A Metric Index for Approximate String Matching ⋆

Gonzalo Navarro [a,1,2] and Edgar Chávez [b,1,3]

[a] *Center for Web Research, Dept. of Computer Science, University of Chile, Chile. Blanco Encalada 2120, Santiago, Chile.* gnavarro@dcc.uchile.cl

[b] *Escuela de Ciencias Físico-Matemáticas, Universidad Michoacana. Edificio "B", Ciudad Universitaria, Morelia, Mich. México 58000.* elchavez@fismat.umich.mx

Abstract

We present a radically new indexing approach for approximate string matching. The scheme uses the metric properties of the edit distance and can be applied to any other metric between strings. We build a metric space where the sites are the nodes of the suffix tree of the text, and the approximate query is seen as a proximity query on that metric space. This permits us finding the *occ* occurrences of a pattern of length m, permitting up to r differences, in a text of length n over an alphabet of size σ, in average time $O(m^{1+\epsilon} + occ)$ for any $\epsilon > 0$, if $r = o(m/\log_\sigma m)$ and $m > \frac{1+\epsilon}{\epsilon} \log_\sigma n$. The index works well up to $r < (3-\sqrt{2})m/\log_\sigma m$, where it achieves its maximum average search complexity $O(m^{1+\sqrt{2}+\epsilon} + occ)$. The construction time of the index is $O(m^{1+\sqrt{2}+\epsilon} n \log n)$ and its space is $O(m^{1+\sqrt{2}+\epsilon} n)$. This is the first index achieving average search time polynomial in m and independent of n, for $r = O(m/\log_\sigma m)$. Previous methods achieve this complexity only for $r = O(m/\log_\sigma n)$. We also present a simpler scheme needing $O(n)$ space.

Key words: Indexed approximate string matching, metric spaces, metric indexes, suffix trees, suffix arrays.

⋆ A conference version of this paper appeared in [11].

[1] Supported by CYTED VII.19 RIBIDI Project.

[2] Funded by Millennium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile.

[3] Supported by CONACyT grant 36911, Mexico.

1 Introduction and Related Work

Indexing text to permit efficient approximate searching on it is one of the main open problems in combinatorial pattern matching. The approximate string matching problem is: Given a long text T of length n, a (comparatively short) pattern P of length m, and a threshold r to the number of "errors" or "differences" permitted in the matches, retrieve all the pattern occurrences, that is, text substrings whose *edit distance* to the pattern is at most r. Text and pattern are sequences over an alphabet of size σ. The *edit distance* between two strings is defined as the minimum number of character insertions, deletions and substitutions needed to make them equal. This distance is used in many applications, but several other distances are of interest.

In the on-line version of the problem, the pattern can be preprocessed but the text cannot. There are numerous solutions to this problem [36], but none is acceptable when the text is too long since the search time is proportional to the text length. Indexing text for approximate string matching has received attention only recently. Despite some progress in the last decade, the indexing schemes for this problem are still rather immature [37].

There exist some indexing schemes specialized on word-wise searching on natural language text [32,5]. These indexes perform quite well but they cannot be extended to handle the general case, which includes extremely important applications such as DNA, proteins, music or Oriental languages.

The indexes that solve the general problem can be divided into three classes. *Backtracking* techniques [24,47,14,20] use the suffix tree [2], suffix array [31] or DAWG [16] of the text to factor out its repetitions. A sequential algorithm on the text is simulated by backtracking on the data structure. These algorithms achieve worst-case time $O((m\sigma)^r)$ or $O(\sigma^{m+r})$, and average-case time $\sigma^{O(r)}$. Those are exponential in m or r but independent of n, the text size. This makes them attractive when searching for very short patterns.

Partitioning methods [44,43,4] extract pattern pieces to ensure that some of the pieces must appear without alterations inside every occurrence. An index able of exact searching is used to detect the pieces and the text areas that have enough evidence of containing an occurrence are checked with a sequential algorithm. These algorithms achieve average search times as low as $O(m)$, but they work only when $r/m = O(1/\log_\sigma n)$, which is too restrictive in many applications.

Hybrid algorithms [35,6,38] extract from the pattern large pieces that can still contain (less) errors, they are searched for using backtracking, and the potential text occurrences are checked as in partitioning methods. These methods can balance between length of the pieces to search for and error level permit-

ted. They achieve on average $O(mn^\lambda)$ search time, for some $0 < \lambda < 1$ that depends on r. They tolerate high error ratios, $r/m = O(1)$.

All the methods above use for approximate searching a data structure aimed at exact searching. There are a few proposals of structures specifically designed for approximate searching, and which require more than $O(n)$ space for the index. Some are restricted to $r = 1$, requiring $O(n \log n)$ space and time to build the index, and $O(m \log\log n + occ)$ search time [9]; or requiring $O(n \log n)$ average space, $O(n \log^2 n)$ average construction time, and $O(m\sigma + occ)$ search time [30]. Others work for any r, requiring $O(n \log^r n)$ space and $O(m + \log^r n \log\log n + occ)$ search time [15].

We propose in this paper a brand new approach to the problem, which is in the line of designing a specific index for this type of search. We take into account that the edit distance satisfies the triangle inequality and hence it defines a metric space on the set of text substrings. We can re-express the approximate search problem as a range search problem on this metric space. This approach has been attempted before [10,3,18], but in those cases the particularities of the application made it possible to index $O(n)$ elements. In the general case we have $O(n^2)$ text substrings.

The main contribution of this paper is to devise a method (based on the suffix tree of the text) to meaningfully collapse the $O(n^2)$ text substring into $O(n)$ sets, to find a way to build a metric space out of those sets, and to devise a pivot selection policy that ensures good complexities. The result is an indexing method that, for any constant $\epsilon > 0$ and some constant $0 < \alpha < 1$, requires $O(m^{1/\alpha+\epsilon} n)$ space and $O(m^{1/\alpha+\epsilon} n \log n)$ worst-case construction time, and it permits finding the *occ* approximate occurrences of the pattern in $O(m^{1/\alpha+\epsilon} + occ)$ average time, for any $(r+2)\log_\sigma(m)/m = \gamma < 3-2\sqrt{2} \approx 0.172$ and $m > \frac{\log_\sigma m}{1-\alpha-\gamma(1+1/\alpha)}$. For example, if $r = o(m/\log_\sigma m)$, that is, $\gamma = o(1)$, we achieve $O(m^{1+\epsilon})$ average search time for $m > \frac{1+\epsilon}{\epsilon}\log_\sigma n$, with $O(m^{1+\epsilon} n)$ space and $O(m^{1+\epsilon} n \log n)$ construction time. As γ approaches its maximum permissible value, the search complexity reaches $O(m^{1+\sqrt{2}+\epsilon}) = O(m^{2.41422})$ for small ϵ.

This is a complexity breakthrough over previous work, as we are able to obtain complexities polynomial in m and independent of n for $r/m = O(m/\log_\sigma m)$, instead of the much stricter limit $r/m = O(m/\log_\sigma n)$ given by partitioning methods. We are still unable to reach $r/m = O(1)$ as hybrid methods, but those have the much higher $O(mn^\lambda)$ search time complexity. Fig. 1 illustrates.

Moreover, our index represents an original approach to the problem that opens a number of possibilities for improvements. For example, it is easier than in other approaches to extend our idea to other distance functions such as reversals.

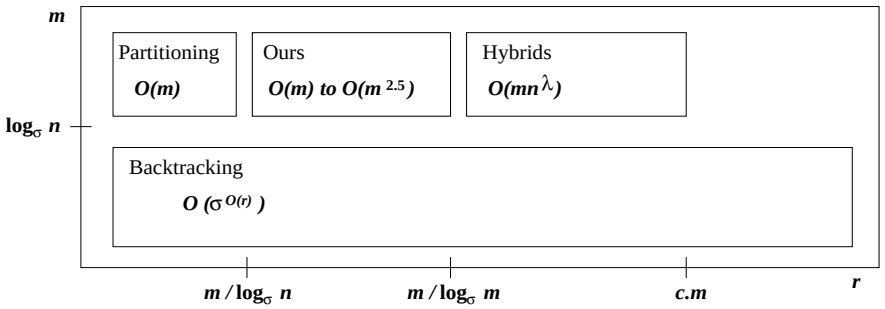

Fig. 1. Existing search complexities and our contribution in context.

The main drawbacks of our index is that it needs superlinear space and that it needs to know m and r at index construction time. We also present a simpler version of the index needing $O(n)$ space and that, despite not involving a complexity breakthrough, promises to be interesting in practice.

We use the following notation in the paper. Given a string $s \in \Sigma^*$ we denote its length as $|s|$. We also denote s_i the i-th character of s, for an integer $i \in \{1..|s|\}$. We denote $s_{i...j} = s_i s_{i+1} \dots s_j$ (which is the empty string ε if $i > j$) and $s_{i...} = s_{i...|s|}$. A string x is said to be a *prefix* of xy, a *suffix* of yx and a *substring* of yxz.

2 Metric Spaces and Edit Distance

We describe in this section some concepts related to searching metric spaces. We have concentrated only in the part that is relevant for this paper. There exist recent surveys if more complete information is desired [13,23].

A metric space is, informally, a set of black-box objects and a distance function defined among them, which satisfies the triangle inequality. The problem of *proximity searching* in metric spaces consists of indexing the set such that later, given a query, all the elements of the set that are close enough to the query can be quickly found. This has applications in a vast number of fields, such as non-traditional databases (where the concept of exact search is of no use and we search for similar objects, e.g. databases storing images, fingerprints or audio clips); machine learning and classification (where a new element must be classified according to its closest existing element); image quantization and compression (where only some vectors can be represented and those that cannot must be coded as their closest representable point); text retrieval (where we look for documents that are similar to a given query or document); computational biology (where we want to find a DNA or protein sequence in a database allowing some errors due to typical variations); function prediction (where we want to search for the most similar behavior of a function in the past so as to predict its probable future behavior); etc.

Formally, a *metric space* is a pair $(\mathbb{X}, d)$, where $\mathbb{X}$ is a "universe" of objects and $d : \mathbb{X} \times \mathbb{X} \longrightarrow \mathbb{R}^+$ is a distance function defined on it that returns non-negative values. This distance satisfies the properties of reflexivity ($d(x, x) = 0$), strict positiveness ($x \neq y \Rightarrow d(x, y) > 0$), symmetry ($d(x, y) = d(y, x)$) and triangle inequality ($d(x, y) \leq d(x, z) + d(z, y)$).

A finite subset $\mathbb{U}$ of $\mathbb{X}$, of size $n = |\mathbb{U}|$, is the set of objects we search. Among the many queries of interest on a metric space, we are interested in the so-called *range queries*: Given a query $q \in \mathbb{X}$ and a tolerance radius r, find the set of all elements in $\mathbb{U}$ that are at distance at most r to q. Formally, the outcome of the query is $(q, r)_d = \{u \in \mathbb{U},\ d(q, u) \leq r\}$. The goal is to preprocess the set so as to minimize the computational cost of producing the answer $(q, r)_d$.

The plethora of existing algorithms to index metric spaces can be roughly divided into two classes. The first, *pivot-based* techniques, are built on a single general idea: Select k elements $\{p_1, \ldots, p_k\}$ from $\mathbb{U}$ (called *pivots*), and identify each element $u \in \mathbb{U}$ with a k-dimensional point $(d(u, p_1), \ldots, d(u, p_k))$ (i.e. its distances to the pivots). The index is basically the set of kn coordinates. At query time, map q to the k-dimensional point $(d(q, p_1), \ldots, d(q, p_k))$. With this information at hand, we can filter out using the triangle inequality any element u such that $|d(q, p_i) - d(u, p_i)| > r$ for some pivot p_i, since in that case we know that $d(q, u) > r$ without need to evaluate $d(u, q)$. Those elements that cannot be filtered out using this rule are directly compared against q. The more pivots we use, the more elements are discarded, but the index needs more space and computing the coordinates of the query gets more expensive.

If one is not only interested in the number of distance evaluations performed but also in the total CPU time required, then scanning all the n elements to filter out some of them may be unacceptable. In that case, one needs *multi-dimensional range search* methods, which include data structures such as the kd-tree, R-tree, X-tree, etc. [49,19,8]. Those structures permit indexing a set of objects in k-dimensional space in order to process range queries.

The second class of techniques for metric space searching is *compact partitioning*, where the set $\mathbb{U}$ is partitioned into classes which are spatially as compact as possible, that is, trying to reduce intra-class distances. Some information on the classes is stored so as to discard them wholly with a few comparisons. For example, one can store minimum an maximum distances, $m(c)$ and $M(c)$, between the class and a distinguished element $c \in \mathbb{U}$. At query time, if $d(q, c) + r < m(c)$ or $d(q, c) - r > M(c)$, then no element in the class can be relevant for the query $(q, r)_d$. Otherwise, the class must be inspected.

In this paper we are interested in a metric space where the universe is the set of strings over some alphabet, i.e. $\mathbb{X} = \Sigma^*$, and the distance function is the so-called *edit distance* or *Levenshtein distance*. This is defined as the minimum

number of character insertions, deletions and substitutions necessary to make two strings equal [29,36]. The edit distance, and in fact any other distance defined as the best way to convert one element into the other, is reflexive, strictly positive (as long as there are no zero-cost operations), symmetric (as long as the operations allowed are symmetric), and satisfies the triangle inequality.

The algorithm to compute the edit distance $ed()$ is based on dynamic programming. Imagine that we need to compute $ed(a, b)$. A matrix $C_{0..|a|,0..|b|}$ is filled, where $C_{i,j} = ed(a_{1..i}, b_{1..j})$, so $C_{|a|,|b|} = ed(a, b)$. This is computed as

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

The algorithm takes $O(|a||b|)$ time. The matrix can be filled column-wise or row-wise. If we choose row-wise filling, for example, the space required is only $O(|b|)$, since only the previous row must be stored in order to compute the new one, and therefore we just keep one row and update it.

3 Text Indexing

Suffix trees are widely used data structures for text processing [1,21]. Any position i in a text T defines a *suffix* of T, namely $T_{i...}$. A *suffix trie* is a trie data structure built over all the suffixes of T. At the leaf nodes the pointers to the suffixes are stored. Every substring of T can be found by traversing a path from the root. Roughly speaking, each suffix trie leaf represents a suffix and each internal node represents a different repeated substring of T.

To improve space utilization, this trie is compacted into a Patricia tree [34] by compressing unary paths. The edges that replace a compressed path store the whole string that they represent (via two pointers to their initial and final text position). Once unary paths are not present the trie, now called *suffix tree*, has $O(n)$ nodes instead of the original worst-case $O(n^2)$. The suffix tree can be directly built in $O(n)$ time [33,48]. Any algorithm on a suffix trie can be simulated at the same cost in the suffix tree.

We call *explicit* those suffix trie nodes that survive in the suffix tree, and *implicit* those that are collapsed. Fig. 2 shows the suffix trie and tree of the text `"abracadabra"`. Note that a special endmarker `"$"`, smaller than any other character, is appended to the text so that all the suffixes are external nodes.

The figure shows the *internal* nodes of the trie (numbered 0 to 9 in italics inside

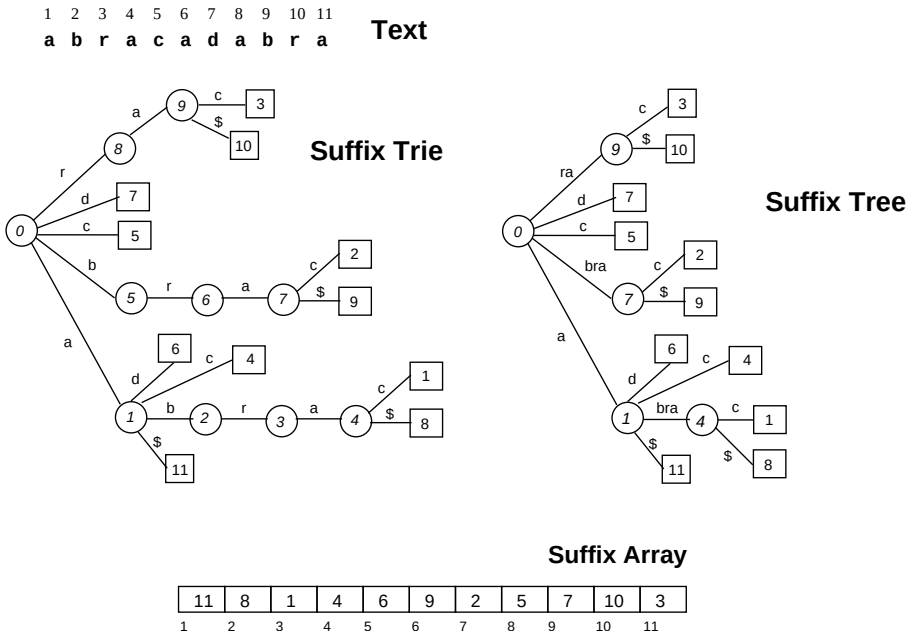

Fig. 2. The suffix trie, suffix tree and suffix array of the text `"abracadabra"`.

circles, and that will be called $i(0)$ to $i(9)$), which represent text substrings that appear more than once, and the *external* nodes (numbered 1 to 11 inside squares, and that will be called $e(1)$ to $e(11)$), which represent text substrings that appear just once. Those leaves do not only represent the unique substrings but all their extensions until the full suffix. In the suffix tree, only some internal nodes are left, and they represent the same substring as before plus some prefixes of it that may have been collapsed. For example node $i(7)$ of the suffix tree represents now the compressed nodes $i(5)$ and $i(6)$, and hence the strings `"b"`, `"br"` and `"bra"`. The node $e(1)$ represents `"abrac"`, but also `"abraca"`, `"abracad"`, etc. until the full suffix `"abracadabra"`.

The suffix array [31] is a more compact version of the suffix tree, which requires much less space and poses a small penalty over the search time. If the leaves of the suffix tree are traversed in left-to-right order, all the suffixes of the text are retrieved in lexicographical order. A suffix array is simply an array containing all the pointers to the text suffixes listed in lexicographical order, as shown in Fig. 2. The suffix array stores one pointer per text position.

The suffix array can be directly built (without building the suffix tree) in $O(n)$ time [26–28]. While suffix trees are searched as tries, suffix arrays are binary searched. However, almost every algorithm on suffix trees can be adapted to work on suffix arrays at an $O(\log n)$ penalty factor in the time cost. This is because each subtree of the suffix tree corresponds to an interval in the suffix array, namely the one containing all the leaves of the subtree. To follow an edge of the suffix trie, we use binary search to find the new limits in the suffix array. For example, the node $i(7)$ in the suffix tree corresponds to the interval

$\langle 6, 7\rangle$ in the suffix array. Note that implicit nodes have the same interval than their representing explicit node.

4 Our Algorithm

4.1 Indexing

A straightforward approach to text indexing for approximate string matching using metric space techniques has the problem that, in principle, there are $O(n^2)$ different substrings in a text, and therefore we should index $O(n^2)$ objects, which is unacceptable.

The suffix tree provides a concise and implicit representation of all the substrings of a text in $O(n)$ space. So instead of indexing all the text substrings explicitly, we index only the (explicit) suffix tree nodes. This is equivalent to a compact partitioning method, where we have derived $O(n)$ classes from $O(n^2)$ objects. This particular partitioning has several interesting properties that will become apparent soon.

Each explicit internal node represents itself and the nodes that descend to it by a unary path. Hence, each explicit node that corresponds to a string xy, where its parent corresponds to the string x, represents the following set of strings

$$x[y] \quad = \quad \{xy_1,\ xy_1y_2,\ \ldots,\ xy\}$$

where $x[y]$ is a notation we have just introduced. For example, the node $i(4)$ in Fig. 2 represents the strings `"a[bra]"` = {"`ab`", "`abr`", "`abra`"}.

The leaves of the suffix tree represent a unique text substring and all its extensions until the full text suffix is obtained. Hence, if $T = zxcy$, for a character c, so that xc is a unique text substring and x is not unique, then the suffix tree node for xc is an explicit leaf, which for us represents the set $x[cy]$. Table 1 shows the substrings represented by each node in our running example. Note that the external nodes that descend by the terminator character `"$"`, i.e. e(8–11), represent a substring that is also represented at its parent and hence it can be disregarded.

Hence, instead of indexing all the $O(n^2)$ text substrings individually, we first partition them into $O(n)$ sets of strings, which are the sets represented by the explicit internal and the external nodes of the suffix tree. In our example, this set is

Node	Suffix trie	Suffix tree	Node	Suffix trie/tree
$i(0)$	ε	ε	$e(1)$	`abra[cadabra]`
$i(1)$	`a`	`[a]`	$e(2)$	`bra[cadabra]`
$i(2)$	`ab`		$e(3)$	`ra[cadabra]`
$i(3)$	`abr`		$e(4)$	`a[cadabra]`
$i(4)$	`abra`	`a[bra]`	$e(5)$	`[cadabra]`
$i(5)$	`b`		$e(6)$	`a[dabra]`
$i(6)$	`br`		$e(7)$	`[dabra]`
$i(7)$	`bra`	`[bra]`	$e(8)$	`abra`
$i(8)$	`r`		$e(9)$	`bra`
$i(9)$	`ra`	`[ra]`	$e(10)$	`ra`
			$e(11)$	`a`

Table 1
The text substrings represented by each node of the suffix trie and tree of Fig. 2. The suffix tree has only the explicit internal nodes.

$$\mathbb{U} = \{\varepsilon, [\texttt{a}], \texttt{a}[\texttt{bra}], [\texttt{bra}], [\texttt{ra}], \texttt{abra}[\texttt{cadabra}], \texttt{bra}[\texttt{cadabra}], \texttt{ra}[\texttt{cadabra}], \texttt{a}[\texttt{cadabra}], [\texttt{cadabra}], \texttt{a}[\texttt{dabra}], [\texttt{dabra}]\}$$

We have now to decide how to index these $O(n)$ sets of strings. Many options are possible, but we have chosen a pivot based approach. We select k different text substrings that will be our *pivots*. We consider in the next section how to choose the pivots. For each explicit suffix tree node $x[y]$ and each pivot p_i, we compute the distance between p_i and all the strings represented by $x[y]$. From the set of distances from a node $x[y]$ to p_i, we store the minimum and maximum ones. Since all these strings are of the form $\{xy_1 \dots y_j,\ 1 \leq j \leq |y|\}$, all the edit distances can be computed in $O(|p_i||xy|)$ time. In Eq. (1), take $a = xy$ and $b = p_i$. The computation of $ed(xy, p_i)$ will also yield $C_{|x|+j,|p_i|} = ed(xy_1 \dots y_j, p_i)$ for all j.

Following our example, let us assume that we have selected $k = 5$ pivots $p_1 = \varepsilon$, $p_2 =$ `"a"`, $p_3 =$ `"br"`, $p_4 =$ `"cad"` and $p_5 =$ `"raca"`. Fig. 3 (left) shows the computation of the edit distances between $i(4) =$ `"a[bra]"` and $p_4 =$ `"cad"`. The result shows that the minimum and maximum values of this node with respect to this pivot are 2 and 4, respectively.

In the case of external suffix tree nodes, the string y tends to be quite long ($O(n)$ length on average), which yields a very high computation time for all the edit distances and anyway a very large value for the maximum edit distance (note that $ed(p_i, xy) \geq |xy| - |p_i|$). We solve this by pessimisti-

		c	a	d
	0	1	2	3
a	1	1	1	2
b	2	2	2	**2**
r	3	3	3	**3**
a	4	4	3	**4**

		c	a	d
abra	4	4	3	4
c	5	4	4	**4**
a	6	5	4	**5**
d	7	6	5	**4**
a	8	7	6	**5**
b	9	8	7	**6**
r	10	9	8	**7**
a	11	10	9	**8**

Fig. 3. The dynamic programming matrix to compute the edit distance between `"cad"` and `"a[bra]"` (left) or `"abra[cadabra]"` (right). The emphasized area is where the minima and maxima are taken from.

cally assuming that the maximum distance is n when the suffix tree node is external. The minimum edit distance can be found in $O(|p_i| \max(|p_i|, |x|))$ time, because it is not necessary to consider arbitrarily long strings $xy_1...y_j$: If we compute the matrix row by row, then after having processed $xy_1...y_j$ we have a minimum value seen up to now, $v_j = \min_{1 \leq j' \leq j} C_{|x|+j',|p_i|}$. Then there is no point in considering rows $|x| + j''$ such that $|x| + j'' - |p_i| > v_j$, as $ed(xy_{1...j''}, p_i) \geq |x| + j'' - |p_i| > v_i$, and this gives us an early termination criterion. In particular, just considering $j = 1$ we have that we work at most until row $j'' = v_1 + |p_i| - |x| = ed(xy_1, p_i) + |p_i| - |x| \leq \max(|p_i| + 1, 2|p_i| - |x|)$ (as $ed(xy_1, p_i) \leq \max(|x| + 1, |p_i|)$). Thus the total amount of work is $O(|p_i|(|x| + j'')) = O(|p_i| \max(|p_i|, |x|))$.

Fig. 3 (right) illustrates this case with $e(1) =$ `"abra[cadabra]"` and the same $p_4 =$ `"cad"`. Note that it is not necessary to compute the last 4 rows, since they measure the edit distance between strings of length 8 or more against one of length 3. The distance cannot be smaller than 5 and we have found at that point a minimum equal to 4 (actually we could have even avoided computing the row for `"d"`). As we assume that the maximum is 11, the minimum and maximum value for this external node and this pivot are 4 and 11.

Note also that to compute the new set of edit distances we have started from $i(4)$, which is the parent node of $e(1)$ in the suffix tree. This can always be done in a depth first traversal of the suffix tree and saves construction time. Since when indexing external nodes $x[y]$ we always have $ed(p_i, x)$ already computed, the cost to index an external node becomes $O(|p_i|j'') = O(|p_i|^2)$.

Once the distances are computed between all the suffix tree nodes and all the pivots, we have a set of k minimum and maximum values for each explicit

suffix tree node. This can be regarded as a hyperrectangle in k dimensions:

$$x[y] \rightarrow \langle \quad (\min(ed(x[y], p_1)), \ldots, \min(ed(x[y], p_k))), \\ (\max(ed(x[y], p_1)), \ldots, \max(ed(x[y], p_k))) \quad \rangle$$

where we are sure that all the strings in $x[y]$ lie inside its rectangle. In our example, the minima and maxima for $i(4)$ with respect to $p_{1\ldots5}$ are $\langle 2, 4\rangle$, $\langle 1, 3\rangle$, $\langle 1, 2\rangle$, $\langle 2, 4\rangle$ and $\langle 3, 3\rangle$. Therefore $i(4)$ is represented by the hyperrectangle $\langle (2, 1, 1, 2, 3), (4, 3, 2, 4, 3)\rangle$. On the other hand, the ranges for $e(1)$ are $\langle 5, 11\rangle$, $\langle 4, 11\rangle$, $\langle 3, 11\rangle$, $\langle 4, 11\rangle$ and $\langle 2, 11\rangle$ and its hyperrectangle is therefore $\langle (5, 4, 3, 4, 2), (11, 11, 11, 11, 11)\rangle$.

4.2 Searching

Let us now consider a given query pattern $P_{1\ldots m}$ searched for with at most r errors. This is a range query with radius r in the metric space of the substrings of $T_{1\ldots n}$. As for pivot based algorithms, we compare the pattern P against the k pivots and obtain a k-dimensional coordinate $(ed(P, p_1), \ldots, ed(P, p_k))$, and as for compact partitioning algorithms, we use bounds to discard the whole classes (that is, suffix tree nodes).

Let p_i be a given pivot and $x[y]$ a given node. If it holds that

$$ed(P, p_i) + r < \min(ed(x[y], p_i)) \quad \vee \quad ed(P, p_i) - r > \max(ed(x[y], p_i)) \quad (2)$$

then, by the triangle inequality, we know that $ed(P, xy') > r$ for any $xy' \in x[y]$. The elimination can be done using any pivot p_i. In fact, the nodes that are not eliminated are those whose rectangle has nonempty intersection with the rectangle $\langle (ed(P, p_1) - r, \ldots, ed(P, p_k) - r), (ed(P, p_1) + r, \ldots, ed(P, p_k) + r)\rangle$.

Fig. 4 illustrates. The node contains a set of points and we store their minimum and maximum distance to two pivots. These define a (2-dimensional) rectangle where all the distances from any substring of the node to the pivots lie. The query is a pattern P and a tolerance r, which defines a circle around P. After taking the distances from P to the pivots we create a hypercube (a square in this case) of width $2r + 1$. If the square does not intersect the rectangle, then no substring in the node can be close enough to P.

We have to solve the problem of finding all the k-dimensional rectangles that intersect a given query rectangle. This is a classical multidimensional range search problem [49,19,8]. We could for example use some variant of R-trees [22,7], which would also yield a good data structure to work on secondary memory.

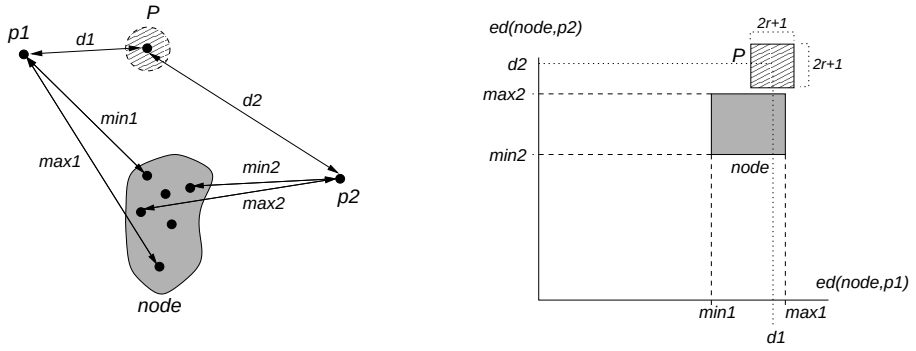

Fig. 4. The elimination rule using two pivots.

Those nodes $x[y]$ that cannot be eliminated using any pivot must be directly compared against P. For those whose minimum distance to P is at most r, we report all their occurrences, whose starting points are written in the leaves of the subtree rooted by the node that has matched. In our running example, if we are searching for `"cab"` with tolerance $r = 1$, then implicit node $i(2)$ matches, and thus explicit node $i(4)$ qualifies, so we report the text positions in the corresponding tree leaves: 1 and 8. Node $e(5)$ also qualifies and we report text position 5.

Observe that, in order to compare P against a given suffix tree node $x[y]$, the edit distance algorithm forces us to compare it against every prefix of x as well. Those prefixes correspond to suffix tree nodes in the path from the root to $x[y]$. In order not to repeat work, we first mark in the suffix tree the nodes that we have to compare explicitly against P, and also mark every node in their path to the root. Then, we backtrack on the suffix tree entering every marked node and keeping track of the edit distance between P and the node. The new row is computed using the row of the parent, just as done with the pivots. This avoids recomputing the same prefixes for different suffix tree nodes, and incidentally is similar to the simplest backtracking approach [20], except that in this case we only follow marked paths. In this respect, our algorithm can be thought of as a preprocessing to a backtracking algorithm, which filters out some paths.

As a practical matter, note that this is the only step where the suffix tree is required. We can even print the text substrings that match the pattern without the help of the suffix tree, but we need it in order to report all their text positions. For this sake, a suffix array is much cheaper and does a better job (because all the text positions are listed in a contiguous interval). In fact, the suffix array can also replace the suffix tree at indexing time, leaving the suffix tree just as a conceptual device to group text substrings.

A simpler index considers only the n text suffixes and no internal nodes. Each set $[T_{j...}]$ represents all the text substrings starting at j, and it is indexed according to the minimum distance between those substrings and each pivot.

The good point of the approach is reduced space. Not only the set $\mathbb{U}$ can have up to half the elements of the original approach, but also only k values (not $2k$) are stored for each element, since all the maximum values are the same. This permits using up to four times the number of pivots of the previous approach at the same memory requirement. Note that we do not even need to build or store the suffix tree or array: We just read the suffixes from the text and index them. Our only storage need is that of the metric index.

The bad point is that the selectivity of the pivots is reduced and some redundant work is done. The first is a consequence of storing only minimum values, while the second is a consequence of not factoring out repeated text substrings. That is, if some substring P' of T is close enough to P and it appears many times in T, we will have to check all its occurrences one by one.

Without using a suffix tree structure, the construction of the index can be done in worst case time $O(k|p_i|n)$ as follows. The algorithm depicted in Section 2 to compute edit distance can be modified so as to make $C_{0,j} = 0$, in which case $C_{i,j}$ becomes the minimum edit distance between $x_{1...i}$ and a suffix of $y_{1...j}$. If x is the reverse of p_i and y the reverse of T, then $C_{|p_i|,j}$ will be the minimum edit distance between p_i and a prefix of $T_{n-j+1...}$, which is precisely $\min(ed(p_i, [T_{n-j+1...}]))$. So we need $O(|p_i|n)$ time per pivot. The space to compute this is just $O(|p_i|)$ by doing the computation column-wise.

5 Analysis and a Specific Index

Although our approach is general, obtaining a given complexity requires careful pivot selection. We start with a general analysis of our method. Then, we give a specific pivot selection technique, which yields our final complexities.

5.1 Index Space and Construction Time

The index space is just the necessary to store the suffix tree, the set of k pivots, and the R-tree. Assume our pivots are of length ℓ. The suffix tree is $O(n)$ size, the pivots require $k\ell$ characters (unless they are implicit somehow), and the R-tree requires $O(kn)$ space, as it stores $O(n)$ objects of k coordinates

each. Thus the overall space is $O(kn)$. This is the same space complexity of the simplified version of Section 4.3, where we do not need any suffix tree or array.

Let us now consider construction cost. We have first to build the suffix tree of T, which can be done in $O(n)$ time. We have then to compute minimum and maximum distances from each p_i and each suffix tree node $x[y]$. Let us first consider internal tree nodes. As we start the matrix computation of node $x[y]$ starting from its parent string x, we pay only $O(|p_i||y|)$ time for each pivot p_i, in node $x[y]$. Since the pivot lengths are ℓ and there are k such pivots, we pay overall $O(k\ell|y|)$ in internal suffix tree node $x[y]$. This suffix tree node represents precisely $|y|$ suffix trie nodes xy_1, xy_1y_2, and so on until xy. Thus, the construction cost for internal suffix tree nodes is $O(k\ell)$ times the number of suffix trie nodes. This number of nodes is $O(n)$ on average and with high probability [45,42], thus the construction cost for internal nodes is $O(k\ell n)$ on average. In worst case, there could be $O(n^2)$ suffix trie nodes and our construction would take time $O(k\ell n^2)$.

Consider now external suffix tree nodes. As explained, for node $x[y]$ we pay $O(|p_i| \max(|p_i|, |x|))$. Yet, we have node x already computed at its parent, so the time becomes $O(|p_i|^2) = O(\ell^2)$. Multiplied by the k pivots and the $O(n)$ external nodes, we get $O(k\ell^2 n)$ construction time for external nodes.

Overall, construction cost is $O(k\ell^2 n)$ on average and $O(k\ell n^2)$ in the worst case. To this we must add the cost to populate the R-tree with the $O(n)$ hyperrectangles, $O(kn \log n)$. In the simpler version of Section 4.3, the worst-case cost is $O(k\ell n)$ plus the cost to populate the R-tree, $O(kn \log n)$.

5.2 Query Time

The query cost has four parts: (1) computing distances against the pivots, (2) searching for the candidates in the R-tree, (3) comparing the query against the candidates found, (4) reporting all the *occ* positions.

The first part is simply k times the cost to compute a distance between the query and a pivot, $O(k\ell m)$.

The second part is the search for the hyperrectangles that intersect the query hyperrectangle. Many analyses of the performance of R-trees exist in the literature [46,25,39,40,17]. Despite that most of them deal with the exact number of disk accesses, their abstract result is that the expected amount of work on the R-tree (and variants such as the KDB-tree [41]) is $O(np \log n)$, where p is the probability that the query hyperrectangle intersects a random hyperrectangle in the set of $O(n)$ elements we index. Put another way, p is the

probability that the k pivots do not permit us discarding a random suffix tree node. This probability depends on m, k, and r, and we consider it soon.

The third part, is the direct verification of the pattern against the suffix tree nodes whose rectangles intersect the query rectangle. Since the probability of not discarding a node with the pivots is p, and there are $O(n)$ suffix tree nodes, we check on average $O(pn)$ nodes. This is true even if the suffix tree nodes are not independent of each other, since the expectation commutes with the sum even on dependent random variables. The overall verification work can be seen as the sum of random variables X_v with value 1 if suffix tree node v is not discarded. Those $O(pn)$ verifications yield a total verification cost of $O(pnm^2)$. The m^2 is the cost to compute the edit distance between a pattern of length m and a candidate whose length must be between $m-r$ and $m+r$ (otherwise we know that the candidate is irrelevant). Note that one suffix tree node might correspond to several substrings, but those are of the form $x[y]$, and all those can be computed in time $O(m|xy|)$.

At the end, we report the *occ* results in $O(occ)$ time using a suffix tree traversal. Hence our total average cost is bounded by $km\ell + np(\log n + m^2) + occ$. The problem we address next is how to choose pivots to obtain a given p. This analysis applies verbatim to the simpler structure of Section 4.3, where now p is the probability that we cannot discard a suffix $[T_{j\ldots}]$.

5.3 Choosing Pivots

Not any pivot selection strategy gives good performance. In particular, choosing k random pivots does not work well, as a term of the form σ^r appears in the average time complexity.

A strategy that works well is as follows. Choose a small positive integer q (according to some bounds to be derived later). Pad T with dummy characters at the end so that its length is a multiple of q. Create a new text T' as the sequence of q-grams of T, that is, $T' = T_{1\ldots q}T_{q+1\ldots 2q}\ldots T_{n-q+1\ldots n}$ (each character in T' is a q-gram in T). This is equivalent to having a text of length $\lceil n/q \rceil$ over alphabet Σ^q, of size $\sigma' = \sigma^q$. To search for $P_{1\ldots m}$ we truncate P to length m_t, $m - q < m_t \leq m$, so that q divides $m_t + 1$, and then perform q searches for the patterns $P^s = P_{s+1\ldots s+q}P_{s+q+1\ldots s+2q}\ldots P_{s+m_t-2q+2\ldots s+m_t-q+1}$, $0 \leq s < q$. All those P^s are of length $m' = (m_t+1)/q - 1 > m/q - 2$ (measured in q-grams). It is necessary to perform the q searches to account for all the possible different alignments of an occurrence of P in the q-grams of T'. Finally, we must maintain the same threshold r, since if P is at distance d to $T_{i\ldots j}$, then their q-gram versions can be at distance at most d too (each error modifying a different q-gram).

For text T', we choose $k = \sigma'$ pivots of the form $p_i = c_i^{m'}$ (that is, a sequence of m' q-grams c_i), for each $c_i \in \Sigma^q$. In this case, $ed(P^s, p_i)$ is m' minus the number of occurrences of c_i in P^s (note that we regard c_i as a character, not as a string of length q). All those distances to all the P^s patterns can be computed in $O(q\sigma^q + m)$ time: We initialize all the $q\sigma^q$ distances at $ed(P^s, p_i) = m'$ and then traverse all the $m_t - q + 1$ q-grams of P once, so that at step j we decrease by 1 the distance $ed(P^s, p_i)$ such that $P_{j\ldots j+q-1} = c_i$ and $s = j \mod q$. This is different from our general analysis, as we take advantage of the particular structure of our pivots to pay less than $O(k\ell m)$.

Let us concentrate on one of those P^s patterns for a while. Assume pessimistically that, for the $O(n')$ suffix tree nodes $x[y]$, we discard them only if $ed(P^s, p_i) + r < \min ed([xy], p_i)$, instead of the more relaxed condition of Eq. (2). In complexity terms, this is equivalent to indexing only the n' text suffixes $[T'_{j\ldots}]$. In particular, this is precisely the case of the simpler index of Section 4.3. For our pivots, it is easy to see that $\min ed([T'_{j\ldots}], p_i) \geq ed(T'_{j\ldots j+m'-1}, p_i)$. Thus we discard text position j whenever $ed(P^s, p_i) + r < ed(T'_{j\ldots j+m'-1}, p_i)$.

Given that we use all the σ' pivots to discard, text position j will be discarded if, for some q-gram c_i, the number of occurrences of c_i in P^s plus r still does not reach the number of occurrences of c_i in $T'_{j\ldots j+m'-1}$. Given a pattern P^s, the set of different strings it is unable to discard is obtained by permuting its q-grams in every possible way, then choosing r positions, and changing the q-grams at those positions (possibly by themselves). An upper bound to the number of such strings is $m'!\binom{m'}{r}(\sigma')^r = O((m')^{m'+r+1/2}(\sigma')^r/e^{m'})$.

The probability of a suffix $[T'_{j\ldots}]$ not being discarded is precisely that of string $T'_{j\ldots j+m'-1}$ belonging to the set above, that is, at most

$$p \;=\; p(m', r) \;=\; O\left(\frac{(m')^{m'+r+1/2}(\sigma')^r}{e^{m'}(\sigma')^{m'}}\right) \;=\; O\left(\frac{(m/q)^{m/q+r+1/2}\sigma^{qr}}{e^{m/q}\sigma^{m-2q}}\right).$$

We recall that we have to perform q searches for different P^s patterns. Following the general analysis of the previous section, the overall search cost is

$$O(q\sigma^q + m + qn'p(m', r)(m^2 + \log n')) \tag{3}$$

where we note that $qn' = n$. We examine now for which q can we make the third term insignificant. It is sufficient that $p(m', r) = O(1/(nm \log n))$. Taking logarithms, we get

$$\left(\frac{m}{q} + r + \frac{1}{2}\right)\log_\sigma \frac{m}{q} + qr - \frac{m}{q}\log_\sigma e - m + 2q \leq -\log_\sigma n - \log_\sigma m - \log_\sigma \log n,$$

or, slightly more stringent,

$$m \;\geq\; \log_\sigma n + \log_\sigma \log n + q(r+2) + \left(\frac{m}{q} + r + \frac{3}{2}\right) \log_\sigma m,$$

where we remark that it is sufficient that this condition holds for sufficiently large m.

It is clear that, if $(m/q)\log_\sigma m \geq m$, the condition cannot hold. Therefore, we require $q \geq (1/\alpha)\log_\sigma m$ for some constant $\alpha < 1$. Similarly, we need $q \leq \beta m/(r+2)$ for some constant $\beta < 1$, so that $q(r+2)$ is bounded away from m. These conditions on q together imply $r + 2 \leq \alpha\beta m/\log_\sigma m$. If, in addition, it holds $\alpha + \beta + \alpha\beta < 1$, then the condition is satisfied for

$$m \;\geq\; \frac{\log_\sigma n + \log_\sigma \log n}{1 - \alpha - \beta - \alpha\beta}.$$

Note that the limit on r, together with condition $\alpha+\beta+\alpha\beta < 1$, implies that $r + 2 < (3 - 2\sqrt{2})m/\log_\sigma m$ ($3 - 2\sqrt{2} \approx 0.172$). Given an r satisfying this condition, we can always find proper α and β values so that there is a feasible q in the interval $[(1/\alpha)\log_\sigma m, \beta m/(r+2)]$.

Thus, we have found the conditions for the third term in Eq. (3) to be $O(m)$. The first term is $q\sigma^q$. Thus we wish to keep q as small as possible, $q = (1/\alpha)\log_\sigma m$, thus $q\sigma^q = O(m^{1/\alpha}\log_\sigma m)$. This term dominates the complexity, so we wish to make α as close as possible to 1. This maximum possible value is $\alpha < (1 - \gamma + \sqrt{\gamma^2 - 6\gamma + 1})/2$, where $\gamma = (r+2)\log_\sigma(m)/m$. Therefore the complexity we can achieve is $O(m^{1/\alpha+\epsilon}) = O(m^{\frac{2}{1-\gamma+\sqrt{\gamma^2-6\gamma+1}}+\epsilon})$, for any constant $\epsilon > 0$. This is $O(m^{1+\epsilon})$ if $r = o(m/\log_\sigma m)$ (that is, $\gamma = o(1)$). As γ grows until $\gamma < 3 - 2\sqrt{2}$, we have decreasing α limits, up to $\alpha < \sqrt{2} - 1$. Hence, the maximum complexity we can have, for the highest error level we can handle, is $O(m^{1+\sqrt{2}+\epsilon})$ ($1 + \sqrt{2} \approx 2.4142$).

Note that the applicability condition on m depends on the complexity we wish. If $r = o(m/\log_\sigma m)$, then we can obtain search time $O(m^{1/\alpha+\epsilon})$ for $m > \frac{\log_\sigma n}{1-\alpha}$, that is, $O(m^{1+\epsilon})$ for $m > \frac{1+\epsilon}{\epsilon}\log_\sigma n$. For constant γ, the limit is $m > \frac{\log_\sigma n}{1-\alpha-(1+1/\alpha)\gamma}$.

Let us consider construction time and space. It is not hard to compute the distances to all the pivots simultaneously in a single suffix tree traversal, as these are just counts of how many occurrences of each character are there in the path from the root to the current node. Therefore this part is just $O(\sigma^q + n)$ time on average and $O(\sigma^q + n^2)$ in the worst case, using our specific pivots. However, the simpler index of Section 4.3 requires only $O(\sigma^q + n)$ time, by sliding a window of length m' over T', so as to update in $O(q)$

time the number of occurrences of the c_i's for $[T'_{j...}]$ from those of $[T'_{j-1...}]$. As the index of Section 4.3 gives improved construction time and similar search complexity, we stick to its analysis. Thus, construction time is dominated by the time to load the R-tree, $O(\sigma^q(n/q)\log n) = O(m^{1/\alpha}n\log n)$. The space is $O(\sigma^q(n/q)) = O(m^{1/\alpha}n)$.

A problem with our approach is that we must know m and r at indexing time in order to choose q. Other m and r different from those used at indexing time are correctly handled, but may not yield the complexities we give here if the differences are significant. In many applications it is perfectly reasonable to assume that tight bounds on the m and r values of interest are known. Another alternative is to use different sets of pivots to reasonably cover the range of relevant m, r values.

6 A Linear Space Index

The index we have presented has another drawback, namely its superlinear space requirement. We now present a variant which, although does not give relevant complexity bounds, takes linear space and promises to be practical.

The space of strings has a distance distribution that is rather concentrated around its mean μ [36]. The same happens to the distances between a pivot p_i and sets $[T_{j...}]$ or the pattern P. If we only discard sets $[T_{j...}]$ such that $ed(p_i, P)+r < \min(ed(p_i, [T_{j...}]))$, only the suffixes with a large $\min(ed(p_i, [T_{j...}]))$ value are likely to be discarded using p_i. Storing all the other $O(n)$ distances to p_i is likely to be a waste of space. Moreover, we can use that memory to introduce more pivots. Fig. 5 illustrates.

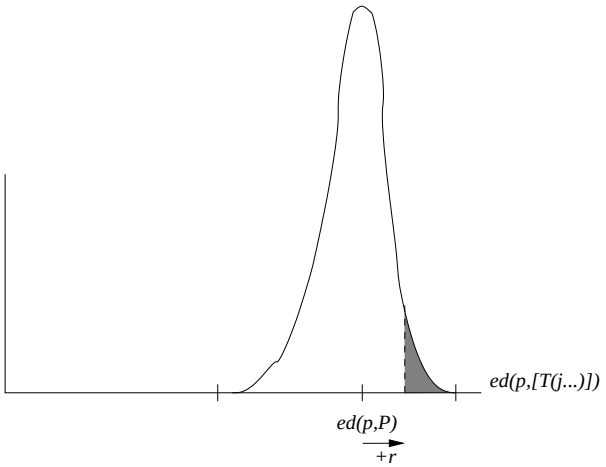

Fig. 5. The distance distribution to a pivot p, including that of pattern P. The grayed area represents the suffixes that can be discarded using p.

The idea is to fix a number s and, for each pivot p_i, store only the s largest $\min(ed(p_i, [T_{j\ldots}]))$ values. Only those suffixes can be discarded using pivot p_i. The space of this index is $O(ks)$ and its construction time is unchanged. We can still use an R-tree for the search, although the rectangles will cover all the space except on s coordinates. The selectivity is likely to be similar since we have discarded uninteresting coordinates, and we can tune number k versus selectivity s of the pivots for the same space usage $O(ks)$.

One can go further to obtain $O(n)$ space as follows. Choose the first pivot and determine its s farthest suffixes. Store a list (in increasing distance order) of those suffixes and their distance to the first pivot and remove them from further consideration. Then choose a second pivot and find its s farthest suffixes from the remaining set. Continue until every suffix has been included in the list of some pivot. Note that every suffix appears exactly in one list. At search time, compare P against each pivot p_i, and if $ed(P, p_i) + r$ is smaller than the smallest (first) distance in the list of p_i, skip the whole list. Otherwise traverse the list until its end or until $ed(P, p_i)+r$ is smaller than the next element. Each traversed suffix must be directly compared against P. A variant of this idea has proven extremely useful to deal with concentrated histograms [12]. It also permits efficient secondary storage implementation by packing the pivots in disk pages and storing the lists consecutively in the same order of the pivots.

Since we choose $k = n/s$ pivots, the construction time is high, $O(n^2|p_i|/s)$. However, the space is $O(n)$, with a low constant (close to 5 in practice) that makes it competitive against the most economical structures for the problem. The search time is $O(|p_i|mn/s)$ to compare P against the pivots, whereas the time to traverse the lists is difficult to analyze.

The pivots chosen must not be very short, because their minimum distance to any $[T_{j\ldots}]$ is at most $|p_i|$. In fact, any pivot not longer than $(m+r)/2$ is useless, as $ed(p_i, P) \geq m-|p_i| \geq m-(m+r)/2 = (m-r)/2$, so $ed(P, p_i)+r \geq (m+r)/2$, which can never be smaller than $\min ed(p_i, [T_{j\ldots}]) \leq |p_i| \leq (m+r)/2$.

We can complement the information given by the metric index with knowledge of the string properties we are indexing to increase suffix pruning. For example, if the set $[T_{j\ldots}]$ is proven to be at distance $r+t$ from P, then we can also discard sets $[T_{j-t+1\ldots} \ldots T_{j+t-1\ldots}]$.

Another idea is to compute the edit distance between the reverse pivot and the reverse pattern. Although the result is the same, we learn also the distances between the pivot and suffixes of the pattern, thus we learn lower bounds to $ed(P_{\ell\ldots}, [T_{i'+1\ldots}])$ for all ℓ. This can be useful to discard suffixes at verification time: If we are verifying $[T_{i\ldots}]$ and, at some point, it holds $\min_\ell ed(P_{1\ldots\ell-1}, T_{i\ldots i'}) + ed(P_{\ell\ldots}, [T_{i'+1\ldots}]) > r$, then a match is not possible starting at text position i.

7 Conclusions

We have presented a novel approach to the approximate string matching problem. The idea is to give the set of text substrings the structure of a metric space and then use an algorithm for range queries on metric spaces. The suffix tree is used as a conceptual device to map the $O(n^2)$ text substrings to $O(n)$ sets of strings. For (roughly) $r = O(m/\log_\sigma m)$ and $m = \Omega(\log_\sigma n)$, we can search at an average cost lower than $O(m^{2.5} + occ)$ using less than $O(m^{2.5}n)$ space. This is a breaktrhough over previous methods, which achieved complexities polynomial in m and independent of n only for $r = O(m/\log_\sigma n)$.

Moreover, our technique can be extended to any other distance function among strings, some of which, like the reversals distance, are problematic to handle with the previous approaches.

The proposal opens a number of possibilities for future work. We plan to explore other methods to reduce the number of substrings (we have used the suffix tree nodes and the suffixes), other metric space indexing methods (we have used a combination of pivots and compact partitions, but there are many choices [13,23]), other multidimensional range search techniques (we have used R-trees, but there are better choices for higher dimensions [8]), other pivot selection techniques (which might work better in practice), etc. In particular, we are interested in an index that does not need knowledge on m or r at construction time and that requires linear space. A setup needing $O(n)$ space has been described in Section 6.

Finally, the method promises an efficient implementation on secondary memory (e.g., with R-trees), which is a weak point in most current approaches.

Acknowledgements

We thank Paolo Ferragina for pointing out some mistakes in our conference paper.

References

[1] A. Apostolico. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words*, NATO ISI Series, pages 85–96. Springer-Verlag, 1985.

[2] A. Apostolico and Z. Galil. *Combinatorial Algorithms on Words*. Springer-Verlag, New York, 1985.

[3] R. Baeza-Yates and G. Navarro. Fast approximate string matching in a dictionary. In *Proc. SPIRE'98*, pages 14–22. IEEE Computer Press, 1998.

[4] 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`.

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

[6] R. Baeza-Yates and G. Navarro. A hybrid indexing method for approximate string matching. *J. of Discrete Algorithms (JDA)*, 1(1):205–239, 2000.

[7] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proc. ACM SIGMOD'90*, pages 322–331, 1990.

[8] C. Böhm, S. Berchtold, and D. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Comp. Surv.*, 33(3):322–373, September 2001.

[9] A. Buchsbaum, M. Goodrich, and J. Westbrook. Range searching over tree cross products. In *Proc. ESA'00*, LNCS 1879, pages 120–131, 2000.

[10] E. Bugnion, T. Roos, F. Shi, P. Widmayer, and F. Widmer. Approximate multiple string matching using spatial indexes. In *Proc. WSP'93*, pages 43–54, 1993.

[11] E. Chávez and G. Navarro. A metric index for approximate string matching. In *Proc. LATIN'02*, LNCS 2286, pages 181–195, 2002.

[12] E. Chávez and G. Navarro. A compact space decomposition for effective metric indexing. *Pattern Recognition Letters*, 26(9):1363–1376, 2005.

[13] E. Chávez, G. Navarro, R. Baeza-Yates, and J. Marroquín. Proximity searching in metric spaces. *ACM Comp. Surv.*, 33(3):273–321, September 2001.

[14] A. Cobbs. Fast approximate matching using suffix trees. In *Proc. CPM'95*, pages 41–54, 1995. LNCS 937.

[15] R. Cole, L. Gottlieb, and M. Lewenstein. Dictionary matching and indexing with errors and don't cares. In *Proc. STOC'04*, pages 91–100, 2004.

[16] M. Crochemore. Transducers and repetitions. *Theor. Comp. Sci.*, 45:63–86, 1986.

[17] C. Faloutsos and I. Kamel. Beyond uniformity and independence: analysis of R-trees using the concept of fractal dimension. In *Proc. ACM PODS'94*, pages 4–13, 1994.

[18] K. Fredriksson. Metric indexes for approximate string matching in a dictionary. In *Proc. SPIRE'04*, LNCS 3246, pages 212–213, 2004.

[19] V. Gaede and O. Günther. Multidimensional access methods. *ACM Comp. Surv.*, 30(2):170–231, 1998.

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

[21] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[22] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proc. ACM SIGMOD'84*, pages 47–57, 1984.

[23] G. Hjaltason and H. Samet. Index-driven similarity search in metric spaces. *ACM Trans. on Database Systems*, 28(4):517–580, 2003.

[24] P. Jokinen and E. Ukkonen. Two algorithms for approximate string matching in static texts. In *Proc. MFCS'91*, volume 16, pages 240–248, 1991.

[25] I. Kamel and C. Faloutsos. On packing R-trees. In *Proc. ACM CIKM'93*, pages 490–499, 1993.

[26] J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proc. ICALP'03*, LNCS 2719, pages 943–955, 2003.

[27] D. Kim, J. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In *Proc. CPM'03*, LNCS 2676, pages 186–199, 2003.

[28] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. In *Proc. CPM'03*, LNCS 2676, pages 200–210, 2003.

[29] V. Levenshtein. Binary codes capable of correcting spurious insertions and deletions of ones. *Problems of Information Transmission*, 1:8–17, 1965.

[30] M. Maas and J. Nowak. A new method for approximate indexing and dictionary lookup with one error. *Inf. Proc. Lett.*, 2005. To appear.

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

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

[33] E. McCreight. A space-economical suffix tree construction algorithm. *J. of the ACM*, 23(2):262–272, 1976.

[34] D. Morrison. PATRICIA — practical algorithm to retrieve information coded in alphanumeric. *J. of the ACM*, 15(4):514–534, 1968.

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

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

[37] G. Navarro, R. Baeza-Yates, E. Sutinen, and J. Tarhio. Indexing methods for approximate string matching. *IEEE Data Eng. Bull.*, 24(4):19–27, 2001.

[38] G. Navarro, E. Sutinen, and J. Tarhio. Indexing text with approximate q-grams. *Journal of Discrete Algorithms (JDA)*, 3(2–4):157–175, 2005.

[39] B. Pagel, H. Six, H. Toben, and P. Widmayer. Towards an analysis of range queries. In *Proc. ACM PODS'93*, pages 241–221, 1993.

[40] D. Papadias, Y. Theodoridis, and E. Stefanakis. Multidimensional range queries with spatial relations. *Geographical Systems*, 4(4):343–365, 1997.

[41] J. Robinson. The K-D-B-tree: a search structure for large multidimensional dynamic indexes. In *Proc. ACM PODS'81*, pages 10–18, 1981.

[42] R. Sedgewick and P. Flajolet. *Analysis of Algorithms*. Addison-Wesley, 1996.

[43] F. Shi. Fast approximate string matching with q-blocks sequences. In *Proc. WSP'96*, pages 257–271. Carleton University Press, 1996.

[44] E. Sutinen and J. Tarhio. Filtration with q-samples in approximate string matching. In *Proc. CPM'96*, LNCS 1075, pages 50–61, 1996.

[45] W. Szpankowski. Probabilistic analysis of generalized suffix trees. In *Proc. CPM'92*, LNCS 644, pages 1–14, 1992.

[46] Y. Thedoridis and T. Sellis. A model for the prediction of R-tree performance. In *Proc. ACM PODS'96*, pages 161–171, 1996.

[47] E. Ukkonen. Approximate string matching over suffix trees. In *Proc. CPM'93*, pages 228–242, 1993.

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

[49] D. White and R. Jain. Algorithms and strategies for similarity retrieval. Technical Report VCL-96-101, Visual Comp. Lab., Univ. of California, July 1996.