A New Indexing Method for Approximate String Matching *

Gonzalo Navarro and Ricardo Baeza-Yates

Dept. of Computer Science, University of Chile
Blanco Encalada 2120 - Santiago - Chile
{gnavarro,rbaeza}@dcc.uchile.cl

Abstract. We present a new indexing method for the approximate string matching problem. The method is based on a suffix tree combined with a partitioning of the pattern. We analyze the resulting algorithm and show that the retrieval time is $O(n^\lambda)$, for $0 < \lambda < 1$, whenever $\alpha < 1 - e/\sqrt{\sigma}$, where α is the error level tolerated and σ is the alphabet size. We experimentally show that this index outperforms by far all other algorithms for indexed approximate searching, also being the first experiments that compare the different existing schemes. We finally show how this index can be implemented using much less space.

1 Introduction

Approximate string matching is a recurrent problem in many branches of computer science, with applications to text searching, computational biology, pattern recognition, signal processing, etc.

The problem is: given a long text of length n, and a (comparatively short) pattern of length m, retrieve all the text segments (or "occurrences") whose *edit distance* to the pattern is at most k. The *edit distance* between two strings is defined as the minimum number of character insertions, deletions and replacements needed to make them equal. We define the "error level" as $\alpha = k/m$.

In the on-line version of the problem, the pattern can be preprocessed but the text cannot. The classical solution uses dynamic programming and is $O(mn)$ time [27, 28]. A number of algorithms improved later this result [34, 20, 16, 11, 35, 32, 12, 30, 36, 9, 8, 24]. The lower bound of the on-line problem (proved and reached in [12]) is $O(n(k + \log_\sigma m)/m)$, which is of course $\Omega(n)$ for constant m.

If the text is large even the fastest on-line algorithms are not practical, and preprocessing the text becomes necessary. However, just a few years ago, indexing text for approximate string matching was considered one of the main open problems in this area [35, 3]. Despite some progress in the last years, the indexing schemes for this problem are still rather immature.

There are two types of indexing mechanisms for approximate string matching, which we call "word-retrieving" and "sequence-retrieving". Word retrieving

* This work has been supported in part by Fondecyt grant 1-990627 and Fondef grant 96-1064.

indices [22, 6, 2] are more oriented to natural language text and information retrieval. They can retrieve every *word* whose edit distance to the pattern is at most k. Hence, they are not able to recover from an error involving a separator, such as recovering the word `"flowers"` from the misspelled text `"flo wers"` or from `"manyflowers"`, if we allow one error[1]. These indices are more mature, but their restriction can be unacceptable in some applications, especially where there are no words (as in DNA) or in agglutinating languages such as Finnish or German.

Our focus in this paper is sequence retrieving indices. Among these, we find two types of approaches.

A first type is based on simulating a sequential algorithm, but running it on the suffix tree [19, 1] or DAWG [14, 10] of the text instead of the text itself. Since every different substring in the text is represented by a single node in the tree or the DAWG, it is possible to avoid redoing the same work when the text has repetitions. Those indices take $O(n)$ space and construction time, but their construction is not optimized for secondary memory and is very inefficient in this case (see, however, [15]). Moreover, the structure is very inefficient in space requirements, since it takes 12 to 70 times the text size.

In [18, 33, 13], different algorithms that traverse the least possible nodes in the suffix tree (or in the DAWG) are presented. The idea is to traverse all the different tree nodes that represent "viable prefixes", which are text substrings that can be prefixes of an approximate occurrence of the pattern.

In [17], a simplified version of the above technique was independently proposed, consisting of a limited depth-first search (DFS) on the suffix tree. Since every substring of the text (i.e. every potential occurrence) can be found from the root of the suffix tree, it is sufficient to explore every path starting at the root, descending by every branch up to where it can be seen that that branch does not represent the beginning of an occurrence of the pattern. This algorithm inspects more nodes than the previous ones, but it is simpler. For instance, with an additional $O(\log n)$ time factor, the algorithm runs on suffix arrays, which take 4 times the text size instead of 12. This algorithm was analyzed in [4].

The second type of sequence-retrieving indices is based on adapting an on-line filtering algorithm. The filters are based in matching substrings of the patterns without errors, and checking for potential occurrences around those matches. The index is used to quickly find those substrings, and is based on storing some text q-grams (substrings of length q) and their positions in the text.

Different filtration indices [23, 31, 29, 7] differ mostly in how the text is sampled (distance between consecutive text samples, whether they overlap or not, etc.), in how the pattern is sampled, in how many matching samples are needed to verify their neighborhood in the text, etc. Depending on this and on q they achieve different space-time tradeoffs. In general, filtration indices are much smaller than suffix trees (1 to 10 times the text size), although they are less tolerant to the error level α. They can also be built in linear time.

[1] Although some, like Glimpse [22], can match the pattern inside a text word.

Somewhat special is [23], because it does not reduce the search to exact but to approximate search of pattern pieces. To search for a pattern of length $m \leq q-k$, all the maximal strings with edit distance $\leq k$ to the pattern are generated and searched in the set of q-grams. Later, all the occurrences are merged. Longer patterns are split in as many pieces as necessary to make them short enough.

In this paper we present a hybrid indexing scheme for this problem. It uses a suffix tree, where the pattern is partitioned in subpatterns which are searched with less errors in suffix tree. All the occurrences of the subpatterns are later verified for a complete match. The goal is to balance between the cost to search in the suffix tree (which grows with the size of the subpatterns) and the cost to verify the potential occurrences (which grows when shorter patterns are searched). This method shows experimentally to be by far superior to all other implemented proposals, and we show analytically that the average retrieval time can be made $O(n^{2(\alpha+H_\sigma(\alpha))/(1+\alpha)})$, where $H_\sigma(\alpha)$ is the base-σ entropy function. This is sublinear for $\alpha < 1-e/\sqrt{\sigma}$. This limit on α cannot probably be improved [8, 25]. We finally propose an alternative data structure to reduce the space requirements of the suffix tree, with little time penalty.

2 Combining Suffix Trees and Pattern Partitioning

We present now our alternative proposal. The general idea is to partition the pattern in pieces, search each piece in the suffix tree in the classical way, and check all the positions found for a complete match. We first consider how to search a piece in the suffix tree and later address the pattern partitioning issue.

2.1 DFS Using a Bit-parallel Automaton

Let us consider the existing algorithms to traverse the suffix tree. While [33, 13] minimize the number of nodes traversed, [17] is simpler but inspects more nodes. We show that [17], thanks to its simplicity, can be adapted to use a node processing algorithm which is faster than dynamic programming, namely our on-line algorithm of [8][2]. The tradeoff is: we can explore less nodes at higher cost per node or more nodes at less cost per node. We show later experimentally that this last alternative is much faster when [8] is used to process the nodes.

We recall that the idea of [17] is a limited depth-first search on the suffix tree, starting at the root and stopping when it can be seen that the current text substring cannot start an approximate pattern occurrence. No text occurrence can be missed because every text substring can be found starting from the root.

More specifically, we compute the edit distance between the tree path and the pattern, and if at some node the distance is $\leq k$ we know that the text substring represented by the node matches the pattern. We report all the leaves of the suffix tree which descend from those nodes, since their text positions start with the matching substring. On the other hand, when we can determine that the

[2] Probably [24] would also fit well.

edit distance cannot be as low as k, we abandon the path. This surely happens at depth $m+k+1$ but normally happens before.

We implement this traversal using our algorithm of [8] instead of dynamic programming. This algorithm uses bit parallelism to simulate a non-deterministic finite automaton (NFA) that recognizes the approximate pattern. We modify this automaton to compute edit distance (removing the initial self-loop it has in [8]).

Figure 1 shows the automaton to recognize "**patt**" with $k=2$ errors. Every row denotes the number of errors seen. Every column represents matching a pattern prefix. Horizontal arrows represent matching a character (i.e. if the pattern and text characters match, we advance in the pattern and in the text). All the others increment the number of errors (move to the next row): vertical arrows insert a character in the pattern (we advance in the text but not in the pattern), solid diagonal arrows replace a character (we advance in the text and pattern), and dashed diagonal arrows delete a character of the pattern (they are empty transitions, since we advance in the pattern without advancing in the text). The automaton signals (the end of) a match whenever a rightmost state is active. If we do not care about the number of errors of the occurrences, we can consider final states those of the last full diagonal.

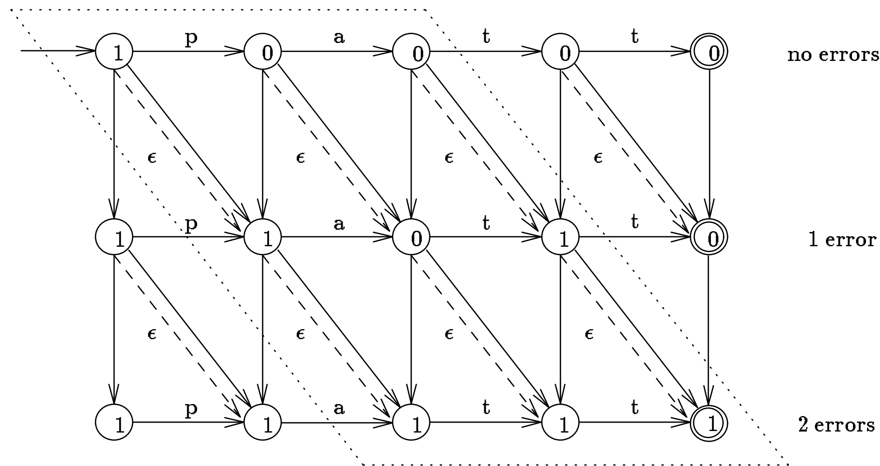


Fig. 1. An NFA for approximate string matching. Unlabeled transitions match any character. Dotted lines enclose the states actually represented in our algorithm.

Initially, the active states at row i are at the columns from 0 to i, to represent the deletion of the first i characters of the pattern. We do not need in fact to represent the initial lower-left triangle, since if a substring matches with initial insertions we will find (in other branch of the suffix tree) a suffix of it which does not need the insertions[3]. On the other hand, unlike [8], we need to represent the

[3] If, after traversing a text substring s, a 1 finally exits from the lower-left triangle, then a suffix of s will do the same without entering into the triangle.

first full diagonal, since now it will not be always active. We start the automaton with only this first full diagonal active, and traverse the suffix tree path until the automaton runs out of active states or the lower right state is activated.

The simulation of this automaton needs $(m-k+1)(k+2)$ bits. If we call w the number of bits in the computer word, then when the previous number is $\leq w$ we can put all the states in a single computer word and work $O(1)$ per traversed node of the suffix tree. For longer patterns, the automaton is split in many computer words, at a cost of $O(k(m-k)/w)$. For moderate-size patterns this improves over dynamic programming, which costs $O(m)$ per suffix tree node.

This bit-parallel variation is only possible because of the simplicity of the traversal. For instance, the idea does not work on the more complex setup of [33, 13], since these need some adaptations of the dynamic programming algorithm that are not easy to parallelize. Note that this algorithm can be seen as a particular case of automaton searching over a trie [5].

2.2 Partitioning the Pattern

It is well known [33, 4] that the search cost using the suffix tree grows exponentially with m and k, no matter which of the two techniques we use (optimum traversal or DFS). Hence, we prefer that m and k are small numbers. We present in this section a new technique based in partitioning the pattern, so that the pattern is split in many sub-patterns which are searched in the suffix tree, and their occurrences are directly verified in the text for a complete match. We show in the experiments that this technique outperforms all the others.

This method is based on the pattern partitioning technique of [23, 8]. The core of the idea is that, if a pattern of length m occurs with k errors and we split the pattern in j parts, then at least one part will appear with $\lfloor k/j \rfloor$ errors inside the occurrence. In fact, the case $j = k+1$ is the basis for the algorithm [9] and the q-gram index [7].

The new algorithm follows. We evenly divide the pattern in j pieces (j is unspecified by now). Then we search in the suffix tree the j pieces with $\lfloor k/j \rfloor$ errors using the algorithm of Section 2.1. For each match found ending at text position i we check the text area $[i-m-k..i+m+k]$.

The reason why this idea works better than a simple suffix tree traversal with the complete pattern is that, since the search cost on the suffix tree is exponential in m and k, it may be better to perform j searches of patterns of length m/j and k/j errors. However, the larger j, the more text positions have to be verified, and therefore the optimum is in between. In the next section we find analytically the optimum j and the complexity of the search

One of the closest approaches to this idea is Myers' index [23], which collects all the text q-grams (i.e. prunes the suffix tree at depth q), and given the pattern it generates all the strings at distance at most k from it, searches them in the index and merges the results. This is the same work of a suffix tree provided that we do not enter too deep (i.e. $m+k \leq q$). If $m+k > q$, Myers' approach splits the pattern and searches the subpatterns in the index, checking all the potential occurrences. The main difference with our proposed approach is that

Myers' index generates all the strings at a given distance and searches them, instead of traversing the structure to see which of them exist. This makes that approach degrade on biased texts, where most of the generated q-grams do not exist (in the experimental section we show that it works well on DNA but quite bad on English). Moreover, we split the pattern to optimize the search cost, while the splitting in Myers' index is forced by indexing constraints (i.e. q).

3 Analysis

3.1 Searching One Piece

An asymptotic analysis on the performance of a depth-first search over suffix trees is immediate if we consider that we cannot go deeper than level $m+k$ since past that point the edit distance between the path and our pattern is larger than k and we abandon the search. Therefore, we can spend at most $O(\sigma^{m+k})$ time, which is independent on n and hence $O(1)$. Another way to see this is to use the analysis of [5], where the problem of searching an arbitrary automaton over a suffix trie is considered. Their result for this case indicates constant time (i.e. depending on the size of the automaton only) because the automaton has no cycles.

However, we are interested in a more detailed average analysis, especially the case where n is not so large in comparison to σ^{m+k}. We start by analyzing which is the average number of nodes at level ℓ in the suffix tree of the text, for small ℓ. Since almost all suffixes of the text are longer than ℓ (i.e. all except the last ℓ), we have nearly n suffixes that reach that level. The total number of nodes at level ℓ is the number of different suffixes once they are pruned at ℓ characters. This is the same as the number of different ℓ-grams in the text. If the text is random, then we can use a model where n balls are thrown into σ^ℓ urns, to find out that the average number of filled urns (i.e. suffix tree nodes at level ℓ) is

$$\sigma^\ell\left(1-\left(1-1/\sigma^\ell\right)^n\right) \;=\; \sigma^\ell\left(1-e^{-\Theta(n/\sigma^\ell)}\right) \;=\; \Theta\left(\min\left(n,\sigma^\ell\right)\right)$$

which shows that the average case is close to the worst case: up to level $\log_\sigma n$ all the possible σ^ℓ nodes exist, while for deeper levels all the n nodes exist.

We also need the probability of processing a given node at depth ℓ in the suffix tree. In the Appendix we prove that the probability is very high for $\beta = k/\ell \geq 1-c/\sqrt{\sigma}$ (Eq. (3)), and otherwise it is $O(\gamma(\beta)^\ell)$, where $\gamma(\beta) < 1$. The constant c can be proven to be smaller than $e = 2.718...$, and is empirically known to be close to 1. The $\gamma(x)$ function (Eq. (1)) is $1/(\sigma^{1-x}x^{2x}(1-x)^{2(1-x)})$, which goes from $1/\sigma$ to 1 as x goes from 0 to $1-c/\sqrt{\sigma}$.

Therefore, we pessimistically consider that in levels

$$\ell \;\leq\; L(k) \;=\; \frac{k}{1-c/\sqrt{\sigma}} \;=\; O(k)$$

all the nodes in the suffix tree are visited, while nodes at level $\ell > L(k)$ are visited with probability $O(\gamma(k/\ell)^\ell)$, where $\gamma(k/\ell) < 1$. Finally, we never work

past level $m + k$. We are left with three disjoint cases to analyze, illustrated in Figure 2.

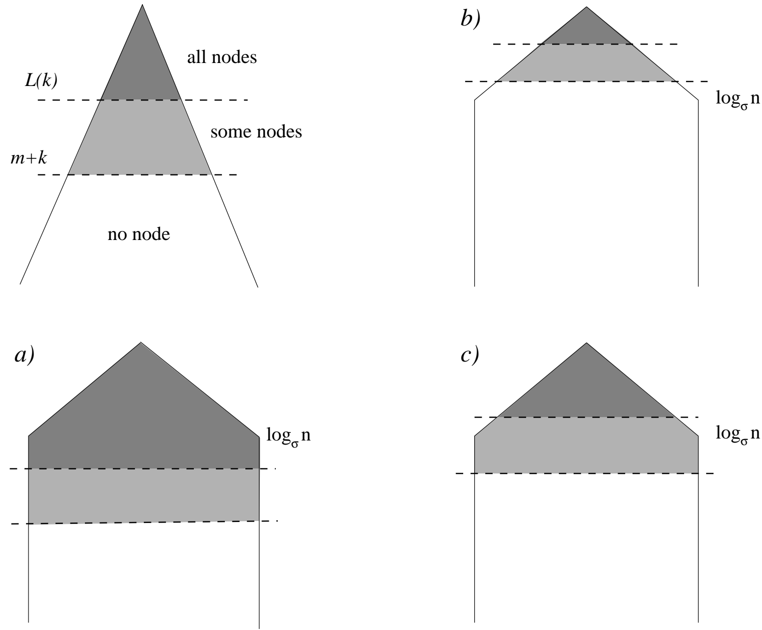


Fig. 2. The upper left figure shows the visited parts of the tree. The rest shows the three disjoint cases in which the analysis is split.

(a) $L(k) \geq \log_\sigma n$, i.e. $n \leq \sigma^{L(k)}$, or "small n"
In this case, since on average we work on all the nodes up to level $\log_\sigma n$, the total work is n, i.e. the amount of work is proportional to the text size. This shows that the index simply does not work for very small texts, being an on-line search preferable as expected.

(b) $m + k < \log_\sigma n$, i.e. $n > \sigma^{m+k}$ or "large n"
In this case we traverse all the nodes up to level $L(k)$, and from there on we work at level ℓ with probability $\gamma(k/\ell)^\ell$, until $\ell = m + k$. Under case (b), there are σ^ℓ nodes at level ℓ. Hence the total number of nodes traversed is

$$\sum_{\ell=0}^{L(k)} \sigma^\ell \quad + \sum_{\ell=L(k)+1}^{m+k} \gamma(k/\ell)^\ell \sigma^\ell$$

where the first term is $O(\sigma^{L(k)})$. For the second term, we see that $\gamma(x) > 1/\sigma$, and hence $(\gamma(k/\ell)\sigma)^\ell > 1$. More precisely,

$$(\gamma(k/\ell)\sigma)^\ell \quad = \quad \frac{\sigma^k \ell^{2\ell}}{k^{2k}(\ell - k)^{2(\ell-k)}}$$

which grows as a function of ℓ. Since $(\gamma(k/\ell)\sigma)^\ell > 1$, we have that even if it were constant with ℓ, the last term would dominate the summation. Hence, the total cost in case (b) is

$$\sigma^{L(k)} \quad + \quad \frac{\sigma^k(1+\alpha)^{2(m+k)}}{\alpha^{2k}}$$

which is independent of n.

(c) $L(k) < \log_\sigma n \le m+k$, i.e. "intermediate n"
In this case, we work on all nodes up to $L(k)$ and on some nodes up to $m+k$. The formula for the number of visited nodes is

$$\sum_{\ell=0}^{L(k)} \sigma^\ell \quad + \sum_{\ell=L(k)+1}^{\log_\sigma(n)-1} \gamma(k/\ell)^\ell \sigma^\ell \quad + \sum_{\ell=\log_\sigma n}^{m+k} \gamma(k/\ell)^\ell n$$

The first sum is $O(\sigma^{L(k)})$. For the second sum, we know already that the last term dominates the complexity (see case (b)). Finally, for the third sum we have that $\gamma(k/\ell)$ decreases as ℓ grows, and therefore the first term dominates the rest (which would happen even for a constant γ).
Hence, the case $\ell = \log_\sigma n$ dominates the last two sums. This term is

$$n\gamma(k/\log_\sigma n)^{\log_\sigma n} \;=\; \frac{\sigma^k(\log_\sigma n)^{2\log_\sigma n}}{k^{2k}(\log_\sigma(n)-k)^{2(\log_\sigma(n)-k)}} \;=\; \frac{\sigma^k(\log_\sigma n)^{2k}}{k^{2k}}\,(1+o(1))$$

(this can be bounded by $(\sigma(1+1/\alpha)^2)^k$ by noticing that we are inside case (c), but we are interested in how n affects the growth of the cost).

The search time is then sublinear for $\log_\sigma n > \max(L(k), m+k)$, or which is the same, $\alpha < \max(\log_\sigma(n)/m\ (1 - c/\sqrt{\sigma}), \log_\sigma(n)/m - 1)$. Figure 3 illustrates.

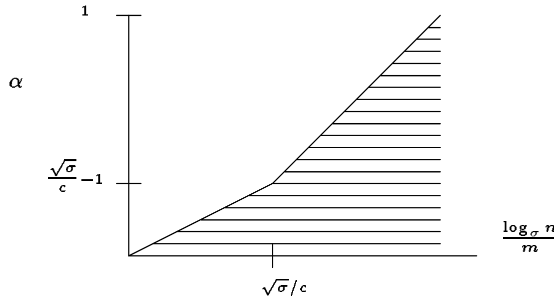


Fig. 3. Area of sublinearity for suffix tree traversal.

3.2 Pattern Partitioning

When pattern partitioning is applied, we perform j searches of the same kind of Section 2.1, this time with patterns of length m/j and k/j errors. We also need to verify all the possible matches.

As shown in [8], the matching probability for a text position is $O(\gamma(\alpha)^m)$, where $\gamma(\alpha)$ is that of Eq. (1). From now on we use $\gamma = \gamma(\alpha)$. Using dynamic programming, a verification costs $O(m^2)$ [4]. Hence, our total search cost is

$$j \times suffix_tree_traversal(m/j, k/j) \quad + \quad j \times \gamma^{m/j} m^2 n$$

and we want the optimum j. First, notice that if $\gamma = 1$ (that is, $\alpha \geq 1 - c/\sqrt{\sigma}$), the verification cost is as high as an on-line search and therefore pattern partitioning is useless. In this case it may be better to use plain DFS. In the analysis that follows, we assume that $\gamma < 1$ and hence $\alpha < 1 - c/\sqrt{\sigma}$.

According to Section 3.1, we divide the analysis in three cases. Notice that now we can adjust j to select the best case for us.

(a) $\sigma^{L(k/j)} \geq n$, or $j \log_\sigma n \leq k/(1 - c/\sqrt{\sigma})$
In this case the search cost is $\Omega(n)$ and the index is of no use.

(b) $\sigma^{(m+k)/j} < n$, or $j \log_\sigma n > m + k$
In this case the total search cost is

$$j \left(\sigma^{L(k/j)} \; + \; \frac{\sigma^{k/j}(1+\alpha)^{2(m+k)/j}}{\alpha^{2k/j}} \; + \; \gamma^{m/j} m^2 n \right)$$

where the first two terms decrease and the last one increases with j. Since $a + b = \Theta(\max(a, b))$, the minimum order is achieved when increasing and decreasing terms meet. When equating the first and third terms we obtain that the optimum j is

$$j_1 \quad = \quad \frac{m}{\log_\sigma(m^2 n)} \left(\frac{\alpha}{1 - c/\sqrt{\sigma}} + \log_\sigma(1/\gamma) \right)$$

and the complexity (only considering n) is $O\left(n^{\alpha/(\alpha + (1-c/\sqrt{\sigma}) \log_\sigma(1/\gamma))}\right)$.
On the other hand, if we equate the second and third term, the best j is

$$j_2 \quad = \quad \frac{m}{\log_\sigma(m^2 n)} \; (1 + 2((1+\alpha)\log_\sigma(1+\alpha) + (1-\alpha)\log_\sigma(1-\alpha)))$$

and the complexity is $O\left(n^{1 - \log_\sigma(1/\gamma)/(1 + 2((1+\alpha)\log_\sigma(1+\alpha) + (1-\alpha)\log_\sigma(1-\alpha)))}\right)$.
In any case, we are able to achieve a sublinear complexity of $O(n^\lambda)$, where

$$\lambda = \max\left(\frac{\alpha}{\alpha + (1-c/\sqrt{\sigma})\log_\sigma(1/\gamma)}, \; 1 - \frac{\log_\sigma(1/\gamma)}{1 + 2((1+\alpha)\log_\sigma(1+\alpha) + (1-\alpha)\log_\sigma(1-\alpha))}\right)$$

Which of the two complexities dominates yields a rather complex condition that depends on the error level α, but in both cases $\lambda < 1$ if $\alpha < 1 - c/\sqrt{\sigma}$.

[4] It can be done in $O((m/j)^2)$ time [23, 26], but this does not affect the result here.

If σ is large enough ($\sigma \geq 24$ for $c = e$), the complexity corresponding to j_2 always dominates. However, it is possible that j_1 or j_2 are outside the bounds of case (b) (i.e. they are too small). In this case we would use the minimum possible $j = (m+k)/\log_\sigma n$, and the third term would dominate the cost, for an overall complexity of $O(n^{1-\log_\sigma(1/\gamma)/(1+\alpha)})$. This complexity is also sublinear if $\alpha < 1 - c/\sqrt{\sigma}$.

(c) $\sigma^{L(k/j)} < n \leq \sigma^{(m+k)/j}$, or $k/(1-c/\sqrt{\sigma}) < j \log_\sigma n \leq m+k$
The search cost in this intermediate case is

$$j\left(\sigma^{L(k/j)} + \frac{\sigma^{k/j}(\log_\sigma n)^{2k/j}}{(k/j)^{2k/j}} + \gamma^{m/j} m^2 n\right)$$

where the first two terms decrease with j and the last one increases. Repeating the same process as before, we find that the first and third term meet again at $j = j_1$ with the same complexity. We could not solve exactly where the second and third term meet. We found

$$j_3 = \frac{m(\alpha + 2\alpha \log_\sigma \log_\sigma n + \log_\sigma \frac{1}{\gamma} - 2\alpha \log_\sigma \frac{m}{j_3})}{\log_\sigma(m^2 n)} \approx \frac{m(\alpha + \log_\sigma \frac{1}{\gamma})}{\log_\sigma(m^2 n)}$$

and since the solution is approximate, the terms are not exactly equal at j_3. The second term is $O\left(n^{\alpha(1+2\log_\sigma(1/\gamma))/(\alpha+\log_\sigma(1/\gamma))}\right)$, slightly higher than the third. Again, it is possible that j_3 is out of the bounds of case (c) and we have to use the same limiting value as before.

The conclusion is that, despite that the exact formulation is complex, we have sublinear complexity for $\alpha < 1 - c/\sqrt{\sigma}$, as well as formulas for the optimum j to use, which is $\Theta(m/\log_\sigma n)$ with a complicated constant.

For larger α values the pattern partitioning method gives linear complexity and we need to resort to the traditional suffix tree traversal ($j = 1$). As shown in [8, 25], it is very unlikely that this limit of $1 - c/\sqrt{\sigma}$ can be improved, since there are too many real approximate occurrences in the text.

An interesting fact that is shown in the experiments is that in many cases the optima are out of bounds and hence the best is to put j in the limit of cases (b) and (c), just where the search of the subpieces become full searches. This shows that a technique that is simple and the best choice in most cases is to select $j = (m+k)/\log_\sigma n$, for a complexity of

$$O\left(n^{1-\frac{\log_\sigma(1/\gamma)}{1+\alpha}}\right) = O\left(n^{\frac{2(\alpha+H_\sigma(\alpha))}{1+\alpha}}\right)$$

where $H_\sigma(\alpha) = -\alpha \log_\sigma \alpha - (1-\alpha)\log_\sigma(1-\alpha)$ is the base-σ entropy function.

3.3 The Limits of the Method

Let us pay some attention to the limits of our hybrid method (Figure 4).

Since $j = \Theta(m/\log_\sigma n)$, the best j becomes 1 (i.e. no pattern partitioning) when $n > \sigma^{\Theta(m)}$ (this is because the cost of verifications dominates over suffix

tree traversal). The best j is $\geq k+1$ for $n < \sigma^{\Theta(1/\alpha)}$. Since in this case we search the pieces with zero errors (i.e. $\lfloor k/(k+1) \rfloor = 0$, recall Section 2.2), the search in the suffix tree costs $O(m)$, and later we have to verify all their occurrences. This is basically what the q-gram index of [7] does, except it prunes the suffix tree at depth q.

Finally, the only case where the index is not useful is when n is very small. We can increase j to be more resistant to small texts, but the limit is $j = k+1$, and using that j the index ceases to be useful for $n < \sigma^{\frac{1}{1-c/\sqrt{\sigma}}} \leq \sigma^{1/\alpha}$. We have also to keep sublinear the cost of verifications, i.e. $n\gamma^{1/\alpha} = o(1)$, which happens for $\alpha < 1/\log_{1/\gamma} n$. This requires, in particular, that $m = \Omega(\log n)$.

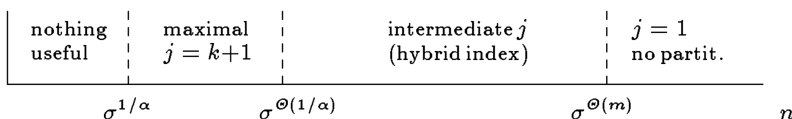


Fig. 4. The j values to be used according to n.

This last consideration helps also to understand how is it possible to have a sublinear-time index based on filtering when there is a fixed matching probability per text position (γ^m), and therefore the verification cost must be $\Omega(n)$. The trick is that in fact we assume $m = \Omega(\log n)$, that is, we have to search longer patterns as the text grows. As we can tune j, we softly move to $j = 1$ (then eliminating verification costs) when n becomes large with respect to m. This "trick" is also present in the sublinearity result of Myers' index [23], and implicit in similar results on natural language texts [6, 25].

4 Experimental Results

We first validate some of the analytical results of the paper and later compare our indices against the other existing proposals. We used two different texts:

- DNA text ("h.influenzae"), a 1.34 Mb file. This file is called DNA in our tests, and H-DNA is the first half megabyte of it. In this case $\sigma = 4$.
- English literary text (from B. Franklin), filtered to lower-case and the separators converted into a single space. This text has 1.26 Mb, and is called FRA in the experiments. H-FRA is the first half megabyte of FRA. Given how the analysis uses the σ value, it is unrealistic to set it to the alphabet size, because the text is biased. It is much better to consider that $1/\sigma$ must be the probability that two random letters are equal. This sets $\sigma = 12.85$.

The texts are rather small, in some cases too small to appreciate the speedup obtained with some indices. This is because we had RAM problems to build suffix trees for larger texts. However, the experiments still serve to obtain basic performance numbers on the different indices.

We have tested short and medium-size patterns, searching with 1, 2 and 3 errors the short ones and with 2, 4 and 6 the medium ones. The short patterns were of length 10 for DNA and 8 for English, and the medium ones were of length 20 and 16, respectively[5]. We selected 1000 random patterns from each file and use the same set for all the k values of that length, and for all the indices.

4.1 Validating the Analysis

We first show that the suffix tree traversal has sublinear complexity. We built the suffix tree of incremental prefixes of FRA and DNA, from 100 Kb to 800 Kb (larger texts start to give I/O problems that disturb the CPU measures). According to our analysis, the m, k and σ values used correspond to intermediate text sizes (case (b) of Section 3.1) for n = 4Kb..4Mb on DNA and for n = 40Kb..8Gb on FRA. Hence, we are clearly in case (b) in all our experiments. The analysis predicts a complexity of $O((\log n)^{2k})$.

Figure 5 shows the user time as n grows, from where the sublinearity is clear. We have used least squares with the model $t = a \ln(n)^b$ to find out the empirical complexity and present it compared to the analytical complexity. The error of the approximation is always below 5%. We see that the analysis approximates reasonably the empirical results, despite the many simplifications done.

We consider now the optimal j value for pattern partitioning. Table 1 presents the query time using different j values in our index, for the FRA, H-FRA, DNA, and H-DNA texts. As it can be seen, there are big differences in time depending on j, and the optimum is a rather small j value (always 1 on short patterns). This matches reasonably our formulas. In fact, once properly rounded, our analysis recommends the correct j values. As mentioned before, the relevant value is always in the limit between cases (b) and (c).

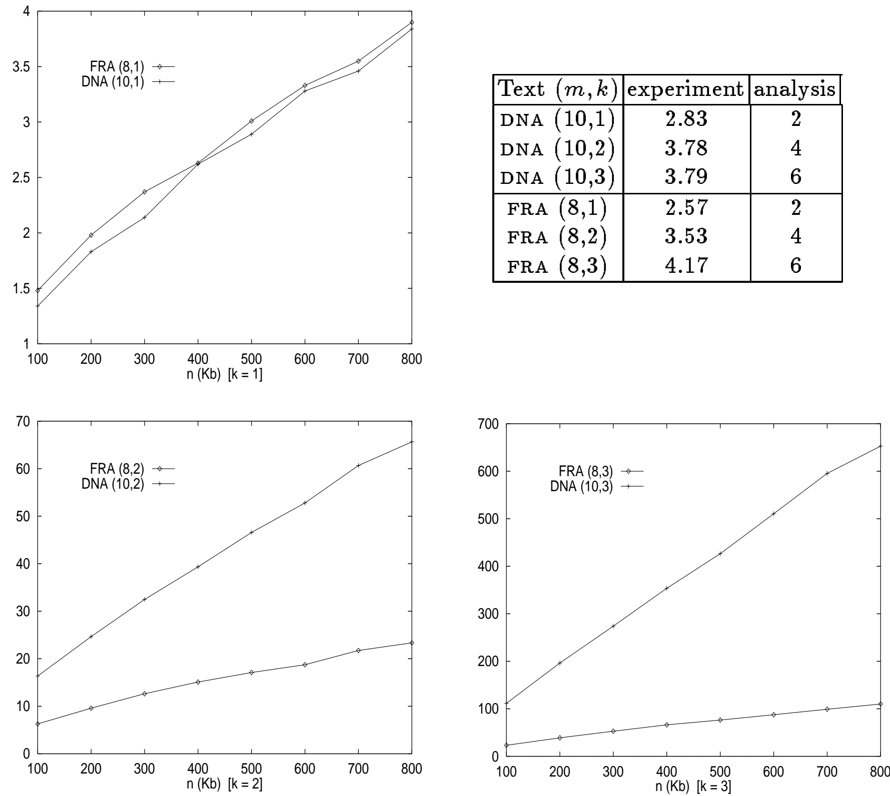
Figure 6 shows the user time for long patterns, as n grows, using pattern partitioning with $j = 2$. This time we have used least squares with the model $t = an^b$. The error of the approximation is always below 2%. It can be seen that also in this case the analysis approximates reasonably the empirical results, slightly overestimating in most cases. The combination DNA (20,6) is not included because it takes too long and already the case (20,4) was clearly linear.

4.2 Comparison Against Others

We compare our index with the other existing proposals. However, as the task to program an index is rather heavy, we have only considered the other indices that are already implemented. The indices included in this comparison are

Myers': The index proposed by Myers [23]. We use the implementation of the author, which works for some m values only (that depend on σ and n).

[5] This is because of the restrictions of Myers' index intersected with our interest in moderate-length patterns.



Text (m,k)	experiment	analysis
DNA (10,1)	2.83	2
DNA (10,2)	3.78	4
DNA (10,3)	3.79	6
FRA (8,1)	2.57	2
FRA (8,2)	3.53	4
FRA (8,3)	4.17	6

Fig. 5. User query time (in milliseconds) for short patterns as n grows for $k = 1$ to 3, using $j = 1$. On the top right, the empirical and analytical exponent of $\log n$.

Cobbs': The index proposed by Cobbs [13]. We use the implementation of the author, not optimized for space. The code is restricted to work on an alphabet of size 4 or less, so it is only built on DNA.

Samples(q)**:** Our index based on q-grams presented in [7]. We show the results for $q = 4$ to 6.

Dfs(a/p)**:** Our new index based on suffix trees. We show the results for the base technique (a) and pattern partitioning (p) with optimal j.

In particular, approximate searching on other q-gram indices [31] is not yet implemented and therefore is excluded from our tests. We know, however, that their space requirements are low (close to a word-retrieving index), but also that since the index simulates the on-line algorithm [30], its tolerance to errors is quite low (see [8, 25], for example).

All the indices were set to show the matches they found, in order to put them in a reasonably real scenario. We present the time to build the indices and the space they take in Table 2.

Text	(10, 1)	(10, 2)	(10, 3)	(20, 2)	(20, 4)	(20, 6)
DNA	**1: 6.81** 2: 2391	**1: 134.4** 2: 2585	**1: 1044** 2: 2756	1: 56.81 **2: 15.80** 3: 802.8	1: 1989 2: 1033 **3: 1010** 4: 5862	1: 10075 2: 9525 **3: 8841** 4: 39077
H-DNA	**1: 2.71** 2: 645.0	**1: 44.29** 2: 715.3	**1: 394.6** 2: 860.4	1: 23.72 **2: 6.01** 3: 232.7	1: 499.9 **2: 305.2** 3: 305.9 4: 1520	**1: 2308** 2: 2482 3: 2464 4: 10339
Text	(8, 1)	(8, 2)	(8, 3)	(16, 2)	(16, 4)	(16, 6)
FRA	**1: 6.11** 2: 180.6	**1: 42.82** 2: 1754	**1: 215.2** 2: 19600	1: 35.98 **2: 13.30** 3: 90.71	1: 482.9 **2: 88.22** 3: 736.6	1: 2204 **2: 464.0** 3: 4718
H-FRA	**1: 2.68** 2: 61.43	**1: 14.28** 2: 601.1	**1: 60.91** 2: 4920	1: 13.39 **2: 5.30** 3: 32.72	1: 126.4 **2: 30.70** 3: 255.5	1: 542.4 **2: 146.4** 3: 1538

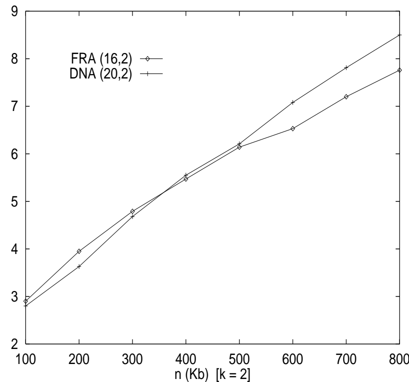
Table 1. User query time (in milliseconds) for different (m, k) values (heading rows). Inside each cell we show the cost for different j values. The optimum is in boldface.

The first clear result of the experiment is that the space usage of the indices is very high. In particular, the indices based on suffix trees (Dfs and Cobbs') take 35 to 65 times the text size. This outrules them except for very small texts (for instance, building Cobbs' index on 1.34 Mb took 12 hours of real time in our machine of 64 Mb of RAM). From the other indices, Myers' took 7-9 times the text size, which is much better but still too much in practice. The best option in terms of space is the Samples index, which takes from 1.5 to 7 times the text size, depending on q and σ. The larger q or σ, the larger the index. Samples(5), which takes 2-5 times the text size, performs well at query time.

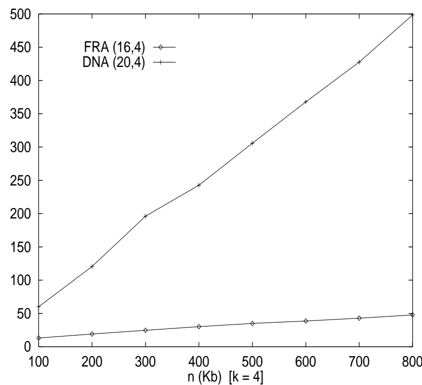
Compared to its size, Myers' index was built very quickly. The Dfs index, on the other hand, was built faster than Cobbs'. Notice that suffix trees are built quickly when they fit in RAM (as in the half-megabyte texts), but for larger texts the construction time is dominated by the I/O, and it increases sharply.

We consider now query time. Tables 3 and 4 present a comparison between the different indices, using for Dfs(p) the optimum j value of Table 1 (only for medium patterns, since for short ones Dfs(a) is always better). The system time is included because it is dominant in many cases. We include also the time of on-line searching for comparison purposes (we use the fastest on-line algorithm for each case). The results clearly show a number of facts.

- The indices work well only for moderate error levels. For larger texts the ratio indexed/on-line should improve. However, when I/O time is considered many indices seem useless, and it is not so clear that this improves for larger texts. This depends on the amount of main memory available, and is a consequence of most indices not being designed to work on secondary memory. This is a very important issue that has been rarely addressed.



Text (m,k)	experiment	analysis
DNA (20,2)	0.547	0.608
DNA (20,4)	1.009	0.935
FRA (16,2)	0.470	0.485
FRA (16,4)	0.624	0.752
FRA (16,6)	0.753	0.922



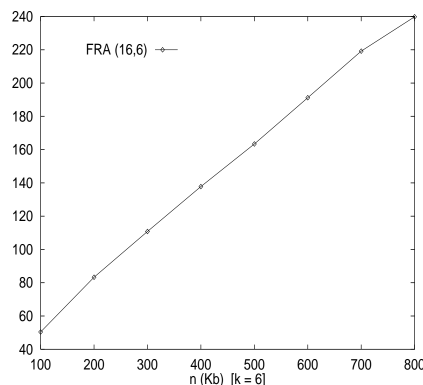


Fig. 6. User query time (in milliseconds) for medium patterns as n grows for $k = 2$ to 6, using $j = 2$. On the top right, the empirical and analytical exponent of n.

- Our strategy Dfs(a) of using a simpler traversal algorithm on the suffix tree and in return using a faster search algorithm definitively pays off, since our implementation is 3 to 40 times faster than Cobbs', and it is the fastest choice for small m and k values. Independently of this fact, the suffix tree indices improve on larger alphabets, but they are much more sensitive to the growth of m or k. In fact, the differences between FRA and DNA are due to the different values of m used. The big problem with this type of index is of course the huge space requirements it poses.
- Myers' index behaves better on short patterns, when less splitting is necessary. It works well for DNA but it worsens on English text. We conjecture that the non-randomness may play a role here: the index takes internally $q = \log_\sigma n$ to avoid searching a number of nonexistent samples that are at distance k or less from the pattern (in our case it took $q = 10$ for DNA and $q = 4$ for English). However, in biased texts like English, a lot of q-grams are not present anyway, and the index pays to search all of them. For DNA the index is a good alternative, since although it is up to 13 times slower than

Index	DNA	H-DNA	FRA	H-FRA
Myers'	5.84u+0.35s 10.68 Mb (7.97X)	2.08u+0.12s 4.50 Mb (9.00X)	5.22u+0.34s 9.39 Mb (7.46X)	2.01u+0.12s 4.18 Mb (8.35X)
Samples (4)	5.53u+0.19s 2.04 Mb (1.52X)	1.95u+0.10s 0.77 Mb (1.53X)	15.05u+0.41s 3.48 Mb (2.77X)	5.90u+0.24s 1.48 Mb (2.98X)
Samples (5)	7.37u+0.24s 2.48 Mb (1.85X)	2.62u+0.08s 0.94 Mb (1.87X)	20.82u+0.70s 5.18 Mb (4.11X)	8.70s+0.35s 2.32 Mb (4.65X)
Samples (6)	10.53u+0.32s 2.90 Mb (2.16X)	3.88u+0.13s 1.11 Mb (2.23X)	32.86u+1.34s 7.65 Mb (6.07X)	13.19u+0.97s 3.54 Mb (7.07X)
Cobbs'	108.70u+532.81s 87.99 Mb (65.67X)	30.50u+76.06s 32.93 Mb (65.85X)	n/a	
Dfs	30.89u+104.17s 52.25 Mb (38.99X)	6.48u+0.42s 19.55 Mb (39.10X)	28.46u+76.86s 44.66 Mb (35.45X)	6.43u+0.61s 17.66 Mb (35.32X)

Table 2. Times (in seconds) to build the indices and their space overhead. The time is separated in the CPU part ("u") and the I/O part ("s"). The space is expressed in megabytes, and also the ratio index/text is shown in the format rX, meaning that the index takes r times the text size.

Dfs(a), it takes 4 times less space. It is also better than the Samples index when the pattern is short, but not when pattern partitioning is necessary.
- The Samples index reaches its optimum performance for q between 5 and 6, depending on the case. Unlike Myers', this index works better on English text than on DNA. In DNA it produces a small index (4 times smaller than Myers') but in general has worse search time. The index for $q = 5$ on English text is half the size of Myers' index, and it also obtains good results for medium patterns and low error levels.
- Dfs(p), which works on the same data structure of Dfs(a), improves over it when the patterns are not very short and the error level is not too high. When applicable, its query time is by far the lowest among all the indices.

5 Conclusions and Future Work

We have presented a new indexing scheme for approximate string matching. The main idea is to split the pattern in pieces to be searched with less errors, and use a suffix tree to find their approximate matches in the text. Later, we verify all their matches for an occurrence of the complete pattern. The splitting technique balances between traversing too many nodes of the suffix tree and verifying too many text positions. We have also shown how to traverse the suffix tree efficiently in practice. We have proved analytically that the resulting index has sublinear retrieval time (of the form $O(n^\lambda)$, where $0 < \lambda < 1$ if the error level is moderate). Finally, we have presented the first (as far as we know) experimental results that compare the different implemented indexing schemes, which show that the proposed idea improves over all the previously implemented approaches.

Index	k	DNA ($m=10$)	H-DNA ($m=10$)	FRA ($m=8$)	H-FRA ($m=8$)
On-line	1	*131.0/21.35*	*55.01/15.24*	*59.74/17.31*	*29.99/9.00*
	2	*152.6/20.56*	*62.41/15.48*	*114.8/20.86*	*52.77/11.56*
	3	*188.7/20.36*	*84.20/15.33*	*142.2/20.56*	*60.30/13.76*
Myers'	1	0.29/1.74	0.64/2.15	7.04/8.04	6.17/7.29
	2	0.97/2.18	1.53/2.74	23.5/21.4	20.2/18.2
	3	6.29/6.79	8.17/8.10	22.4/20.8	20.9/18.5
Samples (4)	1	1.80/6.66	1.72/5.48	0.75/2.01	0.75/1.76
	2	9.33/26.7	9.10/23.4	3.30/9.54	2.69/2.68
	3	30.7/93.4	25.5/73.6	13.8/30.3	13.6/26.7
Samples (5)	1	0.91/2.81	0.93/2.38	0.75/1.91	0.77/1.74
	2	9.88/27.7	9.35/23.5	4.92/10.4	3.47/7.07
	3	36.4/97.2	30.9/77.3	23.9/38.9	21.5/33.5
Samples (6)	1	0.90/2.71	0.93/2.35	0.89/2.06	0.86/1.82
	2	11.3/29.4	10.9/24.6	6.81/12.5	4.81/8.99
	3	57.3/119	49.0/92.5	39.3/52.8	38.9/47.7
Cobbs'	1	0.83/1.98	1.85/3.67	n/a	n/a
	2	3.85/14.9	6.04/19.1		
	3	17.9/84.5	21.8/79.3		
Dfs(a)	1	**0.05/0.15**	**0.05/0.04**	**0.10/0.25**	**0.09/0.07**
	2	**0.88/2.72**	**0.71/0.57**	**0.37/0.96**	**0.27/0.22**
	3	5.53/16.9	4.69/3.96	1.51/4.39	**1.01/0.82**

Table 3. Query time for short patterns and for 1, 2 and 3 errors. The on-line algorithm shows time in milliseconds in the format "user/system", in italics. The indexed algorithms show the fraction they take of the time of the on-line algorithm. The format is "a/b", where a considers only user time and b considers both. The fastest indexed times are in boldface.

A remaining problem is that the suffix tree data structure needs too much space. We plan to replace it by a suffix array [21]. The suffix array contains the leaves of the suffix tree in left-to-right order, or equivalently the pointers to all the text suffixes in lexicographical order. The space requirement is in practice 4 times the text size, which is reasonable. Suffix tree nodes (i.e. subtrees) correspond to suffix array intervals. Any movement in the suffix tree can be simulated in $O(\log n)$ time in the suffix array, and therefore the final complexity is multiplied by $O(\log n)$ and the condition for time sublinearity is not affected. Finally, we are still free to use the j value we like (unlike q-gram indices, which are limited by q). In particular, we can easily implement specialized pattern partitioning approaches for biased texts as in [7], where the partitioning minimizes the total number of text positions to verify.

Index	k	DNA ($m=20$)	H-DNA ($m=20$)	FRA ($m=16$)	H-FRA ($m=16$)
On-line	2	*184.6/22.18*	*75.16/16.61*	*60.59/17.56*	*29.91/9.48*
	4	*311.4/21.70*	*116.0/15.79*	*116.3/20.83*	*50.71/14.98*
	6	*779.2/21.42*	*297.4/15.77*	*205.6/20.58*	*92.36/13.37*
Myers	2	0.67/1.69	0.91/1.97	7.03/8.06	10.9/10.9
	4	5.13/5.50	5.61/5.74	32.7/29.2	31.9/26.3
	6	16.9/16.8	17.7/17.3	26.5/25.0	25.2/23.1
Samples (4)	2	1.55/5.10	1.60/4.55	0.44/0.95	0.63/1.03
	4	6.14/13.4	6.16/12.4	2.08/4.62	2.03/4.01
	6	9.10/25.4	9.48/27.9	9.59/18.6	8.85/16.1
Samples (5)	2	0.60/1.93	0.64/1.73	0.38/0.75	0.62/0.91
	4	5.26/11.3	5.77/11.9	2.21/4.87	2.15/4.19
	6	10.0/25.5	10.7/26.4	14.8/23.6	12.7/19.6
Samples (6)	2	0.31/0.83	0.41/0.84	0.39/0.70	0.60/0.91
	4	5.61/11.7	6.18/12.1	2.71/5.13	2.51/4.42
	6	15.2/31.5	15.3/30.8	22.9/31.1	19.3/25.3
Cobbs'	2	3.93/11.7	6.60/16.0	n/a	n/a
	4	***	69.5/171		
	6	***	***		
Dfs(a)	2	0.31/1.19	0.32/0.26	0.59/1.49	0.45/0.34
	4	6.39/30.8	4.31/3.79	4.15/14.4	2.49/1.93
	6	14.6/64.9	7.76/7.37	10.7/42.0	5.87/5.13
Dfs(p)	2	**0.09/0.23**	**0.08/0.07**	**0.22/0.43**	**0.18/0.13**
	4	3.24/6.42	2.63/2.32	**0.76/1.92**	**0.61/0.47**
	6	11.3/12.6	7.76/7.37	2.26/6.05	1.59/1.38

*** One single query took more than 2 hours of elapsed time.

Table 4. Query time for medium patterns and for $k = 2$, 4 and 6. The on-line algorithm shows time in milliseconds in the format "user/system", in italics. The indexed algorithms show the fraction they take of the time of the on-line algorithm. The format is "a/b", where a considers only user time and b considers both. The fastest indexed times are in boldface.

Acknowledgements

We thank the nice comments of two referees, which helped to improve this work. We also thank Erkki Sutinen for his code to build the suffix tree, and Gene Myers and Archie Cobbs for sending us their implemented indices.

References

1. A. Apostolico and Z. Galil. *Combinatorial Algorithms on Words*. Springer-Verlag, New York, 1985.
2. M. Araújo, G. Navarro, and N. Ziviani. Large text searching allowing errors. In *Proc. WSP'97*, pages 2–20. Carleton University Press, 1997.

3. R. Baeza-Yates. Text retrieval: Theory and practice. In *12th IFIP World Computer Congress*, volume I, pages 465–476. Elsevier Science, September 1992.
4. R. Baeza-Yates and G. Gonnet. All-against-all sequence matching. Dept. of Computer Science, University of Chile, 1990.
5. R. Baeza-Yates and G. Gonnet. Fast text searching for regular expressions or automaton searching on a trie. *J. of the ACM*, 43, 1996.
6. R. Baeza-Yates and G. Navarro. Block-addressing indices for approximate text retrieval. In *Proc. ACM CIKM'97*, pages 1–8, 1997.
7. R. Baeza-Yates and G. Navarro. A practical q-gram index for text retrieval allowing errors. *CLEI Electronic Journal*, 1(2), 1998. `http://www.clei.cl`.
8. R. Baeza-Yates and G. Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999. Preliminary version in *Proc. CPM'96, LNCS 1075.*
9. R. Baeza-Yates and C. Perleberg. Fast and practical approximate pattern matching. *Information Processing Letters*, 59:21–27, 1996.
10. A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. Chen, and J. Seiferas. The samllest automaton recognizing the subwords of a text. *Theoretical Computer Science*, 40:31–55, 1985.
11. W. Chang and J. Lampe. Theoretical and empirical comparisons of approximate string matching algorithms. In *Proc. CPM'92*, LNCS 644, pages 172–181, 1992.
12. W. Chang and T. Marr. Approximate string matching and local similarity. In *Proc. CPM'94*, LNCS 807, pages 259–273, 1994.
13. A. Cobbs. Fast approximate matching using suffix trees. In *Proc. CPM'95*, pages 41–54, 1995. LNCS 937.
14. M. Crochemore. Transducers and repetitions. *Theoretical Computer Science*, 45:63–86, 1986.
15. M. Farach, P. Ferragina, and S. Muthukrishnan. Overcoming the memory bottleneck in suffix tree construction. In *Proc. SODA'98*, pages 174–183, 1998.
16. Z. Galil and K. Park. An improved algorithm for approximate string matching. *SIAM J. on Computing*, 19(6):989–999, 1990.
17. G. Gonnet. A tutorial introduction to Computational Biochemistry using Darwin. Technical report, Informatik E.T.H., Zuerich, Switzerland, 1992.
18. P. Jokinen and E. Ukkonen. Two algorithms for approximate string matching in static texts. In *Proc. MFCS'91*, volume 16, pages 240–248. Springer-Verlag, 1991.
19. D. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, 1973.
20. G. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *J. of Algorithms*, 10:157–169, 1989.
21. U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. In *Proc. ACM-SIAM SODA'90*, pages 319–327, 1990.
22. U. Manber and S. Wu. GLIMPSE: A tool to search through entire file systems. In *Proc. USENIX Technical Conference*, pages 23–32, Winter 1994.
23. E. Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica*, 12(4/5):345–374, Oct/Nov 1994.
24. G. Myers. A fast bit-vector algorithm for approximate pattern matching based on dynamic progamming. In *Proc. CPM'98*, LNCS 1448, pages 1–13, 1998.
25. G. Navarro. *Approximate Text Searching.* PhD thesis, Dept. of Computer Science, Univ. of Chile, December 1998. Technical Report TR/DCC-98-14. `ftp://-ftp.dcc.uchile.cl/pub/users/gnavarro/thesis98.ps.gz`.
26. G. Navarro and R. Baeza-Yates. Improving an algorithm for approximate pattern matching. Technical Report TR/DCC-98-5, Dept. of Computer Science, Univ. of Chile, 1998. Submitted.

27. S. Needleman and C. Wunsch. A general method applicable to the search for similarities in the amino acid sequences of two proteins. *J. of Molecular Biology*, 48:444–453, 1970.
28. P. Sellers. The theory and computation of evolutionary distances: pattern recognition. *J. of Algorithms*, 1:359–373, 1980.
29. F. Shi. Fast approximate string matching with q-blocks sequences. In *Proc. WSP'96*, pages 257–271. Carleton University Press, 1996.
30. E. Sutinen and J. Tarhio. On using q-gram locations in approximate string matching. In *Proc. ESA'95*, LNCS 979, pages 327–340, 1995.
31. E. Sutinen and J. Tarhio. Filtration with q-samples in approximate string matching. In *Proc. CPM'96*, LNCS 1075, pages 50–61, 1996.
32. J. Tarhio and E. Ukkonen. Approximate Boyer-Moore string matching. *SIAM J. on Computing*, 22(2):243–260, 1993.
33. E. Ukkonen. Approximate string matching over suffix trees. In *Proc. CPM'93*, pages 228–242, 1993.
34. Esko Ukkonen. Finding approximate patterns in strings. *J. of Algorithms*, 6:132–137, 1985.
35. S. Wu and U. Manber. Fast text searching allowing errors. *Comm. of the ACM*, 35(10):83–91, October 1992.
36. S. Wu, U. Manber, and E. Myers. A sub-quadratic algorithm for approximate limited expression matching. *Algorithmica*, 15(1):50–67, 1996.

Appendix: Probability of Reaching a Suffix Tree Node

We need to determine which is the probability of the automaton being active at a given node of depth ℓ in the suffix tree. Notice that the automaton is active if and only if some state of the last row is active (recall Figure 1). This is equivalent to some *prefix* of the pattern matching with k errors or less the text substring represented by the suffix tree node under consideration.

We are therefore interested in the probability of a pattern prefix of length m' matching a text substring of length ℓ. This analysis is an extension of that of [8]. As Figure 7 illustrates, at least $\ell - k$ text characters text must match the pattern when $\ell \geq m'$, and at least $m' - k$ pattern characters must match the text whenever $m' \geq \ell$. Hence, the probability of matching is upper bounded by

$$\frac{1}{\sigma^{\ell-k}} \binom{\ell}{\ell-k} \binom{m'}{\ell-k} \qquad \text{or} \qquad \frac{1}{\sigma^{m'-k}} \binom{\ell}{m'-k} \binom{m'}{m'-k}$$

depending on whether $\ell \geq m'$ or $m' \geq \ell$, respectively (the combinatorials count all the possible locations for the matching characters in both strings). Notice that this imposes that $m' - k \leq \ell \leq m' + k$. We also assume $m' \geq k$, since otherwise the matching probability is 1. Since $k \leq m' \leq m$, we have that $\ell \leq m + k$, otherwise the matching probability is zero. Hence the matching probability is 1 for $\ell \leq k$ and 0 for $\ell > m + k$, and we are interested in what happens in between.

Since we are interested in any pattern prefix matching the current text substring, we add up all the possible lengths from k to m:

$$\sum_{m'=k}^{\ell} \frac{1}{\sigma^{\ell-k}} \binom{\ell}{\ell-k} \binom{m'}{\ell-k} + \sum_{m'=\ell+1}^{m} \frac{1}{\sigma^{m'-k}} \binom{\ell}{m'-k} \binom{m'}{m'-k}$$

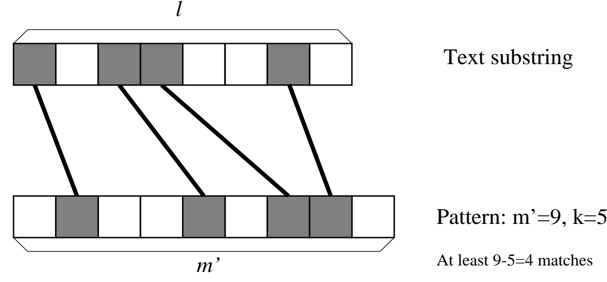


Fig. 7. Upper bound for the probability of matching. At least $\max(m'-k, \ell-k)$ characters must match, since otherwise it would not be possible to convert one string into the other.

In the analysis that follows, we call $\beta = k/\ell$, where $\alpha/(1+\alpha) \leq \beta \leq 1$. We will prove that, after some depth ℓ in the suffix tree, the matching probability is $O(\gamma(\beta)^\ell)$, for some $\gamma(\beta) < 1$. We begin with the first summation. We analyze its largest term (the last one), which is

$$\frac{1}{\sigma^{\ell-k}} \binom{\ell}{k}^2$$

and by using Stirling's approximation $x! = (x/e)^x \sqrt{2\pi x}(1 + O(1/x))$ we have

$$\frac{1}{\sigma^{\ell-k}} \left(\frac{\ell^\ell \sqrt{2\pi\ell}}{k^k (\ell-k)^{\ell-k} \sqrt{2\pi k}\sqrt{2\pi(\ell-k)}} \right)^2 \left(1 + O\left(\frac{1}{\ell}\right)\right)$$

which is

$$\left(\frac{1}{\sigma^{1-\beta}\beta^{2\beta}(1-\beta)^{2(1-\beta)}} \right)^\ell \ \ell^{-1} \ \left(\frac{1}{2\pi\beta(1-\beta)} + O\left(\frac{1}{\ell}\right)\right)$$

where the last step is done using Stirling's approximation to the factorial. This formula is of the form $\gamma(\beta)^\ell \ O(1/\ell)$, where we define

$$\gamma(x) \quad = \quad \frac{1}{\sigma^{1-x} x^{2x} (1-x)^{2(1-x)}} \tag{1}$$

The whole first summation is bounded by $\ell - k$ times the last term, which gives $(\ell-k)\gamma(\beta)^\ell O(1/\ell) = O(\gamma(\beta)^\ell)$. Therefore the first summation is exponentially decreasing with ℓ if and only if $\gamma(\beta) < 1$, i.e.

$$\sigma \quad > \quad \left(\frac{1}{\beta^{2\beta}(1-\beta)^{2(1-\beta)}} \right)^{\frac{1}{1-\beta}} \quad = \quad \frac{1}{\beta^{\frac{2\beta}{1-\beta}}(1-\beta)^2} \tag{2}$$

It is easy to show analytically that $e^{-1} \leq \beta^{\frac{\beta}{1-\beta}} \leq 1$ if $0 \leq \beta \leq 1$, so it suffices that $\sigma > e^2/(1-\beta)^2$, or equivalently

$$\beta \quad < \quad 1 - \frac{e}{\sqrt{\sigma}} \tag{3}$$

is a sufficient condition for the largest (last) term to be $O(\gamma(\beta)^\ell)$, as well as the whole first summation.

We address now the second summation, which is more complicated. In this case, it is not clear which is the largest term. We can see each term as

$$\frac{1}{\sigma^r}\binom{\ell}{r}\binom{k+r}{k}$$

where $\ell - k < r \leq m - k$. By considering $r = x\ell$ ($x \in [1-\beta, m/\ell - \beta]$) and applying again Stirling's approximation, we maximize the base of the resulting exponential, which is

$$h(x) = \frac{(x+\beta)^{x+\beta}}{\sigma^x x^{2x}(1-x)^{1-x}\beta^\beta}$$

Elementary calculus leads to solve a second-degree equation that has roots in the interval $[1-\beta,\infty)$ only if $\sigma \leq \beta/(1-\beta)^2$. Since due to Eq. (3) we are only interested in $\sigma \geq 1/(1-\beta)^2$, $\delta h(x)/\delta x$ does not have roots, and the maximum of $h(x)$ is at $x = 1-\beta$. That means $r = \ell - k$, i.e. the first term of the second summation, which is the same largest term of the first summation.

We conclude that the probability of being active at a node of level ℓ is upper bounded by

$$\frac{m-k}{\ell}\,\gamma(\beta)^\ell\,\left(1+O\left(\frac{1}{\ell}\right)\right) \;=\; O\left(\gamma(\beta)^\ell\right)$$

and therefore Eq. (3) is valid for the whole summation. When $\gamma(\beta)$ is 1, the probability is very high: only considering the term $m' = \ell$ we have $\Omega(1/\ell)$.

Hence, the result is that the matching probability is very high for $\beta = k/\ell \geq 1 - e/\sqrt{\sigma}$, and otherwise it is $O(\gamma(\beta)^\ell)$, where $\gamma(\beta) < 1$.

Although the e appeared via a bounding condition, we can see that this bound is tight: we take $\log_\sigma$ on both sides of the condition $\gamma(\beta) < 1$ and get

$$1-\beta+2(\beta\log_\sigma\beta+(1-\beta)\log_\sigma(1-\beta)) \;>\; 0$$

and by replacing $x = 1-\beta$ and using $\ln(1-x) = -x + O(x^2)$ we have

$$x\ln\sigma + 2(x\ln x - (1-x)(x+O(x^2))) \;=\; x\ln\sigma + 2x\ln x - 2x + O(x^2) \;>\; 0$$

from where divide by x to obtain

$$x \;>\; \frac{e}{\sqrt{\sigma}}\,e^{O(x)} \;=\; \frac{e}{\sqrt{\sigma}}\,(1+O(x)) \;=\; \frac{e}{\sqrt{\sigma}}\,(1+O(1/\sqrt{\sigma}))$$

We conclude that the precise limit for $\beta = 1-x$ is

$$\beta \;<\; 1-\frac{e}{\sqrt{\sigma}} \;+\; O(1/\sigma)$$

As we show experimentally in [8], however, the real β limit is very close to the same formula if e is replaced by $c = 1.09$. The reason is that the bounding condition (Figure 7) we use is not strong enough: for instance, we could avoid replacements in the edit distance and the bound would be the same. In the paper we use a limit of the form $\beta = 1 - c/\sqrt{\sigma}$, knowing that we can prove $c \leq e$ but in practice it holds $c \approx 1$.