	1.5

(c) Random text over $|\Sigma| = 4$ (n=64M)

	m			
	10	20	30	40
$\alpha = 10\%$	1.1	1.0	0.9	0.9
20%	4.6	3.1	3.6	5.2
30%	1.4	4.1	2.9	5.1
40%	3.9	4.3	4.8	6.5

(d) Random text over $|\Sigma| = 16$ (n=64M)

Table 1: Suffix filters vs. factor filters. The numbers are the ratios of the running times (factor filter time / suffix filter time).

Pattern Matching, number 937 in LNCS, pages 41–54. Springer, 1995.

[5] L. Coelho and A. Oliveira. A structure for approximate text indexing. In *Proc. 13th Symposium on String Processing and Information Retrieval*. Springer, Oct. 2006. To appear.

[6] R. Cole, L.-A. Gottlieb, and M. Lewenstein. Dictionary matching and indexing with errors and don't cares. In *Proceedings of the thirty-sixth annual ACM Symposium on Theory of Computing (STOC-04)*, pages 91–100. ACM Press, 2004.

[7] H. Hyyrö. Explaining and extending the bit-parallel algorithm of Myers. Technical Report A-2001-10, Department of Computer and Information Sciences, University of Tampere, 2001.

[8] B. Ma, J. Tromp, and M. Li. PatternHunter: Faster and more sensitive homology search. *Bioinformatics*, 18:440–445, 2002.

[9] M. G. Maaß and J. Nowak. Text indexing with errors. In *Proc. 16th Annual Symp. on Combinatorial Pattern Matching (CPM)*, volume 3537 of *LNCS*, pages 21–32. Springer, June 2005.

[10] E. W. Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica*, 12(4/5):345–374, 1994.

[11] G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *J. ACM*, 46(3):395–415, 1999.

[12] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[13] G. Navarro and R. Baeza-Yates. A hybrid indexing method for approximate string matching. *Journal of Discrete Algorithms*, 1(1):205–239, 2000. Special issue on Matching Patterns.

[14] G. Navarro, R. Baeza-Yates, E. Sutinen, and J. Tarhio. Indexing methods for approximate string matching. *IEEE Data Engineering Bulletin*, 24(4):19–27, 2001. Special issue on Managing Text Natively and in DBMSs.

[15] P. Weiner. Linear pattern matching algorithm. In *Proc. 14th Symposium on Switching and Automata Theory*, pages 1–11. IEEE Press, 1973.

[16] S. Wu and U. Manber. Fast text searching allowing errors. *Commun. ACM*, 35(10):83–91, 1992.