String Matching with Metric Trees Using an Approximate Distance

Ilaria Bartolini, Paolo Ciaccia, and Marco Patella

DEIS - CSITE-CNR
University of Bologna, Italy
{ibartolini, pciaccia, mpatella}@deis.unibo.it

Abstract. Searching in a large data set those strings that are more similar, according to the *edit distance*, to a given one is a time-consuming process. In this paper we investigate the performance of metric trees, namely the M-tree, when they are extended using a cheap approximate distance function as a filter to quickly discard irrelevant strings. Using the *bag distance* as an approximation of the edit distance, we show an improvement in performance up to 90% with respect to the basic case. This, along with the fact that our solution is independent on both the distance used in the pre-test and on the underlying metric index, demonstrates that metric indices are a powerful solution, not only for many modern application areas, as multimedia, data mining and pattern recognition, but also for the string matching problem.

1 Introduction

Many modern real world applications, as text retrieval, computational biology, and signal processing, require searching in a huge string database for those sequences that are more similar to a given *query* string. This problem, usually called *approximate string matching*, can be formally defined as follows: Let Σ be a finite alphabet of symbols, let $\mathcal{O} \subseteq \Sigma^*$ be a database of finite length strings over the Σ alphabet, let $q \in \Sigma^*$ be a string, and let d_{edit} be the edit distance function, find the strings in $\mathcal{O}$ which are sufficiently similar to q, up to a given threshold. The edit distance $d_{edit}(X, Y)$ between two strings $X, Y \in \Sigma^*$ counts the minimum number of atomic edit operations (insertions, deletions, and substitutions of one symbol) needed to transform X in Y. As an example, the distance between the strings `"spire"` and `"peer"` is 4, $d_{edit}(\texttt{"spire"}, \texttt{"peer"}) = 4$, because to transform `"spire"` in `"peer"` we need at least 4 edit operations, i.e. 1 substitution (replace `i` with `e`), 2 deletions (`s` and `e`), and 1 insertion (`e`).

In the *online* version of the problem [13], the query string can be preprocessed, but the database cannot, thus the best algorithms are at least linear in the database size. An alternative approach, first proposed in [2], considers the strings to be searched as points in a metric space: A metric index is then built on the strings to reduce the number of distance computations needed to solve a query. Metric indices organize strings using relative distances and are able to consistently improve search performance over the simple sequential scan,

since only a fraction of database strings has to be compared against q. In some cases, however, response times are still unsatisfactory, particularly in the case of long strings, due to the fact that computing $d_{edit}(X, Y)$ using dynamic programming is $O(|X| \cdot |Y|)$, where $|X|$ (resp. $|Y|$) is the length of string X (resp. Y) [19]. To overcome such limitation, we propose the use of an approximate and cheap distance function to quickly discard objects that cannot be relevant for the query. We then show how, using a simple "bag distance", whose complexity is $O(|X| + |Y|)$, performance of metric trees can be improved up to 90%. Moreover, our results can also be applied to other approximate distances, thus making metric trees a powerful solution also for the text application domain, a fact that only in recent times has been considered [2, 14].

The paper is organized as follows: In Section 2 we give background information on solving similarity queries with metric trees. Section 3 presents our approach, and Section 4 contains results obtained from experimentations with real data sets. Finally, in Section 5 we conclude and discuss about possible extensions for our work.

2 Background

Metric trees are access methods that are able to efficiently solve range and k-nearest neighbor queries over general metric spaces. A metric space is a pair $(\mathcal{U}, d)$, where $\mathcal{U}$ is a set of objects and $d : \mathcal{U} \times \mathcal{U} \rightarrow \Re^+$ is a symmetric binary function that also satisfies the triangle inequality: $d(O_i, O_j) \leq d(O_i, O_k) + d(O_k, O_j)$, $\forall O_i, O_j, O_k \in \mathcal{U}$. Given a data set $\mathcal{O} = \{O_1, \ldots, O_N\}$, an object $q \in \mathcal{U}$, and a value $r \in \Re^+$, a range query selects all the objects O in $\mathcal{O}$ whose distance d from q is not higher than r. A k-nearest neighbor query, k being a natural number, retrieves the k objects in $\mathcal{O}$ having the minimum distance d from q. This general view of the problem is shared by many application domains (e.g. image databases [15], computational biology [8], and pattern recognition [3]).

To reduce the number of distance evaluations needed to solve a query, a number of index structures has been recently introduced for the general case of metric spaces [6, 5, 4, 9]. Among them, metric trees, like the M-tree [9] and the Slim-tree [17], are secondary memory structures whose goal is to reduce both CPU and I/O times for solving queries over very large data sets, that cannot fit into main memory.

A characteristic shared by all metric trees is that they partition the data space $\mathcal{U}$ into regions and assign each region to a node n stored on a disk page, whereas data objects are stored into leaf nodes [7]. In detail, at index-construction time, a hierarchical subdivision of the space is performed, such that each region of a node in a sub-tree is enclosed within the region associated to the root of the tree (thus, the root of a sub-tree is a coarser description of its descendant nodes).

Even if metric trees can be used to solve both range and k-nearest neighbors queries, in the following we will concentrate on the former type, since the latter can be built over range queries by using a priority search on the tree [9, 7]. At query time, the index is explored downwards, and distances between the query

object q and objects in internal nodes are computed in order to prune away subtrees that cannot lead to any qualifying object, since their regions are disjoint from the query region. At the leaf level, objects that have not been filtered out, which are called *candidate objects*, are directly compared to the query to check if they should be included in the result (for a candidate object O this happens if $d(q, O) \leq r$). Following the terminology introduced in [7], the cost for determining the candidate objects is named *internal complexity*, while the cost for checking all candidates is referred as *external complexity* (see Figure 1).

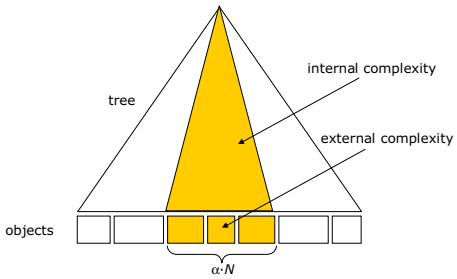


Fig. 1. Internal and external complexity when querying a tree.

Usually, the internal complexity is (much) lower than the external complexity. For a given range query, with query object q and radius r, let $\alpha_q(r) \cdot N$ be the number of candidate objects, N being the cardinality of the data set and $\alpha_q(r) \in [0, 1]$ being the *selectivity factor* of the tree with respect to the query. Although the definition of $\alpha_q(r)$ depends on both q and r, we can eliminate the dependency on q by simply averaging $\alpha_q(r)$ over a sample of query objects. Moreover, let $cost_d$ be the average cost of computing d.[1] The external complexity needed to solve a range query is therefore:

$$\alpha(r) \cdot N \cdot cost_d \tag{1}$$

The goal of metric trees, therefore, is to (consistently) reduce the external complexity, i.e. access only the objects relevant to the query, by (slightly) increasing the internal complexity (for the sequential scan, the internal complexity is 0 and the external complexity is $N \cdot cost_d$).

2.1 The M-tree

The M-tree is a paged, dynamic, and balanced metric tree. Each node n corresponds to a *region* of the indexed metric space $(\mathcal{U}, d)$, defined as $Reg(n) =$

[1] Note that this is a simplification, since, in the general case, the cost of a single distance does depend on the objects for which it is computed, e.g. this is the case for the edit distance.

$\{O \in \mathcal{U} | d(O^{[n]}, O) \leq r^{[n]}\}$, where $O^{[n]}$ is called the *routing object* of node n and $r^{[n]}$ is its *covering radius*. All the objects in the sub-tree rooted at n are then guaranteed to belong to $Reg(n)$, thus their distance from $O^{[n]}$ does not exceed $r^{[n]}$. Both $O^{[n]}$ and $r^{[n]}$ are stored, together with a pointer to node n, $ptr(n)$, in an entry of the parent node of n. In order to save distance computations, the M-tree also stores pre-computed distances between each object and its parent. Thus, if n_p is the parent node of n, the entry for n in node n_p also includes the value of $d(O^{[n_p]}, O^{[n]})$.

When the metric space is the space of strings over an alphabet Σ equipped with d_{edit}, each object corresponds to a string. Figure 2 shows an example of (a part of) an M-tree indexing strings: Strings in each node are connected to the routing string of that node with a line whose length is proportional to the edit distance between them.

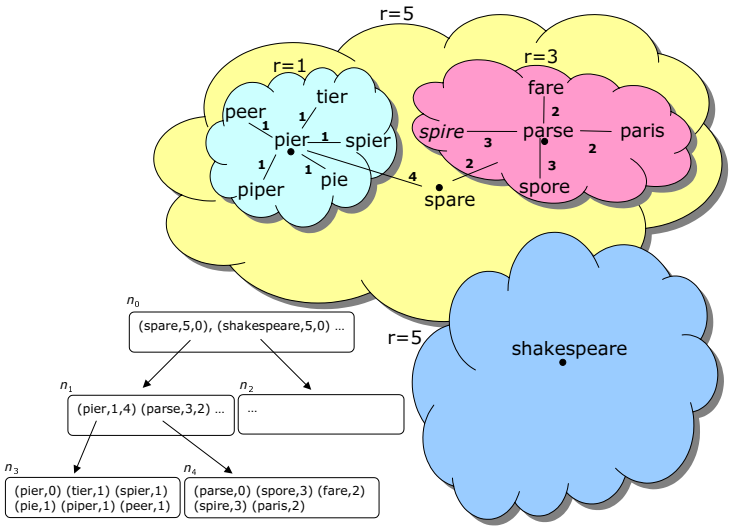


Fig. 2. An example of an M-tree and regions associated to each node. For each entry in an internal node, the routing object, the covering radius, and the distance from the parent object are shown, whereas entries in leaf nodes only contain a data string and its distance to the routing object.

Given a query string q and a search radius r, the M-tree is recursively descended and sub-trees are accessed iff the region associated with their root node overlaps the query region (see Figure 3). For a given node n with routing object $O^{[n]}$ and covering radius $r^{[n]}$, this amounts to check if $d(q, O^{[n]}) \leq r + r^{[n]}$ holds, since from the triangle inequality it follows:

$$d(q, O^{[n]}) > r + r^{[n]} \implies d(q, O) > r \qquad \forall O \in Reg(n) \tag{2}$$

In the example of Figure 3, where q = `"spire"` and $r = 1$, node n_3 can be pruned since it is $d_{edit}(\texttt{"spire"}, \texttt{"pier"}) = 3 > r + r^{[n_3]} = 1 + 1 = 2$. The distance $d(O^{[n_p]}, O^{[n]})$ can be used to prune out nodes without actually computing $d(q, O^{[n]})$, since the triangle inequality guarantees that:

$$|d(q, O^{[n_p]}) - d(O^{[n_p]}, O^{[n]})| > r + r^{[n]} \implies d(q, O) > r \quad \forall O \in Reg(n) \qquad (3)$$

Note that, since the tree is recursively descended, $d(q, O^{[n_p]})$ has been already calculated, thus in Equation 3 no new distance is computed. In the example of Figure 3, we can avoid to compute $d_{edit}(\texttt{"spire"}, \texttt{"pier"})$, since it is $|d_{edit}(\texttt{"spire"}, \texttt{"spare"}) - d_{edit}(\texttt{"spare"}, \texttt{"pier"})| = |1 - 4| = 3 > 1 + 1 = 2$. Thus, the internal complexity for solving the query with the given tree is $3 \cdot cost_{d_{edit}}$, since we have to compare `"spire"` with the routing strings `"spare"`, `"shakespeare"`, and `"parse"`, whereas the external complexity is $4 \cdot cost_{d_{edit}}$, since candidate strings are those contained in node n_4, except for `"parse"` which can again be eliminated using Equation 3. Result strings are `"spire"` and `"spore"`.[2]

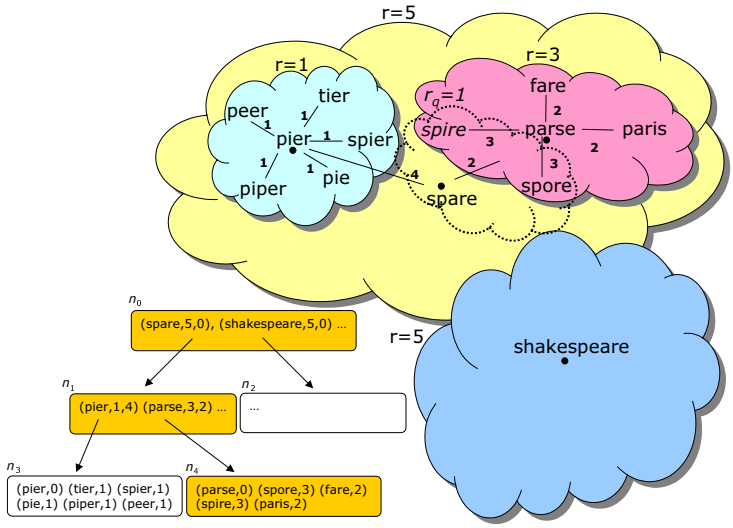


Fig. 3. Solving a range query with an M-tree: Highlighted nodes are those accessed during the search. Nodes n_2 and n_3 are pruned since their regions do not overlap the query region.

[2] Note that routing objects do not necessarily belong to the data set [9], as it is the case for `"spare"` in Figure 2.

3 Proposed Solution

The internal complexity for solving range queries with metric indices is usually very low. On the other hand, when using the edit distance, the external complexity, expressed by Equation 1, can be too high due to $cost_{d_{edit}}$. Thus, to reduce the external complexity, we need a way to decrease the number of strings for which d_{edit} has to be computed. The solution we propose here is to use another distance d_c, which approximates d_{edit}, to quickly filter out candidates. All the strings remaining after this filtering phase are then checked, using d_{edit}, against q to see if they qualify for the result. The test to filter out a candidate string X is:

$$d_c(q, X) > r \tag{4}$$

All the strings satisfying Equation 4 are discarded without further processing. In order to guarantee that no string which should be included in the result is filtered out (no *false dismissals* are allowed), we need a constraint over d_c:

$$d_c(X, Y) \leq d_{edit}(X, Y) \qquad \forall X, Y \in \Sigma^* \tag{5}$$

i.e. d_c should be a lower bound for d_{edit}. If this is not the case, in fact, it could be $d_{edit}(q, X) \leq r$, thus X satisfies the query, yet $d_c(q, X) > r$, thus X is filtered out by d_c.

In order to keep costs low, d_c should be also very cheap to compute. A good candidate for approximating d_{edit} is the "bag distance" (also called "counting filter" in [12]), which is defined as follows.

Definition 1 (Bag Distance). *Given a string X over an alphabet Σ, let $x = ms(X)$ denote the multiset (bag) of symbols in X. For instance, $ms($"peer"$) = \{\{$e, e, p, r$\}\}$. The following can be easily proved to be a metric on multisets:*

$$d_{bag}(x, y) = \max\{|x - y|, |y - x|\}$$

where the difference has a bag semantics (e.g. $\{\{$a, a, a, b$\}\} - \{\{$a, a, b, c, c$\}\} = \{\{$a$\}\}$), and $|\cdot|$ counts the number of elements in a multiset (e.g. $|\{\{$a, a$\}\}| = 2$). In practice, $d_{bag}(x, y)$ first "drops" common elements, then takes the maximum considering the number of "residual" elements. For instance:

$$d_{bag}(\{\{\texttt{a}, \texttt{a}, \texttt{a}, \texttt{b}\}\} - \{\{\texttt{a}, \texttt{a}, \texttt{b}, \texttt{c}, \texttt{c}\}\}) = \max\{|\{\{\texttt{a}\}\}|, |\{\{\texttt{c}, \texttt{c}\}\}|\} = 2$$

It is immediate to observe that $d_{bag}(X, Y) \leq d_{edit}(X, Y)$, $\forall X, Y \in \Sigma^*$.[3] Further, since computing $d_{bag}(X, Y)$ is $O(|X| + |Y|)$, d_{bag} can indeed be effectively used to quickly filter out candidate strings.[4] In the example of Figure 3, the filtering step would exclude the candidate string `"fare"` since it is

[3] For the sake of brevity, here and in the following, we will replace the bag $ms(X)$ of symbols in a string with the string X itself, with a slight abuse of notation.

[4] Other distances can be used to approximate d_{edit} [11]; d_{bag} has, however, the advantage that it does not require further processing of strings and is very fast to compute.

d_{bag}("spire","fare") = 3, whereas the string "paris" cannot be excluded since d_{bag}("spire","paris") = 1. Thus, d_{edit}("spire","paris") = 4 has to be computed to exclude the string from the result. The external complexity now is $4 \cdot cost_{d_{bag}} + 3 \cdot cost_{d_{edit}}$.

3.1 How Much Do we Save?

In order to compute the reduction in the external complexity obtained by using d_{bag}, we need to know the probability that the filter condition of Equation 4 succeeds for a string X. To this end, we use the *distance distribution* of objects, which has been profitably adopted several times to predict performance of metric structures [10, 16, 7, 17]. Formally, for a generic distance d over a domain $\mathcal{U}$, we denote with $F_{\mathbf{d}}(\cdot)$ the distance distribution of d, that is:

$$F_{\mathbf{d}}(x) = \Pr\{d(\mathbf{O_i}, \mathbf{O_j}) \leq x\} = \Pr\{\mathbf{d} \leq x\} \qquad x \in [0, d^+]$$

where $\mathbf{O_i}$ and $\mathbf{O_j}$ are randomly chosen objects of $\mathcal{U}$, d^+ is a finite upper bound on distance values, and $\mathbf{d} = d(\mathbf{O_i}, \mathbf{O_j})$ is a random variable with distribution $F_{\mathbf{d}}(\cdot)$. Figure 4 shows two sample distance distributions. Of course, since $d_{bag}(X, Y) \leq d_{edit}(X, Y)$, $\forall X, Y \in \Sigma^*$, it is also $F_{\mathbf{d_{bag}}}(x) \geq F_{\mathbf{d_{edit}}}(x)$, $\forall x \in [0, d^+]$.

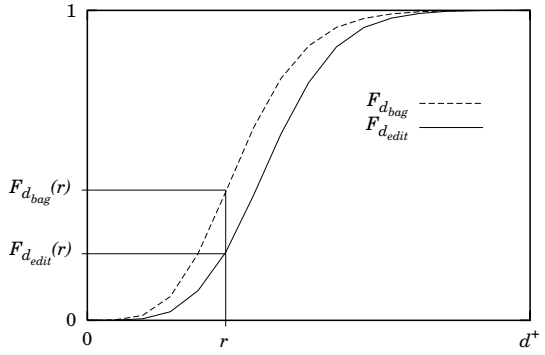


Fig. 4. Distance distributions for d_{bag} and d_{edit}.

The probability that a random string $\mathbf{X}$ is not filtered away by d_{bag} can be, therefore, computed as:

$$\Pr\{d_{bag}(q, \mathbf{X}) \leq r\} = F_{\mathbf{d_{bag}}}(r) \tag{6}$$

We are now ready to compute the overall cost for solving a query. For each candidate string X, $d_{bag}(q, X)$ has to be computed, with a cost of $cost_{d_{bag}}$. Only if the test of Equation 4 fails, which happens with probability given by Equation 6,

we have to compute $d_{edit}(q, X)$, paying $cost_{d_{edit}}$. The overall external complexity is therefore given as:

$$\alpha(r) \cdot N \cdot cost_{d_{bag}} + F_{\mathbf{d_{bag}}}(r) \cdot \alpha(r) \cdot N \cdot cost_{d_{edit}} \tag{7}$$

Comparing Equation 7 with Equation 1, we obtain a saving in search time whenever it is:

$$\alpha(r) \cdot N \left(cost_{d_{bag}} + F_{\mathbf{d_{bag}}}(r) \cdot cost_{d_{edit}}\right) \leq \alpha(r) \cdot N \cdot cost_{d_{edit}}$$

that is when:

$$F_{\mathbf{d_{bag}}}(r) \leq 1 - \frac{cost_{d_{bag}}}{cost_{d_{edit}}} \tag{8}$$

The saving S in search time can be computed as:

$$\begin{aligned} S &\approx \frac{\alpha(r) \cdot N \cdot cost_{d_{edit}} - \left(\alpha(r) \cdot N \cdot cost_{d_{bag}} + F_{\mathbf{d_{bag}}}(r) \cdot \alpha(r) \cdot N \cdot cost_{d_{edit}}\right)}{\alpha(r) \cdot N \cdot cost_{d_{edit}}} = \\ &= 1 - F_{\mathbf{d_{bag}}}(r) - \frac{cost_{d_{bag}}}{cost_{d_{edit}}} \end{aligned} \tag{9}$$

where the approximation is due to the fact that, in Equation 9, the internal complexity is ignored.[5] Finally, if $F_{\mathbf{d_{bag}}}$ is invertible, we can obtain the maximum search radius r_{max} for which it is convenient to use d_{bag}:

$$r_{max} = F^{-1}_{\mathbf{d_{bag}}}\left(1 - \frac{cost_{d_{bag}}}{cost_{d_{edit}}}\right) \tag{10}$$

3.2 Generalizing the Approach

In the previous Section we showed how we can reduce the external complexity for solving a query by filtering out objects using an approximate distance. In line of principle, nothing would prevent to use not only a single distance, but several ones. However, one should consider the fact that each filtering step requires the computation of some distances. As an example, consider the case where two approximate distances, d_{c_1} and d_{c_2}, are present, with $d_{c_1}(X, Y) \leq d_{c_2}(X, Y) \leq d_{edit}(X, Y)$. We can first use d_{c_1} to prune out some candidate strings, and then use d_{c_2} to discard some of the remaining ones. Finally, $d_{edit}(q, X)$ is computed for strings not filtered out by the second step. The external complexity for this solution can be computed as (see Figure 5):

$$\alpha(r) \cdot N \left(cost_{d_{c_1}} + F_{\mathbf{d_{c_1}}}(r) \cdot cost_{d_{c_2}} + F_{\mathbf{d_{c_2}}}(r) \cdot cost_{d_{edit}}\right) \tag{11}$$

Now, one can compare the cost obtained from Equation 11 with that given by Equation 7 (by replacing d_{bag} with either d_{c_1} or d_{c_2}) to choose whether it is convenient to use both distances or only one of them. Of course, it can be expected that $cost_{d_{c_1}} < cost_{d_{c_2}}$, since d_{c_2} is a better approximation of d_{edit}.

[5] We will experimentally validate this assumption in Section 4.

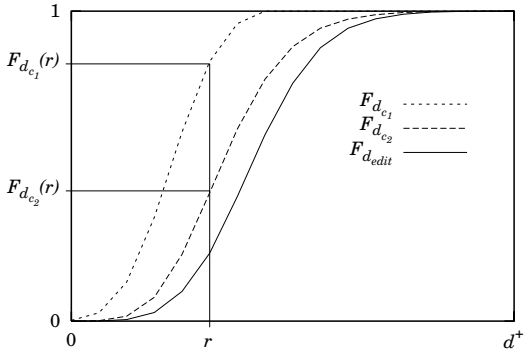


Fig. 5. Distance distributions for d_{c_1}, d_{c_2}, and *edit*.

It is now easy to generalize the above discussion to the case where m distances, $d_{c_1}, \ldots, d_{c_m}$, each lower-bounding d_{edit}, are present. Possible relations existing among the d_{c_i}s allow for a variety of scenarios:

- If it is $d_{c_i}(X,Y) \leq d_{c_{i+1}}(X,Y)$ $(i = 1, \ldots, m-1)$, then each distance can be used to filter out a fraction of candidate objects. Equation 11 can be easily extended to deal with m different distances, obtaining:

$$\alpha(r) \cdot N\left(cost_{d_{c_1}} + F_{\mathbf{d_{c_1}}}(r) \cdot cost_{d_{c_2}} + \ldots + F_{\mathbf{d_{c_m}}}(r) \cdot cost_{d_{edit}}\right) \qquad (12)$$

- A second scenario is obtained whenever it is $d_{c_i}(X,Y) \leq d_{c_{i+1}}(X,Y)$, yet $d_{c_{i+1}}(X,Y)$ can be incrementally obtained from $d_{c_i}(X,Y)$. As an example, this is the case we get considering the bag distance as computed only on the first i symbols of the Σ alphabet: This can be easily obtained starting from the distance computed considering only the first $i-1$ symbols. Of course, in computing $cost_{d_{c_{i+1}}}$, we should not take into account the cost already payed for computing $cost_{d_{c_i}}$.
- The more general case is obtained when it is $d_{c_i}(X,Y) \leq d_{edit}(X,Y)$, yet $d_{c_i}(X,Y) \not\leq d_{c_j}(X,Y), i \neq j$. In this case, the external complexity can still be computed from Equation 12, but now an important optimization issue regards the order in which the m distances should be considered.

It has to be noted that, even if in this paper we only use d_{bag} to approximate the edit distance, *any* function that lower bounds d_{edit} can do the job, since it is not required that d_c is a metric, but only that Equation 5 holds.

4 Experimental Results

In this Section we experimentally evaluate the solution proposed in Section 3. To this purpose, we provide results obtained from the following *real* data sets:

`BibleWords`: This data set consists of all the 12,569 distinct words occurring in the English King James version of the Holy Bible (as provided by [1]).

`BibleLines`: The 74,645 variable-length lines of the Holy Bible.

`BibleLines20`: We took the Holy Bible and segmented it into lines of length 20, which resulted in a total of 161,212 lines.

For each data set, we used a sample of the data (approximately 1% of the whole data set in size) as query objects and built the tree using the remaining strings. In order to predict the time saving when using d_{bag}, in Figure 6 distance distributions for the three data sets are shown, whereas Table 1 presents the average time needed to compute d_{bag} and d_{edit} for the three data sets.

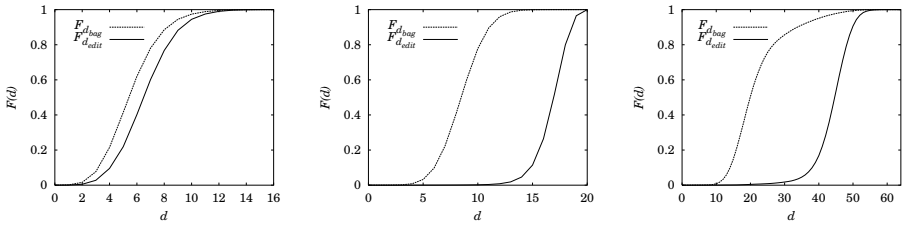
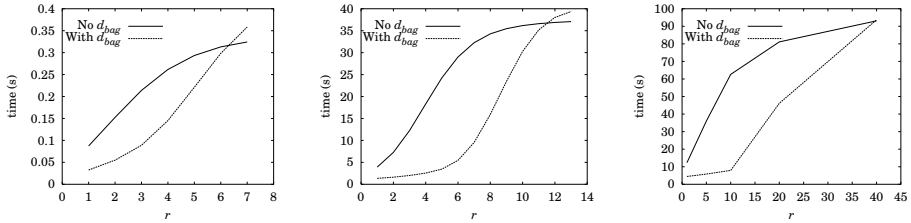


Fig. 6. Distributions of d_{edit} and d_{bag} for the `BibleWords` (a), `BibleLines20` (b), and `BibleLines` (c) data sets.

Data set	$cost_{d_{bag}}$ (s)	$cost_{d_{edit}}$ (s)
`BibleWords`	7.78×10^{-6}	26.1×10^{-6}
`BibleLines20`	17.7×10^{-6}	231.6×10^{-6}
`BibleLines`	66.1×10^{-6}	1300×10^{-6}

Table 1. Distance computation times for data sets used in the experiments.

We ran all the experiments on a Linux PC with a Pentium III 450 MHz processor, 256 MB of main memory, and a 9 GB disk. The node size of the M-tree was set to 8 Kbytes. In Figure 7, we show average search times for solving a range query as a function of the query radius r. The graphs indeed demonstrate that search times rapidly increase for long strings (see Figures 7 (b) and (c)), but also that, for a sufficiently wide interval of query radius values, the use of an approximate distance is very effective in reducing search costs. In particular, for the `BibleLines` data set, when $r = 10$ (0.1% of the data set is retrieved), the average search time when d_{bag} is not used is 64 seconds, which obviously makes this solution unaffordable, whereas the cost can be reduced to less than 8 seconds by using d_{bag}.

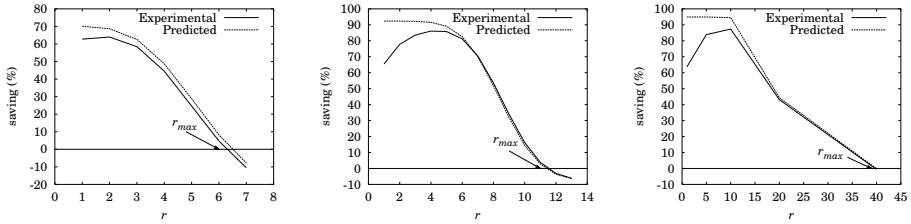


(a) (b) (c)

Fig. 7. Search times when d_{bag} is used or not for the `BibleWords` (a), `BibleLines20` (b), and `BibleLines` (c) data sets.

In Figure 8 we compare the saving in search time obtained through the experiments with the value predicted using Equation 9. As the graphs show, the actual searching performance can be very well estimated. The accuracy of Equation 9 is only slightly reduced for low values of the query radius, i.e. when the internal complexity for searching in the M-tree cannot be neglected wrt the external complexity. In Figure 8 the maximum search radius for which is convenient to use d_{bag}, obtained through Equation 10, is also shown. In particular, using values for distance computation times in Table 1 and the corresponding distance distributions (Figure 6), we obtain:

- $r_{max}(\texttt{BibleWords}) = F^{-1}_{\mathbf{d_{bag}}}(1 - 7.78/26.1) \approx F^{-1}_{\mathbf{d_{bag}}}(0.703) \approx 6$,
- $r_{max}(\texttt{BibleLines20}) = F^{-1}_{\mathbf{d_{bag}}}(1 - 17.7/232) \approx F^{-1}_{\mathbf{d_{bag}}}(0.924) \approx 11$, and
- $r_{max}(\texttt{BibleLines}) = F^{-1}_{\mathbf{d_{bag}}}(1 - 66.1/1300) \approx F^{-1}_{\mathbf{d_{bag}}}(0.949) \approx 39$.



(a) (b) (c)

Fig. 8. Real and predicted time saving for the `BibleWords` (a), `BibleLines20` (b), and `BibleLines` (c) data sets. Also shown are the r_{max} values which limit the usefulness of using d_{bag}.

The graphs indeed demonstrate the accuracy of Equation 10 that can, therefore, be used to reliably predict whether, for a given query, it is worth using an approximate distance or not. The values of r_{max} are high enough to allow efficient search performance over a wide interval of values of the query radius.

It is also worth noting that even if, for the `BibleLines20` and the `BibleLines` data sets, d_{edit} is not very well approximated by d_{bag} (see Figure 6), we can still obtain significant performance improvements (see Figure 8). This is due to the fact that, in such cases, d_{bag} is very cheap to compute compared to d_{edit}, because of the increased strings length (see Table 1), and this indeed allows to reduce search times, as also demonstrated by Equation 9.

5 Conclusions

In this paper we considered the problem of searching in a string database for those strings which are similar, according to the *edit distance*, to a given one. Metric trees, as the M-tree [9], can be used to efficiently reduce the time needed to solve such queries. However, metric trees do not always succeed in consistently reducing the number of strings that have to be directly compared to the query, a fact that can indeed abate search performance, particularly when long strings are used. To solve this problem we proposed the use of the approximate and cheap *bag distance*, to quickly get rid of irrelevant strings. We also used the distribution of distances to predict the effectiveness of this pre-filtering step, showing how the saving in search times can be analytically computed, thus one can decide in advance if it is worth to use an approximate distance. Experimental results with large real data sets demonstrate that our approach achieves savings up to 90% with respect to the case when the approximate distance is not used, and that analytical predictions are very accurate.

Even if in this paper we concentrated on the M-tree, our solution and results apply to any other metric index, e.g. the recent Slim-tree [17] or the main memory structures considered in [2]. Moreover, we would also point out that our approach allows different functions to be used for approximating the edit distance, since we only require the former to lower bound the latter. Together, these two facts indeed broaden the applicability of our approach.

An issue not covered in this work is that of *local string alignment*, where the longest common substrings between a query and a database string are searched [11]. Since, however, such problem can be reduced to that of string matching, we believe that the use of a cheap function to approximate the edit distance can be effective also in this case. We leave this research topic as future work.

Finally, we would like to point out that in this paper we did not consider other techniques that are commonly used to reduce the complexity of the online problem, e.g. cutoff heuristics that can reduce the complexity of computing the edit distance to $O(r \cdot |X|)$ [18], where r is the query radius and $|X|$ is the length of a string in the data set. We plan to investigate in the future how such techniques can be embedded in our approach to further improve its efficiency. Moreover, we also plan to compare our approach with other state-of-the-art main memory solutions [13].

References

1. Project Gutenberg official home site. `http://www.gutenberg.net/`.

2. R. Baeza-Yates and G. Navarro. Fast approximate string matching in a dictionary. In *Proceedings of the 5th String Processing and Information Retrieval Symposium (SPIRE'98)*, Santa Cruz, Bolivia, Sept. 1998.
3. S. Berretti, A. Del Bimbo, and P. Pala. Retrieval by shape similarity with perceptual distance and effective indexing. *IEEE Transaction on Multimedia*, 2(4):225–239, Dec. 2000.
4. T. Bozkaya and M. Özsoyoglu. Indexing large metric spaces for similarity search queries. *ACM Transactions on Database Systems*, 24(3):361–404, Sept. 1999.
5. S. Brin. Near neighbor search in large metric spaces. In *Proceedings of the 21st International Conference on Very Large Data Bases (VLDB'95)*, pages 574–584, Zurich, Switzerland, Sept. 1995.
6. W. A. Burkhard and R. M. Keller. Some approaches to best-match file searching. *Communications of the ACM*, 16(4):230–236, Apr. 1973.
7. E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquín. Proximity searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, Sept. 2001.
8. W. Chen and K. Aberer. Efficient querying on genomic databases by using metric space indexing techniques. In *1st International Workshop on Query Processing and Multimedia Issues in Distributed Systems (QPMIDS'97)*, Toulouse, France, Sept. 1997.
9. P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB'97)*, pages 426–435, Athens, Greece, Aug. 1997.
10. P. Ciaccia, M. Patella, and P. Zezula. A cost model for similarity queries in metric spaces. In *Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'98)*, pages 59–68, Seattle, WA, June 1998.
11. T. Kahveci and A. K. Singh. An efficient index structure for string databases. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB 2001)*, pages 351–360, Rome, Italy, Sept. 2001.
12. G. Navarro. Multiple approximate string matching by counting. In *Proceedings of the 4th South American Workshop on String Processing (WSP'97)*, pages 125–139, Valparaiso, Chile, Nov. 1997.
13. G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, Mar. 2001.
14. G. Navarro, R. Baeza-Yates, E. Sutinen, and J. Tarhio. Indexing methods for approximate string matching. *IEEE Data Engineering Bulletin*, 24(4):19–27, Dec. 2001. Special Issue on Text and Databases.
15. S. Santini. *Exploratory Image Databases: Content-Based Retrieval*. Series in Communications, Networking, and Multimedia. Academic Press, 2001.
16. C. Traina Jr., A. J. M. Traina, and C. Faloutsos. Distance exponent: A new concept for selectivity estimation in metric trees. In *Proceedings of the 16th International Conference on Data Engineering (ICDE 2000)*, page 195, San Diego, CA, Mar. 2000.
17. C. Traina Jr., A. J. M. Traina, C. Faloutsos, and B. Seeger. Fast indexing and visualization of metric data sets using Slim-trees. *IEEE Transactions on Knowledge and Data Engineering*, 14(2):244–260, Mar. 2002.
18. E. Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms*, 6(1):132–137, Mar. 1985.
19. R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, Jan. 1974.