A Hybrid Indexing Method for Approximate String Matching

GONZALO NAVARRO[1], *Dept. of Computer Science, University of Chile. Blanco Encalada 2120, Santiago, Chile.*
gnavarro@dcc.uchile.cl

RICARDO BAEZA-YATES[1], *Dept. of Computer Science, University of Chile. Blanco Encalada 2120, Santiago, Chile.*
rbaeza@dcc.uchile.cl

ABSTRACT: We present a new indexing method for the approximate string matching problem. The method is based on a suffix array combined with a partitioning of the pattern. We analyze the resulting algorithm and show that the average retrieval time is $O(n^{\lambda} \log n)$, for some $\lambda > 0$ that depends on the error fraction tolerated α and the alphabet size σ. It is shown that $\lambda < 1$ for approximately $\alpha < 1 - e/\sqrt{\sigma}$, where $e = 2.718...$. The space required is four times the text size, which is quite moderate for this problem. We experimentally show that this index can outperform by far all the existing alternatives for indexed approximate searching. These are also the first experiments that compare the different existing schemes.

Keywords: Suffix tries, suffix trees, text searching allowing errors, text indexing, computational biology.

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* (ed) 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 $O(mn)$ worst case time [33]. A number of algorithms improved later this result [26]. The lower bound of the on-line problem (proved and reached in [10]) is $O(n(k + \log_\sigma m)/m)$ average search time, where σ is the size of the alphabet Σ. This is of course $\Omega(n)$ for constant m.

If the text is large even the fastest on-line algorithms are not practical, and prepro-

[1]This work has been supported in part by Fondecyt grants 1-990627 and 1-000929.

cessing 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 [40, 3].

Despite some progress in the last years, the indexing schemes for this problem are still rather immature. The existing indexes do not perform well in practice. Some of them, based on suffix trees, are impractically large (12 to 70 times the text size), do not behave well in secondary memory, and suffer from an exponential dependence on the pattern length or on k in their search time. Others, based on filtration, have reasonable space requirements but permit only a very low error level α at search time.

We present an index which is practical and can be considered as a hybrid between the two approaches mentioned above. The main novelty is that it uses a suffix tree-like approach but, to avoid the exponential dependence on m or k, the search pattern is split in smaller subpatterns and their occurrences are later integrated. It is shown that this is an intermediate option between the extremes represented by the two previous approaches and that the optimum search time is not on these extremes but in between. We show analytically that the average search time obtained using the optimal partitioning scheme is $O(n^\lambda \log n)$, where $\lambda < 1$ when $\alpha < 1 - e/\sqrt{\sigma}$, where $e = 2.718...$[2].

On the practical side, we replace the suffix tree by a suffix array, which takes only four times the text size (this idea is not new) and use a novel node processing algorithm. We show experimentally that the search time on the suffix array is in practice slower than using the suffix tree only in some cases, for very short texts, and by a small percentage. On longer texts, on the other hand, the suffix array becomes much superior. We show experimentally that the hybrid partition scheme is better than the two extremes. Finally, we compare our index against the other implemented indexes, showing that it can be much faster than any alternative approach. In particular, an approach closer to ours is Myers' index [24], and we show experimentally that each index is faster than the other in different cases. This constitutes the first experimental comparison that includes most of the implemented indexes for the problem.

This paper is organized as follows. Section 2 covers related work in more detail and puts our results in context. Section 3 explains the basic techniques our approach builds on. Section 4 presents the core of our contribution. In Section 5 we analyze its average search time. Section 6 presents all the experimental results and Section 7 our conclusions. This paper is an extended and improved version of [29].

2 Previous Work and Our Contribution

There are two types of indexing mechanisms for approximate string matching, which we call "word-retrieving" and "sequence-retrieving". Word retrieving indexes [23, 8, 2] are more oriented to natural language text and information retrieval. They can retrieve every *word* whose edit distance to the pattern *word* 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"`, if we allow one error.

[2] All the average case results of this paper assume that the text is random and that the letters are independently and uniformly distributed.

These indexes are more mature, but their restriction can be unacceptable in some applications, especially where there are no words (as in DNA), where the concept of word is difficult to define (as in oriental languages) or in agglutinating languages such as Finnish.

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

A first type is based on simulating a sequential algorithm, running it on the suffix tree [21, 1, 38] or DAWG (directed acyclic word graph) [12, 9] of the text instead of the text itself. A *suffix trie* is a trie where all the suffixes of the text string have been inserted. A *suffix tree* achieves $O(n)$ worst-case space and construction time by compressing unary paths of the suffix trie. A DAWG is the minimal automaton that recognizes all the substrings of the text and is obtained by compressing the suffix tree via identifying all the final states.

Since every different substring in the text is represented by a single node in the suffix tree or the DAWG, it is possible to avoid redoing the same work when the text has repetitions. Those indexes take $O(n)$ space and construction time, but their construction is not optimized for secondary memory and they are very inefficient in this case (see, however, [13]). Moreover, the structure is very inefficient in space requirements, since it takes 12 to 70 times the text size (see, e.g. [14]).

In [15, 6], a limited depth-first search (DFS) technique on the suffix tree was introduced. Since every substring of the text (that is, every potential pattern occurrence) can be found by descending 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 was analyzed in [6]. With an additional $O(\log n)$ time factor, it also runs on suffix arrays [22, 16], which take four times the text size instead of (at least) twelve [3].

In [20, 37, 11], on the other hand, more sophisticated algorithms were presented that traverse the least possible nodes in the suffix tree (or in the DAWG). The idea is to traverse all the different tree nodes that represent "viable prefixes", which are minimal text substrings that can be prefixes of an approximate occurrence of the pattern. Compared to the simple DFS approach, this one avoids entering so deep in the suffix tree, but its node processing is more expensive and cannot be implemented on the cheaper suffix array.

In all cases the search time of these algorithms is asymptotically independent on n but exponentially dependent on m or k, e.g. $O(\min(3^m, (2\sigma m)^k))$ in the average and worst cases [37].

The second type of sequence-retrieving indexes is based on adapting an on-line filtering algorithm. Filters are fast algorithms that discard large parts of the text checking for a necessary condition (simpler than the matching condition). Most existing filters are based in finding substrings of the pattern without errors, and checking for potential occurrences around those matches. The index is used to quickly find those pattern substrings, and is in most cases based on storing some text q-grams (substrings of

[3] These figures come from considering that the suffix array needs $n \log_2 n$ bits, which is limited by $4n$ bytes if $n \leq 4$ Gb. The text needs $n \log_2 \sigma$ bits and it is customary to use one byte per character (that is, $\sigma \leq 256$). The suffix tree needs at least $3n \log_2 n$ bits. In this paper we use the convention that characters take 1 byte and pointers and integers take 4 bytes.

length q) and their positions in the text.

Different filtration indexes [20, 24, 18, 36, 34, 28] 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, their alignment conditions, etc. Depending on this and on q they achieve different space-time tradeoffs. In general, filtration indexes need $O(n)$ space but they are much smaller than suffix trees (1 to 10 times the text size), and can also be built in linear time. The price is that they perform badly for medium and large error level α. Different analyses show that these indexes can achieve sublinear search time on average only for $\alpha = O(1/\log_\sigma n)$.

Somewhat outside the previous classification is [24], 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, the pieces are searched and their occurrences used to assemble the final answers. The analysis shows that the average search time is $O(kn^\lambda \log n)$ for $\lambda = \log_\sigma((c+1)/(c-1)) + \alpha \log_\sigma c + \alpha$, where $c = 1/\alpha + \sqrt{1 + 1/\alpha^2}$. This is sublinear when $\lambda < 1$, which puts an upper bound on α (which can be numerically computed for each σ).

In this paper we study more in depth the technique of reducing the problem to approximate searching of pattern pieces. We show that this is essentially a hybrid between the extremes of suffix tree traversal and filtering by finding exact pattern pieces. Moreover, we show that the optimal reduction scheme is between both extremes, namely partitioning the pattern in $\Theta(m/\log_\sigma n)$ pieces. 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). Unlike [24], which fixes the partitioning at indexing time because of the constraints of q, we propose an index than can handle any pattern length (conceptually, a suffix tree) and therefore it can select the best partition at query time. We show analytically that the average search 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 $\alpha < 1 - e/\sqrt{\sigma}$, where $e = 2.718...$. On the other hand, the results of [7, 25] show that sublinearity cannot be achieved for $\alpha \geq 1 - e/\sqrt{\sigma}$.

We implement the index using a suffix array instead of a suffix tree. The suffix array takes only 4 times the text size and multiplies the above search cost only by $O(\log n)$. We also use a faster node processing algorithm based on previous work [7] which is especially well suited for this case.

Ours and Myers' can be considered as a third class of algorithms, based on reducing the problem to approximate search of pattern pieces (an intermediate between the two extremes of pure suffix tree searching and partitioning into exact searching of pattern pieces). A very recent work in this line is [31], although they show no analysis and their comparison against previous work shows that our search times are superior (albeit they need less space).

3 Basics

We present in this section the basic algorithms on which our approach builds.

3.1 Computing Edit Distance

We start with the classical algorithm to compute the edit distance (ed) between two strings and later show how it is extended for online text searching allowing errors [33].

The algorithm is based on dynamic programming. Imagine that we need to compute $ed(x, y)$. A matrix $C_{0\ldots|x|,0\ldots|y|}$ is filled, where $C_{j,i}$ represents the minimum number of operations needed to match $x_{1\ldots j}$ to $y_{1\ldots i}$. This is computed as follows

$$\begin{aligned} C_{j,0} &= j, \quad C_{0,i} = i \\ C_{j,i} &= \text{if } (x_j = y_i) \text{ then } C_{j-1,i-1} \\ &\quad \text{else } 1 + \min(C_{j-1,i}, C_{j,i-1}, C_{j-1,i-1}) \end{aligned}$$

where at the end $C_{|x|,|y|} = ed(x, y)$. The rationale of the above formula is as follows. First, $C_{j,0}$ and $C_{0,i}$ represent the edit distance between a string of length j or i and the empty string. Clearly j (respectively i) deletions are needed on the non-empty string. For two non-empty strings of length j and i, we assume inductively that all the edit distances between shorter strings have already been computed, and try to convert $x_{1\ldots j}$ into $y_{1\ldots i}$.

Consider the last characters x_j and y_i. If they are equal, then we do not need to consider them and we proceed in the best possible way to convert $x_{1\ldots j-1}$ into $y_{1\ldots i-1}$. On the other hand, if they are not equal, we must deal with them in some way. Following the three allowed operations, we can delete x_j and convert in the best way $x_{1\ldots j-1}$ into $y_{1\ldots i}$, insert y_i at the end of $x_{1\ldots j}$ and convert in the best way $x_{1\ldots j}$ into $y_{1\ldots i-1}$, or replace x_j by y_i and convert in the best way $x_{1\ldots j-1}$ into $y_{1\ldots i-1}$. In all cases, the cost is 1 plus the cost for the rest of the process (already computed). Notice that the insertions in one string are equivalent to deletions in the other. Figure 1 (left) illustrates this algorithm to compute $ed($`"survey"`, `"surgery"`$)$.

		s	u	r	g	e	r	y
	0	1	2	3	4	5	6	7
s	1	0	1	2	3	4	5	6
u	2	1	0	1	2	3	4	5
r	3	2	1	0	1	2	3	4
v	4	3	2	1	1	2	3	4
e	5	4	3	2	2	1	2	3
y	6	5	4	3	3	2	2	**2**

		s	u	r	g	e	r	y
	0	0	0	0	0	0	0	0
s	1	0	1	1	1	1	1	1
u	2	1	0	1	2	2	2	2
r	3	2	1	0	1	2	2	3
v	4	3	2	1	1	2	3	3
e	5	4	3	2	2	1	2	3
y	6	5	4	3	3	**2**	**2**	**2**

FIG. 1: On the left, the dynamic programming algorithm to compute the edit distance between `"survey"` and `"surgery"`. The bold entry shows the final result. On the right, the variation to search `"survey"` in the text `"surgery"`. Bold entries indicate matching text positions when $k = 2$.

Therefore, the algorithm is $O(|x||y|)$ time in the worst and average case. However, the space required is only $O(\min(|x|, |y|))$. This is because, by a column-wise processing, only the previous column must be stored in order to compute the new one, and therefore we just keep one column and update it. We can process the matrix row-wise or column-wise so that the space requirement is minimized.

We show now how to adapt this algorithm to search a short pattern P in a long text T. The algorithm is basically the same, with $x = P$ and $y = T$ (proceeding column-wise so that $O(m)$ space is required). The only difference is that we must allow that any text position is the potential start of a match. This is achieved by setting $C_{0,i} = 0$ for all $i \in 0 \ldots n$. That is, the empty pattern matches with zero errors at any text position (because it matches with a text substring of length zero).

The algorithm then initializes its column $C_{0 \ldots m}$ with the values $C_j = j$, and processes the text character by character. At each new text character T_i, its column vector is updated to $C'_{0 \ldots m}$. The update formula is

$$\begin{aligned} C'_j &= \text{if } (P_j = T_i) \text{ then } C_{j-1} \\ &\quad\ \text{else } 1 + \min(C'_{j-1}, C_j, C_{j-1}) \end{aligned}$$

and the text positions where $C_m \leq k$ are reported.

The search time of this algorithm is $O(mn)$ and its space requirement is $O(m)$. Figure 1 (right) exemplifies.

3.1.1 A Bit-parallel Simulated NFA

An alternative and very useful way to consider the problem is to model the search with a non-deterministic automaton (NFA) [39, 40, 4, 7].

Consider the NFA for $k = 2$ errors under edit distance shown in Figure 2. Every row denotes the number of errors seen (the first row zero, the second row one, etc.). 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 ε-transitions, since we advance in the pattern without advancing in the text). The initial self-loop allows a match to start anywhere in the text. The automaton signals (the end of) a match whenever a rightmost state is active. Without the initial self-loop, the automaton computes edit distance.

It is not hard to see that once a state in the automaton is active, all the states of the same column and higher rows are active too. Moreover, at a given text position, if we collect the smallest active rows at each column, we obtain the vertical vector of the dynamic programming algorithm (in this case $\{0, 1, 2, 3, 3, 3, 2\}$, compare to right table of Figure 1).

This NFA is implemented in [7] using "bit-parallelism": the states of the NFA are mapped to bits in a computer word, and all the updates required to all the states when

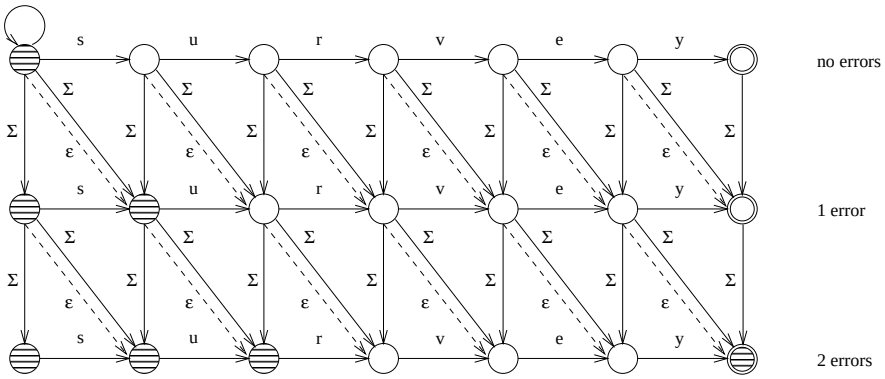


FIG. 2: An NFA for approximate string matching of the pattern `"survey"` with two errors. The shaded states are those active after reading the text `"surgery"`.

a new text character is read are performed in $O(1)$ operations on the whole machine word. The simulation needs $(m-k)(k+2)$ bits for the simulation, since only the full diagnoals need to be represented. The first full diagonal is furthermore excluded because the initial self-loop makes it always active. If those bits do not fit in a single computer word then the cost to update the NFA is $O((m-k)(k+2)/w)$ per text character, where w is the number of bits in the machine word. This yields a total worst case time of $O(mkn/w)$, which is very fast for short patterns. The reader is referred to the original paper for the details of the simulation.

3.2 DFS over Suffix Trees

The number of strings that match a pattern P with at most k errors is finite. This is immediately clear if we see that the length of any such string must be between $m-k$ and $m+k$, since otherwise more than k deletions or insertions would be necessary to convert one into the other. We call this set of strings the "k-neighborhood" of P, and denote it

$$U_k(P) = \{x \in \Sigma^*,\ ed(x,P) \leq k\}$$

The idea of this approach is, in essence, to generate all the strings in the neighborhood of P and search them in the text (without errors). It is clear that the answer is the set of all positions where the strings in $U_k(P)$ appear. Each such string can be found by using an exact search technique, as done in [24], or with a more sophisticated technique, as explained shortly.

The main problem with this approach is that $U_k(P)$ is quite large. For instance, U_1(`"hello"`) is

$$\{\,\texttt{ello},\ \texttt{hllo},\ \texttt{helo},\ \texttt{hell}\,\} \cup \bigcup_{x\in\Sigma} \{\,x\texttt{ello},\ \texttt{h}x\texttt{llo},\ \texttt{he}x\texttt{lo},\ \texttt{hel}x\texttt{o},\ \texttt{hell}x\,\} \cup$$

$$\bigcup_{x \in \Sigma} \{\ x\texttt{hello},\ \texttt{h}x\texttt{ello},\ \texttt{he}x\texttt{llo},\ \texttt{hell}x\texttt{o},\ \texttt{hello}x\ \}$$

An analysis in [39] shows that, by considering the number of different sequences of k edit operations that can be performed over P, the number of different resulting elements can be upper bounded by

$$|U_k(P)| \ \leq\ \frac{12}{5}(m+1)^k(\sigma+1)^k \ \ =\ \ O(m^k\sigma^k)$$

which shows an exponential growth with k. A slightly more elaborated upper bound is given in [24], where "maximal" neighborhood elements, i.e. neighborhood elements that are not prefixes of others, are considered. We call this set $U_k^t(P) \subseteq U_k(P)$. An upper bound for $|U_k^t(P)|$ is obtained by working on the recurrence that defines an approximate occurrence (Section 3.1). The result is shown to be

$$|U_k^t(P)| \ \leq\ \frac{c^{k+1}(c+1)^m}{(c-1)^{m+1}}\,\sigma^k$$

for any $c > 1$, a formula that is minimized for $c \approx 1/\alpha + \sqrt{1+1/\alpha^2}$ (recall that $\alpha = k/m$). This is still exponential on k.

Both bounds show that this approach works well for small m and k values. Otherwise the number of elements to search makes the problem intractable.

However, searching all the strings one by one is not the smartest choice. A more clever search strategy is possible if we use the suffix array (or tree) as the data structure to implement this search [15, 6, 37]. Since every substring of the text (i.e. every potential occurrence) can be found by traversing the suffix tree from the root, 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 cannot be the beginning of a string in $U_k(P)$.

We explain now the detailed algorithm (on a suffix trie for simplicity), although it is not hard to adapt it to the suffix tree or array. We choose an algorithm that determines the edit distance between P and any other string x. The algorithm must be able to: (a) consider the string x incrementally, (b) determine when $ed(P,x) \leq k$, and (c) determine when $ed(P,xy) > k$ for any y. The state of the algorithm is initialized and we start at the root of the tree. Now, we descend recursively by every branch of the suffix tree. When we descend by a branch labeled by the letter a, the comparison algorithm adds a to the current string x. If the algorithm determines that $ed(P,x) \leq k$, we report all the leaves of the current subtree as answers. If, on the other hand, the algorithm determines that $ed(P,xy) > k$ for any string y, we abandon immediately that branch. Otherwise, we continue recursively descending in the suffix tree.

The algorithm is shown in Figure 3. Recall that each suffix tree node N corresponds to a different text substring x. At each invocation S reflects the result of comparing P against the text substring x represented by N. The first `"if"` corresponds to action (b) in the comparison algorithm, while the second corresponds to action (c). The operation $Update$ corresponds to action (a), where the letter a is added to the current string x corresponding to state S. Notice that the algorithm is recursive and a set of states S is stacked along the execution. This stack has height at most $m+k$.

```
Search (Suffix Tree Node N, Search State S)

   if (S implies a match between P and N)
      Report all the leaves below N
   else if (S implies that N can be extended to match P)
           for each tree edge N → N' labeled a
              Search (N', Update(S, a))
```

FIG. 3. The algorithm to find the neighborhood of a pattern using the suffix tree.

In [15, 6] the comparison algorithm used is the dynamic programming algorithm we have presented in Section 3.1. Each new character of x corresponds to a new column in the matrix, and the state S is simply the last column processed ($O(m)$ space). Adding a new letter (action (a)) can be done by updating the last column S, in $O(m)$ time. A match is detected (action (b)) when the last element of the column S is $\leq k$. Also, it is known that x cannot be extended to match P (action (c)) when all the values of the last column are $> k$.

Figure 4 illustrates the process. We show the path that spells out the string `"surgery"`. The matrix can be seen now as a stack (that grows to the right). With $k = 2$ the backtracking ends indeed after reading `"surge"` since that string matches the pattern (action (b)). If we had instead $k = 1$ the search would have been pruned (action (c)) after considering `"surger"`, and in the alternative path shown, after considering `"surga"`, since in both cases no entry of the matrix has a 1 or 0.

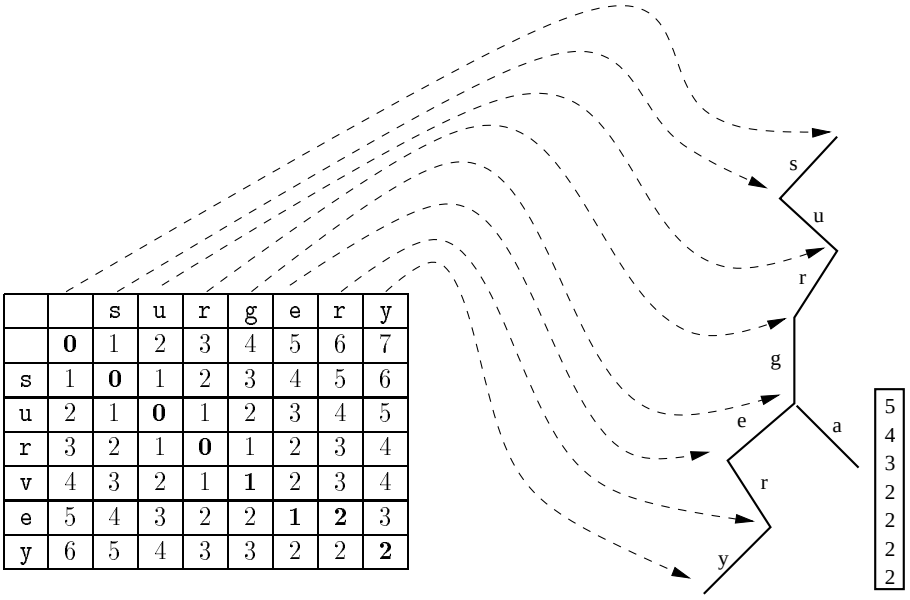
Yet other even more sophisticated traversal techniques are possible [37, 11] but, as we show later, they are not better in practice.

3.3 Filtration Techniques

Each approximate match of a pattern contains some pattern substrings that match without errors. This means that it is possible to derive sufficient conditions for an approximate match based on exact matching of one or more carefully selected pattern pieces. The text can be scanned for the exact occurrences of the pattern pieces selected, and the text areas surrounding those occurrences can be verified for an approximate occurrence of the complete pattern. This has been well known for a long time in on-line algorithms, giving rise to the so-called "filtration" techniques [26].

The application of the idea to indexed searching is as follows: some kind of index is used to help locate quickly the *exact* occurrences of the selected pattern pieces, and once their positions in the text are known, a classical on-line algorithm is run on the candidate text areas to check for real occurrences of the pattern. A general limitation of all those methods (on-line and indexed) is due to the nature of the problem: there is always a maximum error ratio α up to where they are useful, as for larger error levels the text areas to verify cover almost all the text.

A general lemma can be used to abstract from the many existing variants of exact partitioning.



		s	u	r	g	e	r	y
	0	1	2	3	4	5	6	7
s	1	**0**	1	2	3	4	5	6
u	2	1	**0**	1	2	3	4	5
r	3	2	1	**0**	1	2	3	4
v	4	3	2	1	**1**	2	3	4
e	5	4	3	2	2	**1**	**2**	3
y	6	5	4	3	3	2	2	**2**

FIG. 4: The dynamic programming algorithm run over the suffix tree. We show just one path and one additional link.

LEMMA 3.1
Let A and B be two strings such that $ed(A, B) \leq k$. Let $A = A_1 x_1 A_2 x_2 ... x_{k+s-1} A_{k+s}$, for strings A_i and x_i and for any $s \geq 1$. Then, at least s strings $A_{i_1} \ldots A_{i_s}$ appear in B. Moreover, their relative distances inside B cannot differ from those in A by more than k.

That is, we can select $k + s$ non-overlapping pieces from A, and at least s of them must appear unaltered in B. This is clear if we consider the sequence of at most k edit operations that convert A into B. As each edit operation can affect at most one of the A_i's, at least s of them must remain unaltered. The extra requirement on relative distances follows by considering that k edit operations cannot produce misalignments larger than k. Figure 5 illustrates the lemma.

The lemma can be used in different ways. In particular, two main branches of algorithms based on it exist, differing essentially in where (pattern or text) are the errors assumed to occur. That is, the branch explored in [34, 28] consider $P = A$ and $T' = B$ (where T' is an occurrence of P in T), while the branch explored in [20, 18, 36] consider $T' = A$ and $P = B$. A very simple application of the lemma [30] is to split the pattern in $k + 1$ pieces and check the text area surrounding each exact occurrence of a piece in the text.

However, Lemma 3.1 can be relaxed to permit the presence of some errors in the pieces:

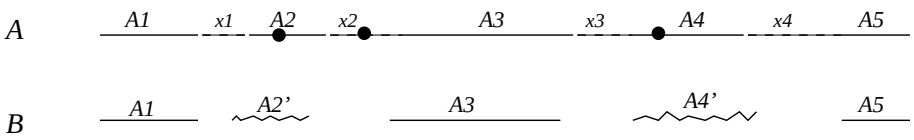


FIG. 5: An example of Lemma 3.1 with $k = 3$ and $s = 2$. At least 2 of the $A_i's$ survive unaltered. They are actually 3 such segments in this example because one of the errors appeared in x_2. Another possible reason could have been more than one error occurring in a single A_i.

LEMMA 3.2
Let A and B be two strings such that $ed(A, B) \leq k$. Let $A = A_1 x_1 A_2 x_2 ... x_{j-1} A_j$, for strings A_i and x_i and for any $j \geq 1$. Then, at least one string A_i appears in B with at most $\lfloor k/j \rfloor$ errors.

The proof is similar to that of Lemma 3.1: since at most k errors are performed on A to convert it into B, at least one of the A_i's get no more than $\lfloor k/j \rfloor$ of them. Note that Lemmas 3.1 and 3.2 have a common point at $j = k + 1$ and $s = 1$. Figure 6 illustrates.

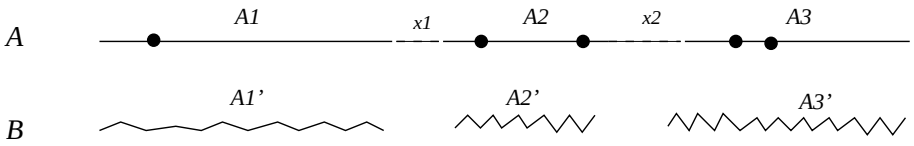


FIG. 6: Illustration of Lemma 3.2, where $k = 5$ and $j = 3$. At least one of the A_i's has at most one error (in this case A_1).

It is worthwhile to note that it is possible that $j\lfloor k/j \rfloor < k$, so we are not only "distributing" the errors across pieces but also "removing" some of them.

In this case, the idea is to partition the pattern in less than $k+1$ pieces, so one cannot guarantee that there are pieces free of errors. However, one can reduce the number of errors that may appear in at least one of the pieces. There exist filtration approaches based on different interpretations for A and B [24, 31]. The one we use in this paper corresponds to $P = A$, $x_i = \varepsilon$ and $B = T'$, where T' is an occurrence of P in T. The pattern P is split in j pieces and these are searched allowing $\lfloor k/j \rfloor$ errors in the text. Only the text areas surrounding those occurrences can contain a complete occurrence of P.

4 Combining Suffix Trees/Arrays and Pattern Partitioning

We present now our 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, then we address the pattern partitioning issue, and finally discuss a suffix array implementation.

4.1 DFS Using the NFA

We combine the DFS over suffix trees (Section 3.2) with our NFA simulation (Section 3.1.1). Recall that the former consists of 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.

The reason to combine DFS with our NFA as its node processing algorithm is as follows. According to [26], the NFA simulation [7] is the fastest algorithm for short patterns. This is precisely the type of patterns that we are going to search with this method, since longer patterns will be split in many subpatterns. As the next section makes clear, if the pattern is long enough to make another node processing algorithm better, it is because the pattern pieces are so long that the exponential nature of the search cost on suffix trees will make the whole approach useless. On the other hand, there exist (filtering) algorithms that for low error levels are faster than our choice, e.g. [30], but those algorithms need to skip over the text, which is not possible in this setup.

The use of the NFA node processing algorithm is only possible because of the simplicity of the DFS traversal. For instance, the idea does not work on the more complex setup of [37, 11], since these need some adaptations of the dynamic programming algorithm that are not easy to parallelize. 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 the NFA is used to process the nodes.

The NFA is modified as follows. We remove the initial self-loop of the automaton, so that it forces the whole string read to match the pattern. 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. Hence, we start the automaton with its first full diagonal active. The other states in the lower left triangle represent initial insertions in the pattern and hence need not be represented, 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[4].

Relating this to Section 3.2, we have that the three actions needed are (a) add a new letter to the text, which is accomplished by changing the active states of the NFA; (b) recognize a match, which is signaled by the fact that the lower-right state of the automaton is active; (c) determine that the current string cannot be extended to match the pattern, which is determined when the NFA runs out of active states.

On the other hand, unlike for the online algorithm [7], we need to represent the first full diagonal of the NFA, since now it will not be always active. 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

[4] 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.

single computer word and work $O(1)$ per traversed node of the suffix tree, using the bit-parallel simulation algorithm depicted in [7]. 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.

An additional twist is possible: every approximate occurrence of the pattern in the text must start with one of the first $k+1$ pattern characters, since otherwise a match is not possible. Hence, in the first level of the tree we need to enter only into those (at most) $k+1$ different characters.

4.2 Partitioning the Pattern

As seen in the Introduction and Section 3.2, the search cost using the suffix tree grows exponentially with m and k, no matter which of the two techniques we use (DFS or optimal traversal). 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.

This method is based on the pattern partitioning technique of Section 3.3. 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.

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 4.1. 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 Section 5 we find analytically the optimum j and the complexity of the search. Figure 7 illustrates the idea.

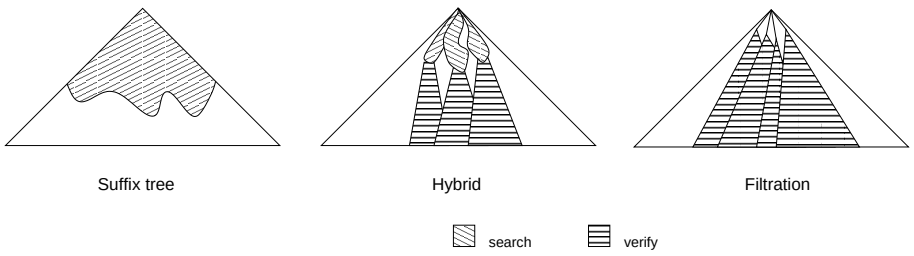


FIG. 7. Illustration of the tradeoff between indexing techniques.

One of the closest approaches to this idea is Myers' index [24], which collects all

the text q-grams (that is, 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 (that is, $m + k \leq q$). 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 edit distance to the pattern and searches them, instead of traversing the structure to see which of them actually 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 (that is, q) and cannot be adapted at query time.

4.3 A Suffix Array Implementation

As discussed, the suffix tree poses very high space requirements, so high that the data structure becomes impractical for many applications. We show now how our search algorithms can be implemented on a suffix array [22]. This was already done in [15, 6], so we explain the idea again and give some optimization details.

The basic idea is to simulate over the suffix array the algorithm designed for the suffix tree. Nodes of the suffix tree correspond to intervals in the suffix array. Specifically, the interval is that of all the leaves of the subtree rooted by the node. So each time the suffix tree algorithm is at a given node, its suffix array simulation is at a given interval. When the backtracking algorithm is at a given node, it either (1) stops and prunes the search in that branch, (2) stops and reports all the subtree leaves as occurrences, or (3) enters recursively into all its children. The first two cases are easily handled in the suffix array (reporting the leaves is particularly easy because they form precisely the current interval).

For the third case, we need to find all the subintervals of the current interval that correspond to the children of the current suffix tree node. A first choice is to binary search the children in alphabetical order (that is, left to right). We prefer to start by searching the child whose interval contains the middle position of the current interval, and proceed recursively with the two halves left. The net effect is the same, but we search in smaller intervals on average. Additionally, we switch to sequential traversal when the intervals are too short.

Finally, to speed up the first level of the search we precompute a supraindex that gives direct access to the suffix array with the first letter. This needs only $O(\sigma)$ extra space, and is particularly helpful to start the search entering only in the intervals corresponding to the first $k + 1$ characters of the pattern.

Our prototype uses a very simple construction algorithm: the n suffixes are pointed to and then quicksorted. There are many other more efficient construction algorithms [22, 32, 19]. The last reference shows to be about 8 times faster than using quicksort. However, we concentrate on the search algorithm in this work, as the data structure is well known.

5 Analysis

In which follows we analyze the query time complexity of our hybrid algorithm over suffix trees. The analysis over suffix arrays yields the same results for the optimal partitioning (up to lower order terms), but the final complexities over suffix arrays have to be multiplied by $O(\log n)$.

5.1 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 (that is, depending on the size of the automaton only) because the automaton of Figure 2 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 (that is, 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 (that is, 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. (A.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. (A.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 8.

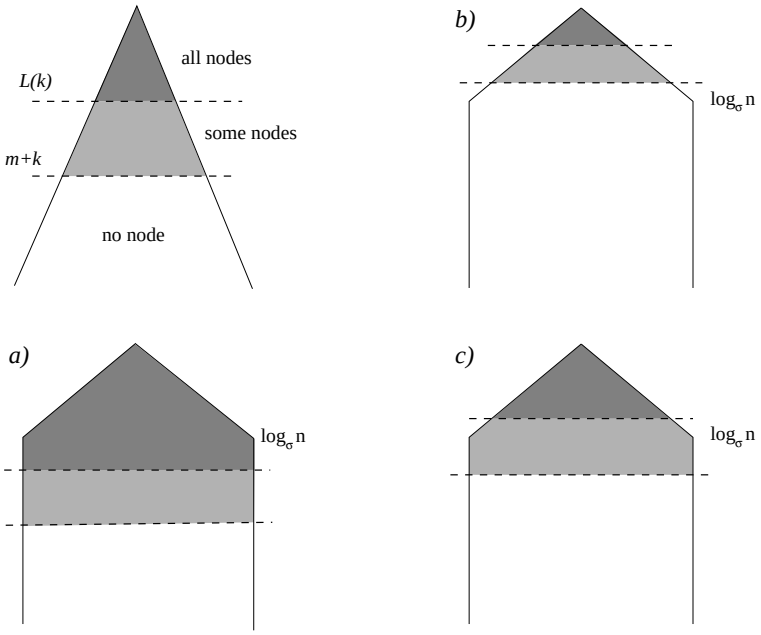


FIG. 8: The upper left figure shows the visited parts of the tree. The rest shows the three disjoint cases in which the analysis is split.

(a) $L(k) \geq \log_\sigma n$, that is, $n \leq \sigma^{L(k)}$, 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, that is, 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) $m + k < \log_\sigma n$, that is, $n > \sigma^{m+k}$ 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) $L(k) < \log_\sigma n \leq m+k$, that is, "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 $\log_\sigma n > \min(L(k), m+k)$, or which is the same, $\alpha < \max(\log_\sigma(n)/m\ (1-c/\sqrt{\sigma}), \log_\sigma(n)/m - 1)$. Figure 9 illustrates.

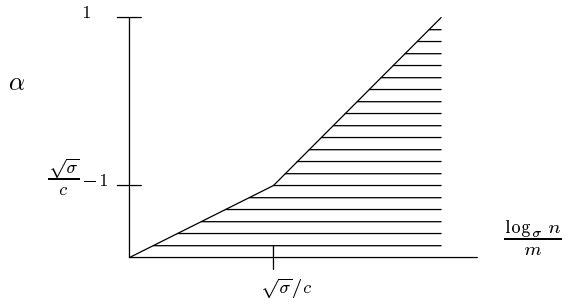


FIG. 9. Area of sublinearity for suffix tree traversal.

5.2 Pattern Partitioning

When pattern partitioning is applied, we perform j searches of the same kind of Section 4.1, this time with patterns of length m/j and k/j errors. We also need to verify all the possible matches.

As shown in [7], the matching probability for a text position is $O(\gamma(\alpha)^m)$, where $\gamma(\alpha)$ is that of Eq. (A.1). From now on we use $\gamma = \gamma(\alpha)$. Using dynamic programming, a verification costs $O(m^2)$ [5]. Hence, our total search cost is

$$j \times suffix\_tree\_traversal(m/j, k/j) \;\; + \;\; j \times \gamma^{m/j} m^2 n$$

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 5.1, 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

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

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

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

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 $\alpha < 1 - c/\sqrt{\sigma}$. 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) (that is, 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 $\alpha < 1 - c/\sqrt{\sigma}$.

[5] It can be done in $O((m/j)^2)$ time [24, 27], but this does not affect the result here.

(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 [7, 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.

A simplified technique that gives a reasonable result 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. This is the complexity that we claim in the beginning of this work, despite that it is not necessarily the best that can be obtained.

5.3 The Limits of the Method

Let us pay some attention to the limits of our hybrid method (Figure 10).

Using $j = (m+k)/\log_\sigma n$, the best j becomes 1 (that is, no pattern partitioning) when $n > \sigma^{m+k}$ (this is because the cost of verifications dominates over suffix tree traversal). The best j is $\geq k+1$ for $n < \sigma^{1+1/\alpha}$. Since in this case we search the pieces with zero errors (that is, $\lfloor k/(k+1)\rfloor = 0$, recall Section 4.2), 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 [28] does, except because 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, that is, $n\gamma^{1/\alpha} = o(1)$, which happens for $\alpha < 1/\log_{1/\gamma} n$. This requires, in particular, that $m = \Omega(\log n)$.

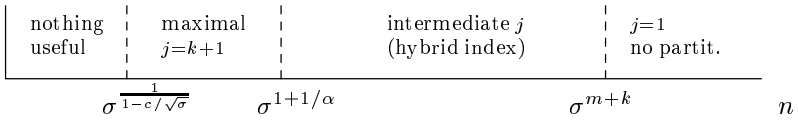


FIG. 10. The j values to be used according to n.

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 [24], and implicit in similar results on natural language texts [8, 25].

Finally, it is interesting to compare the limit error level α for which our index remains sublinear in its average search time against that of Myers' [24] and typical filtration indexes. As explained in Section 4.2, ours and Myers' index should share a single analysis, as the idealized method turns out to be the same (the differences in practice are explained in Section 4.2 and made apparent in the next section on experimental results). Since our analysis and that of [24] differ, we show in Figure 11 the numerical solution of $\lambda = 1$ in both cases. As can be seen, each analysis is tighter than the other for different σ values (on the x axis): Myers' is tighter for $\sigma \leq 30$ and ours is tighter for larger σ. We note that the curves are the exact numerical solutions to the equation $\lambda = 1$, while our gross conservative bound $\alpha \leq 1 - e/\sqrt{\sigma}$ approaches both curves from below and reaches Myers' for $\sigma = 60$.

Despite that Myers' limit error level for sublinearity has to be numerically computed, it is interesting to mention it can be fairly well approximated by the model $\alpha = 1 - 1.78/(1.09 + \ln \sigma)$, with a percentual error close to 1% for $\sigma \leq 200$. This shows a deeper difference in models: Myers' solution is of the type $1 - a/(b + \ln \sigma)$ while ours is of the type $1 - a/\sqrt{\sigma}$. As shown in [26], this last is the theoretically correct one asymptotically, which explains that our model becomes better than Myers' from some σ value on.

To give an idea of how much these indexes improve over the maximum α tolerated by typical filtration indexes, we plot one of them [28] (as explained, all of them have similar limits on the error level). The limit for this index is $\alpha \leq 1/(3\log_\sigma m + \log_\sigma n)$. Assuming very moderate values $m = 10$ and $n = 1$ Mb yields the third curve of the figure.

6 Experimental Results

We present in this section a number of experimental results to test the performance of our algorithm and to compare it against others. We first compare the existing algo-

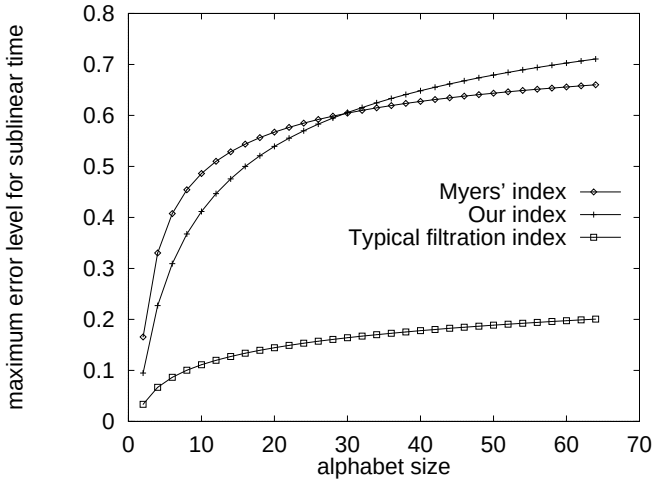


FIG. 11: The maximum α values for which different indexes have sublinear average search time, according to the analyses in the source papers.

rithms on suffix trees and arrays, showing that our implementation on suffix arrays is the best choice. We then show the behavior of our hybrid search algorithm. Finally, we compare our algorithm against the others.

We have tested short ($m = 10$) and medium-size ($m = 20$) patterns, searching with 1, 2 and 3 errors the short ones and with 2, 4 and 6 the medium ones. We used two types of text: English and DNA. The specific texts used vary because we are severely limited in the text sizes that can be handled with suffix trees (little more than 1 megabyte in our machine), while we use texts of up to 10 megabytes in the rest of the experiments. In all cases, we selected 1000 random patterns from each text file and used the same set for all the k values of that length, and for all the indexes.

Our machine is a Sun UltraSparc-1 of 167 MHz and 64 Mb of RAM, running Solaris.

6.1 Suffix Trees vs Suffix Arrays

Our first experiment aims at determining the most convenient traversal strategy over suffix trees and suffix arrays. 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 that the character distribution is not uniform, the best choice is to consider the alphabet size as the inverse probability of two random letters being equal, which gives σ around 13.

On the other hand, three indexes are compared:

Cobbs': The index proposed by Cobbs [11], which minimizes the number of suffix tree nodes traversed. We use the implementation of the author, not optimized for space (and actually implemented over a DAWG instead of a suffix tree). The code is restricted to work on an alphabet of size 4 or less, so it is only built on DNA.

Dfs(ST): The depth-first search technique of Gonnet [15, 6] run over a suffix tree and using our bit-parallel NFA simulation to process the nodes. The code is ours[6].

Dfs(SA): The same depth-first search technique run over a suffix array and using our bit-parallel NFA simulation to process the nodes. The code is ours.

Table 1 shows the construction cost and space requirements of the indexes. While Cobbs' implementation requires about 65 times the text size and our suffix tree needs 35 to 39 times the text size, the suffix array requires only 4 times the text size. Its construction time is close to that of suffix trees, despite that we have not used the fastest algorithms for suffix array construction but just a simple quicksort of pointers (as shown in [19], this means that the construction could be 8 times faster). In particular, note that suffix arrays are built faster than suffix trees when the capacity of the RAM memory is reached by the size of the data structure and paging comes into play.

Index	DNA	H-DNA	FRA	H-FRA
Cobbs'	108.70u/532.81s 65.67X	30.50u/76.06s 65.85X	n/a	
Dfs(ST)	30.89u/104.17s 38.99X	6.48u/0.42s 39.10X	28.46u/76.86s 35.45X	6.43u/0.61s 35.32X
Dfs(SA)	31.19u/0.04s 4.00X	9.27u/0.021s 4.00X	24.95u/0.05s 4.00X	7.57u/0.02s 4.00X

TABLE 1: Times (in seconds) to build the suffix tree and array indexes and their space overhead. The time is separated in the CPU part ("u") and the I/O part ("s"). The space is expressed in terms of the ratio index/text: rX means that the index takes r times the text size.

Table 2 shows query time for short and medium length patterns searched with an error level of 10% to 30%.

Three conclusions are clear from this comparison:

- Suffix trees are not practical except when the text size to handle is so small that the suffix tree fits in main memory. In the experiments we could use texts of little more than one megabyte only, because in our machine of 64 Mb of RAM larger texts were unmanageable. Even for these small texts, it took up to 12 hours to build Cobb's index.
- Even when the suffix trees fit in main memory, the suffix array is a better alternative (despite its theoretically worse complexity). In our experiments the suffix

[6] The implementation of the suffix tree is from Erkki Sutinen.

Short patterns ($m = 10$)					
Index	k	DNA	H-DNA	FRA	H-FRA
Cobbs'	1	110.0u/192.5s	101.8u/156.0s	n/a	
	2	588.1u/1989s	377.0u/1113s		
	3	3370u/14291s	1835u/6060s		
Dfs(ST)	1	6.81u/15.45s	2.59u/0.25s	6.11u/13.01s	2.12u/0.17s
	2	134.4u/337.2s	48.55u/0.91s	42.82u/87.50s	13.71u/0.27s
	3	1044u/2482s	446.4u/5.25s	215.2u/500.1s	51.38u/0.33s
Dfs(SA)	1	5.90u/0.31s	2.83u/0.22s	4.74u/0.15s	2.27u/0.19s
	2	77.32u/1.58s	33.86u/0.75s	25.91u/0.23s	12.74u/0.22s
	3	615.8u/12.42s	240.6u/4.83s	90.32u/0.50s	42.14u/0.32s

Medium patterns ($m = 20$)					
Index	k	DNA	H-DNA	FRA	H-FRA
Cobbs'	2	726.1u/1700s	496.3u/974.0s	n/a	
	4	***	8060u/14447s		
	6	***	***		
Dfs(ST)	2	56.80u/189.5s	18.60u/0.31s	35.98u/80.30s	12.93u/0.31s
	4	1989u/8269s	432.6u/0.22s	482.9u/1488s	125.6u/0.36s
	6	11341u/40604s	2185u/0.20s	2204u/7286s	516.2u/0.20s
Dfs(SA)	2	35.26u/0.22s	18.81u/0.21s	23.59u/0.31s	11.94u/0.22s
	4	713.4u/0.31s	336.3u/0.42s	202.5u/0.20s	96.12u/0.23s
	6	4005u/0.44s	1751u/0.45s	806.8u/0.20s	382.1u/0.27s

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

TABLE 2: Times (in milliseconds) to search approximate patterns in suffix tree/array indexes. They are separated in CPU time ("u") and I/O time ("s").

array beats the suffix tree even for 0.5 Mb texts. Where the suffix array does not beat the suffix tree, it gets very close. This is probably due to better locality of reference, which translates into more efficient cache usage.

- Among suffix tree algorithms, the idea of a more complex node processing algorithm in exchange for traversing less nodes does not pay off. In practice, the simpler depth-first search strategy performs better. The experiments show that Cobbs' index is 20 to 30 times slower than the suffix array implementation.

Therefore, in which follows we keep only our traversal algorithm on suffix array, and discard suffix tree implementations. This gives us the best representative of these methods and allows us to use much larger texts in the rest of the experiments that follow. As the data structures that remain fit in main memory, we do not consider the I/O time anymore.

6.2 The Performance of the Hybrid Index

Our aim in this section is to show how our hybrid algorithm behaves under different choices of j, as well as to show its sublinear behavior. From now on, we use a different set of texts:

- English natural language text (articles from The Wall Street Journal taken from the TREC collection [17]), filtered to lower-case and the separators converted into a single space. We use 10 megabytes of the collection and call it WSJ in the experiments. Again, σ can be considered to be around 13.
- DNA-like text, which consists of 10 megabytes of text randomly generated with a uniform distribution over a 4-letter alphabet. This file is called DNA in the tests that follow. We made this choice because we did not have a real DNA text of that size.

Figure 12 shows the result of different choices for j when using our hybrid algorithm on the WSJ text. We present only the j values that are interesting (the others give very bad results). First consider $m = 10$. Although for one and two errors the simple backtracking algorithm is the best, we can see that a partition in $j = 2$ pieces is better for 3 errors for 2 Mb of text or less. This matches with the analysis in the sense that, the larger j, the worse the complexity with respect to n, so as n grows the optimal j is reduced (recall that the optimal j is $\Theta(m/\log_\sigma n)$).

The same phenomenon can be observed for $m = 20$ and $k = 2$. In this case the best choice in the range of n values tested is $j = 3$, that is, $j = k + 1$. However, it is clear that soon after $n = 10$ Mb the situation will change and $j = 2$ will dominate (since the complexity in terms of n is better than with $j = 3$). Much later $j = 1$ would become the best choice.

A case where an intermediate $1 < j \leq k$ is the best choice appears for $m = 20$ and $k = 4$. The best j is disputed between 2 and 3, becoming 2 the best choice for $n = 3$ Mb or more. This plot shows nicely how the optimal j is in between, as the two extremes ($j = 1$ and 4) are much more expensive. The same happens with the combination $m = 20$ and $k = 6$.

Figure 13 shows the same experiments on DNA. In this case $j = 1$ is the only reasonable option for $m = 10$, and an intermediate $j = 2$ becomes the best choice for $m = 20$ and all the different k values.

In the sections that follow, we use the best j when comparing our index against others. It should be noted, however, that despite that our big-O analysis predicts correctly the growth rates, it does not predict well the exact j values that should be used.

6.3 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 indexes that are already implemented. These are

Myers': The index proposed by Myers [24]. We use the implementation of the author,

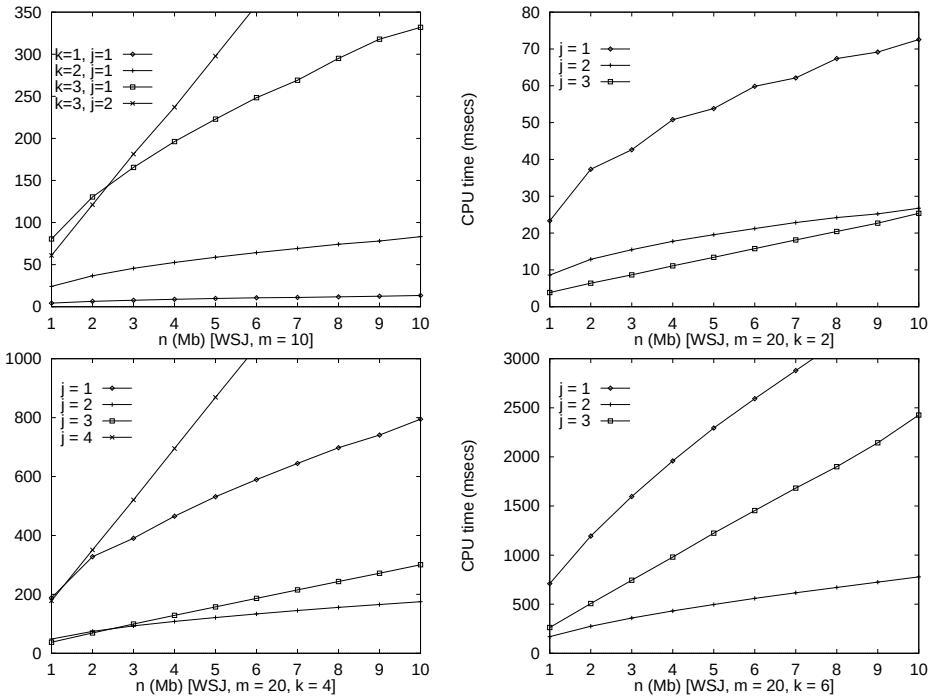


FIG. 12: Different alternatives for our hybrid index on English text and for increasing n. In reading order, the cases $m = 10$; $m = 20$ with $k = 2$; $m = 20$ with $k = 4$; and $m = 20$ with $k = 6$.

which is a prototype that works only for some m values that depend on σ and n (the algorithm is generic but the implementation is not). In particular, the range of n values that fit our pattern lengths in WSJ goes from 2 Mb to 6 Mb. On DNA we have used $m = 11$ and $m = 22$ (instead of 10 and 20) for this index, which allowed us to use it from 2 Mb to 10 Mb. However, we also limited DNA to 6 Mb because the time becomes unmanageable as soon as the index ceases to fit in main memory.

Exact(q): The index based on q-grams presented in [28]. This is quite similar to partition in $k + 1$ pieces, except that the index stores only q-grams and some extra work may be necessary when the length of the pattern pieces is not q. We show the results for $q = 4$ and 5 on WSJ and $q = 5$ and 6 on DNA.

Dfs: Our suffix array implementation of the depth-first search traversal [15, 6].

Hybrid: Our new index based on suffix arrays and pattern partitioning, using optimal j. This index is not included in the tests for $m = 10$ because $j = 1$ (that is, the Dfs index) is already the optimal value.

Online: is the best online algorithm for each case (according to [26]). This is included for comparison purposes.

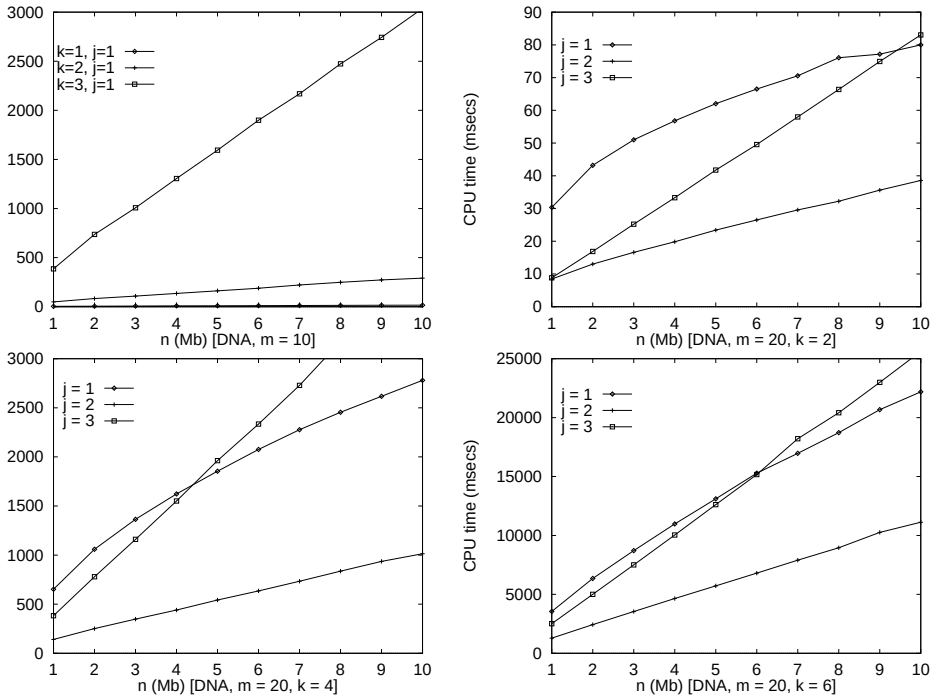


FIG. 13: Different alternatives for our hybrid index on DNA and for increasing n. In reading order, the cases $m = 10$; $m = 20$ with $k = 2$; $m = 20$ with $k = 4$; and $m = 20$ with $k = 6$.

In particular, one of the most relevant q-gram indexes [36] is not yet implemented and therefore is excluded from our tests. We know, however, that its space requirement is low (close to a word-retrieving index), but also that since the index simulates the on-line algorithm [35], its tolerance to errors is quite low (see [7, 25], for example).

All the indexes 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 indexes and the space they take in Table 3. We show the time per megabyte and the proportional extra space needed to index 6 Mb (since Myers' index could not be built for larger texts), although this time is almost independent on the text size. As can be seen, Myers' index is efficiently built but is the most space demanding. The suffix array takes half the space, and could be built in less time if the algorithm of [19] were used. Finally, Exact(q) gets heavier as q grows, but in general is the most compact index.

Figure 14 compares the search time (only CPU) of the different algorithms for $m = 10$ as n grows. For these short patterns our hybrid scheme does not apply, so only Dfs is shown. Myers' index (automatically) uses $q = 5$ on WSJ and $q = 11$ on DNA. Therefore, on WSJ we are comparing indeed $j = 1$ (Dfs), $j = 2$ (Myers') and $j = 3$ (Exact), albeit the implementation details are quite different. As in our results with

Index	Myers'	Exact(4)	Exact(5)	Exact(6)	S. Array
WSJ	4.51 sec 8.12 X	13.33 sec 2.50 X	18.06 sec 3.89 X	n/a	25.17 sec 4.00 X
DNA	4.11 sec 7.67 X	n/a	5.33 sec 1.93 X	7.50 sec 2.15 X	27.50 sec 4.00 X

TABLE 3: Times (in seconds of CPU per megabyte) to build the different indexes and their space overhead (in the format rX, meaning that the index takes r times the text size). The text had 6 Mb of size. The suffix array is the data structure for the Dfs and Hybrid methods.

our own index, the best choice is $j = 1$ (except for $k = 3$ where the choice $j = 2$ is very close). The different choices of the Exact index show their space-time tradeoff: the larger q, the fastest the index but, it demands more space. On DNA, Myers' index is also using $j = 1$, and its better search time shows that the choice of generating all the neighborhood of the pattern is better than a suffix array approach when the alphabet is small.

Figure 15 shows the results for $m = 20$. We first consider WSJ. For $k = 2$ we already know that $j = k + 1$ is the best choice, and this is reflected in the fact that the best is the Hybrid index with $j = 3$ together with Exact(5), which is a very close approximation. Myers' and Dfs are using $j = 2$ and $j = 1$ respectively, and hence perform worse.

For $k = 4$ the best option is $j = 2$ and therefore the Hybrid beats Exact and Dfs. Myers' index is beaten too, although it is also using $j = 2$. The reason is that over this large and biased alphabet the technique of generating all the possible q-grams that match the pattern piece with errors is not a good choice, because a huge number of nonexistent strings are generated and searched. The suffix tree or array, on the other hand, are used to search only the strings that actually exist in the text. It is also interesting that the differences blur for $k = 6$.

On DNA, the optimal j is always 2, so Dfs and Exact(q) are ruled out. Myers' index is faster than the Hybrid, showing again that generating the strings close to the pattern pays off if the alphabet is small. We remark also that this text is "perfect", in the sense that it is randomly generated and not truly DNA. A less random text negatively affects Myers' index, while the suffix array technique should benefit from less different strings close to the pattern.

Note also that for $k = 6$ the online algorithm is much faster than any indexed scheme on DNA. In general, the online algorithms are less sensitive to the error level.

Some more general conclusions that can be extracted from the experiments are:

- The most important feature that affects the performance of the indexes is the amount of pattern partitioning performed. Under this light, suffix tree/array traversal algorithms do not partition the pattern ($j = 1$), traditional filtering indexes partition the pattern in $k + 1$ pieces, and Myers' index uses an intermediate partition fixed at indexing time. Our hybrid index has the potential of selecting the best

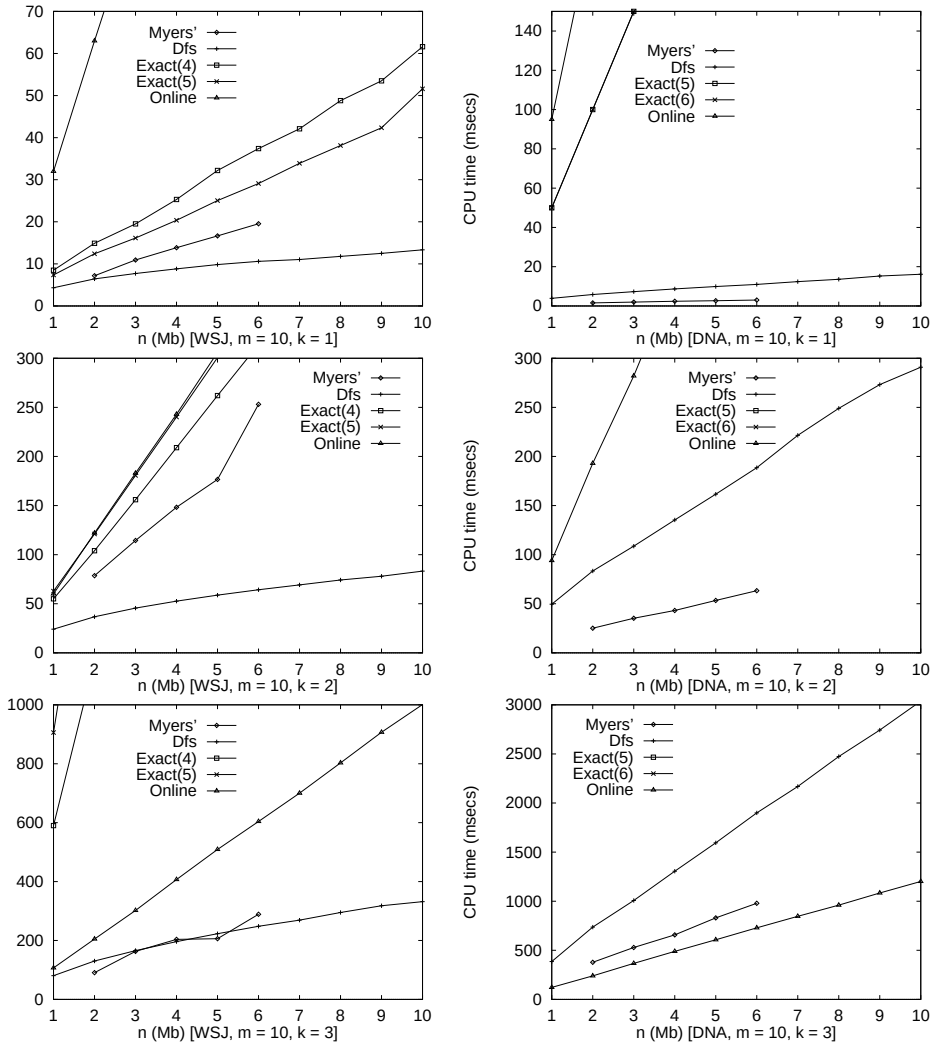


FIG. 14: Comparison between different indexes for $m = 10$ and $k = 1, 2$ and 3 (first to third rows, respectively). The left plots correspond to WSJ and the right plots to DNA.

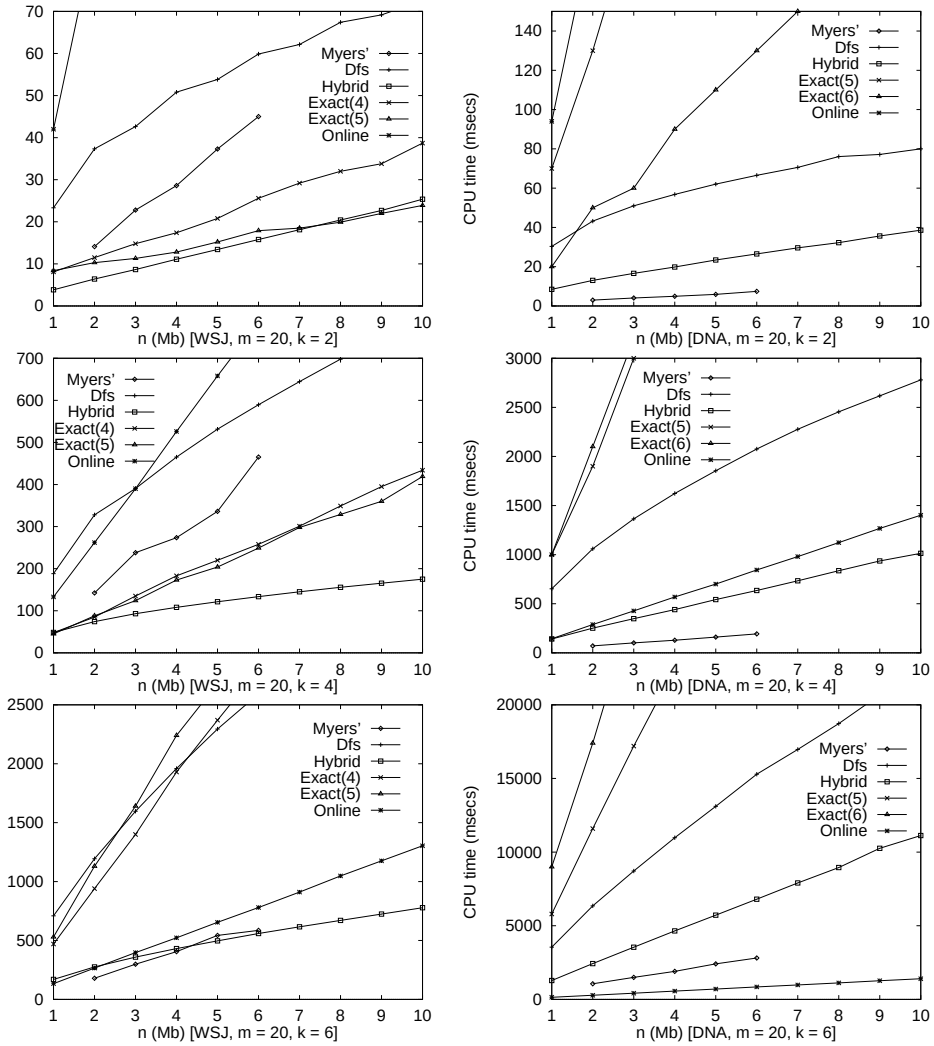


FIG. 15: Comparison between different indexes for $m = 20$ and $k = 2$, 4 and 6 (first to third rows, respectively). The left plots correspond to WSJ and the right plots to DNA.

partition at query time and needs half the space of Myers'.

- Among the suffix tree/array traversal algorithms, depth-first search over a suffix array shows to be the best choice in practice.
- When Myers' index can use the best j, its main difference with our index is that it generates all the text substrings at some edit distance to the pattern and searches all them in the index. We instead use the suffix array to find the strings that actually exist in the text. Myers' technique shows to be faster on random texts over small alphabets (as in random DNA and perhaps on real DNA), while it is slower in other cases (as in English text).
- The Exact(q) is a low-cost alternative in terms of space, and performs reasonably well for low error levels and not too small alphabets.
- The indexed schemes are more sensitive to the error level allowed than the online algorithms, so for high error levels still there is no way to improve over online searching.

7 Conclusions and Future Work

We have proposed a hybrid 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 shown that this hybrid is an intermediate approach between the extremes of pure suffix tree traversal and filtering using exact searching of pattern pieces.

We have proved analytically that the optimal number of pieces is indeed between both extremes, and that the resulting index has sublinear retrieval time (of the form $O(n^\lambda)$, where $0 < \lambda < 1$ if the error level is moderate).

We have implemented this approach using a fast node processing algorithm and simulating the suffix tree on the less space demanding suffix array. We have shown experimentally that this approach performs better in practice than those based on suffix trees. We have also shown the effect of partitioning the pattern in different number of pieces. Finally, we have presented the first experimental results that compare the different implemented indexing schemes, which show that the proposed idea can improve over the previously implemented approaches.

We have implemented a crude technique to verify the occurrences of the pieces in order to check if they form an occurrence of the complete pattern. A better approach is shown in [24, 27]. Since this would make a difference only for larger j, we have not considered that improvement for this paper, but it would make an important difference for longer patterns.

Our analysis predicts the asymptotic behavior of the index but it is too crude to help determine the correct number of pieces in which the pattern has to be partitioned. A finer analysis or empirical procedure able to determine this number automatically would be of great practical importance. Related to this issue is optimizing the partitioning: there is no need to split the pattern in equal-length pieces. Although this

is the best choice on random text, in cases like English a very frequent pattern piece will trigger many more verifications. In that case we want to split in pieces so that the overall number of text positions to verify is minimized. In [28] an $O(mk^2)$ dynamic programming algorithm is presented that finds the best partition in this sense. Adapting such an algorithm to this case, where the pattern pieces are not searched exactly, is another interesting problem.

Acknowledgements

We thank the useful comments of the referees of the [29] version, 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 indexes.

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. 4th South American Workshop on String Processing (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. A unified view of string matching algorithms. In *SOFSEM'96: Theory and Practice of Informatics*, LNCS 1175, pages 1–15, 1996. Invited paper.

[5] R. Baeza-Yates and G. Gonnet. Fast text searching for regular expressions or automaton searching on a trie. *Journal of the ACM*, 43, 1996.

[6] R. Baeza-Yates and G. Gonnet. A fast algorithm on average for all-against-all sequence matching. In *Proc. 6th Symposium on String Processing and Information Retrieval (SPIRE'99)*. IEEE CS Press, 1999. Previous version unpublished, Dept. of Computer Science, Univ. of Chile, 1990.

[7] R. Baeza-Yates and G. Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999. Preliminary version in *Proc. CPM'96, LNCS 1075*.

[8] R. Baeza-Yates and G. Navarro. Block-addressing indices for approximate text retrieval. *Journal of the American Society for Information Science (JASIS)*, 51(1):69–82, January 2000.

[9] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. Chen, and J. Seiferas. The smallest automaton recognizing the subwords of a text. *Theoretical Computer Science*, 40:31–55, 1985.

[10] W. Chang and T. Marr. Approximate string matching and local similarity. In *Proc. 5th Annual Symposium on Combinatorial Pattern Matching (CPM'94)*, LNCS 807, pages 259–273, 1994.

[11] A. Cobbs. Fast approximate matching using suffix trees. In *Proc. 6th Annual Symposium on Combinatorial Pattern Matching (CPM'95)*, LNCS 937, pages 41–54, 1995.

[12] M. Crochemore. Transducers and repetitions. *Theoretical Computer Science*, 45:63–86, 1986.

[13] M. Farach, P. Ferragina, and S. Muthukrishnan. Overcoming the memory bottleneck in suffix tree construction. In *Proc. 9th Symposium on Discrete Algorithms (SODA'98)*, pages 174–183, 1998.

[14] R. Giegerich, S. Kurtz, and J. Stoye. Efficient implementation of lazy suffix trees. In *Proc. 3rd Workshop on Algorithm Engineering (WAE'99)*, LNCS 1668, pages 30–42, 1999.

[15] G. Gonnet. A tutorial introduction to Computational Biochemistry using Darwin. Technical report, Informatik E.T.H., Zuerich, Switzerland, 1992.

[16] G. Gonnet, R. Baeza-Yates, and T. Snider. *Information Retrieval: Data Structures and Algorithms*, chapter 3: New indices for text: Pat trees and Pat arrays, pages 66–82. Prentice-Hall, 1992.

[17] D. Harman. Overview of the Third Text REtrieval Conference. In *Proc. Third Text REtrieval Conference (TREC-3)*, pages 1–19, 1995. NIST Special Publication 500-207.

[18] N. Holsti and E. Sutinen. Approximate string matching using q-gram places. In *Proc. 7th Finnish Symposium on Computer Science*, pages 23–32. University of Joensuu, 1994.

[19] H. Itoh and H. Tanaka. An efficient method for in memory construction of suffix arrays. In *Proc. 6th Symposium on String Processing and Information Retrieval (SPIRE'99)*, pages 81–87. IEEE CS Press, 1999.

[20] P. Jokinen and E. Ukkonen. Two algorithms for approximate string matching in static texts. In *Proc. 2nd Annual Symposium on Mathematical Foundations of Computer Science (MFCS'91)*, volume 16, pages 240–248, 1991.

[21] D. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, 1973.

[22] U. Manber and E. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, pages 935–948, 1993.

[23] U. Manber and S. Wu. GLIMPSE: A tool to search through entire file systems. In *Proc. USENIX Technical Conference*, pages 23–32, Winter 1994.

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

[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. A guided tour to approximate string matching. Technical Report TR/DCC-99-5, Dept. of Computer Science, Univ. of Chile, 1999. To appear in *ACM Computing Surveys*. `ftp://-ftp.dcc.uchile.cl/pub/users/gnavarro/survasm.ps.gz`.

[27] 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.

[28] G. Navarro and R. Baeza-Yates. A practical q-gram index for text retrieval allowing errors. *CLEI Electronic Journal*, 1(2), 1998. `http://www.clei.cl`.

[29] G. Navarro and R. Baeza-Yates. A new indexing method for approximate string matching. In *Proc. 10th Annual Symposium on Combinatorial Pattern Matching (CPM'99)*, LNCS 1645, pages 163–186, 1999.

[30] G. Navarro and R. Baeza-Yates. Very fast and simple approximate string matching. *Information Processing Letters*, 72:65–70, 1999.

[31] G. Navarro, E. Sutinen, J. Tanninen, and J. Tarhio. Indexing text with approximate q-grams. In *Proc. 11th Annual Symposium on Combinatorial Pattern Matching (CPM'2000)*, Montreal, Canada, 2000. To appear.

[32] K. Sadakane. A fast algorithm for making suffix arrays and for the Burrows-Wheeler transformation. In *Proc. Data Compression Conference (DCC'98)*, pages 129–138, 1998.

[33] P. Sellers. The theory and computation of evolutionary distances: pattern recognition. *Journal of Algorithms*, 1:359–373, 1980.

[34] F. Shi. Fast approximate string matching with q-blocks sequences. In *Proc. 3rd South American Workshop on String Processing (WSP'96)*, pages 257–271. Carleton University Press, 1996.

[35] E. Sutinen and J. Tarhio. On using q-gram locations in approximate string matching. In *Proc. ESA'95*, LNCS 979, pages 327–340, 1995.

[36] E. Sutinen and J. Tarhio. Filtration with q-samples in approximate string matching. In *Proc. 7th Annual Symposium on Combinatorial Pattern Matching (CPM'96)*, LNCS 1075, pages 50–61, 1996.

[37] E. Ukkonen. Approximate string matching over suffix trees. In *Proc. 4th Annual Symposium on Combinatorial Pattern Matching (CPM'93)*, pages 228–242, 1993.

[38] E. Ukkonen. Constructing suffix trees on-line in linear time. *Algorithmica*, 14(3):249–260, Sep 1995.

[39] Esko Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms*, 6:132–137, 1985.

[40] S. Wu and U. Manber. Fast text searching allowing errors. *Comm. of the ACM*, 35(10):83–91, October 1992.

A 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 2). 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 [7]. As Figure 16 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.

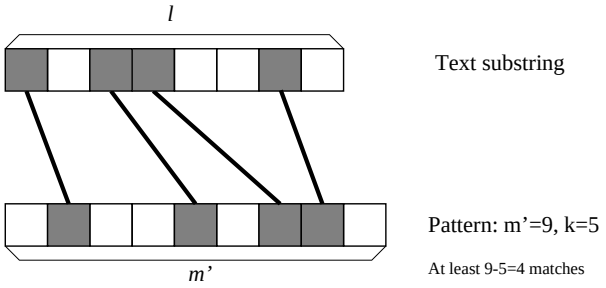


FIG. 16: Upper bound for the probability of matching. At least $\max(m' - k, \ell - k)$ characters must match, since otherwise it would not be possible to convert one string into the other.

Since we are interested in any pattern prefix matching the current text substring, we add up all the possible lengths from $\ell - k$ to $\ell + k$:

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

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

$$\gamma(x) \;=\; \frac{1}{\sigma^{1-x}\,x^{2x}\,(1-x)^{2(1-x)}} \tag{A.1}$$

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 $\gamma(\beta) < 1$, that is,

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

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

$$\beta \;<\; 1 - \frac{e}{\sqrt{\sigma}} \tag{A.3}$$

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 $\ell-k < r \le \ell$. By considering $r = x\ell$ ($x \in [1-\beta, 1]$) 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 $\sigma \le \beta/(1-\beta)^2$. Since due to Eq. (A.3) we are only interested in $\sigma \ge 1/(1-\beta)^2$, $\delta h(x)/\delta x$ does not have roots, and the maximum of $h(x)$ is at $x = 1-\beta$. That means $r = \ell - k$, that is, the first term of the second summation, which is the same as the 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. (A.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 $\beta = k/\ell \ge 1 - e/\sqrt{\sigma}$, 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 dividing by x we 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 [7], 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 16) we use is not strong enough: for instance, we could avoid replacements in the edit distance and the bound would be the same. In this 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 $c \approx 1$.