An empirical evaluation of a metric index for approximate string matching

Bilegsaikhan Naidan and Magnus Lie Hetland

Abstract

In this paper, we evaluate a metric index for the approximate string matching problem based on suffix trees, proposed by Gonzalo Navarro and Edgar Chávez [9]. Suffix trees are used during the index construction to generate intermediate data (pivot table) that to be indexed and the query processing. One of the main problems with suffix trees is their space requirements. To address this, we proposed as an alternative a linear-time algorithm that simulates suffix trees in the suffix arrays. The proposed algorithm is more space-efficient and is more suited for disk-based implementation. Even so, experimental results on two real-world data sets show that the metric index is beaten by straightforward, slightly enhanced linear scan.

1 Introduction

Approximate string matching is crucial for many modern applications, such as in computational biology. The main goal of this problem is to find all the occurences of a given query string in a large string permitting a given amount of error in the matching.

Nowadays, the metric space model is becoming a favored approach for solving many approximate or distance-based search problems. Given a metric d over a universe $\mathbb{U}$, and a data set $\mathbb{D} \subset \mathbb{U}$, the task is to quickly retrieve the objects in $\mathbb{D}$ that are within a given search radius (or the k nearest neighbors) of some query $q \in \mathbb{U}$ according to the metric d. For strings, a metric d can be the edit distance (or Levenstein distance), which gives the minimum number of insertions, deletions, and replacements required to transform one string into another. The edit distance between strings x and y can be computed in $\mathcal{O}(|x| \cdot |y|)$ time and $\mathcal{O}(\min(|x|, |y|))$ space. Several approaches [8] have been proposed to improve the efficiency of approximate string matching. However, most of these approaches are only focused on short string matching. In DNA sequences, it is necessary to align long queries in a large sequence. For more details on various approaches to DNA sequence alignment, see the recent survey by Li and Homer [7] and the comparative analysis by Ruffalo et al. [10].

Gonzalo Navarro et al. [9] proposed a metric indexing structure for approximate string matching based on suffix trees. According to Kurtz [6], improved implemen-

This paper was presented at the NIK-2012 conference; see http://www.nik.no/.

tations of linear-time suffix tree construction algorithm require 20 times more space than the input string in the worst case. For instance, the human genome is about 3 GiB of symbols and its suffix tree requires about 60 GiB space. Thus it may not fit in the main memory of many systems. Also, disk-based implementation of suffix trees is not straightforward. A more space-efficient data structure is the *suffix array*. To summarize, the main contributions of this paper are outlined as follows:

- We propose a more efficient algorithm that simulates the index construction process, using suffix arrays.
- We have conducted experiments evaluating the metric index structure, providing empirical results. (The original authors provided only theoretical results.)

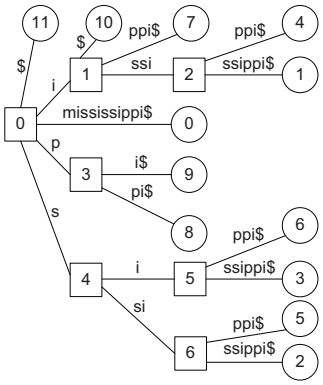
The remainder of this paper is organized as follows. In Section 2, some notation and preliminary definitions are given. Section 3 presents the original version of the metric index while Section 4 describes our algorithm. Experimental results are provided in Section 5 and finally Section 6 concludes the paper.

2 Preliminaries

In this section we introduce some notation and definitions that are used in the rest of the paper.

Let Σ be a finite ordered alphabet and let S be a string over Σ. We assume that S ends with a special end symbol \$ that is not included in Σ and lexicographically ordered before any symbol in Σ. The length of S is denoted by $n = |S|$. A substring $S[i \dots j]$ of S starts at position i and ends at position j $(0 \leq i \leq j < n)$. For simplicity, we let $S[i]$ denote $S[i \dots n-1]$ $(0 \leq i < n)$.

The suffix tree $\mathcal{T}$ for S is a tree that compactly represents all the suffixes of S and has exactly n leaves. The leaves contain unique integers in the range $[0, n-1]$ that indicate the starting positions of the suffixes. Each edge of the tree is labeled with a substring of S, so that the concatenated edge labels on a path from root to leaf form the suffix represented by the given leaf node. An example of a suffix tree for string "mississippi\$" is shown in Figure 1a. For more detailed explanation see book by Gusfield [3].



(a)

i	$sa[i]$	$lcp[i]$	$S[sa[i]]$
0	11	0	\$
1	10	0	i\$
2	7	1	ippi\$
3	4	1	issippi\$
4	1	4	ississippi\$
5	0	0	mississippi\$
6	9	0	pi\$
7	8	1	ppi\$
8	6	0	sippi\$
9	3	2	sissippi\$
10	5	1	ssippi\$
11	2	3	ssissippi\$

(b)

Figure 1: Illustrations of a suffix tree for string "mississippi\$" (a) a suffix array for the same string and its lcp array (b).

The suffix array *sa* is an integer array which contains the starting positions of lexicographically ordered suffixes of S. The longest common prefix array *lcp* is an

array of integers in which $lcp[0]$ is initialized to 0 and $lcp[i]$ indicates the length of the common prefix of $S[sa[i]]$ and $S[sa[i-1]]$, for $1 \leq i < n$. Figure 1b shows an example of a suffix and a lcp array for string "mississippi$". The average value of lcp will be $\mathcal{O}(\log n)$ [1].

We use the notation $x[y]$ from the original work by Navarro et al. [9], which represents the group of substrings $\{xy_1, xy_1y_2, ..., xy\}$ on an explicit node of suffix tree, where the current node's parent corresponds to the string x and the label of current node corresponds to the string y. For instance, the explicit internal node $i(2)$ in Figure 1a represents the strings "i[ssi]" = {"is", "iss", "issi"} and the external node $e(8)$ represents the strings "p[pi]" = {"pp", "ppi"}. Let $\mathcal{I}$ be the set of all the strings of explicit internal nodes of $\mathcal{T}$. For the string "mississippi$", $\mathcal{I}$ is {"[i]", "i[ssi]", "[p]", "[s]", "s[i]", "s[si]"}. Let $\mathcal{E}$ be a set of all the strings of external nodes of $\mathcal{T}$. Let $\mathcal{E}^*$ be $\mathcal{E} \setminus \mathcal{I}$. For the string "mississippi$", the substring "[i]" of $i(1)$ is already in $\mathcal{I}$, and thus "i" of $e(10)$ is not included in $\mathcal{E}^*$. Therefore, $\mathcal{E}^*$ is {"$", "i[ppi]", "issi[ppi]", "issi[ssippi]", "[mississippi]", "p[i]", "p[pi]", "si[ppi]", "si[ssippi]", "ssi[ppi]", "ssi[ssippi]"}.

3 The metric index

As can be seen in Section 1, some distance functions (e.g., the edit distance) are computationally expensive and to answer similarity queries by performing a linear scan on large data sets is impractical. Thus we usually build an index structure over the data set to reduce the overall query processing costs by exploiting the triangle inequality.

Most metric indexing methods are based on a filter-refine principle. One important example is so-called *pivot*-based methods, where a set of reference objects (the pivots) are selected from the dataset, and the distances to these form coordinates in a *pivot space*. In other words, the objects of the original space are embedded into the pivot space by computing the distances between pivots and those objects. Some of these distances are stored to reduce the number of distance computations during query processing. As the number of pivot increases, query processing in the pivot space becomes expensive. We note that query processing in the transformed space should be cheaper than the original one, otherwise, this entire effort is useless. In the filtration step, objects that can not qualify as relevant to a query are filtered out by establishing lower bounds of the real distances; this is done by using the pre-computed distances together with the query-pivot distances and the triangle inequality. If the lower bound is greater than the query radius the object can safely be filtered out. In the refinement step, the real distances between the query and the candidate objects obtained from the previous step are computed, and the objects that qualify as relevant reported.

A naïve approach of metric indexing for S is to build an object-pivot distance table over all the $\mathcal{O}(n^2)$ substrings of S, which is unacceptable for large strings. Thus, Navarro et al. [9] proposed an indexing algorithm that uses suffix trees during index construction and query processing. With suffix trees, we collapse all the $\mathcal{O}(n^2)$ substrings of S into the $\mathcal{O}(n)$ strings of explicit suffix tree nodes (i.e., $\{\mathcal{I} \cup \mathcal{E}^*\}$) as well as speeding up the computation of the edit distance between pivots and $\{\mathcal{I} \cup \mathcal{E}^*\}$ by traversing the tree in a depth-first manner. The general schema of the metric index is given in Figure 2.

We introduce the term *last split point* of $x[y]$ to refer to the starting position of

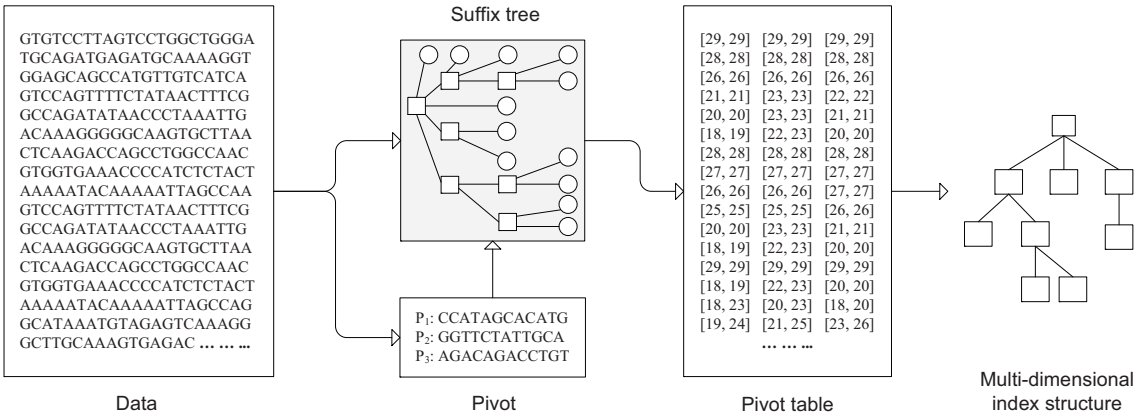


Figure 2: General schema of the metric index.

y in $x[y]$ (i.e., $|x|$). We now briefly describe the metric index.

First, a suffix tree $\mathcal{T}$ is built over S and a set of D pivots $\{P_1, \ldots, P_D\}$ selected from S. The pivot table is filled by columns as follows. For each pivot P_i, the algorithm traverses the suffix tree $\mathcal{T}$ in a depth first manner by filling the edit distance matrix row-wise and computing the *minimum* (*mind*) and *maximum* (*maxd*) distances between P_i and all the strings of $\{\mathcal{I} \cup \mathcal{E}^*\}$. We note that *mind* and *maxd* distances for a string of $\{\mathcal{I} \cup \mathcal{E}^*\}$ are computed from the last split point of that string not from its beginning. For instance, the last split point of string "ssi[ssippi]" is 3 therefore we start considering *mind* and *maxd* from the position 3. Let N be an internal node of $\mathcal{T}$ and x be a substring of S which is obtained by visiting the nodes from the root to N by combining the labels of that path. Let us assume that the traversal algorithm has reached N. Then the overall costs for the computations of the edit distance between P_i and any child node of N can be reduced. Since they share same prefix (i.e., x) and the distance matrix values already filled up to N. Thus those distance values can be used for the distance computation for any child node of N.

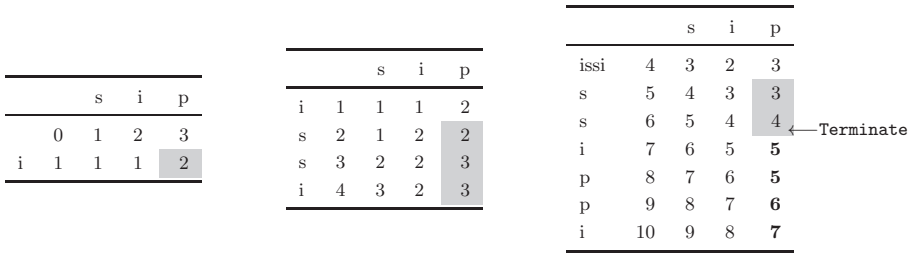
For each string of $\mathcal{I}$, we compute *mind* and *maxd* while for each string $x[y]$ of $\mathcal{E}$, we compute only *mind* and set *maxd* as $max(|x[y]|, |P_i|)$ because y can be very long. Let us assume that we have processed the string $xy_1 \ldots y_j$ of xy by using a matrix $ed_{0\ldots|xy_1\ldots y_j|,0\ldots|P_i|}$ and let v_j be $min_{1 \leq j' \leq j} ed_{|x|+j',P_i}$. The distance computation for *mind* is early terminated at row j'' $(j < j'' < n)$ if the following condition holds:

$$|x| + j'' - |P_i| > v_j \tag{1}$$

This is because $d(xy_1 \ldots y_{j''}, P_i) \geq |x| + j'' - |P_i| > v_j$. Let us consider an example of computing *mind* and *maxd* between the pivot "sip" and strings "[i]", "i[ssi]" and "issi[ssippi]". The example is illustrated in Figure 3. For "issi[ssippi]", we stop the process after the sixth row due to the early termination condition (1). Thus *mind* and *maxd* between the pivot and strings "[i]", "i[ssi]", "issi[ssippi]" are $[2, 2]$, $[2, 3]$ and $[3, 10]$, respectively.

After the traversal of each pivot in P, we have a D-dimensional hyperrectangle with coordinates $[[mind_{x[y],P_1}, maxd_{x[y],P_1}], \ldots, [mind_{x[y],P_D}, maxd_{x[y],P_D}]]$ for each string $x[y]$ in $\{\mathcal{I} \cup \mathcal{E}^*\}$. Once we obtain the pivot table we can use any multidimensional index structure such as R-trees [4] to index it.

A query q with a radius r is resolved as follows. First, we compute



(a) (b) (c)

Figure 3: Examples of computations of the edit distance between pivot "sip" and strings "[i]" (a), "i[ssi]" (b) and "issi[ssippi]" (c). The values for *mind* and *maxd* are taken from the numbers in gray area. In the last figure, the last four numbers in bold listed after the termination point are not neccessary to be computed due to the condition 1.

distances between P and q to obtain a D-dimensional vector with coordinate $[d(q, P_1), \ldots, d(q, P_D)]$. With this vector, we define a query hyperrectangle with coordinates $[[l_1, h_1], \ldots, [l_D, h_D]]$, where $l_i = d(q, P_i) - r$ and $h_i = d(q, P_i) + r$ $(1 \leq i \leq D)$. The filtration of a string $x[y]$ can be done by using any pivot P_i $(1 \leq i \leq D)$ if at least one of $l_i > maxd_{x[y],p_i}$ or $h_i < mind_{x[y],p_i}$ holds, because $d(q, xy') > r$ then holds for any $xy' \in x[y]$. Informally, a candidate set consists of all the objects of $\{\mathcal{I} \cup \mathcal{E}^*\}$ whose hyperrectangles intersect the query hyperrectangle. In the refinement step, the candidate objects obtained from the previous step are checked against q with the suffix tree. For each candidate object $x[y]$, we may compute the edit distance between q and every prefix of x several times. In order to avoid this redundant work, we first mark every node in the path that represents every candidate object. After that, we traverse the suffix tree again, computing the edit distance between q and strings of those marked nodes of the suffix tree in the same way that we used in the pivot table construction. If $x[y]$ is part of the result set for q (i.e., $d(q, x[y]) \leq r$), we report all the starting points in the leaves of the subtree rooted by the node that represents $x[y]$.

4 Our algorithm

In the construction phase of the metric index, the replacement of the suffix tree with a suffix array leads to a new problem: finding all the substrings of explicit internal nodes of the suffix tree in the suffix array. Because suffix arrays do not have any hierarchical information that suffix trees have, these substrings are not readily available. However, by using some auxiliary information, we can deal with this problem without increasing the asymptotic running time. Our algorithm is based on the following two simple observations. First, all the paths through a node will be adjacent in the suffix array. Thus all split points (i.e., all nodes of $\mathcal{I}$) can be detected by examining all the adjacent pairs of the suffix array by analyzing their lcp values. For instance, the string "i[ssi]" of $i(2)$ is detected with the strings "issi[ppi]" of $e(3)$ and "issi[ssippi]" of $e(4)$. The corresponding lcp value of $e(4)$ in the lcp array is 4, therefore we selected the prefix "issi" of $e(3)$. Second, even with suffix trees we still need $\mathcal{O}(max(lcp) \cdot max(|P_i|))$ additional space in the worst case for a matrix that is used during computations of the edit distance between $\{\mathcal{I} \cup \mathcal{E}^*\}$ and P, because the distance values computed earlier are required in backtracking and

visiting other nodes of the suffix tree. Thus the *mind* and *maxd* of explicit internal node can be obtained from the matrix when the traversal algorithm is visiting a node or backtracking.

Algorithm 1 outlines the simplest version of our algorithm that generates all the equivalent strings of $\{\mathcal{I} \cup \mathcal{E}^*\}$ of $\mathcal{T}$ for S. There are at most $n-1$ explicit internal nodes in $\mathcal{T}$. Thus, the algorithm generates at most $\mathcal{O}(2n)$ non-empty substrings of S. In the pseudocode, we use the standard stack operations such as *push*() (which adds a new element to the top of the stack), *pop*() (which removes an element from the top of the stack) and *top*() (which returns an element at the top of the stack). First, a stack *seen* is initialized in line 1. For any iteration i of the for-loop, any element e of *seen* represents that we have seen the internal node represented by the prefix of length e of $S[sa[i]]$. In line 6, the condition $seen.top() > next$ means that the algorithm needs backtracking. Thus, we need to pop out those positions greater than *next* from *seen*, because we check a different prefix with same length in the next time. However, the top element of *seen* that equals to *next* is not popped out, because that information is used in line 7 to check whether a node has been reported before. Also, the condition $next \neq 0$ ensures that we are not back to the root. The strings for $\mathcal{I}$ are reported in line 9. In line 10, each element of the suffix array is reported as an external node of $\mathcal{E}^*$, unless it has already been reported as explicit internal. Those nodes would be adjacent, so we only need to check the lengths of their strings.

Algorithm 1: Generate all the strings of $\{\mathcal{I} \cup \mathcal{E}^*\}$

```
1: initialize a stack seen
2: for i = 1 to n do
3:     cur ← lcp[i]; next ← 0
4:     if i + 1 ≤ n then
5:         next ← lcp[i + 1]
6:         while seen ≠ ∅ and seen.top() > next do seen.pop()
7:         if next ≠ 0 and (seen = ∅ or seen.top() < next) then
8:             seen.push(next)
9:             report S[sa[i], next]
10:    if next ≠ n − sa[i] then
11:        report S[sa[i]]
```

Algorithm 1 has a running time complexity of $\mathcal{O}(n)$. (The running time of the algorithm is equal to the number of times *seen.pop*() in line 6 is executed, and in line 8, *seen.push*() is executed at most $n-1$ times.)

We need two auxiliary arrays for filling up the pivot table columns with *mind* and *maxd*. The last split point of any string of $\mathcal{E}^*$ is defined by the maximum values of the current and next lcp while the last split point (*sp*) of any string of $\mathcal{I}$ is defined as follows. We can not direclty assign the lcp values to *sp* while detecting the nodes, because we may not obtain explicit internal nodes in depth-first traversal order of suffix tree nodes (for example, in our example $i(4)$ is reported after $i(5)$). Thus we maintain a stack that stores pairs of *sp* indexes and lcp values. The values of *sp* are assigned when we backtrack and select as the maximum value of the top element's lcp and the next lcp. The suffix array index *si* for an explicit internal node is an array of integers in which an element $si[i]$ indicates a smallest index of the suffix array where the string of the current explicit internal node is identical to the prefix

of $S[sa[i]]$ $(0 \leq i \leq n)$. This array is used during query processing. Now we modify algorithm 1 to generate the arrays of *sp* (Algorithm 2) and *si* (Algorithm 3) and to construct the pivot table (Algorithm 4). In Algorithm 4, the variable *row* points the last row in the matrix *ed* that has processed. We report *mind* and *maxd* and the suffix array indexes for each string in $\{\mathcal{I} \cup \mathcal{E}^*\}$.[1]

Sample runs of Algorithm 1, 2, 3 for string "mississippi$" are illustrated in Figure 4. Algorithm 1 produces the strings of $\{\mathcal{I} \cup \mathcal{E}^*\}$ exactly in the same order that is shown the figure.

i	$sa[i]$	$lcp[i]$	$\mathcal{E}^*$
0	11	0	
1	10	0	
2	7	1	i[ppi]
3	4	1	issi[ppi]
4	1	4	issi[ssippi]
5	0	0	[mississippi]
6	9	0	p[i]
7	8	1	p[pi]
8	6	0	si[ppi]
9	3	2	si[ssippi]
10	5	1	ssi[ppi]
11	2	3	ssi[ssippi]

(a)

internal node	$\mathcal{I}$	$sp[i]$	$si[i]$
$i(1)$	[i]	0	1
$i(2)$	i[ssi]	1	3
$i(3)$	[p]	0	6
$i(5)$	s[i]	1	8
$i(4)$	[s]	0	8
$i(6)$	s[si]	1	10

(b)

Figure 4: Example runs of the algorithms for string "mississippi$". The strings of $\mathcal{E}^*$ (a) and the strings of $\mathcal{I}$ and the split points and the suffix array indexes for the strings of $\mathcal{I}$ (b).

The main principle of query processing for our approach is almost the same as in the original version. The only difference is that we mark the indexes of the suffix array instead of suffix tree nodes. Let us assume that we have processed a row k of the matrix for a candidate object. If the candidate is the result of a query at this time, the candidate is reported and we directly report those objects after the candidate object in the suffix array in contiguous order such that their lcp values are greater than or equal to k.

5 Experiments

In this section we provide experimental results. We are particularly interested in answering the following questions:

- How expensive is our simulation algorithm in terms of running time and memory requirement?
- What is the effect of varying the number of pivots?
- How does the performance of the metric index with *mind* and *maxd* compare to the one with only *mind*?
- How does the metric index work in practice?
- What are the main problem with the metric index, if any?

[1] In the pseudocode, the expression ($cond$? $expr_1$: $expr_2$) evaluates to $expr_1$ if $cond$ is true, and to $expr_2$, otherwise. Also, we let a_{-i} denote an element at position $|a| - 1 - i$ of the array a $(0 \leq i < |a|)$.

Experiment Settings

First, we implemented our algorithm with full of indexing and query processing. For the original suffix tree based version, we implemented only the pivot table generation part (i.e., not query processing). We used two real-world datasets, namely DNA and protein datasets, which were obtained from the Pizza & Chilli corpus [2] and used a 10 MiB prefix of the datasets. The total number of objects (we recall that the size of $\{\mathcal{I} \cup \mathcal{E}^*\}$ is less than $2n$) to be indexed was 17 508 956 for DNA while for the protein data set the number was 17 341 406. The length of pivot and query was 35. Our query set consisted of 1000 queries, which were selected randomly from the datasets. For each query, we intentionally introduced 0–3 errors at random. We performed range searches with query radii varying from 0 to 3. The results were averaged over 10 runs. We did not use any compression algorithm during the construction of the index. For indexing the hyperrectangles, we used the R-tree implementation from the spatial index library [5][2]. All the programs were compiled by gcc 4.6.2 with the -O3 option and were run on a PC with a 3.3 GHz Intel Core i5-2500 processor and 8 GiB RAM. After testing several pivot selection algorithms, where none was clearly better than the others, we decided to use pivots that were randomly selected from the datasets.

Filtering effect of *maxd* and of number of pivot

Let MetricD be the metric index with only *mind* and MetricDD, the metric index with *mind* and *maxd*. The numbers of pivots used were 10 (p=10) and 20 (p=20). Figure 5 shows that the filtering effect of varying the number of pivot and MetricD versus MetricDD on various datasets.

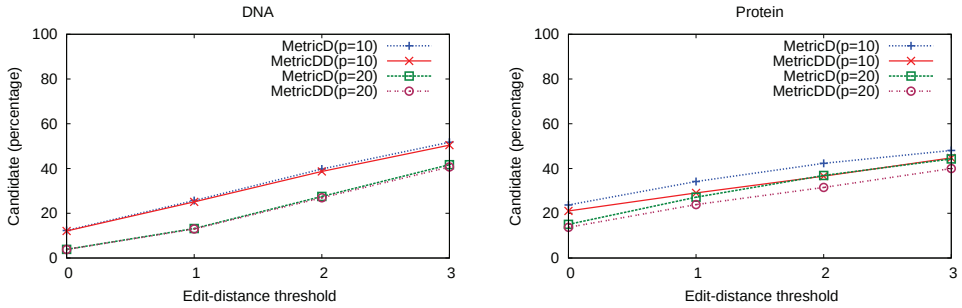


Figure 5: The percentage of candidates left after pivot filtration.

The experiments show that MetricDD filters out slightly more of the unpromising objects than MetricD. However, the difference is not really significant. It is also worth mentioning that MetricDD requires twice as much memory as MetricD.

Comparison of our simulation algorithm with the original one

We compared our simulation algorithm against the original one in terms of total memory and time required to generate pivot table. By total memory we mean the memory required to build a suffix tree for the original version while for our algorithm, we mean the sum of *sa*, *sp*, *si* and stacks and arrays that were used during the simulation. The comparison is shown in Table 1. The auxiliary arrays *sp* and *si* were generated in 0.19 s for both of the datasets.

[2]This implementation is commonly used in many papers of the database community

	Our simulation algorithm			The original algorithm		
Dataset	Total Mem. (MiB)	p=10 Time (s)	p=20 Time (s)	Total Mem. (MiB)	p=10 Time (s)	p=20 Time (s)
DNA	169	376	755	551	394	787
Protein	169	935	1844	545	937	1872

Table 1: Comparison of total memory and pivot table construction time.

Table 1 shows that our algorithm performs better than the original version in all of the experiments. We note that auxiliary arrays *sp* and *si* are not needed anymore after the pivot table construction.

Query response time

To decide on a multidimensional index structure to use in our experiments, we compared the performance of linear scan and R-tree on a pivot table with *mind* and *maxd* which was generated for 10 pivots. The R-tree was built over the pivot table. We set the query radius to 0 and measured the total time to answer a query set and required disk space. The results are shown in Table 2.

	Linear scan		R-tree	
Dataset	Time (s)	Disk space (MiB)	Time (s)	Disk space (MiB)
DNA	970	1967	4799	5291
Protein	1368	2058	5681	5250

Table 2: Comparison of linear scan and R-tree on pivot table.

Table 2 shows that the linear scan outperforms the R-tree on the pivot table. In light of this, the linear scan is used as the multidimensional index part of the metric index.

As a baseline competitor for the metric index, we used an enhanced linear scan. We performed a linear scan using the suffix and longest common prefix arrays to speed up the computations of the edit distance between query and suffixes. Figure 6 shows the comparison of query execution time for both indexes.

The experiments show that the metric index was beaten the enhanced linear scan by several order of magnitude. The reason for this bad performance of the metric index is due to the filtration step (we discussed about the space transformation in the second paragraph of Section 3). We divided query set execution time for each of filtration and refine steps. Then we converted them in the scale of 100 % and the results are presented in Figure 7.

The figures show that the filtration step took most of the query execution time. Because of this filtration effect, we did not compare our method to the original one in terms of search time. The distance distribution histograms for *mind* of both pivot tables are shown in Figure 8. We see that the distances are highly concentrated between 13 to 23 for DNA and 22 to 27 for protein data set.

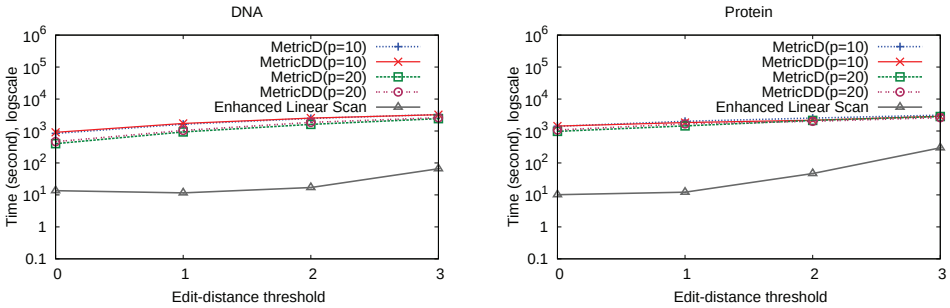


Figure 6: Comparison of query set execution time between the metric index and the enhanced linear scan.

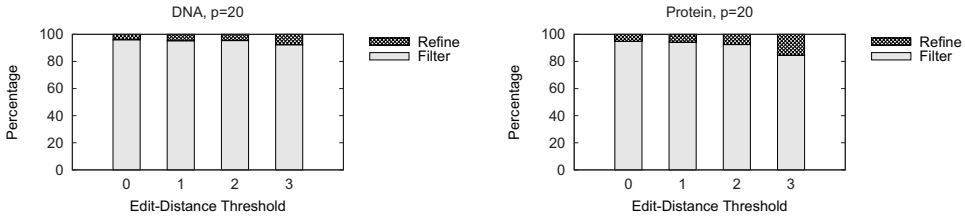


Figure 7: The percentage of filtration and refine steps in the query set execution time.

The histograms show that all the hyperrectangles more or less coincide. The *mind*s of the query object are also in this highly concentrated region. Because of this issue, the query hyperrectangle intersects with almost every indexed hyperrectangle. Thus the multidimensional search algorithm fails.

6 Conclusions

The primary goal of this paper was to empirically evaluate the metric index for approximate string matching based on suffix trees, introduced by Navarro et al. [9], as their original paper contained only theoretical results. In order to give the method a fair chance, we have proposed improvements that would reduce its memory consumption. We achieved this by simulating the index construction process using suffix arrays. Even with this improvement, though, our experiments on two real-data sets show that the metric index is impractical for real-world use, as it was clearly beaten by an enhanced version of a linear scan.

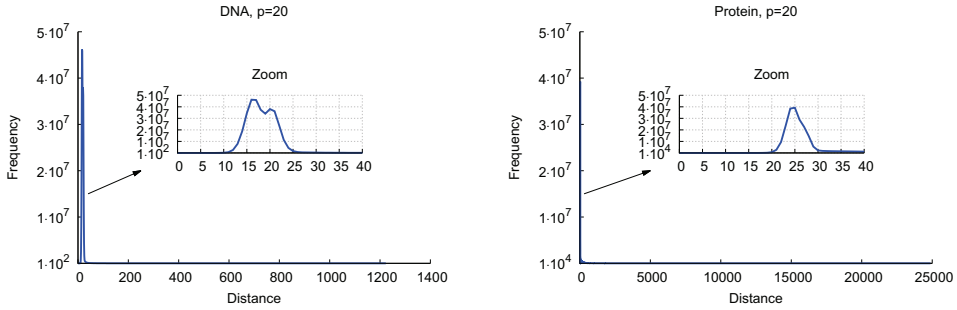


Figure 8: Distance distribution histograms of *mind* and its zoomed areas covering the distance values between 0 to 40 on X-axis.

Acknowledgements

We wish to thank Pål Sætrom for helpful discussions.

Appendix

Algorithm 2: Generate all the last split points for the strings of $\mathcal{I}$

```
1: initialize a stack seen
2: initialize a stack s that can store a pair of (lcp, pos)
3: for i = 1 to n − 1 do
4:     cur ← lcp[i]; next ← lcp[i + 1]
5:     while s ≠ ∅ and s.top().lcp > next do
6:         c = s.pop().pos
7:         sp[c] ← s = ∅ ? next : max(next, s.top().lcp)
8:     while seen ≠ ∅ and seen.top() > next do seen.pop()
9:     if next ≠ 0 and (seen = ∅ or seen.top() < next) then
10:        seen.push(next)
11:        s.push(next, i)
12: while s ≠ ∅ do
13:    c = s.pop().pos
14:    sp[c] ← (s = ∅) ? 0 : s.top().lcp
```

Algorithm 3: Generate all the suffix array indexes for the strings of $\mathcal{I}$

```
1: initialize a stack seen
2: initialize an array a that can store a pair of (lcp, idx)
3: for i = 1 to n − 1 do
4:     cur ← lcp[i]; next ← lcp[i + 1]
5:     if |a| = 0 then
6:         a.PushBack((next, i))
7:     else if a_{−1}.lcp = 0 then
8:         a_{−1} ← (next, i)
9:     else if a_{−1}.lcp < next then
10:        a.PushBack((next, i))
11:    else
12:        a_{−1}.lcp ← next
13:        while |a| ≥ 2 and a_{−2}.lcp > a_{−1}.lcp do
14:            a_{−2}.lcp ← a_{−1}.lcp
15:            a.PopBack()
16:    while seen ≠ ∅ and seen.top() > next do seen.pop()
17:    if next ≠ 0 and (seen = ∅ or seen.top() < next) then
18:        seen.push(next)
19:        si[i] ← a_{−1}.idx
```

Algorithm 4: Generate a pivot table column for a pivot p

initialize a stack *seen*
initialize a two dimensional array *ed*
$row \leftarrow 0$
for $i = 1$ **to** n **do**
  $cur \leftarrow lcp[i]; next \leftarrow 0$
  if $i + 1 \leq n$ **then**
    $next \leftarrow lcp[i + 1]$
    while $seen \neq \emptyset$ **and** $seen.top() > next$ **do** $seen.pop()$
    if $next \neq 0$ **and** ($seen = \emptyset$ **or** $seen.top() < next$) **then**
      $seen.push(next)$
      if $row + 1 \leq next$ **then**
        compute $d(S[row + 1 \ldots next], p)$
        $row \leftarrow next$
      $(mind, maxd) \leftarrow (min, max)\{ed[sp[i] + 1 \ldots next][|p|]\}$
      report $mind, maxd, si[i]$
  if $next \neq n - sa[i]$ **then**
    compute $d(S[row + 1 \ldots n - sa[i]], p)$ and let us assume that we are processing a row k ($row + 1 \leq k \leq n - sa[i]$) and then $mind \leftarrow min\{ed[row + 1 \ldots k][|p|]\}$ and early terminate the distance computation if the condition 1 holds
    $row \leftarrow$ (early terminated) ? $k : n - sa[i]$
    $row \leftarrow min(row, next)$
    report $mind$, $max(n - sa[i], |p|), i$

References

[1] L. Devroye, W. Szpankowski, and B. Rais. A note on the height of suffix trees. *SIAM J. Comput.*, 21(1):48–53, 1992.

[2] P. Ferragina and G. Navarro. Pizza & chili corpus. DNA and protein datasets downloaded on December 5th, 2011 from `http://pizzachili.dcc.uchile.cl`.

[3] D. Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology.* Cambridge University Press, 1997.

[4] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of SIGMOD*, pages 47–57, 1984.

[5] M. Hadjieleftheriou. Spatial index library v1.6.1. `http://research.att.com/~marioh/spatialindex`.

[6] S. Kurtz. Reducing the space requirement of suffix trees. *SoftwarePractice & Experience*, 29:1149–1171, 1999.

[7] H. Li and N. Homer. A survey of sequence alignment algorithms for next-generation sequencing. *Briefings in Bioinformatics*, 11(5):473–483, 2010.

[8] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33, 1999.

[9] G. Navarro and E. Chávez. A metric index for approximate string matching. *Theoretical Computer Science*, 352:266–279, March 2006.

[10] M. Ruffalo, T. LaFramboise, and M. Koyutrk. Comparative analysis of algorithms for next-generation sequencing read alignment. *Bioinformatics*, 27(20):2790–2796, 2011.