A Metric Index for Approximate String Matching*

Edgar Chávez[1]** and Gonzalo Navarro[2]

[1] Escuela de Ciencias Físico-Matemáticas, Universidad Michoacana.
Edificio "B", Ciudad Universitaria, Morelia, Mich. México 58000.
elchavez@fismat.umich.mx
[2] Depto. de Ciencias de la Computación, Universidad de Chile.
Blanco Encalada 2120, Santiago, Chile.
gnavarro@dcc.uchile.cl

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 R occurrences of a pattern of length m in a text of length n in average time $O(m \log^2 n + m^2 + R)$, using $O(n \log n)$ space and $O(n \log^2 n)$ index construction time. This complexity improves by far over all other previous methods. We also show a simpler scheme needing $O(n)$ space.

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, retrieve all the pattern occurrences, that is, text substrings whose *edit distance* to the pattern is at most r. 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 [25], 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.

There exist some indexing schemes specialized to word-wise searching on natural language text [21,3]. These indexes perform quite well in that case but

* Supported by CYTED VII.19 RIBIDI project (both authors), CONACyT grant 36911 (first author), and Fondecyt grant 1-000929 (second author).

** On leave of absence at Universidad de Chile.

they cannot be extended to handle the general case. Extremely important applications such as DNA, proteins, music or oriental languages fall outside this case.

The indexes that solve the general problem can be divided into three classes. *Backtracking* [17,34,11,15] uses the suffix tree [2], suffix array [20] or DAWG [12] of the text in order to factor out its repetitions. A sequential algorithm on the text is simulated by backtracking on the data structure. These algorithms take time exponential on m or r but in many cases independent of n, the text size. This makes them attractive when searching for very short patterns.

Partitioning [31,30,5] partitions the pattern into 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 work well only when r/m is small.

The third class [24,6] is a hybrid between the other two. The pattern is divided into large pieces that can still contain (less) errors, they are searched for using backtracking, and the potential text occurrences are checked as in the partitioning methods. The hybrid algorithms are more effective because they can find the right point between length of the pieces to search for and error level permitted. Using the appropriate partition of the pattern, these methods achieve on average $O(n^\lambda)$ search time, for some $0 < \lambda < 1$ that depends on r. They tolerate moderate error ratios r/m.

We propose in this paper a brand new approach to the problem. 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 [8,4], but in those cases the particularities of the problem 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, and to find a way to build a metric space out of those sets. The result is an indexing method that, at the cost of requiring on average $O(n \log n)$ space and $O(n \log^2 n)$ construction time, permits finding the R approximate occurrences of the pattern in $O(m \log^2 n + m^2 + R)$ average time. This is a complexity breakthrough over previous work, and it is easier than in other approaches to extend the idea to other distance functions such as reversals. Moreover, it represents an original approach to the problem that opens a vast number of possibilities for improvements. We consider also a simpler version of the index needing $O(n)$ space and that, despite not involving a complexity breakthrough, promises to be better 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 $s_{i...j} = s_i s_{i+1} \ldots s_j$ (which is the empty string if $i > j$) and

$s_{i...} = s_{i...|s|}$. The empty string is denoted as ε. A string x is said to be a *prefix* of xy, a *suffix* of yx and a *substring* of yxz.

2 Metric Spaces

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 [10].

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 $(q, r)_d$.

From the plethora of existing algorithms to index metric spaces, we focus on the so-called *pivot-based* ones, which 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.

An interesting feature of pivot-based algorithms is that they can reduce the number of final distance evaluations by increasing the number of pivots. Define $D_k(x,y) = \max_{1 \le j \le k} |d(x,p_j) - d(y,p_j)|$. Using the pivots $p_1, ..., p_k$ is equivalent to discarding elements u such that $D_k(q,u) > r$. As more pivots are added we need to perform more distance evaluations (exactly k) to compute $D_k(q,*)$ (these are called *internal* evaluations), but on the other hand $D_k(q,*)$ increases its value and hence it has a higher chance of filtering out more elements (those comparisons against elements that cannot be filtered out are called *external*). It follows that there exists an optimum k.

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 *multidimensional range search* methods, which include data structures such as the kd-tree, R-tree, X-tree, etc. [36,14]. Those structures permit indexing a set of objects in k-dimensional space in order to process range queries.

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 [19,25]. 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 $ed(x,y)$. A matrix $C_{0..|x|,0..|y|}$ is filled, where $C_{i,j} = ed(x_{1..i}, y_{1..j})$, so $C_{|x|,|y|} = ed(x,y)$. This is computed as

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

The algorithm takes $O(|x||y|)$ time. The matrix can be filled column-wise or row-wise (there are more sophisticated ways as well). For reasons that will be made clear later, we prefer the row-wise filling. The space required is only $O(|y|)$, 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 [2,1]. 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 substring of T.

To improve space utilization, this trie is compacted into a Patricia tree [23] 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 worst-case $O(n^2)$ of the trie. The suffix tree can be directly built in $O(n)$ time [22,35]. 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. Figure 1 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.

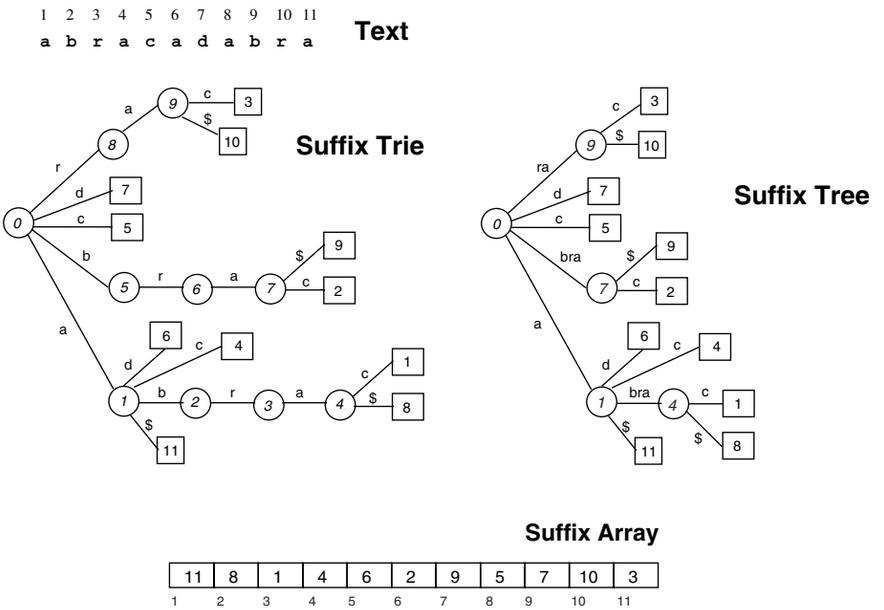

Fig. 1. The suffix trie, suffix tree and suffix array of the text `"abracadabra"`.

The figure shows the *internal* nodes of the trie (numbered 0 to 9 in italics inside circles), which represent text substrings that appear more than once, and the *external* nodes (numbered 1 to 11 inside squares), 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 the prefixes that may have been collapsed. For example the internal node (7) of the suffix tree represents now the compressed nodes (5) and (6), and hence the strings `"b"`, `"br"` and `"bra"`. The external node (1) represents `"abrac"`, but also `"abraca"`, `"abracad"`, etc. until the full suffix `"abracadabra"`.

Finally, the suffix array [20] 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 Figure 1. The suffix array stores one pointer per text position.

The suffix array can be directly built (without building the suffix tree) in $O(n \log n)$ worst case time and $O(n \log \log n)$ average time [20]. 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 internal node (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 spaces 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 representation of all the substrings of a text in $O(n)$ space. So instead of indexing all the text substrings, we index only the (explicit) suffix tree nodes. Therefore, we have $O(n)$ objects to be indexed as a metric space under the edit distance.

Now, 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 and 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 internal node (4) in Figure 1 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 $T = zxy$ and x is a unique text substring (whose prefixes are not unique), then the corresponding suffix tree node is an explicit leaf, which for us represents the set $\{x\} \cup x[y]$. 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.

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 Figure 1. Internal nodes are represented as $i(x)$ and externals as $e(x)$.

Hence, instead of indexing all the $O(n^2)$ text substrings, we index $O(n)$ 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 $\mathbb{U} = \{\varepsilon$, [a], a[bra], [bra], [ra], abra[cadabra], bra[cadabra], ra[cadabra], a[cadabra], [cadabra], a[dabra], [dabra]$\}$.

We have now to decide how to index this metric space formed by $O(n)$ sets of strings. Many options are possible, but we have concentrated on a pivot based approach. We select at random k different text substrings that will be our *pivots*. For reasons that are made clear later, we choose to select pivots of lengths 0, 1, 2, $\cdots$, $k-1$. 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 $\{xy_1...y_j,\ 1 \leq j \leq |y|\}$, all the edit distances can be computed in $O(|p_i||xy|)$ time.

Following our example, let us assume that we have selected $k = 5$ pivots p_0 = `""`, p_1 = `"a"`, p_2 = `"br"`, p_3 = `"cad"` and p_4 = `"raca"`. Figure 2 (left) shows the computation of the edit distances between $i(4)$ = `"a[bra]"` and p_3 = `"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 pessimistically 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 x we have a minimum value seen up to now, v. Then there is no point in considering rows j such that $|x| + j - |p_i| > v$. Hence we work until row $j = v + |p_i| - |x| \leq |p_i|$.

		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. 2. 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.

Figure 2 (right) illustrates this case with $e(1) =$ `"abra[cadabra]"` and the same $p_4 =$ `"cad"`. Note 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. Note also 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. In fact we just assume that the maximum is 11, so the minimum and maximum value for this external node and this pivot are 4 and 11. In particular, since when indexing external nodes $x[y]$ we always have $ed(p_i, x)$ already computed, they can be indexed in $O(|p_i|^2)$ time.

Once this is done for 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:

$$x[y] \rightarrow \langle \quad (\min(ed(x[y], p_0)), \ldots, \min(ed(x[y], p_{k-1}))), \\ (\max(ed(x[y], p_0)), \ldots, \max(ed(x[y], p_{k-1}))) \quad \rangle$$

where we are sure that all the strings in $x[y]$ lie inside the rectangle. In our example, the minima and maxima for $i(4)$ with respect to p_0 to p_4 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 P searched for with at most r errors. This is a range query with radius r in the metric space of the suffix tree nodes. As for pivot based algorithms, we compare the pattern P against the k pivots and obtain a k-dimensional coordinate $(ed(P, p_1), \ldots, ed(P, p_k))$.

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

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

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 $\langle (ed(P, p_1) - r, \ldots, ed(P, p_k) - r), (ed(P, p_1) + r, \ldots, ed(P, p_k) + r) \rangle$.

Figure 3 illustrates. The node contains a set of points and we store its 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.

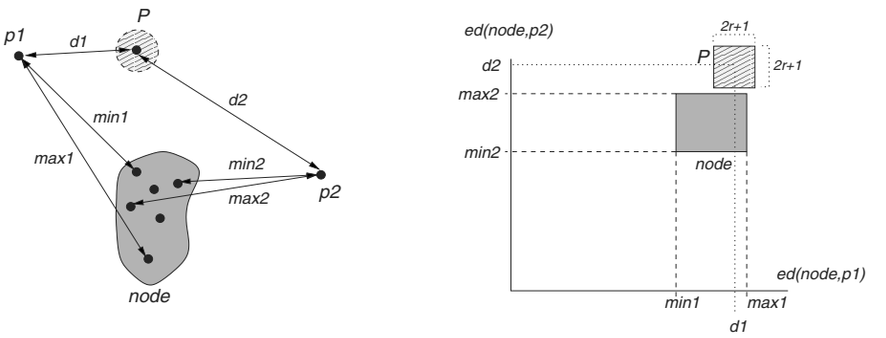

Fig. 3. The elimination rule using two pivots.

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 [36,14]. We could for example use some variant of R-trees [16,7], which would also yield a good data structure to work on secondary memory.

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 `"abr"` with tolerance $r = 1$, then node $i(4)$ qualifies, so we report the text positions in the corresponding tree leaves: 1 and 8.

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 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 [15], 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.

5 Analysis

Let us first consider the construction cost. The maximum length of a repeated text substring, or which is the same, the average height of the suffix trie, is $O(\log n)$ [32,29]. Recall also that our pivots are of length $O(k)$. Hence computing all the kn minima and maxima takes time $O(kn \times k \log n) = O(k^2 n \log n)$, where $O(k \log n)$ is the time to compute each distance. However, we start the computation of the edit distances of a node from the last row of its parent. This reduces the average construction cost to $O(k^2 n)$, since for each pivot we compute one dynamic programming row per suffix trie node, and on average the number of nodes in the suffix trie is $O(n)$ [32,29]. The n leaves are also computed in $O(|p_i|^2 n) = O(k^2 n)$ time.

The total space required by the data structure is $O(kn)$, since we need to store for each explicit node a pointer to the suffix tree and its k coordinates. The suffix tree itself takes $O(n)$ space.

It remains to determine the average search time. A key element of the analysis is a constant α, which is the probability that, for a random hyperrectangle of the set, along some fixed coordinate, the corresponding segment of the query hypercube intersects with the corresponding segment of the hyperrectangle. Another way to put it is that, along that coordinate, the query point falls inside the hyperrectangle projection onto that coordinate after it is enlarged in r units along each dimension. In operational terms, α is the probability that some given pivot (that corresponding to the selected coordinate) does *not* permit discarding a given element. Note that α does not depend on k, only on r.

The first part of the search is the computation of the edit distances between the k pivots and the pattern P of length m. This takes $O(k^2 m)$ time.

The second part is the search for the rectangles that intersect the query rectangle. Many analyses of the performance of R-trees exist in the literature [33,18,26,27,13]. 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 [28]) is $O(n\alpha^k \log n)$.

The third part, finally, is the direct check of the pattern against the suffix tree nodes whose rectangles intersect the query rectangle. Since we discard using any of k random pivots, the probability of not discarding a node is α^k. As there are $O(n)$ suffix tree nodes, we check on average $\alpha^k n$ nodes, with a total cost of $O(\alpha^k n \times m^2)$. 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 $m+r$. This is because the pivot ε removes all shorter or longer candidates.

At the end, we report the R results in $O(R)$ time using a suffix tree traversal. Hence our total average cost is bounded by $k^2 m + n\alpha^k \log n + n\alpha^k m^2 + R$ for $0 \leq \alpha \leq 1$. This is optimized for $k^* = \log_{1/\alpha} n + O(\log\log n) \geq \log_{1/\alpha} n = \Theta(\log n)$.

If we use $\log_{1/\alpha} n$ pivots the search cost becomes $O(m \log^2 n + m^2 + R)$ on average. Note that the influence of the search radius r is embedded in α. This is much better complexity than all previous work, which obtains $O(mn^\lambda)$ time for some $0 < \lambda < 1$. Moreover, much of previous work requires $m = \Omega(\log n)$ to obtain sublinearity, while our approach does not.

The price is in the construction time and space, which become $O(n \log^2 n)$ and $O(n \log n)$, respectively. Especially the latter can be prohibitive and we may have to content ourselves with a smaller k. There seems to be no good tradeoff between space and time, e.g., to obtain $O(n^\lambda)$ time we also need $\Theta(\log n)$ pivots. Most other indexes require $O(n)$ space and construction time.

Finally, it is worth mentioning that, since we automatically discard any internal node not in the length $[m-r, m+r]$ thanks to the pivot $p_0 = \varepsilon$, there is a worst-case limit σ^{m+r} on the number of suffix tree nodes to consider for the last phase. Although this limit is exponential on m and r, it is independent of n. Other indexing schemes based on the suffix tree share the same property.

6 Towards a Practical Implementation

Despite that we have obtained an important reduction in time complexity with respect to n and m, our result is hiding a multiplying factor that depends on the search radius. It is possible that this constant is too large (that is, α too close to 1) and makes the whole approach useless. Also, the extra space requirement (which also increases as α tends to 1) can be unmanageable. In this section we consider an alternative approach that is simpler and likely to obtain better results in practice, despite not involving a complexity breakthrough.

6.1 Indexing Only Suffixes

A simpler index that derives from the same ideas of the paper considers only the n text suffixes and no internal nodes. Each suffix $[T_{j...}]$ represents all the text substrings starting at i, 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 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 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.

6.2 Using an Index for High Dimensions

The space of strings has a distance distribution that is rather concentrated around its mean μ. The same happens to the distances between a pivot p_i and suffixes $[T_{j...}]$ or the pattern P. Since we can only discard suffixes $[T_{j...}]$ 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. Figure 4 illustrates.

The idea is to fix a number s and, for each pivot p_i, store only the s largest $\min(ed(p_i, [T_{j...}]))$ 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 [9]. It also permits

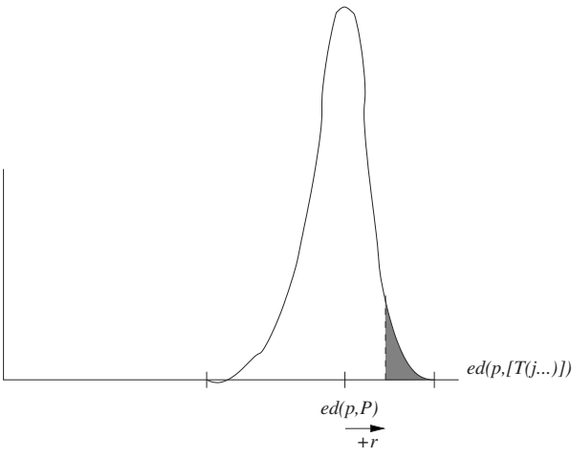

Fig. 4. The distance distribution to a pivot p, including that of pattern P. The grayed area represents the suffixes that can be discarded using p.

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, namely $O(n^2|p_i|/s)$. 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, while the time to traverse the lists is difficult to analyze.

The pivots chosen must not be very short, because their minimum distance to any $[T_{j...}]$ is at most $|p_i|$. In fact, any pivot not longer than $m + r$ is useless.

6.3 Using Specific Strings Properties

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

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. This can also be useful to discard suffixes at verification time: If $d' = ed(P_{1...\ell}, T_{i...i'})$ and we know from the index that $ed(P_{\ell+1...}, T_{i'+1...}) > r - d'$, then a match is not possible.

Other ideas, such as hybrid algorithms that partition the pattern and search for the pieces permitting less errors [6], can be implemented over our metric index instead of over a suffix tree or array. Indeed, our data structure should compete in the area of backtracking algorithms, as the others are orthogonal.

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. We show analytically that the search complexity is better than that obtained in previous work, at the price of slightly higher space usage. More precisely we can search at an average cost of $O(m \log^2 n + m^2 + R)$ time using $O(n \log n)$ space, while by using a suffix tree (the best known technique) one can search in $O(mn^\lambda)$ time for $0 \leq \lambda \leq 1$ using $O(n)$ space. 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 pivots), other multidimensional range search techniques (we have used R-trees), other pivot selection techniques (we took them at random), etc. A more practical 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.

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. Block-addressing indices for approximate text retrieval. In *Proc. ACM CIKM'97*, pages 1–8, 1997.
4. R. Baeza-Yates and G. Navarro. Fast approximate string matching in a dictionary. In *Proc. SPIRE'98*, pages 14–22. IEEE Computer Press, 1998.
5. 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`.
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. Special issue on Matching Patterns.
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. E. Bugnion, T. Roos, F. Shi, P. Widmayer, and F. Widmer. Approximate multiple string matching using spatial indexes. In *Proc. 1st South American Workshop on String Processing (WSP'93)*, pages 43–54, 1993.
9. E. Chávez and G. Navarro. An effective clustering algorithm to index high dimensional metric spaces. In *Proc. SPIRE'2000*, pages 75–86. IEEE CS Press, 2000.
10. E. Chávez, G. Navarro, R. Baeza-Yates, and J. Marroquín. Searching in metric spaces. *ACM Comp. Surv.*, 2001. To appear.

11. A. Cobbs. Fast approximate matching using suffix trees. In *Proc. CPM'95*, pages 41–54, 1995. LNCS 937.
12. M. Crochemore. Transducers and repetitions. *Theor. Comp. Sci.*, 45:63–86, 1986.
13. 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.
14. V. Gaede and O. Günther. Multidimensional access methods. *ACM Comp. Surv.*, 30(2):170–231, 1998.
15. G. Gonnet. A tutorial introduction to Computational Biochemistry using Darwin. Technical report, Informatik E.T.H., Zuerich, Switzerland, 1992.
16. A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proc. ACM SIGMOD'84*, pages 47–57, 1984.
17. P. Jokinen and E. Ukkonen. Two algorithms for approximate string matching in static texts. In *Proc. MFCS'91*, volume 16, pages 240–248, 1991.
18. I. Kamel and C. Faloutsos. On packing R-trees. In *Proc. ACM CIKM'93*, pages 490–499, 1993.
19. V. Levenshtein. Binary codes capable of correcting spurious insertions and deletions of ones. *Problems of Information Transmission*, 1:8–17, 1965.
20. U. Manber and E. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. on Computing*, pages 935–948, 1993.
21. U. Manber and S. Wu. GLIMPSE: A tool to search through entire file systems. In *Proc. USENIX Technical Conference*, pages 23–32, Winter 1994.
22. E. McCreight. A space-economical suffix tree construction algorithm. *J. of the ACM*, 23(2):262–272, 1976.
23. D. Morrison. PATRICIA — practical algorithm to retrieve information coded in alphanumeric. *J. of the ACM*, 15(4):514–534, 1968.
24. E. Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica*, 12(4/5):345–374, Oct/Nov 1994.
25. G. Navarro. A guided tour to approximate string matching. *ACM Comp. Surv.*, 33(1):31–88, 2001.
26. B. Pagel, H. Six, H. Toben, and P. Widmayer. Towards an analysis of range queries. In *Proc. ACM PODS'93*, pages 241–221, 1993.
27. D. Papadias, Y. Theodoridis, and E. Stefanakis. Multidimensional range queries with spatial relations. *Geographical Systems*, 4(4):343–365, 1997.
28. J. Robinson. The K-D-B-tree: a search structure for large multidimensional dynamic indexes. In *Proc. ACM PODS'81*, pages 10–18, 1981.
29. R. Sedgewick and P. Flajolet. *Analysis of Algorithms*. Addison-Wesley, 1996.
30. F. Shi. Fast approximate string matching with q-blocks sequences. In *Proc. WSP'96*, pages 257–271. Carleton University Press, 1996.
31. E. Sutinen and J. Tarhio. Filtration with q-samples in approximate string matching. In *Proc. CPM'96*, LNCS 1075, pages 50–61, 1996.
32. W. Szpankowski. Probabilistic analysis of generalized suffix trees. In *Proc. CPM'92*, LNCS 644, pages 1–14, 1992.
33. Y. Thedoridis and T. Sellis. A model for the prediction of R-tree performance. In *Proc. ACM PODS'96*, pages 161–171, 1996.
34. E. Ukkonen. Approximate string matching over suffix trees. In *Proc. CPM'93*, pages 228–242, 1993.
35. E. Ukkonen. Constructing suffix trees on-line in linear time. *Algorithmica*, 14(3):249–260, 1995.
36. D. White and R. Jain. Algorithms and strategies for similarity retrieval. Technical Report VCL-96-101, Visual Comp. Lab., Univ. of California, July 1996.