Efficient Index Structures for String Databases

Tamer Kahveci   Ambuj Singh
Department of Computer Science
University of California
Santa Barbara, CA 93106
{tamer,ambuj}@cs.ucsb.edu

Abstract

We consider the problem of substring searching in very large string databases. Typical applications of this problem are genetic data, web data, and event sequences. Since the size of such databases grows exponentially, it becomes impractical to use in-memory algorithms for these problems. In this paper, we propose to map the substrings of the data into an integer space with the help of wavelet coefficients. Later, we index these coefficients using Minimum Bounding Rectangles. We define a distance function which is a lower bound to the actual edit distance between strings. We experiment with both nearest neighbor queries and range queries. The results show that our technique prunes significant amount of the database (typically 50-95%), thus reducing both the disk I/O cost and the CPU cost significantly.

1 Introduction

String data naturally arises in many real world applications like genetic data, web data and event sequences. There is a frequent need to find similarities between such data. For example, the similarity of two gene strings from different organisms may correspond to some functional or physical relation between these organisms. The similarity of substrings of a genetic string to a query string may be used to predict some diseases in advance, or to find some functional relationship between different species. Significant breakthrough have already been achieved in genome research using the analysis of similar genetic strings. Identification of the genetic code of the deadly E.coli bacteria, or genetic clues for fibrodysplasia ossificans progressiva (FOP), a disease that affects muscle and skeleton growth, and the vital proteins for the bone growth, or identification of the genes that hasten the healing of some venous ulcers are only a few of the achievements obtained recently.

Another application of substring searching is the identification of similar patterns in large text databases while allowing some amount of typographical errors. This application includes searching a word in a dictionary, or a phrase in a large collection of text. Spell checkers and web searchers are some specific examples of such applications.

Video data can be viewed as an event sequence if some prespecified set of events are detected and stored as a sequence. These events can be voices, faces, objects, or text. Video databases include a wide variety of applications including security cameras, interviews, documentaries, movies, and TV news. Searching similar event subsequences can be used to find related video files in a large collection video database. Some companies like CNN, ABC, CNET, and AltaVista are already encoding and indexing video data. For example, ABC uses a search engine which enables one to search some specific text that appeared in ABC news. A number of universities are also recording lectures and seminars, with the aim of providing online access and search capabilities.

String data applications generally involve very large databases. GenBank [7], a database of nucleotide and protein strings built by National Center for Biotechnology Information (NCBI), is an example for such databases. Figure 1 plots the growth of the number of sequences and the size of the database over years 1982 to 2000. The statistics show that the size of GenBank has doubled every 15 months [8]. Similarly, the size of a video database can also increase dramatically. Only CNN has more than 150 hours of news feed every day, and they will be encoding more than 100,000 hours of archived material.

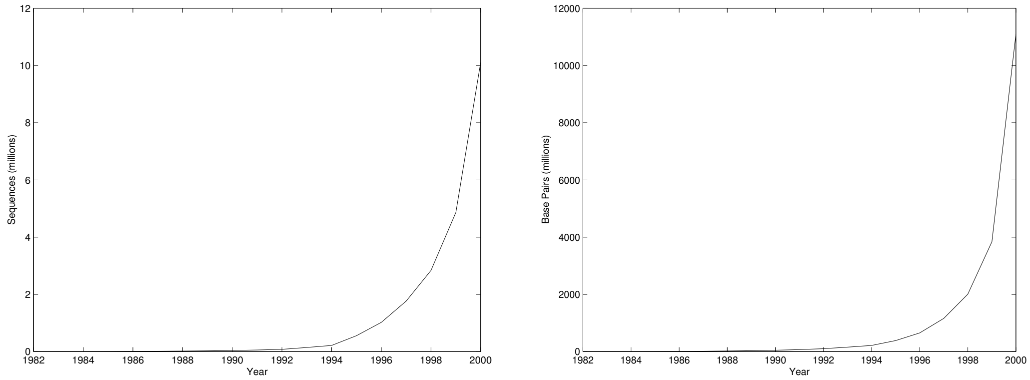

Figure 1. The growth of the number of sequences and the size of the database over years.

String search algorithms proposed so far are all in-memory algorithms [2, 4, 5, 6, 10, 16, 17, 18, 20, 21, 22]. That is, these techniques assume that the database fits into memory. Therefore, these techniques suffer from disk I/Os when the database is too large. In-memory algorithms are impractical for this kind of applications because, the database size grows faster than the available memory capacity, and extensive memory requirements make the data and the search programs impractical for personal computers. Therefore, external memory algorithms are needed for most string comparison applications of the future.

A string s_1 can be transformed into another string s_2 by using three *edit operations*, namely *insert*, *delete*, and *replace*, on individual characters of the string s_1. Figure 2 presents a transformation of the string ACTTAGC to

AATGATAG using edit operations. The transformation given in Figure 2 consists of 1 replace, 2 insert, and 1 delete operations. The difference between two strings s_1 and s_2 is generally defined as the minimum number of edit operations to transform s_1 to s_2, called *edit distance* (ED). Let m and n be the lengths of strings s_1 and s_2, then the edit distance, $ED(s_1, s_2)$, and the corresponding edit operations can be determined in $O(mn)$ time and space using dynamic programming [10]. The space complexity can be reduced to $O(min\{m, n\})$ if only the edit distance is needed (i.e. the corresponding edit operations are not required). Some applications assign different weights to different edit operations or different character pairs [12], leading to a *weighted edit distance*. The time and space complexity of finding the weighted edit distance is also $O(mn)$ by using dynamic programming.

```
A C T - - T A G C
  R   I I       D
A A T G A T A G -
```

Figure 2. Transformation of the string ACTTAGC to AATGATAG using edit operations. Here, the edit operations are represented by I=insert, D=delete, and R=replace.

An *alignment* of strings s_1 and s_2 is obtained by matching each character of s_1 to a character in s_2 in increasing order. All the unmatched characters in both strings are matched with space. An alignment of strings ACTTAGC and AATGATAG is given in Figure 2. The *dashes* (i.e., -) in this figure correspond to spaces. Each character pair is assigned a score based on their similarity, and these values are stored in a *score matrix*. The value of an alignment is defined as the sum of the scores of all of their character pairs. *Global alignment* (or *similarity*) of s_1 and s_2 is defined as the the maximum valued alignment of s_1 and s_2. Finding the similarity of two strings is the dual of finding the distance between them. *Local alignment* [20] of s_1 and s_2 is defined as the highest valued alignment of all the substrings of s_1 and s_2. Both global and local alignments can be determined in $O(mn)$ time using dynamic programming.

Some genetic applications assume that the cost of inserting or deleting a block of characters is less than that of inserting or deleting the same number of characters individually. Inserting or deleting a block of characters corresponds to a *gap*. A common way to compute the cost of a gap of length g is $\beta_1 + \beta_2 \times (g - 1)$, called *affine gap* [2]. Here, β_1 is the cost of starting a gap and β_2 is the contribution of each space in the gap to the total cost, where $\beta_1 > \beta_2$.

In this paper, we consider the problem of range queries and nearest neighbor queries. The typical databases that we work with include very long strings. For example, the string corresponding to chromosome-22 of humans has about 35 million base pairs. (A base pair is one of A,C,G, or T characters corresponding to the four different kinds of nucleic acids.) A k-nearest neighbor returns the k closest substrings from the database for a given query.

A range query, on the other hand, returns the substrings that lie within a given distance of the input query.

We propose a wavelet-based method to map the substrings of the database into a multidimensional integer space. The number of dimensions is determined by the alphabet size and the number of wavelet coefficients. We define a notion of distance in this integer space that is a lower bound to the actual edit distance. A sliding window is used to translate a set of contiguous substrings into an MBR. Repeating this over all the strings generates an array of MBRs corresponding to one resolution (window size) for the database. We use a hierarchical scheme in which windows of successive coarser grain are used. This generates an approximation to the database at different granularities, and results in a grid of MBRs. The resulting index structure is quite compact and can be stored in memory. Typical size of this index structure ranges between 1-2% of the database size. Range queries and nearest-neighbor queries are first performed using this in-memory index structure using the lower-bound distance. The resulting set of candidate pages are then accessed from the disk to remove false hits (using the actual edit distance).

According to experimental results, our method runs 5 to 45 times faster than other techniques for nearest neighbor queries of 10 to 200 nearest neighbors and 2 to 12 times faster than other techniques for range queries.

The rest of the paper is as follows. Section 2 discusses the related work. Section 3 discusses the substring searching problem and defines our index structure and algorithms. Section 4 discusses the experimental results. We end with a brief discussion in Section 5.

2 Related Work

The dynamic programming solution to the problem of finding the substrings of a given string s of length n, which are within a distance of $r = \epsilon \times |q|$ to a query string q of length m, runs in $O(mn)$ time and space. This technique is a variation of the dynamic programming algorithm that finds the edit distance between two strings by generating a distance matrix of size $m \times n$. For long data and query strings, this technique becomes infeasible in terms of both time and space. Myers [17] improved the time and space complexity to $O(rn)$ by maintaining only the required part of the distance matrix. However, for large error rates r is $O(m)$, and hence the complexity is still $O(mn)$.

Wu and Manber [22] proposed a technique that uses r binary masks M_1, M_2, ..., M_r of length m. They scan through the data string s and update these masks for each character in s. After the j^{th} character is processed, the value of $M_k[i]$ becomes 1 if the last i characters of s are within k edit operations to the first i characters of q. If w is the size of a word, the algorithm runs in $O(nr\frac{m}{w})$ time. The space requirement of this technique is $O(r\frac{m}{w})$. The algorithm runs efficiently for small values of m (close to $O(nr)$), but the performance degrades for large m. Furthermore, the space requirement may become much larger than the data string for large m and r.

In another paper, Myers [18] proposed a technique that preprocesses the data string s and creates an index of size $O(n)$. This technique assumes a lower bound l on the length of the query string ($l = log(n)$ here). All possible strings of length l are mapped to integers using a perfect hashing function. Later, the leftmost points of all the occurrences of these strings in s are stored in separate lists. For a given query q and query radius r, the technique generates the set of strings which are within edit distance of r to q, called *condensed r-neighborhood*. The strings in the condensed r-neighborhood are searched in the index to find the answers to the query. If the query length is larger than l, the technique splits the query string into subqueries, searches each subquery separately and combines the results. Since this technique indexes all possible strings of some prespecified length, we call it a *dictionary based technique*. The author proves that if the database is created as a result of equi-probable Bernoulli trials, then the technique runs in sublinear time. There are two drawbacks with this technique. First, although the space complexity is $O(n)$, the index size can be 7-9 times larger than the data size. This may cause a drop in performance if the index does not fit into the memory. Second, the worst case running time complexity of this technique is very high.

Baeza-Yates and Navarro proposed an NFA-based solution in [5]. In this paper, the authors propose to construct an NFA of $(r + 1) \times (m + 1)$ states, which accepts s as the input string. The NFA is constructed using the query string. The NFA goes into an accepting state whenever a substring within edit distance of r is processed. The authors propose to use only the required states of the NFA at any time. The expected running time of this technique is $O(mn\frac{r}{w})$, where w is the size of a word. The experimental results presented in the paper show that for short queries and small alphabets, this technique over performs other techniques. The performance of this technique drops when s is very long (i.e. it does not fit into memory).

Altschul, Gish, Miller, Myers, and Lipman proposed the *BLAST* technique [3] to find local similarities. BLAST, the most popular string matching tool for biologists, runs in two phases. In the first phase, all the substrings of the query of some prespecified length (typically between 3 and 11) are searched in the database for an exact match. In the second phase, all the matches obtained in the first phase are extended in both directions until the similarity between the two substrings falls below some threshold. This technique keeps a pointer to the starting locations of all possible substrings of the prespecified length in the database to speedup the first phase. Therefore, the space requirement of BLAST is more than the size of the database. Furthermore, BLAST does not find a similar substring to the whole query string, only similarities between the query substrings and the database substrings.

Giladi, Walker, Wang, and Volkmuth [9] considered a heuristic solution for the same problem which runs in $O(logn)$ expected time. This technique splits the data strings into overlapping windows of length l for some prespecified overlap amount of Δ. For each such window, they count the number of repetitions of all the possible *k-tuples*, and store this value in a σ^k dimensional vector, where σ is the alphabet size. Later, these vectors are indexed using a hierarchical binary tree. The authors propose to approximate to the actual similarity between the query string and a substring by using D_1 distance between these vectors. Experimental results show that this technique runs 25 to 50 times faster than BLAST [3]. The authors also note that this technique can be used as a preprocessing step to speed up any string search program. There are two drawbacks with this method: it allows false drops and the index size increases exponentially with k.

A special case of the substring matching problem is *exact matching* (i.e. $r = 0$). One can solve this problem using suffix trees [10] in which all the suffixes of s are stored in a tree. However, the size of the suffix tree may be more than 10 times larger than the data. Manber and Myers [16] propose a data structure called *suffix arrays* to reduce the space requirement for the index structure. However, the space requirement is still more than 4 times the data size.

Jagadish, Koudas, and Srivista consider the problem of exact matching with wild-card for multidimensional strings in [13]. The authors map all the strings to real space based on their lexical order. Later, these multidimensional points are indexed using R-trees [11]. The strings within a data page are stored using *elided tries* for each dimension. This technique works efficiently if the database consists of short strings. However, the performance degrades for long strings. This is because it becomes more difficult to distinguish long strings by inspecting (the much shorter) prefixes of these strings.

3 Proposed Solution

The *edit distance* (ED) between two strings is defined as the minimum number of edit operations to transform one string to the other. There are three edit operations: *insert*, *delete*, and *replace*. These edit operations work on individual characters. Figure 2 presents a sample transformation of the string ACTTAGC to AATGATAG using edit operations.

String matching problem can be classified in two groups. These are *whole matching* and *substring matching*. The simpler case, whole matching, considers the problem of finding the edit distance $ED(q, s)$ between a data string s and a query string q. Substring matching searches all the substrings $s[i : j]$ of s which are close to the query string.

Given a string database $S = \{s_1, s_2, ..., s_d\}$ consisting of very long strings, we consider two types of queries:

- *Range search* seeks all the substrings of S which are within an edit distance of r to a given query string q, where r is the query range. We define $\epsilon = \frac{|q|}{r}$ as the *error rate*.
- *k-nearest neighbor search* seeks the k closest substrings of S to q.

There are several challenges in this problem. 1) Finding the edit distance is very costly in terms of both time and space. 2) The strings in the database may be very long. For example, the length of chromosome-22, one of the shortest chromosomes in human genome library, is approximately 35 million bp (base pairs). For a query string of length ten thousands and an error rate of $\epsilon = 0.01$, there are billions of possible substrings. Therefore it is infeasible to check all the substrings. 3) The database size for most applications grows exponentially. Therefore, a solution method based on sequentially scanning all the database will suffer from disk I/Os.

The existing solution techniques work well only for short strings and low error rates. Furthermore, the sizes of most of the proposed index structures are even larger than the database itself. Such an index structure will not fit into the memory, and suffer from disk I/Os even in the index search phase.

The rest of this section is as follows. Section 3.1 defines a lower bound distance for substring searching. Section 3.2 improves this lower bound by using the idea of wavelet transformation. Section 3.3 defines the MRS-index structure based on the aforementioned distance formulations. Sections 3.4 and 3.5 define the algorithms for range queries and nearest neighbor queries.

3.1 A new distance function

We define a transformation, $f(s)$ that maps a string s to a point in a multidimensional integer space as follows:

Definition 1 *Let s be a string from the alphabet $\Sigma = \{\alpha_1, \alpha_2, ..., \alpha_\sigma\}$. Let n_i be the number of occurrences of the character α_i in s for $1 \leq i \leq \sigma$. We define the* frequency vector, *$f(s)$, of s as:*

$$f(s) = [n_1, n_2, ..., n_\sigma].$$

For example, if s=TACTTAG is a genetic string (i.e. from alphabet $\Sigma = \{A, C, G, T\}$), then $f(s) = [2, 1, 1, 3]$ (We use alphabetic order in the construction of $f(s)$.). Three important notes follow this definition. 1) The transformed string $f(s)$ has σ dimensions independent of the length of s. 2) The sum of the entries of $f(s)$ is independent of the contents of s. This fact is explained in Lemma 1. 3) All the entries of $f(s)$ are nonnegative.

Lemma 1 *Let s be a string from the alphabet $\Sigma = \{\alpha_1, \alpha_2, ..., \alpha_\sigma\}$. Let $f(s) = [v_1, v_2, ..., v_\sigma]$ be the frequency vector of s, then $\Sigma_{i=1}^{\sigma} v_i = |s|$.*

Since the sum of the entries of $f(s)$ is n for all strings of length n, the transformations of all strings of length n lie on the $\sigma - 1$ dimensional plane that passes through the point $[n, 0, 0, ..., 0]$ and perpendicular to the normal vector $[1, 1, ..., 1]$. The relationship between the edit operations and the frequency vectors is captured in Theorem 1.

Theorem 1 *Let s be a string from the alphabet $\Sigma = \{\alpha_1, \alpha_2, ..., \alpha_\sigma\}$. Let $f(s) = [v_1, v_2, ..., v_\sigma]$ be the frequency vector of s. An edit operation on s has one of the following effects on $f(s)$, for $1 \leq i, j \leq \sigma$, and $i \neq j$:*

1. $v_i := v_i + 1$
2. $v_i := v_i - 1$
3. $v_i := v_i + 1$ *and* $v_j := v_j - 1$

Proof:

Case 1 corresponds to inserting α_i in some location of s.

Case 2 corresponds to deleting α_i from s.

Case 3 corresponds to replacing α_j with α_i in s. $\square$

Theorem 1 shows that a single edit operation on a string results in a limited change in the corresponding integer space. We can conclude that if string s' is obtained by applying one edit operation to s, then $||f(s) - f(s')||_2 \leq 2^{1/\sigma}$. Keeping this fact in mind, we define *neighbor points* as follows:

Definition 2 *Let u and v be integer points in σ dimensional space, then u and v are called* neighbors *if $||u-v||_2 \leq 2^{1/\sigma}$.*

Definition 2 can be stated in another way: two integer points are neighbors if they are transformations of two strings such that one of them can be obtained from the other using a single edit operation. Next we define a new distance function, *frequency distance* (FD_1), for the frequency vectors, which is a lower bound on the edit distance of the corresponding strings. The idea is based on Theorem 1.

Definition 3 *Let u and v be integer points in σ dimensional space. The* frequency distance*, $FD_1(u, v)$, between u and v is defined as the minimum number of steps in order to go from u to v (or equivalently from v to u) by moving to a neighbor point at each step.*

Theorem 2 proves that the frequency distance between the frequency vectors of two strings is a lower bound on their edit distance.

Theorem 2 *Let s_1 and s_2 be two strings from the alphabet $\Sigma = \{\alpha_1, \alpha_2, ..., \alpha_\sigma\}$, then*

$$FD_1(f(s_1), f(s_2)) \leq ED(s_1, s_2).$$

Since FD_1 is a lower bound on ED, we can conclude the following Corollary:

Corollary 1 *Let q and s be two strings from the alphabet $\Sigma = \{\alpha_1, \alpha_2, ..., \alpha_\sigma\}$, then*

$$\textit{if } r < FD_1(f(q), f(s)) \textit{ then } r < ED(q, s).$$

Given a query string q and a query range r, one can prune a string s without computing $ED(q, s)$ if $r < FD_1(f(q), f(s))$ by Corollary 1. Another important attribute of the frequency distance function is that it satisfies metric properties. One can easily prove this by using the definition of the frequency distance function.

Figure 3 presents an efficient algorithm to compute $FD_1(u, v)$ for σ dimensional integer points u and v. The algorithm computes two values, namely *posDistance*, and *negDistance*. PosDistance defines the number of decrement operations that must be applied to u, and similarly negDistance defines the number of increment operations

Algorithm $FD_1(u,v)$ /* u and v are σ dimensional integer points. */

- $posDistance := negDistance := 0$
- for i:= 1 to σ
  - if $u_i > v_i$ then $posDistance$ += $u_i - v_i$
  - else $negDistance$ += $v_i - u_i$
- if $posDistance > negDistance$ then return $posDistance$ else return $negDistance$

Figure 3. Frequency Distance algorithm

that must be applied to u. Since each edit operation changes the values of posDistance and negDistance by at most one, the larger of the two values equals $FD_1(u,v)$.

3.2 Improving the lower bound distance

The frequency distance can be improved by storing local frequencies of the characters in the string as well as the global frequencies. For this, we define the wavelet transformation of a string. Assume n is a power of 2 for simplicity in the following development.

Definition 4 *Let $s = c_1c_2...c_n$ be a string from the alphabet $\Sigma = \{\alpha_1, \alpha_2, ..., \alpha_\sigma\}$, then k^{th}*-level wavelet transformation, *$\psi_k(s)$, $0 \leq k \leq log_2 n$, of s is defined as:*

$$\psi_k(s) = [v_{k,1}, v_{k,2}, ..., v_{k,\frac{n}{2^k}}], \text{ where } v_{k,i} = [A_{k,i}, B_{k,i}],\ 1 \leq i \leq \tfrac{n}{2^k}, \text{ and}$$

$$A_{k,i} = \begin{cases} f(c_i) & k = 0 \\ A_{k-1,2i-1} + A_{k-1,2i} & 0 < k \leq log_2 n \end{cases}$$

and

$$B_{k,i} = \begin{cases} 0 & k = 0 \\ A_{k-1,2i-1} - A_{k-1,2i} & 0 < k \leq log_2 n \end{cases}$$

As an example of the above definition, it is possible to show that $\psi_3(TCACTTAG) = [[2,2,1,3],[0,2,-1,-1]]$. The above formulation is similar to the Haar wavelet except for a scaling factor. Some important properties of the wavelet transformation of a string s are as follows. 1) The A coefficients of ψ_0 defines the string. 2) Global frequency vector $f(s)$ as developed in Section 3.1 is equal to $A_{log_2 n,0}$ 3) Each $A_{k,i}$ coefficient corresponds to the frequency vector of a substring of s of length 2^k. Formally, $A_{k,i} = f(s[2^k i : 2^k(i+1)-1])$. Hence, $A_{log_2 n,0} = f(s)$.

4) Each $B_{k,i}$ coefficient corresponds to the difference of the frequency vectors of two consecutive substrings of s of length 2^k. Formally, $B_{k,i} = f(s[2^{k-1}(2i-1) : 2^{k-1}k2i-1]) - f(s[2^{k-1}2i : 2^{k-1}k(2i+1)-1])$. Hence, $B_{log_2 n,0} = f(s[1 : \frac{n}{2}-1]) - f(s[\frac{n}{2} : n])$. 5) $A_{k-1,2i-1}$ and $A_{k-1,2i}$ can be constructed using $A_{k,i}$ and $B_{k,i}$.

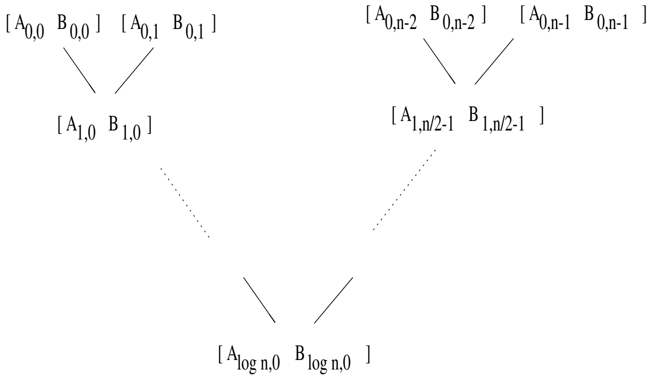

Figure 4. Wavelet decomposition of a string

The hierarchical structure of the wavelet transformation of a string is given in Figure 4. The coefficients $A_{log_2 n-1,0}$ and $A_{log_2 n-1,1}$ can be computed if the coefficients $A_{log_2 n,0}$ and $B_{log_2 n,0}$ are known in advance. We name $A_{log_2 n,0}$ as the *first wavelet coefficient*, and $B_{log_2 n,0}$ as the *second wavelet coefficient*. If the B coefficients of ψ_{k-1} are also known, then the A coefficients of ψ_{k-2} can be computed. The B coefficients of ψ_{k-2} are called third and fourth wavelet coefficients. In general, if the first wavelet coefficient and all the B coefficient of ψ_i for $t \leq i \leq log_2 n$ are known, then all the A coefficients of ψ_{t-1} can be determined exactly.

In Section 3.1, we transformed the strings to their first wavelet coefficients. As the number of wavelet coefficients increases, the accuracy of the lower bound function increases at the cost of a larger index size. This is shown next. We will focus our development on the first two wavelet coefficients; however, the idea can be generalized to any number of coefficients. Here on, we will use $\psi(s)$ instead of $\psi_{log_2|s|}(s)$ for simplicity.

Theorem 3 *Let s be a string from the alphabet $\Sigma = \{\alpha_1, \alpha_2, ..., \alpha_\sigma\}$. Let $\psi(s) = [A, B]$ be the first and the second wavelet coefficients of s. Let $A = [a_1, ..., a_\sigma]$ and $B = [b_1, ..., b_\sigma]$, then an edit operation on s has one of the following effects on A and B for $1 \leq i, j \leq \sigma$, and $i \neq j$:*

1. $a_i := a_i + 1$, $a_j := a_j - 1$, $b_i := b_i + 1$, $b_j := b_j - 1$.

2. $a_i := a_i + 1$, $a_j := a_j - 1$, $b_i := b_i - 1$, $b_j := b_j + 1$.

3. $a_i := a_i \pm 1$, $b_i := b_i \pm 1$.

4. $a_i := a_i \pm 1$, $b_i := b_i + 1$, $b_j := b_j - 2$.

5. $a_i := a_i \pm 1$, $b_i := b_i - 1$, $b_j := b_j + 2$.

Proof:

This can be proven by splitting the string into 2 equal parts and inspecting the effect of the edit operations on these substrings. □

The wavelet transform $\psi(s)$ can be considered as a point in a 2σ dimensional integer space. Theorem 3 lists the legal steps that can be used to move from $\psi(s_i)$ to $\psi(s_j)$, where s_i and s_j are strings. The transformation of s_1 to s_2 using the edit operations corresponds to a legal path between their wavelet transformations. Therefore, the edit distance between s_1 and s_2 is at least the number of steps in the shortest legal path from $\psi(s_i)$ to $\psi(s_j)$. Lemma 2 defines a lower bound, $FD_2(\psi(s_i), \psi(s_j))$, to the number of steps in the shortest legal path in 2σ dimensional integer space based on the legal operations given in Theorem 3.

Lemma 2 *Let s_1 and s_2 be strings from the alphabet $\Sigma = \{\alpha_1, \alpha_2, ..., \alpha_\sigma\}$. Let $\psi(s_1) = [A_1, B_1]$ and $\psi(s_2) = [A_2, B_2]$ be the first and second wavelet coefficients of s_1 and s_2. Let $A_1 = [a_{1,1}, ..., a_{1,\sigma}]$, $B_1 = [b_{1,1}, ..., b_{1,\sigma}]$, $A_2 = [a_{2,1}, ..., a_{2,\sigma}]$, and $B_2 = [b_{2,1}, ..., b_{2,\sigma}]$. Let*

$$pos = \sum_{i, a_{1,i} > a_{2,i}} (a_{1,i} - a_{2,i}) + \sum_{i, b_{1,i} > b_{2,i}} (b_{1,i} - b_{2,i}),$$

$$neg = \sum_{i, a_{1,i} < a_{2,i}} (a_{2,i} - a_{1,i}) + \sum_{i, b_{1,i} < b_{2,i}} (b_{2,i} - b_{1,i}).$$

Let min be the minimum of pos and neg, then

$$FD_2(\psi(s_i), \psi(s_j)) = \begin{cases} \frac{|pos-neg|}{2} & min < \frac{|pos-neg|}{2} \\ \frac{|pos-neg|}{2} + \frac{min - \frac{|pos-neg|}{2}}{2} & \textit{otherwise} \end{cases}$$

Proof:

Let max be the maximum of pos and neg. Steps 4 and 5 of Theorem 3 makes the largest change on the difference $\psi(s_i) - \psi(s_j)$ (+3, -1 or -3, +1). Therefore, these steps must be used whenever $|pos - neg| > 0$. If $min < \frac{|pos-neg|}{2}$, then these steps can be used at most min times. After that, only Step 3 $(\pm 2, 0)$ can be used to increment or decrement the remaining value by 2 at each step. Therefore, at least $\frac{max-3min}{2}$ more steps are needed. This makes the total distance $min + \frac{max-3min}{2} = \frac{|pos-neg|}{2}$.

A similar reasoning can be made for the second part. If $min > \frac{|pos-neg|}{2}$, then steps 4 and 5 can be used at most $\frac{|pos-neg|}{2}$ times. This makes $pos = neg$, so Step 1 and Step 2 (+2, -2 or -2, +2) can be used for the rest. Therefore, at least $\frac{|pos-neg|}{2} + \frac{min - \frac{|pos-neg|}{2}}{2}$ steps are needed to move from $\psi(s_i)$ to $\psi(s_j)$. □

Lemma 2 constructs a lower bound to the edit distance using the first and second wavelet coefficients at the same time. However, $FD_1(f(s_1), f(s_2))$ is not necessarily less than $FD_2(\psi(s_1), \psi(s_2))$. If the first and second wavelet coefficients are known, then the distance $FD_1(s_1, s_2)$ can also be computed efficiently. Therefore, we define the *maximum frequency distance* (FD) between s_1 and s_2 as

$$FD(s_1, s_2) = max\{FD_1(f(s_1), f(s_2)), FD_2(\psi(s_1), \psi(s_2))\}.$$

3.3 The MRS index structure

Let $S = \{s_1, s_2, ..., s_d\}$ be a database consisting of potentially long strings from alphabet $\Sigma = \{\alpha_1, \alpha_2, ..., \alpha_\sigma\}$.. Let $w_1 = 2^a$ be the length of the shortest possible query string. Our index structure stores a grid of trees $T_{i,j}$, where i ranges from a to $a + l - 1$, and j ranges from 1 to d. The parameter l represents the number of resolution levels available in the index structure. Tree $T_{i,j}$ is the set of MBRs for the j^{th} string corresponding to window size 2^i. Figure 5 shows a layout of this index structure.

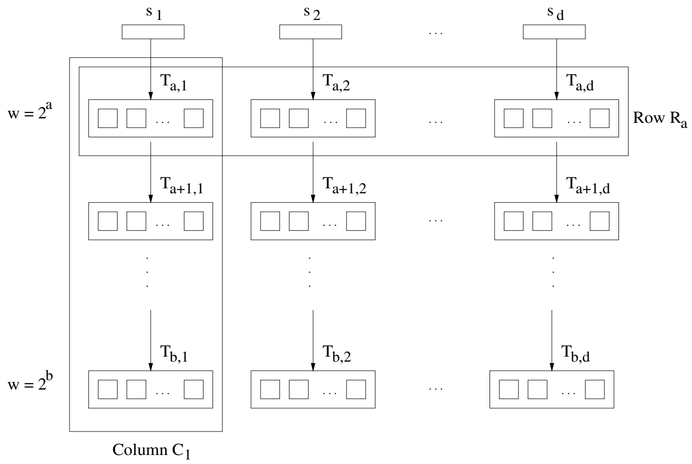

Figure 5. Layout of the MRS-index structure

In order to obtain $T_{i,j}$, we slide a window of length 2^i on string s_j, starting from the leftmost point of s_j. For each possible placement of the window, we compute the wavelet transformation of the corresponding substring of s_j, and store the first two wavelet coefficients. Note that each substring corresponds to a point in the 2σ dimensional integer space [1]. We begin with the initial substring and find the minimum box, called *Minimum Bounding Rectangle* (MBR), that covers the wavelet coefficients of this substring. This box is later extended to cover the transformations of the first c substrings, where c is the box capacity. (We will later discuss the impact of the value of c on the efficiency of the index structure.) After the first c substrings are transformed, a new MBR is created to cover the next c substrings. This process continues until all substrings are transformed. Note that, we only store the lower and higher end points of the MBRs along with the starting locations of the first substring contained in that MBR. Since the index structure stores frequencies at different resolutions, we call this index structure the *Multi Resolution String Index Structure* (MRS).

The i^{th} row of the MRS index structure is represented by R_i, where $R_i = \{T_{i,1}, ..., T_{i,d}\}$ corresponds to the set of all trees at resolution 2^i. Similarly, the j^{th} column of the MRS index structure is represented by C_j, where

[1]One can store any number of wavelet coefficients to construct the index structure. We explain the two coefficient version in this section. We report experimental results for both one coefficient and two coefficient versions in Section 4.

$C_j = \{T_{a,j}, ..., T_{a+l-1,j}\}$ corresponds to the set of all trees for the j^{th} string in the database.

Let q be a query string of length 2^i, where $a \leq i \leq a+l-1$. Given an MBR B, we define $FD(q,B) = min_{s \in B} FD(q,s)$. We write the following:

- if $r \leq FD(q,B)$ then $r \leq FD(q,s)$ for all $s \in B$.
- As the box capacity c increases, the box volume increases. As a result, the query-box distance $FD(q,B)$ decreases, and the performance of the index structure deteriorates.
- For fixed value of c, $FD(q,B)$ decreases as the window size (w) decreases. This can be explained as follows. Recall that the sum of the entries of the frequency vectors of substrings of length w is constant (i.e. w). Furthermore, frequency vectors contain a fixed number of dimensions (i.e. σ), and each dimension has a nonnegative value. Hence, there are $C(w+\sigma-1, w)$ possible frequency vectors for substrings of length w. Consequently, with decreasing w, the set of frequency vectors in the MBR constitutes a higher percentage of the set of all possible frequency vectors. As a result, the probability that $f(q)$ is contained in the MBR increases, and $FD(q,B)$ decreases. We verified this for our datasets by computing the average volume of an MBR for different window sizes. This is plotted in Figure 6. According to this figure, the average box volume increases exponentially as w decreases.
- The wavelet coefficients of the substrings obtained by sliding the window only one character are very close to each other. Therefore, the set of wavelet coefficients in an MBR are generally highly clustered.

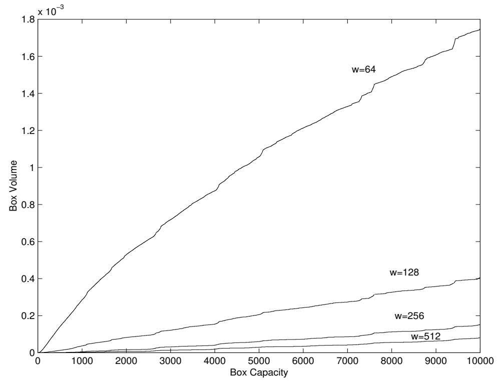

Figure 6. The average volume of MBRs for various box capacities.

3.4 Range Queries

Our search technique partitions a given query string of arbitrary length into a number of subqueries at various resolutions available in our index structure. Later, it performs a *partial range query* for each of these subqueries on the corresponding row of the index structure. This is called a partial range query, because it only computes distance of the wavelet transform of the query substring to the MBRs, not the distance of the query string to the substrings contained in the MBRs.

Given any query q of length $k2^a$ and a range ϵ, there is a unique partitioning[2], $q = q_1 q_2 ... q_t$, with $|q_i| = 2^{c_i}$ and $a \leq c_1 < ... < c_i \leq c_{i+1} \leq ... c_t \leq a + l - 1$. This partitioning technique chooses the longest possible suffix of q, such that its length is equal to one of the resolutions available in the index, as the last query substring. Later, it recursively partitions the rest of the strings to find the other query substrings.

We first perform a search using q_1 on row R_{c_1} of the index structure. As a result of this search, we obtain a set of MBRs that lie within a distance of $r = \epsilon \times |q|$ from q_1. Using the distances to these MBRs, we refine the value of r for each MBR and make a second query using q_2 on row R_{c_2} and the new value of r. This process continues for the remaining rows R_{c_3} ... R_{c_t}.

The relationship between an MBR and the substrings of the string that forms this MBR is captured in the following lemma.

Lemma 3 *Let s be a string and B be the MBR that covers the wavelet transforms of all the substrings of length w in s. Let q be a query string of length w. Let d be the minimum edit distance between q and all substrings of s, then*

$$FD(q, B) \leq d.$$

One can use Lemma 3 to find a lower bound on the distance of the query substring and the substrings of the data string. The sum of the distances for all query substrings defines a lower bound for the distance between the query string and data substrings.

An example partitioning of a query string is presented in Figure 8. In this example, the length of the query string is 1152 and the minimal query length is 128. Assume that the index structure contains three resolutions: 128, 256, and 512. The query is split into 3 pieces of length 128, 512 and 512. Three subqueries are performed, one for each partition. The first subquery q_1 is performed on row R_7 with r as the radius. As a result of this search, r is refined to r'_k for each MBR B_k. The next subquery q_2 is performed on row R_9 with the smaller radius of r'_k. As a result of this subquery, r'_k is refined further to r''_k. For example, if the first MBR B_1 in row R_7 contains substrings of $s[1 : 10000]$, then we need to check the MBRs corresponding to the substrings in $s[128 - r + 1 : 10000 + 512 + r]$ for the second subquery in order to allow at most r insertions and deletions between query substrings. Finally, subquery q_3 is performed on row R_9 with radius r''_k. The resulting set of MBRs is then processed to find the actual substrings using disk I/Os.

[2]If the length of the query string is not a multiple of the minimum window size, then the longest prefix of the query whose length is a multiple of the minimum window size can be used.

Algorithm $RANGE\_SEARCH(q, r)$

1. Partition the query q into t parts as q_1, q_2, ..., q_t such that $|q_i| = 2^{c_i}$ and $a \leq c_1 < c_2 < ... < c_i \leq c_{i+1} \leq$ ...... $\leq c_t \leq b$. Let $T_{c_1,j}$ contain M_j boxes.
2. For $j := 1$ to d
   (a) For $k := 1$ to M_k $distance[k] := r$
   (b) For $i := 1$ to t
      i. For $k := 1$ to M_j
         A. Perform range query on MBRs corresponding to q_i using query radius $distance[k]$. Let $Res_{c_i,j}$ be the resulting set of MBRs whose distances to q_i are less than $distance[k]$.
         B. $distance[k] := max_{B \in Res_{c_i,j}} \{distance[k] - FD(q_i, B)\}$
   (c) Read disk pages corresponding to $Res_{c_t,j}$
   (d) Perform postprocessing to eliminate false retrievals.

Figure 7. Range search algorithm.

Figure 7 presents the complete search algorithm. Step 1 partitions the query q into separate pieces corresponding to a subset of the rows R_{c_1}, R_{c_2}, ..., R_{c_t} of the index structure. In Step 2, these rows are searched from top to bottom independently for each MBR (column wise) in the database. At every row, a partial range query (Step $2b(i)$), and then a range refinement (Step $2b(ii)$) are carried out. When all rows have been searched, the disk pages corresponding to the last result set are read (Step $2c$). Finally, postprocessing is carried out to eliminate false retrievals (Step $2d$). Note that any of the distance computation techniques available [2, 3, 4, 5, 6, 10, 16, 17, 18, 20, 21, 22] can be used in the postprocessing step.

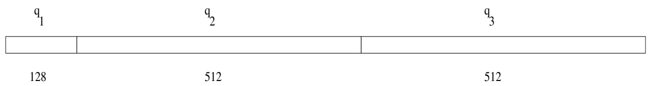

Figure 8. Partitioning for query q**,** $|q| = 1152$

As a consequence of Theorem 2 and Lemma 3 we have the following theorem.

Theorem 4 *The MRS index structure does not incur any false drops.*

We note the following about the search algorithm. 1) For each MBR, the refinement of radius is carried out independently, and proceeds from top to bottom. 2) For each substring, no disk reads are done until the termination

Algorithm $k - NN\_SEARCH(q, k)$

1. $\mathcal{B}$:= The set of k closest MBRs to the query q.
2. $r_1 := max_{B \in \mathcal{B}}\{FD(q, B)\}$.
3. $RANGE\_SEARCH(q, r_1)$
4. Return the k closest strings in the answer set.

Figure 9. k-nearest neighbor algorithm

of the for loop in Step $2b$. Furthermore, the target pages are read in the same order as their location on disk. As a result, the average cost of a page access is much less than the cost of a random page access.

3.5 Nearest neighbor queries

k-nearest neighbor (k-NN) query seeks the k closest substrings from the database to a query string s. We perform a k-NN query in two phases. In the first phase, the k closest MBRs in the index structure are determined by an in-memory search on the index structure. Once the k closest MBRs are determined, the algorithm reads the substrings contained in these MBRs, and finds the k^{th} smallest edit distance of these substrings to the query string. We represent this distance by r_1. Note that, generally a small percentage of the database is processed at this stage. In the second phase, we perform a range query using r_1 as the query radius. Figure 9 presents the complete k-NN search algorithm.

It is guaranteed that the k nearest neighbors are retrieved in the second phase. This can be explained as follows. The edit distance of the query to the actual k^{th} nearest neighbor (say r) is at most r_1, because we search a subset of the whole search space. Let B be an MBR that contains the wavelet coefficients of at least one of the substrings in the actual answer set of the k-NN query, then $FD(q, B) \leq r$. Hence, $FD(q, B) \leq r_1$, and all the MBRs that contain answer strings are retrieved in the second phase.

The substrings retrieved in the first phase exploit a sampling of the strings which are close to the query string. Therefore, one can retrieve any number of MBRs in the first phase as long as they contain more than k substrings. For example, the substrings in the closest MBR is sufficient to prove correctness of the algorithm if the box capacity is at least k. The postprocessing and disk read cost of the first phase decreases if fewer MBRs are retrieved in the first phase. However, as the number of MBRs retrieved in the first phase decreases, the radius of the range query in the second phase increases. Hence, it causes more pages reads and postprocessing in the second phase.

Korn, Sidiropoulos, Faloutsos, Siegel, and Protopapas [15] propose a similar k-NN search. The authors propose a technique in which k closest points are obtained in the first phase using an approximate distance function. The

Dataset	Num A	Num C	Num G	Num T	Num N	Num. Char
chr18	1300930	823103	819387	1282027	30	4225477
chr21	10032226	6921020	6908202	9962534	174	33824156
chr22	8751963	8002860	8000421	8721658	1076929	34553831
chr02	9394422	6251334	6299566	9565401	2266	31512989

Figure 10. Dataset properties.

actual distance to these points is computed, and a range query with the greatest actual distance is performed in the second phase. Seidl and Kriegel [19] propose an optimal iterative k-nearest neighbor search technique. They iterate over both the feature and the object spaces to ensure that no unnecessary objects are accessed. Our algorithm is closer in spirit to the former algorithm except that, we work with MBRs instead of data points in the first phase.

4 Experimental results

We have used four *homo sapiens* chromosome strings in our experiments taken from [1]. These are chromosome 2 (*chr02*), chromosome 18 (*chr18*), chromosome 21 (*chr12*), and chromosome 22 (*chr22*). These chromosome strings are composed of the alphabet $\Sigma = \{A, C, G, T, N\}$. The letter "N" stands for *not known*. We treat the letter N as a different letter, resulting in an alphabet size of 5. The lengths of these strings and the number of occurrences of each letter are presented in Figure 10. The chr18 dataset contains 4M characters, and all the other datasets contain more than 31M characters. The number of A, C, G, and T characters in the chr22 dataset is about equal, while the other datasets contain more A/T than C/G. The percentage of the unknown characters for chr22 is 3%, while it is less than 0.007% for the other datasets.

We implemented single wavelet coefficient and two wavelet coefficient versions of the MRS-index structure for window sizes $w = \{128, 256, 512, 1024\}$. We compared the performance of our technique to the NFA based method proposed by Baeza-Yates and Navarro [5]. This is a recent technique for range queries. This technique sequentially reads the data string and feeds it as an input to an NFA constructed using the query string and the query radius. The NFA goes into the accepting state whenever an answer substring is processed. Our measure of cost is based on the number of disk pages accessed, and the number of string comparisons made later using the accessed disk pages. Since the number of string comparisons needed for a given disk page is constant over all techniques, the total cost is proportional to the number of disk pages read. Note that a sequential scan based technique that searches for all possible substrings has a higher cost than NFA. We assume that the page size is 1K in our experiments.

The rest of this section is as follows. Section 4.1 discusses the effect of box capacity on the performance of queries. Section 4.2 discusses the effect of window size on the performance of queries. Sections 4.3 and 4.4 discusses our results for NN and range queries.

4.1 Effect of box capacity

The first experiment compares the performance of the MRS-index for different box capacities. In this experiment, we perform 100 arbitrary length nearest neighbor queries for box capacities 100, 500, 1000, 2000, 3000, and 4000 for $k = 10$ and 50 on chr18 dataset. The length of the queries varies between 512 and 10000. Figure 11 and 12 plots the cost and the speedup of the MRS-index technique to the NFA technique. Figure 11 corresponds to $k = 10$, and Figure 12 corresponds to $k = 50$. The cost of the MRS-index is much lower than the NFA technique for all these box capacities. The MRS-index runs up to 120 times faster than the NFA technique when the box capacity is 500, and up to 4 when the box capacity is 4000. Although using 2-wavelet coefficient slightly improves the performance for the same box capacity, the size of the index structure is doubled. If we use the same amount of memory, the single coefficient version performs better. Though this experiment shows that the single wavelet coefficient version is better than the two wavelet coefficient version if the index size of the index structure is fixed, we will report the results for both cases for the other experiments for completeness. We also think that better results could be obtained for the two-coefficient version by using more complex distance functions.

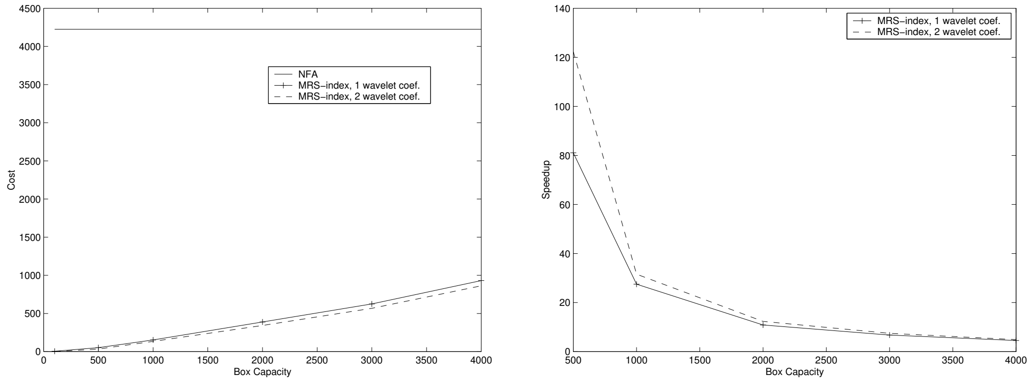

Figure 11. The cost and the speedup of our method with one and two wavelet coefficients versus the NFA technique for different box capacities for 10-NN queries on chromosome_18 dataset.

4.2 Effect of window size

The second experiment reports the impact of the window size (i.e. resolution) on the performance of the MRS-index structure. In this experiment, we use only one row of the index structure. We ran 100 arbitrary length k-NN queries for window sizes 128, 256, 512, 1024 for $k = 10$ and 50 on the chr18 dataset. The queries are

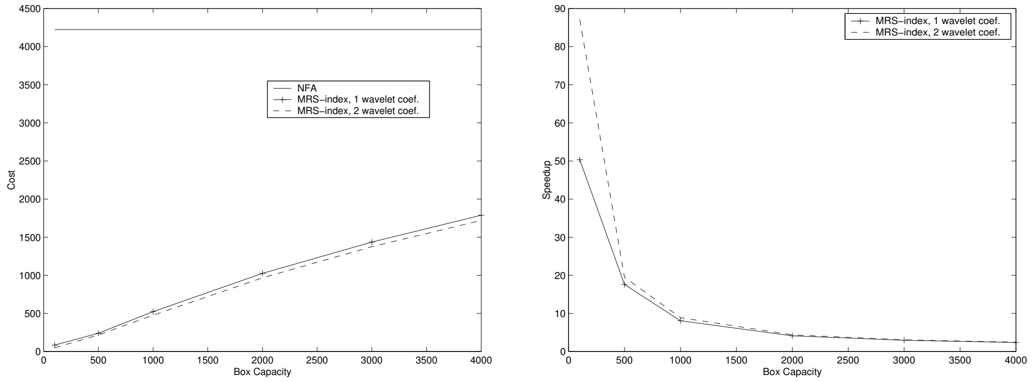

Figure 12. The cost and the speedup of our method with one and two wavelet coefficients versus the NFA technique for different box capacities for 50-NN queries on chromosome_18 dataset.

randomly selected from chr18 dataset. Figures 13 and 14 plot the cost and speedup of the MRS-index structure. The MRS-index structure outperforms the NFA technique for all the window sizes. Furthermore, the performance of the index structure improves as the window size increases. We achieved speedups up to 100 when the window size is 1024.

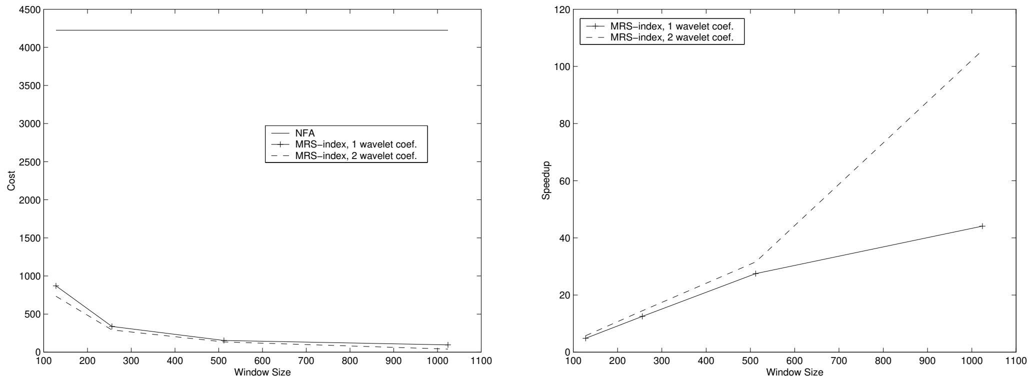

Figure 13. The cost and the speedup of our method with one and two wavelet coefficients versus the NFA technique for different window lengths for 10-NN queries on chromosome_18 dataset.

4.3 Nearest neighbor queries

The third experiment set considers k-NN queries on four datasets (chr02, chr18, chr21, and chr22) for 9 different values of k from 10 to 500 when the box capacity is 1000. In this experiment, we ran 100 random NN queries of arbitrary lengths in the range 512 to 10000 for each value of k. The queries are generated from the datasets.

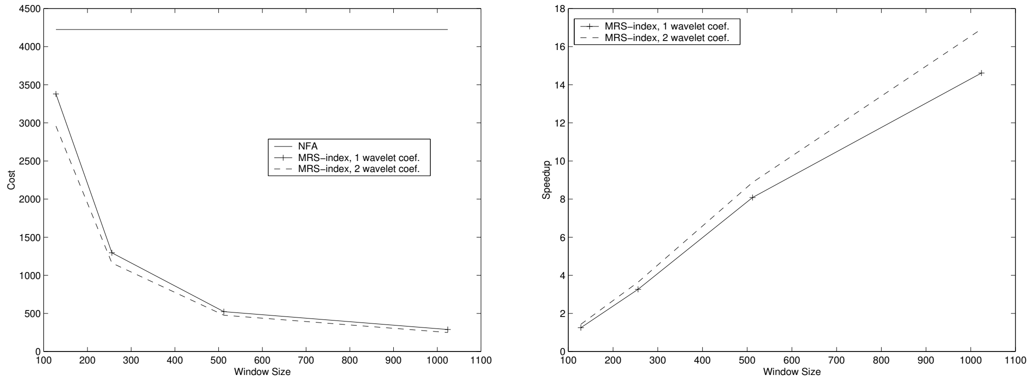

Figure 14. The cost and the speedup of our method with one and two wavelet coefficients versus the NFA technique for different window lengths for 50-NN queries on chromosome_18 dataset.

Figures 15 to 18 present the cost and the speedups of the MRS-index structure. The experimental results for all four data sets are similar. The MRS-index structure outperforms the NFA technique for all values of k. Although the performance of the MRS-index structure drops for large values of k, it still performs better than the NFA technique. We achieved speedups up to 45 for 10 nearest neighbors. The speedup for 200 nearest neighbors is 3. As the number of nearest neighbors increases, the performance of the MRS-index structure approaches to that of the NFA technique.

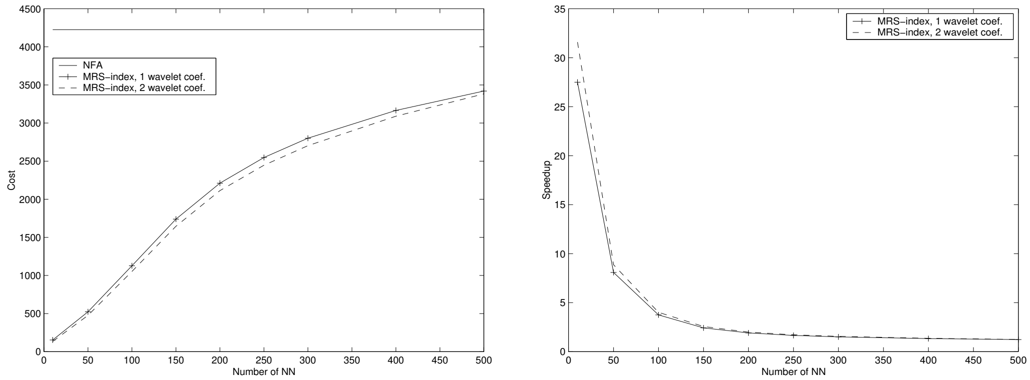

Figure 15. The cost and the speedup of our method with one and two wavelet coefficients versus the NFA technique for different number of nearest neighbors on chr18 dataset.

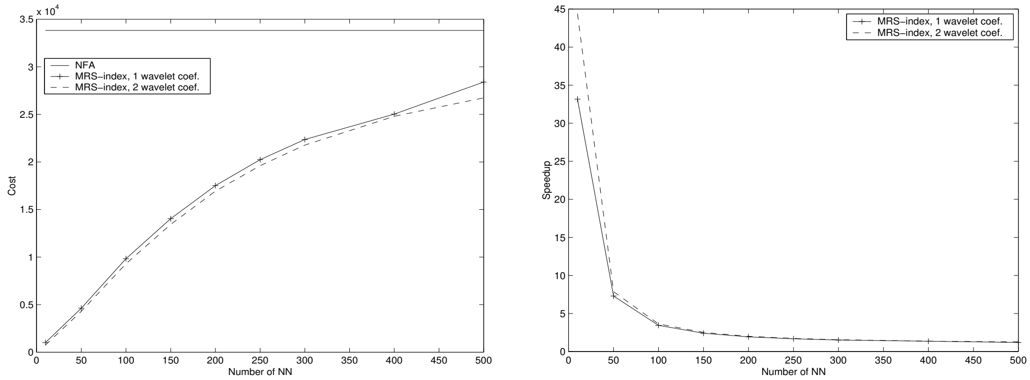

Figure 16. The cost and the speedup of our method with one and two wavelet coefficients versus the NFA technique for different number of nearest neighbors on chr21 dataset.

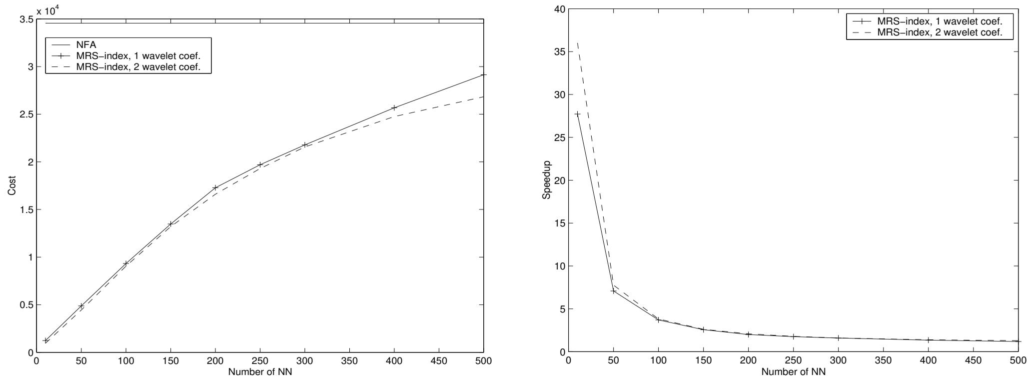

Figure 17. The cost and the speedup of our method with one and two wavelet coefficients versus the NFA technique for different number of nearest neighbors on chr22 dataset.

4.4 Range queries

We considered range queries in the fourth experiment set for $\epsilon = 0.01$, 0.025, 0.05, 0.075, and 0.1 on chr18 and chr22 datasets. We performed 100 arbitrary length queries for each error rate. The box capacity is fixed to 1000 in this experiment. Figures 19 to 22 present the cost and the speedups of the MRS-index structure. Figures 19 and 20 correspond to the case in which the query strings are selected from the same data strings. Figures 21 and 22 correspond to the case in which the query strings are selected from the other data strings. The MRS-index structure performed up to 12 times faster than the NFA technique. The performance of the MRS-index structure improved when the queries are selected from different data strings. This is because the DNA strings have high self similarity. Therefore, if the query string is chosen from the data string itself, the likelihood of having more

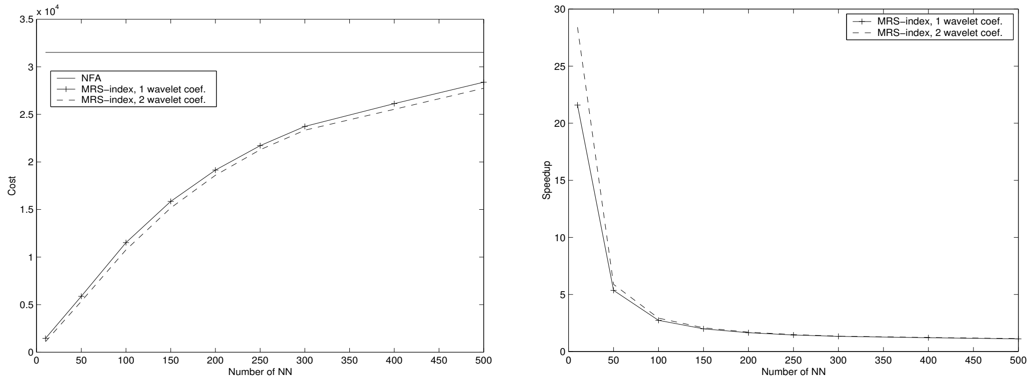

Figure 18. The cost and the speedup of our method with one and two wavelet coefficients versus the NFA technique for different number of nearest neighbors on chr02 dataset.

number of matches within a specified range increases.

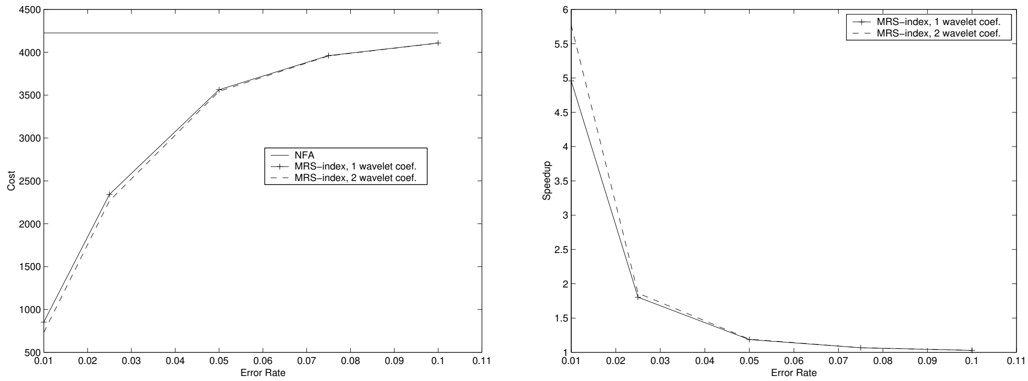

Figure 19. The cost and the speedup of our method with one and two wavelet coefficients versus the NFA technique for range queries of different ranges on chr18 dataset. The query strings are chosen from chr18 dataset.

5 Discussion

In this paper, we considered the problem of searching similar strings in a database consisting of very long strings. The difference between the strings is defined as the minimum number of edit operations (insert, delete, and replace) to transform one string to the other. The existing search techniques are in-memory techniques, and the existing index structures require more space than the database.

We proposed to transform the strings to integer space by mapping them to their frequency vectors. Later, we

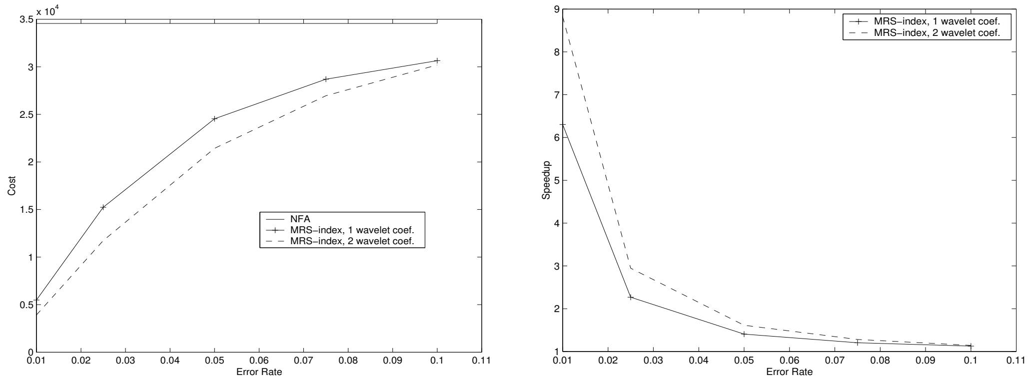

Figure 20. The cost and the speedup of our method with one and two wavelet coefficients versus the NFA technique for range queries of different ranges on chr22 dataset. The query strings are chosen from chr22 dataset.

generalized this idea to map the strings to their local frequencies for different resolutions by using wavelet transformation. We presented an efficient algorithm to find a lower bound on the distance of the wavelet coefficients of the strings.

We adapted the MR-index [14] to cluster the wavelet coefficients of the string. We called this index structure the MRS-index structure. This index structure slides a window on the data string for different resolutions. For each resolution, the index structure clusters the wavelet coefficients of a fixed number (box capacity) of consecutive substrings in an MBR. The MRS-index structure is dynamic, and allows arbitrary length queries.

We presented algorithms for both range queries and nearest neighbor queries. The range query algorithm splits the query into subqueries of available resolutions and performs successive range reduction for each subquery without using the data strings. The k-nearest neighbor algorithm runs in two phases. In the first phase, the distance of the query to the MBRs are computed, and the distance to the k^{th} closest MBR is used as the radius for a range query in the second phase. This technique can be used as a preprocessing to speed up any string search technique including BLAST by pruning large amounts of the data strings. Note that BLAST itself is an in-memory algorithm since its index size is more than the database size.

According to our experimental results with four different human chromosomes, our method runs up to 45 times faster than the NFA technique for 10 nearest neighbor queries. The MRS-index structure works efficiently for up to 200 nearest neighbors. Similarly, the MRS-index structure runs efficiently for range queries if the error rate is less than 0.1. The MRS-index structure performs up to 12 times faster than other techniques for range queries with error rate 0.01.

In the future, we would like to model other distance measures in which different character pairs have different

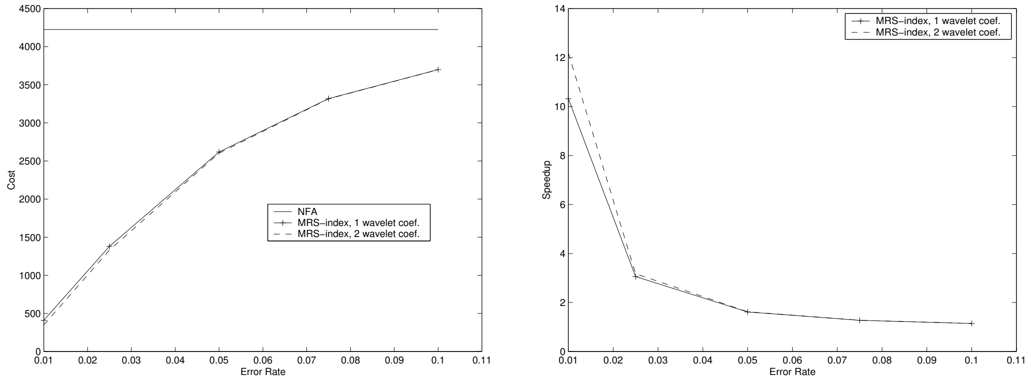

Figure 21. The cost and the speedup of our method with one and two wavelet coefficients versus the NFA technique for range queries of different ranges on chr18 dataset. The query strings are chosen from chr22 dataset.

costs, or where the gaps are affine. Finding high local similarities within the substrings of the query string to the data strings is also an interesting problem. Comparison of the quality of our results with existing software packages is also planned for future. We also would like to test the performance of the MRS-index structure on the event sequences.

References

[1] www.ncbi.nlm.nih.gov.

[2] S. Altschul and B. W. Erickson. Optimal sequence alignment using affine gap costs. *J. Mol. Biol.*, 1986.

[3] S. Altschul and W. Gish. Basic local alignment search tool. *J. Molecular Biology*, 1990.

[4] R. A. Baeza-Yates and G. Navarro. A practical index for text retrieval allowing errors. In *CLEI*, volume 1, pages 273–282, November 1997.

[5] R. A. Baeza-Yates and G. Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999.

[6] R. A. Baeza-Yates and C. H. Perleberg. Fast and practical approximate string matching. In *Combinatorial Pattern Matching, Third Annual Symposium*, pages 185–192, 1992.

[7] D.A. Benson, M.S. Boguski, D.J. Lipman, J. Ostell, and B.F. Ouellette. Genban. *Nucleic Acids Research*, 26(1):1–7, 1998.

[8] D.A. Benson, I. Karsch-Mizrachi, D.J. Lipman, J. Ostell, B.A. Rapp, and D.L. Wheeler. Genbank. *Nucleic Acids Research*, January 2000.

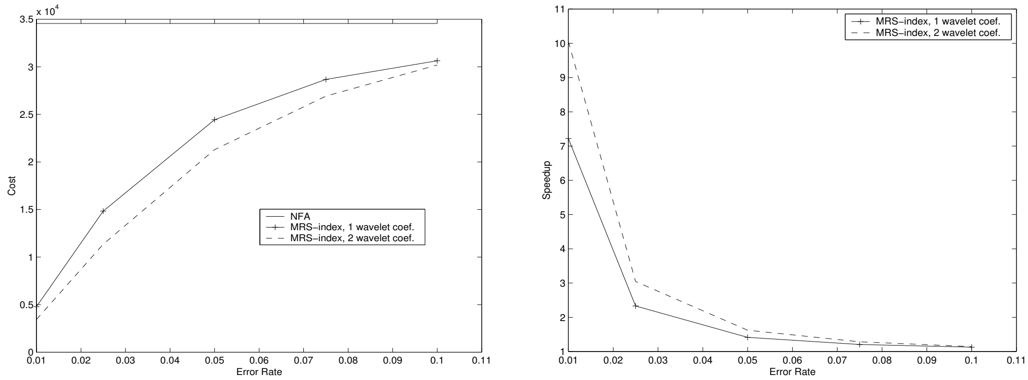

Figure 22. The cost and the speedup of our method with one and two wavelet coefficients versus the NFA technique for range queries of different ranges on chr22 dataset. The query strings are chosen from chr18 dataset.

[9] E. Giladi, M. G. Walker, J. Z. Wang, and W. Volkmuth. Sst: An algorithm for searching sequence databases in time proportional to the logarithm of the database size. In *RECOMB*, Japan, 2000.

[10] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1 edition, January 1997.

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

[12] S. Henikoff and J. G. Henikoff. Amino acid substitution matrices from protein blocks. In *PNAS*, pages 10915–10919, 1989.

[13] H. V. Jagadish, Nick Koudas, and Divesh Srivastava. On effective multi-dimensional indexing for strings. In *ACM SIGMOD*, 2000.

[14] T. Kahveci and A. Singh. Variable length queries for time series data. In *ICDE*, Heidelberg, Germany, 2000.

[15] F. Korn, N. Sidiropoulos, C. Faloutsos, E. Siegel, and Z. Protopapas. Fast nearest neighbor search in medical databases. In *VLDB*, pages 215–226, India, 1996.

[16] U. Manber and E. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[17] E. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, pages 251–266, 1986.

[18] E. Myers. A sublinear algorithm for approximate keyword matching. *Algorithmica*, pages 345–374, 1994.

[19] T. Seidl and H.P. Kriegel. Optimal multi-step k-nearest neighbor search. In *SIGMOD*, 1998.

[20] T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, March 1981.

[21] S. Uliel, A. Fliess, A. Amir, and R. Unger. A simple algorithm for detecting circular permutations in proteins. *Bioinformatics*, 15(11):930–936, 1999.

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