Distance Based Indexing for String Proximity Search

S. Cenk Sahinalp[+†*], Murat Tasan[*+], Jai Macker[*], Z. Meral Ozsoyoglu[*+]

Abstract

In many database applications involving string data, it is common to have near neighbor queries (asking for strings that are similar to a query string) or nearest neighbor queries (asking for strings that are most similar to a query string). The similarity between strings is defined in terms of a distance function determined by the application domain. The most popular string distance measures are based on (a weighted) count of (i) character edit or (ii) block edit operations to transform one string into the other. Examples include the Levenshtein edit distance and the recently introduced compression distance.

The main goal in this paper is to develop efficient near(est) neighbor search tools that work for both character and block edit distances. Our premise is that distance-based indexing methods, which are originally designed for metric distances can be modified for string distance measures, provided that they form almost metrics. We show that several distance measures, such as the compression distance and weighted character edit distance are almost metrics. In order to analyze the performance of distance based indexing methods (in particular VP trees) for strings, we then develop a model based on distribution of pairwise distances. Based on this model we show how to modify VP trees to improve their performance on string data, providing tradeoffs between search time and space. We test our theoretical results on synthetic data sets and protein strings.

1 Introduction

In many database applications, including those in computational genomics and proteomics, text and audio processing and computational finance, it is common to have proximity queries such as asking for strings that are *similar* to a query string (near neighbor search), or are *most similar* to a query string (nearest neighbor search). The similarity between strings is defined in terms of a distance function determined by the application domain. The most popular string distance measures are based on (a weighted) count of (i) character edit or (ii) block edit operations to transform one string into the other.

Character edit distances, such as the (unweighted) Levenshtein edit distance [34] or its weighted version have been used for decades for measuring the functional and evolutionary similarity of DNA and protein strings [41, 46].

Block edit distances, such as the transformation distance of Varre et. al. [49] and its closely related companion, the compression distance of Li et. al. [37, 11, 38] are recent alternatives to the character edit distances. Such distances can be described in terms of the minimum number of single character and block edit operations to transform one string into another. Block edit distances, in particular the compression distance, provide practical upper bounds to the *algorithmic complexity* among strings (i.e. the Kolmogorov complexity of a string when an external string is available for support [37, 38]). Although recently introduced, the compression distance attracted considerable attention from general scientific community [40] due to its demonstrated success in capturing the evolutionary relationships between world languages (through comparing "declaration of human rights" in various languages), [11], DNA and RNA sequences[1], as well as identification of authorship for works of literature [11].

Our aim in this paper is to develop efficient near(est) neighbor search tools that work for both character and block edit distances.

Background. Efficient indexing methods compute the near neighbors of a query item by iteratively pruning subsets of potential answers to a query (desirably a large fraction of them) by partitioning the data set and checking out to which partition the query belongs. For vector spaces (where dimensions are well defined), this pruning can be done by focusing on a small number of dimensions (preferably one) at a time. Spatial in-

[*]Department of EECS,[†]Department of Genetics, [+]Center for Computational Genomics, Case Western Reserve University. This research is supported in part by NSF Awards CCR-0133791 and IIS-0209145, and a grant from the Charles B. Wang Foundation.

[1]Known mitochondrial DNA sequences of various species [37] and families of retrotransposons [49].

dex structures such as R-trees [26], R+-trees [44], R*-trees [8], quad-trees [21], X-trees [10], SR-trees [30], A-trees [43], and others are examples that employ this general strategy. These data structures work quite well for vector spaces with small dimensionality; however as the number of dimensions grow, their performance drops significantly, possibly to the level of the brute-force search [18, 12, 51].

To overcome the curse of dimensionality in vector spaces, several dimension reduction techniques, such as those based on random projections, sampling, etc., have been demonstrated to be effective [1, 17, 22, 27, 33]. For example, one can estimate the Hamming distance between two data items within some $(1+\epsilon)$ approximation factor by comparing a small random sample of their attributes (i.e. dimensions). Clearly this general approach requires that the data items have well defined dimensions, i.e., a priori knowledge of which attribute of one data point is compared with which attribute of another. Strings under edit distances have no clear notion of a dimension: even a single character insertion or deletion may alter which attribute (i.e. character/block) of one string needs to be aligned and compared to which attribute of the other string. Thus these approaches are not directly applicable to string databases.

Dimension reduction techniques for strings under various edit distances rather "reduce" a long query string to one or more of its substrings (usually called q-grams) [23, 31, 6, 5] and identify all strings in the database which also include such substring (i.e. q-grams). Many of these methods are designed for *substring* similarity search problems in which the goal is to find the most similar substrings of a long text string to a query string. When applied to similarity search problems involving *full strings* (which are the main focus of our work), these methods usually provide a tradeoff between the false positive rate and the false negative rate by adjusting the length of the q-grams searched. With proper calibration, such methods can prune out $\geq 50\%$ of the strings in certain data sets, however, for many other data sets their performance drops to the level of brute force search.

An alternative method for dimension reduction for strings was presented in [39] which provides an involved mapping of strings under certain block edit distances (such as the transformation distance [49]) to binary vectors under the Hamming distance. Although the number of dimensions in these binary vectors are high, they can later be decreased substantially through random projections [17, 22, 27, 33]. Unfortunately this approach imposes an approximation factor of $O(\log n \log^* n)$ that must be tolerated, which makes it unsuitable for several applications. Moreover, this general approach is not appropriate for any character edit distance.

Contributions. Our main premise is that string similarity search can be performed efficiently through distance-based indexing [13, 48, 16] . Originally designed for metric spaces, vantage point trees (VP-trees) [52], their variant MVP-trees [14], and M-trees [20] are examples of distance-based indexing methods. Our focus in this paper is on VP trees which iteratively partition a given data set by defining "spheres" around select "vantage" points in the input space. Pruning is achieved if the "query sphere" falls completely within or outside of the vantage point's sphere; this is possible due to triangular inequality, a necessary condition for a space to be metric.

Unfortunately many string distance measures do not satisfy the triangular inequality; examples include the weighted character edit distance [41, 46] and the compression distance [37, 11, 38]. In order to use distance based indexing techniques under such measures we employ the following strategy.

(1) We first show that the compression distance and the weighted character edit distance are *almost metrics*, a new notion we introduce in this paper to describe distance measures which are *reflexive*, *symmetric*, and *satisfy the triangular inequality within a constant factor*.

(2) We then show how to update pruning conditions on distance based indexing methods if the distance measure used is an almost metric. The constant factor in the triangular inequality satisfied by the distance function has an important role in the performance of this modified index.

(3) In high dimensional spaces, many indexing methods perform poorly if the data is uniformly distributed over the input space [9, 12]; however not much is known for other data distributions. However we observe that many applications involve string data with high levels of clustering. Our examples include all known protein strings that are active in the human brain and other organisms, and declaration of human rights in 52 Euro-Asian languages. For both data sets we plot $f(r)$, the number of pairs of strings whose distance is less than some radius r against r to observe that the first data set exhibits a polynomial distribution (a power law) whereas the second exhibits an exponential distribution. We show how to exploit this bias to improve the performance of distance based indexing methods (in particular VP-trees) by providing a tradeoff between query time and space for both polynomial and exponential distributions.

(4) We verify our theoretical results on near(est) neighbor search via updated VP-trees using various data sets under several distance functions. We first test our implementation on synthetic data that enables us to bet-

ter understand its performance dependency to specific parameters of the data set. We also test it on a number of data sets (e.g. the complete human proteome) and obtain good performance; for example we achieve $\geq 90\%$ pruning in nearest neighbor search among proteins that are active in brain.

Notation. Throughout the paper S, Q, R, T denote strings from an arbitrary alphabet; $S[i]$ denotes the ith character of string S and $S[i:j]$ the substring between the ith and j^{th} characters (inclusive) of S. The length of the string S is denoted by $|S|$. Given two strings S and R, we denote by SR the concatenation of S and R, and by S^i the concatenation of i copies of S.

2 String Similarity Measures

Given two strings R and S, one can describe a *transformation* of one string into the other through a set of specified edit operations. Possible edit operations that involve single characters are *character insertion*, *character deletion* and *character replacement*. One may also consider block (i.e. substring) edits such as: *block copying*, *block deletion* and *block relocation*. Each of these edit operations may have a specific *cost*, thus given a transformation from R to S, the *distance* $d(R \to S)$ from R to S can be defined as the minimum total cost of edit operations to transform R to S. Such a distance $d(.,.)$ is not necessarily symmetric, i.e., there may be strings S and R for which $d(R \to S) \neq d(S \to R)$ (consider, for example, transforming an empty string to a long string, and then consider the reverse transformation). Given a transformation, the *symmetric version* of the distance $d()$ implied by this transformation can be defined as $d(S,R) = \frac{1}{2} \cdot [d(S \to R) + d(R \to S)] = d(R,S)$.

It is possible to classify transformations from R into S and the distances between them according to the *source* and the *destination* of edit operations permitted.

External transformations iteratively construct S from an initially empty string S' by performing edit operations whose source is R and whose destination is S'; thus R does not change during the transformation.

Example 1 *Consider an external transformation that allows only block copy operations. One can transform $R = A,C,T,G,A,T,G$ to $S = A,T,C,T,G,T,G,A$ via 3 block copies starting with an empty S: (1) copy block A,T from R to the right end of S to obtain $S = A,T$, (2) copy block C,T,G from R to obtain $S = A,T,C,T,G$, (3) copy block T,G,A to obtain $S = A,T,C,T,G,T,G,A$.*

Internal transformations construct S by iteratively applying edit operations to string S' which is initially set to R.

Example 2 *Consider an internal transformation that allows block deletions, relocations and copies. One can transform $R = A,C,T,G,A,T,G$ to $S = A,T,C,T,G,T,G,A$ via 2 such block operations starting with $S = R = A,C,T,G,A,T,G$, (1) move the rightmost A to the right end of S to obtain $S = A,C,T,G,T,G,A$, (2) copy T to second position in S to obtain $S = A,T,C,T,G,T,G,A$.*

An orthogonal classification of transformations (and corresponding distances) between R and S is according to the *restrictions on the destinations* of the edit operations allowed.

Marginal transformations allow edit operations whose destinations are either a prefix or a suffix of the string that evolves into S.

Unrestricted transformations allow edit operations whose destinations can be any arbitrary location on the string that evolves into S.

Commonly Used String Distance Measures. Arguably the most commonly used similarity measure for strings is the character edit distance, which is also referred as the Levenshtein edit distance or simply the edit distance [34]. It can be defined as the minimum number of single character insertions, deletions and replacements needed to transform one string into another. In terms of the notions introduced above, this transformation is *internal* and *unrestricted*. Weighted versions of the character edit distance assign costs to specific operations to specific characters.

More recently block edit operations and distances based on these operations have received considerable attention [40] especially in the context of evolutionary analysis of world languages and genome strings (e.g. mitochondrial DNA) from various species. Given two strings R and S, the *transformation distance* [49] is defined to be the minimum number of block relocations, copies and deletions as well as single character insertions, deletions and replacements to transform one string to another in the *internal*, *unrestricted* model. Here, string R is transformed into S by assigning $S' = R$ and applying available edit operations directly on S' to transform it to S. This distance is identical to the *the block edit distance* of [39]; because of its generality, it provides a lower bound to any distance based on block edits.

Although the transformation distance (a.k.a. the block edit distance) includes all edit operations responsible from genome sequence or natural language evolution, it is NP-hard to compute [36]. For heuristically approximating the transformation distance [49] it is possible to put a restriction on the edit operations by employ-

ing the *external*, 1-sided *marginal* model. In other words one can start with an empty string S' and transform it to S by (externally) copying blocks of R or inserting single characters to the right end of S'. The number of such edit operations performed gives an upper bound on the transformation distance.

Such limitations on the transformation distance can be relaxed by allowing internal copying of blocks of S' to its right end as well. We will call the resulting distance as the *compression distance*. Introduced in [37, 11, 38], the compression distance basically measures the number of Lempel-Ziv-77 phrases obtained during the compression of a string S when another string R is used as part of the dictionary. By the use of suffix trees the compression distance can be computed in time linear with the lengths of the strings [42].

Summary of Our Results. Although the character edit distance and the transformation distance are metric distances, their more useful siblings, the weighted character edit distance and the compression distance are not. In this section we state that the compression distance and the weighted edit distance[2] are *almost metrics*, i.e. they are symmetric, reflexive and satisfy the triangular inequality within a constant factor. Note that due to space constraints we leave most of the proofs to the full version of the paper.

2.1 Compression Distance is an Almost Metric

Compression distance as used in [37, 11, 38] can be defined in our framework as follows.

Definition 1 *Consider the following internal transformation from R to S and its associated symmetric distance $c(R, S)$, called the compression transformation and the compression distance respectively: R is transformed into S by starting from $R' = R$ and iteratively copying any substring of R' or inserting any single character to its right end. As a final operation, deletion of the original version of R from the left end of R' is permitted.*

The following lemma states that the greedy version of the compression distance is equal to the compression distance, simplifying the job of computing the compression distance.

Lemma 1 *Consider the greedy version of the compression transformation between R and S, where the block copy operations are performed greedily as per the gem_1 transformation. Let $gc(R, S)$ be the symmetric distance associated with greedy compression transformation. Then $gc(R, S) = c(R, S)$.*

[2]under certain conditions on operation costs

Example 3 *Consider the strings $R = A, C, T, A, G, T, A, T$, and $S = A, G, T, C, T, A, A, T$ again. One can compute the $c(R \to S)$ as follows. We start with $S' = R = A, C, T, A, G, T, A, T$. The longest prefix of S which exists as a substring in S' is $S'[4 : 6] = A, G, T$; we update S' to $S' = A, C, T, A, G, T, A, T, \mathbf{A}, \mathbf{G}, \mathbf{T}$. In the remainder of S, which is $S[4 : 8]$, the longest prefix which exists in S' is $S'[2 : 4] = C, T, A,$; we update S' to $S' = A, C, T, A, G, T, A, T, \mathbf{A}, \mathbf{G}, \mathbf{T}, \mathbf{C}, \mathbf{T}, \mathbf{A}$. We finally observe that the remaining of S exists as $S'[7 : 8] = A, T$, and thus update $S' = A, C, T, A, G, T, A, T, \mathbf{A}, \mathbf{G}, \mathbf{T}, \mathbf{C}, \mathbf{T}, \mathbf{A}, \mathbf{A}, \mathbf{T}$. The final operation is the deletion of the original version of R as a prefix of S', leaving us with $S' = S = A, G, T, C, T, A, A, T$. This implies that $c(R \to S) = 3$. The reader can verify that $c(S \to R) = 4$, thus $c(S, R) = c(R, S) = 7/2$. Note that for this example $c(R, S) = gem_1(R, S)$, which is not the case for all R and S.*

Lemma 2 *The compression distance $c(R, S)$ between two strings R and S can be computed in optimal $O(|R| + |S|)$ time.*

The lemma follows from [42] on-line suffix tree construction.

Although compression distance is efficiently computable, it does not satisfy the triangular inequality and hence is not a metric.

Example 4 *Consider $R = v, w, x, y, z$, $Q = z, y, x, w, v$ and $S = w, v, x, w, y, x, z, y, y, x, w, x, w, v, z, y, x, z, y, x, w, y, x, w, v, z, y, x, w, v$. One can verify that $c(R \to Q) = 6 = c(Q \to R)$, $c(Q \to S) = 10$, $c(S \to Q) = 2$, $c(R \to S) = 19$ and $c(S \to R) = 6$. Thus $c(R, S) = \frac{6+19}{2} = 12.5 \geq c(R, Q) + c(Q, S) = \frac{6+6}{2} + \frac{10+2}{2} = 12$.*

The example demonstrates that in order to construct a string S from substrings of another string R, it may be helpful to construct an intermediate string Q. Because compression transformation does not allow deletion of a substring except R itself, Q can not be incorporated in the construction. It is possible to relax this deletion constraint to obtain a metric distance. Such a relaxed version approximates the compression distance within a constant factor, implying that the compression distance is an *almost metric*.

Definition 2 *Consider relaxing the compression transformation from R to S by allowing deletion of any prefix of R' as a final operation. We call this transformation, the relaxed compression transformation and its associated distance the relaxed compression distance, denoted $rc(R, S)$.*

Lemma 3 *The relaxed compression distance, $rc(R,S)$, is a metric.*

Unfortunately the relaxation makes the distance much harder to compute.

Lemma 4 *Computing $rc(R \to S)$, the relaxed compression distance between R and S, is NP-hard.*

Nevertheless, the compression distance (which is easy to compute) provides a constant factor approximation to the relaxed compression distance. Because relaxed compression distance is a metric, the compression distance is an *almost metric*, a notion we introduce in this paper.

Definition 3 *A distance function f provides an almost metric for space S, if it is symmetric, reflexive and satisfies the triangular inequality within a constant factor k; i.e. for all $S, R, Q \in$S, $f(S,R) \leq k \cdot [f(S,Q) + f(Q,R)]$.*

Now we state that the compression distance provides a constant approximation to its relaxed version, which proves that the compression distance is an almost metric.

Theorem 5 $rc(R,S) \leq c(R,S) \leq 3 \cdot rc(R,S)$

2.2 The Weighted Edit Distance is an Almost Metric

Weighted edit distances are commonly used for protein and genome string comparisons. Here each operation has a certain cost, determined in general by the log of the probability of that edit operation occurring in one of the two copies of a string during its evolution. Let h be the highest cost of any operation among the ones allowed and l be the lowest. Then h times the character edit distance between any two strings R and S will provide an upper bound on their weighted edit distance. Similarly l times the edit distance will provide a lower bound, which implies that the weighted edit distance satisfies the triangular inequality within a factor of h/l.

3 String Proximity Search with Distance Based Indexing

There are a number of distance based indexing structures in the literature including [48, 52, 19, 16, 14, 20, 53], which are potentially applicable to string proximity search. In this section we show how to modify one of the distance based indexing method, the VP trees, to *almost metrics*, and in particular to the compression distance. Note that it may be possible to modify other distance based techniques for almost metrics as well. However our goal is to simply show that being almost metric is a property for a distance measure that can be employed to perform efficient proximity search.

For completeness, we briefly describe the main idea behind distance based index structures and VP trees below. We later show how these structures can be generalized so as to perform efficient search under almost metrics, with VP and MVP trees being good representatives of purely distance based index structures.

Distance Based Indexing. The general task of similarity search is a well studied problem in database research. The types of queries can vary, but most can be reduced to one of the two basic forms: the near neighbor query (i.e. range query), and the nearest neighbor query.

More formally, a near neighbor query on a given set of data elements $X = \{x_1, x_2, \ldots, x_n\}$ asks to retrieve all data elements that are within some specified distance r of a given query point q; i.e. the task is to return $X' = \{x_i | x_i \in X \& d(x_i, q) \leq r\}$. The nearest neighbor query, on the other hand, asks for the closest element in the data set to the query element. Other types of queries include k-nearest neighbors, k-farthest neighbors, etc.

Fundamental to the similarity search problem is the nature of the search space and the distance function. When the distance function and search space can convey spatial information (as is the case of Euclidean distances), a wide variety of spatial index structures can be used to efficiently answer the queries. These methods include R-trees [26], R+-trees [44], R*-trees [8], quadtrees [21], X-trees [10], SR-trees [30], A-trees [43], and others. Such index structures take advantage of the fact that each data element has a well-defined location in the search space. The distances of these elements to arbitrary points in the search space can be computed, and using this property the search space is often broken up hierarchically into partitions that can be later pruned by exploiting properties of the particular nature of the partitioning method. Because of the complexity of space partitioning in multi-dimensional spaces, such methods are particularly useful for spaces with small dimensionality.

Distance-based index structures are inherently different than these spatial index structures [13, 48, 16]. Here, only the relative distances of data elements are needed for index construction and search, i.e., no spatial information on the data elements are utilized. The key principle is again eliminating a subset of the search space (through hierarchical partitioning) that can be proven to not contain any of the data points in the answer of the query. This pruning is often done using the triangle inequality, a feature of metric spaces, which provides a necessary condition on the search space. Vantage point

trees (VP-trees) [52], their variant MVP-trees [14], and M-trees [20] are examples of distance-based indexing. Because dimensionality is not a key issue for the pruning technique they employ, such distance based methods are potentially more effective for high dimensional data than the above alternatives.

For certain types of data it is possible to obtain specialized indices [23, 2], optimized (and often designed exclusively) for operation under a unique distance. Unfortunately, not all distance measures provide such specialized structure as per our problem. In particular, when insertion and deletion operations are allowed, strings can not be treated as vectors on which distinct dimensions can be compared. Thus we are confined to methods that use distances between pairs of strings as the primary source of information.

Our primary difficulty is that available distance based methods require the distance measure used to be a metric. In the following sections, we show how the VP and MVP trees can be modified so that it enables efficient proximity search even under almost metrics.

VP trees in a nutshell. Vantage point trees were first introduced by [48], and have subsequently been extended into MVP (multi vantage point) trees [14]. In their most basic form, vantage point trees are binary trees that recursively partition a data set. More specifically the structure of a binary VP-tree can be described as follows. Each internal node is of the form $(X_v, M, Rptr, Lptr)$, where X_v is the vantage point, M is the median distance among the distances of all the points (from X_v) indexed below that node, and $Rptr$ and $Lptr$ are pointers to the right and left branches. Left branch of the node indexes the points whose distances from S_v are less than or equal to M, and right branch of the node indexes the points whose distances from X_v are greater than or equal to M. In leaf nodes, instead of the pointers to the left and right branches, references to the data points are kept.

Construction of VP-trees. Given a finite set $X = \{X_1, X_2, \ldots, X_n\}$ of n elements, and a metric distance function $d(X_i, X_j)$, a binary VP-tree V on X is constructed as follows.
(1) If $|X| = 0$, then create an empty tree.
(2) Else, let X_v be an arbitrary element from X. (X_v is the vantage point.) Also let M be the median of $\{d(X_i, X_v) | \forall Xi \in X\}$; let $X_l = \{X_i | d(X_i, X_v) \leq M, X_i \in X, X_i \neq X_v\}$ and let $X_r = \{X_i | d(X_i, X_v) \geq M, X_i \in X\}$. (Note that the cardinality of X_l and X_r are equal.) Recursively create VP-trees on X_l and on X_r as the left and right branches of the root of V.

The binary VP-tree is balanced and therefore can be easily paged for storage in secondary memory [52]. The construction can be done by performing $O(n \log n)$ pairwise distance computations.

Search in VP-trees. For a given query element Q, the set of data elements that are within distance r of Q are found using the search algorithm below.
(1) If $d(Q, X_v) \leq r$, then X_v (the vantage point at the root) is in the answer set.
(2) If $d(Q, X_v) + r \geq M$, then recursively search the right branch.
(3) If $d(Q, X_v) - r \leq M$, then recursively search the left branch.
Note that both branches can be searched if both search conditions are satisfied.

The correctness of this simple search strategy can be proven easily by using the triangle inequality of distances among any three elements in a metric data space. Multi-vantage-point trees (MVP-trees) have been developed to increase the effectiveness of searches. In an MVP tree, two vantage points (X_{v1} and X_{v2}) are created at each new iteration of its construction. The search space at a given step is divided first using X_{v1}, then each of the subsections are divided again, all using X_{v2}. This basically corresponds to collapsing two levels of a typical vantage-point tree into one, and reduces the number of vantage points needed (since multiple branches share a common vantage point). All variants of the VP-tree still use the basic principal of triangular inequality in their searches, so the criterion for eliminating any particular branch during the search is fundamentally the same and can be easily generalized to each form of VP-tree. We will thus continue to use simple binary vp-trees in our algorithm examples as they are more readily clear to the reader; we use multi-vantage-point trees in our experimental results.

3.1 Updating VP/MVP trees for almost metrics

In this section we show how to update VP (and MVP) trees so that they could be effectively used in proximity search in almost-metric spaces. As described earlier for VP trees, portions of the search space can be eliminated when certain conditions (the distance from the query element to the vantage point and the radius of the query) hold. This elimination is the primary goal and measure of effectiveness of the vantage point tree, as each branch eliminated from search reduces the number of the candidates for answering the query. Using the definition of an almost metric, we now describe how we can eliminate branches in the binary vantage point tree search.

Let Q be the query element, r be the query range, X_v be the vantage point accessed during the search, and M be the median distance value for X_v. Given an almost metric distance $d(.,.)$ which satisfies the triangular inequality within a factor of 3, we first show that if

$d(X_v, Q) + r < M/3$ then we do not have to search the right branch. Then, we show that if $d(X_v, Q) - 3r > 3M$ then we do not have to search the left branch.

Let Z denote any data element indexed in the right branch, and Y any data element indexed in the left branch.

(1a) $d(Z, X_v) \geq M$, (1b) $M/3 > d(X_v, Q) + r$ (hypothesis), (1c) $d(X_v, Q) + d(Q, X) \geq d(X_v, Z)/3$ (triangular inequality), (1d) $d(Q, Z) > r$ (sum up 1a, 1b, 1c).

(2a) $d(Y, X_v) \leq M$, (2b) $3M < d(X_v, Q) - 3r$ (hypothesis), (2c) $d(X_v, Y) + d(Y, Q) >= d(X_v, Q)/3$ (triangular inequality), (2d) $d(Y, Q) > r$ (sum up 2a, 2b, 2c).

One method for performing a near/nearest neighbor search is to pick a "good" value of r (using a heuristic method) to reduce the search space and then perform each actual distance computation in the reduced r-radius set to pick the correct data point. As shown below, vantage point trees can also be used to help reduce the final search space without the need to pick a radius r. In this case, only the right branches of the search can be eliminated.

We now show that if $d(X_v, Q) < M/6$, the right branch can be eliminated from the search. Let Z again denote any data element indexed in the right branch:

(3a) $M/6 > d(X_v, Q)$ (hypothesis), (3b) $d(Z, Q) > d(X_v, Q)$ (sum up 1a, 3a, and 1c).

4 Properties of String Spaces for Proximity Search Applications

There are two important issues that determine whether proximity searches could be performed efficiently by distance based indexing: (1) The distance measure used has to be a metric or an almost metric - this issue was addressed in the earlier sections of the paper. (2) The distribution of data points should be "suitable" for distance based indexing - which is the focus of this section.

Earlier work by [9, 12] consider especially metric spaces where data is distributed uniformly and identically over the input space. They show that for any given point p, as the dimensionality increases the difference between the nearest and the farthest neighbors of a point diminishes. This suggests that nearest neighbor searches in high dimensional spaces may not give meaningful answers for uniformly distributed data. However, to the best of our knowledge, biased/clustered distributions of data have never been investigated under efficiency considerations for distance based indexing.

Our main concern is to understand how typical string data from applications in computational linguistics and computational proteomics is distributed over the input space. Our data sets include: (1) Declaration of human rights in 52 Eurasian languages. This is the same data set used by [37, 11] for demonstrating the power of compression distance. (2) The complete set of protein strings that are known to be active in the brain cells of humans and other organisms from the SwissProt database (93 proteins). Given these data sets we ask a natural question: how does $f(r)$, the number of pairs of strings whose distance is at most r, behave against r. Our tests give the following answers:

1. The linguistic data under compression distance seems to be exponentially distributed over the input space. More specifically the function $f(r)$ seems to satisfy $f(r) = k \cdot c^r$ for some constants c and k as the plot of $\log f(r)$ against r is quite close to a straight line. Clearly $\log f(r) = \log k \cdot c^r$ gives $\log f(r) = \log k + r \cdot \log c$ which enables us to compute constants c and k to be approximately $2^{1/400}$ and $2^{-2.2}$ respectively. Considering that the number of characters (and hence the potential dimensions) in each string is in the order of a few thousands these constants are very small and indicate some clustering (non-uniformity) in the data set. This does not come as a surprise as these languages are evolutionarily related to each other [11].

2. The protein data under both the character edit distance and the compression distance gives a polynomial distribution over the input space (i.e. satisfies a power law). In other words the function $f(r)$ seems to satisfy $f(r) = k \cdot r^c$ for some constants c and k: this can be observed through the log-log plots of $f(r)$ against r which are quite close to straight lines. If $\log f(r) = \log(k \cdot r^c)$, then $\log f(r) = \log k + c \cdot \log r$. The constants k and c vary only slightly among the two distances; they imply some significant non-uniformity in the data set. Again this should not come as a surprise as these proteins are evolutionarily related.

4.1 Performance of VP Trees Under Polynomial and Exponential Pairwise Distance Distributions

In this section we develop a model for analyzing the performance of the VP trees under polynomial and exponential pairwise distance distributions. We assume that: (1) the distribution of the distances between a typical data point to other points in the data set resembles the overall pairwise distance distribution, and (2) the distribution of query points in the input space resemble the distribution of the data points. Based on our model it

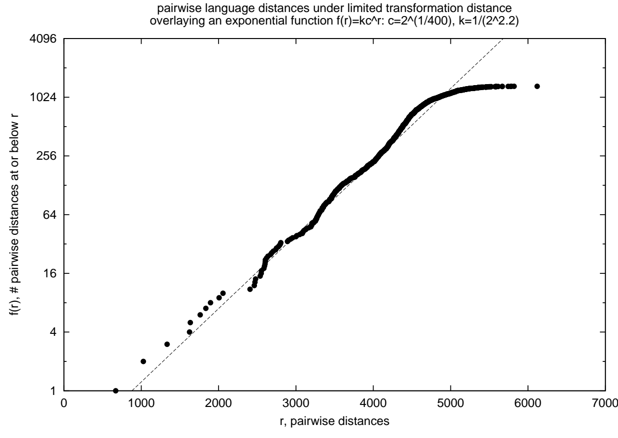


Figure 1. Pairwise distance distribution (compression distance): declaration of human rights in 52 Eurasian languages.

becomes possible to calculate bounds on constants c and k under which the VP trees perform nearest neighbor searches well. Our analysis below assumes that the distance measure $dist()$ provides a metric; however, it can easily be generalized to almost metrics as well.

Nearest neighbor search under exponential distribution. Let a data set contain m points. Given any typical query point q the number of points observed at distance $\leq r$ obeys $f(r) = k \cdot c^r$. One can easily compute the distance between q and its nearest neighbor $nn_1(q)$, namely $dist(q, nn_1(q))$ as follows. By definition $f(dist(q, nn_1(q))) = 1$ and thus $dist(q, nn_1(q)) = \log_c 1/k$.

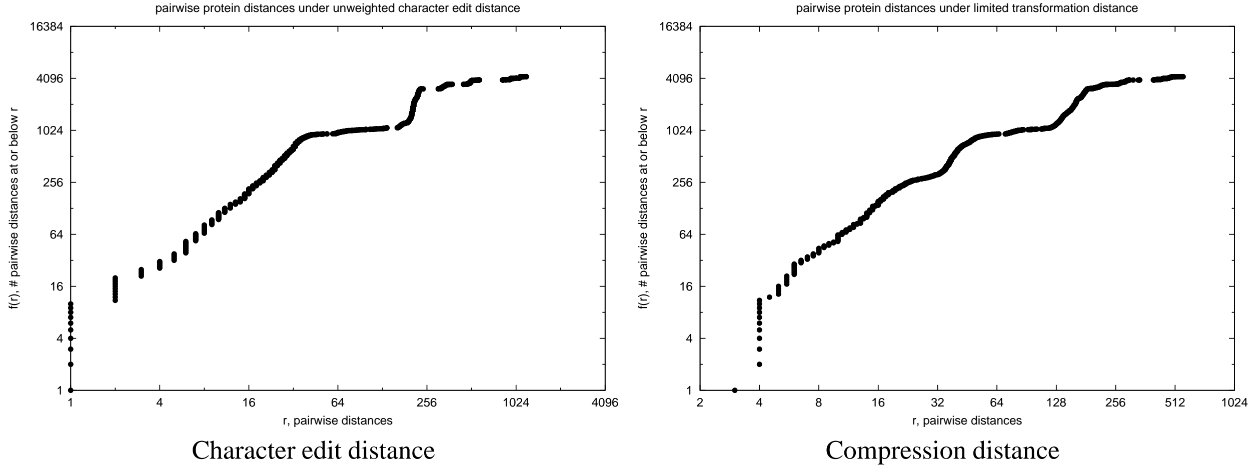
Let p be the topmost vantage point in the VP tree. It is possible to compute the distance between p and its m/ℓ'th nearest neighbor for some constant $\ell \geq 1$ which will be determined soon: $f(dist(p, nn_{m/\ell}(p))) = m/\ell$ and thus $dist(p, nn_{m/\ell}(p)) = \log_c m/k\ell$. The number of points that are within distance $dist(p, nn_{m/\ell}(p)) + dist(p, nn_1(p))$ from p are:
$f(dist(p, nn_{m/\ell}(p)) + dist(p, nn_1(p))) = k \cdot c^{\log_c 1/k} \cdot c^{\log_c m/k\ell} = \frac{m}{k \cdot \ell}$.
Let the VP tree be built in a way that once the vantage point p is determined, it partitions the data set into two: (1) inner partition includes the nearest $m/k\ell$ points to p and (2) the outer partition includes the remaining points.

When searching for the neighbors of q within distance $\epsilon = dist(q, nn_1(q)) = \log_c 1/k$ (i.e. the nearest neighbors) the first step we perform is to compute $dist(q, p)$ and take the following action according to the result:

(1) If $dist(p, q) \leq dist(p, nn_{m/\ell}(p)) = \log_c m/k\ell$ then we eliminate the outer partition and iteratively perform our search on the inner partition. Because the query points are distributed similar to the data points, the probability of this case is $1/\ell$.
(2) If $dist(p, q) > \log_c m/k\ell + \log_c 1/k$ then we eliminate the inner partition and iteratively perform our search on the outer partition. Because the query points are distributed similar to the data points, the probability of this case can be calculated as $1 - 1/\ell k^2$.
(3) Otherwise both the inner and the outer partitions need to be searched.

If $1/k\ell = 1/2$ as per standard VP trees (so that the cardinality of the inner and outer partitions are equal), the probability of case (2) is non-zero if $1/\ell$, the probability of case (1) is > 1. Thus we can ignore case (2) and obtain the following recurrence relation for the query time.
$T(m) \leq 1 + 2k \cdot T(m/2) + (1 - 2k) \cdot 2 \cdot T(m/2)$.
This recursive equation has a solution at $T(m) \leq m^{\log 2 - k/2}$. The space requirements of this implementation will be a small factor of m, much smaller than the data set itself (typical strings of interest are several hundreds or thousands of bytes long).

An alternative implementation. It is possible to modify the standard vantage point trees so that the search time could be substantially improved for the exponential data distribution model. In this version of the VP trees the data set is not partitioned into two by a vantage point. Rather we take the following actions after computing $dist(p, q)$:



Character edit distance
Compression distance

Figure 2. Pairwise distance distribution: proteins that are active in the brain.

(1) If $dist(p,q) \leq dist(p, nn_{m/\ell}(p)) = \log_c m/k\ell$ then we iteratively perform the search within the inner partition again. This case will again occur with probability $1/\ell$.
(2) If on the other hand $dist(p,q) > \log_c m/k\ell$ we perform the search once more on the whole data set, this time using another vantage point.[3] To limit the space complexity we will do at most some $j \cdot \ell$ comparisons of q against vantage points; after that the search will be performed separately on both the inner and the outer partitions. The probability of failing in all $j \cdot \ell$ vantage points is only $(1 - 1/\ell)^{j \cdot \ell} \leq 1/e^j$.

The running time of this modification can be obtained as follows:
$T(m) \leq \ell + (1 - 1/e^j) \cdot T(m/k\ell) + 1/e^j \cdot T(m/k\ell) + 1/e^j \cdot T(m(1 - 1/k\ell))$.
For $\ell = 2/k$ as per the standard VP, $T(m) \leq 2/k + (1 - 1/e^j) \cdot T(m/2) + 2/e^j T(m/2)$ and thus $T(m) \leq 2/k \cdot (1 + 1/e^j)^{\log m} \cdot T(m/2)$ which implies $T(m) \leq 2/k \cdot (1 + 1/e^j)^{\log m} = O(2/k \cdot m^{\log 1 + 1/e^j})$.

The space requirement of this version of the updated VP tree is proportional to the total number of vantage points picked. There are $2/k$ vantage points at level 1 and in level i there are $(2/k)^i$ vantage points. Because the number of levels is $\log m$, the total space requirement will be $O((2/k)^{\log m})$.

Nearest neighbor search with polynomial distribution. Given any typical query point q the number of points observed at distance $\leq r$ obeys $f(r) = k \cdot r^c$ for some $c > 1$. By definition $f(dist(q, nn_1(q))) = 1$ and thus $dist(q, nn_1(q)) = (1/k)^{1/c}$.

[3] This is reminiscent to a data structure suggested -but not analyzed- in [16].

Similarly, given p as the vantage point, $f(dist(p, nn_{m/\ell}(p))) = m/\ell$ thus $dist(p, nn_{m/\ell}(p)) = (m/k\ell)^{1/c}$. It is easy to verify that the number of points that are within distance $dist(p, nn_{m/\ell}(p)) + dist(p, nn_1(p))$ from p is approximately m/ℓ. Thus for $\ell = 2$ as per the standard VP tree (which sets the sizes of partitions to be $m/2$ each), the search time becomes $T(m) = 1 + 3/2 \cdot T(m/2)$ and hence $T(m) = O((3/2)^{\log m}) = O(m^{\log 3/2}) = O(m^{0.58})$. The space complexity of this standard VP tree implementation is again $O(m)$.

A modification for improving the performance of the above VP tree can again be obtained for the polynomial distribution as per the modification for the exponential distribution. For $\ell = 2$ and letting at most j vantage points per level, one can get $T(m) \leq 2 + (1 - 1/2^j)T(m/2) + 2/2^j T(m/2)$. Solving the recurrence relation gives $T(m) \leq 2 \cdot m^{\log(1+1/2^j)}$. By repeating the analysis performed for the exponential distribution, one can show the space complexity to be $O(j^{\log m}) = O(m^{\log j})$. By picking $j = 4$ one can achieve $O(n^{1/11})$ search time (which will be a small constant for all practical data sets) by using space $O(m^2)$.

4.2 Experimental Evaluation of VP Trees

Our theoretical results indicate that the VP trees and the suggested updates would perform well especially under polynomial distance distributions. We first test our VP implementations on synthetic data that exhibits some high degree polynomial distance distribution. The distance function we choose is the compression distance. We then test or implementations on complete human

proteome from the Celera databases [50]. The distance we used for this application is the character edit distance.

Performance of the VP tree on synthetic data. (Figure 3) Our test involves generation of some 2000 (polynomially distributed) random strings among which the nearest 5 strings were searched for a given query string. The query string is generated uniformly at random; its "nearest neighbor" is generated by applying 5 random block edit operations on the query string; 50 of the remaining strings are generated by applying 15 operations; all the rest of the strings are generated by applying 45 operations. We performed experiments on three different VP trees on the same data set, each VP tree with a different query string used for construction.

We performed searches (search radius $\epsilon = 15$) using compression distance using a $O(m)$ space VP tree implementation with the best and worst possible constants in the triangular inequality. As described earlier, this constant can be empirically computed to ensure a most efficient implementation of the modified VP tree.

We performed two sets of experiments depending on the constant k for which for any three of the strings R, S, Q in the data set, $c(R,S) \leq k \cdot [c(R,Q) + c(Q,S)]$. The first experiment assumes the worst possible constant $k = 3$. The second experiment assumes the best possible constant $k = 1$. We verified that the data set satisfies the best case constant $k = 1$, but we provide our experimental results for $k = 3$ for comparison purposes.

In the worst case, the VP tree was able to eliminate only $33 - 45\%$ of the strings in the data set to return the closest 5 strings to the query. This provides the worst case performance of the data set if the data set satisfies the triangular inequality within a constant factor of $k = 3$. In reality, the data set satisfies the triangular inequality with $k = 1$. By using this fact, the index was able to eliminate 90% of the strings in the data set, resulting in significant savings in near neighbor search. We expect that with some constant factor increase in space it will be possible to improve the pruning factor further. We leave the analysis of the tradeoffs between k and the percentage data set elimination to the full version of the paper.

Experimental Results on Protein Data. (Figure 4) Our second data set includes all (about 32K) active and potential proteins derived from the complete human genome sequence database obtained by Celera [50]. We report on the pruning results of near neighbor searches with varying search radii (based on character edit distance) on arbitrarily selected protein strings from the data set. The figure shows that the number of string comparisons made against the size of the data set varies considerably from one search to another.

5 Summary

We investigate the applicability of distance based indexing techniques for string proximity search. We first show that string distance measures of interest such as the compression distance and weighted character edit distance provide almost metrics. We then show how to modify vantage point trees and other distance based indexing techniques to accommodate almost metric distances. Finally we show how to further modify these techniques in order to exploit typical pairwise distance distributions observed in textual and biomolecular string data sets and improve their search performance.

References

[1] M. Ankerst, G. Kastenmuller, H-P. Kriegel, T. Seidl, *Nearest Neighbor classification in 3D Protein Databases*, *Proc. of ISMB*, 1999.

[2] R. Agrawal, C. Faloutsos, A Swami, *Efficient Similarity Search in Sequence Databases*, *Proc. of FODO*, 1993.

[3] A. N. Arslan, O. Egecioglu, P. A. Pevzner *A new approach to sequence comparison: normalized sequence alignment*, *Proceedings of RECOMB 2001*.

[4] R. Agarwal, K. Lin, H. Sawhney and K. Shim. *Fast similarity search in the presence of noise, scaling and translation in time-series databases*, *Proc. VLDB conference*, 1995.

[5] E. Hunt, M.P. Atkinson, R.W. Irving *A database index to large biological sequences Proc. of VLDB conference*, 2001.

[6] Altschul SF, Gish W, Miller W, Myers EW, Lipman DJ. *Basic local alignment search tool J Mol Biol* 1990 Oct 5;215(3):403-10.

[7] Bailey JA, Yavor AM, Massa HF, Trask BJ, Eichler EE. *Segmental duplications: organization and impact within the curren t human genome project assembly*, *Genome Research* 11(6), Jun 2001.

[8] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, Bernhard Seeger *The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles*, *Proc. ACM SIGMOD Conference* 1990: 322-331

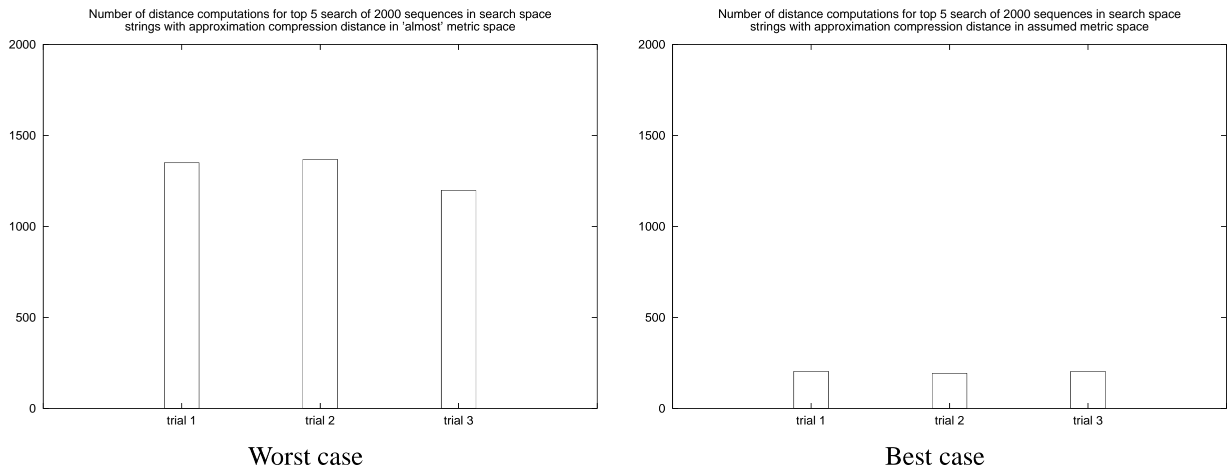
[9] S. Berchtold and D. A. Keim. *High-dimensional Index Structures*, *Proc. ACM SIGMOD*, pp. 501, 1998.

[10] Stefan Berchtold, Daniel A. Keim, Hans-Peter Kriegel. *The X-tree : An Index Structure for High-Dimensional Data*, *VLDB* 1996: 28-39

[11] D. Benedetto, E. Caglioti, V. Lorento. *Language Trees and Zipping*, *Physical Review Letters*, 88(4), Jan 2002.

[12] K. Beyer, J. Golstein, R. Ramakrishnan, U. Shaft. *When is nearest neighbors meaningful?*, *Proc. ICDT*, 1997.

[13] W. A. Burkhard and R. M. Keller. *Some Approaches to Best-Match File Searching Communications of the ACM*, 16(4), pages 230–236, April 1973.



Worst case Best case

Figure 3. Pruning efficiency test on synthetic data. The number of string comparisons made vs the size of the data set.

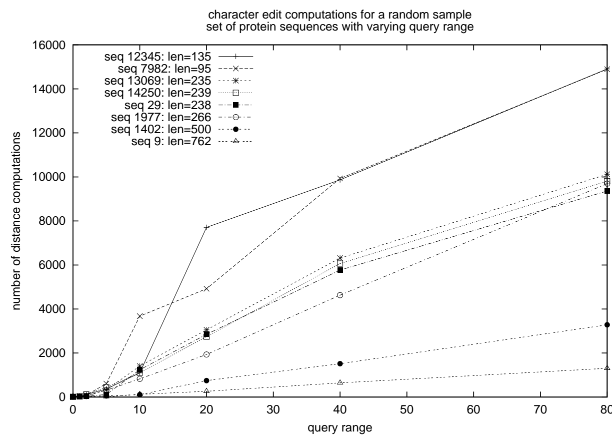


Figure 4. Pruning efficiency test on the complete human proteome.

[14] T. Bozkaya and M. Ozsoyoglu, *Distance-Based Indexing for High-Dimensional Metric Spaces Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pp. 357-368, 1997.

[15] A. Borodin and R. Ostrovsky and Y. Rabani, *Lower Bounds for High Dimensional Nearest Neighbor Search and Related Problems*, *In Proc. of ACM STOC*, 1999.

[16] S. Brin, *Near Neighbor Search in Large Metric Spaces*, *In Proc. VLDB*, 574-584, 1995.

[17] J. Buhler and M. Tompa *Finding Motifs Using Random Projections*, *In Proc. of RECOMB 2001*.

[18] Chakrabarti, Chazelle, Gum and Lvov, *A Lower Bound on the Complexity of Approximate Nearest-Neighbor Searching on the Hamming Cube*, *Proc. ACM Symposium on Theory of Computing*, 1999.

[19] T. Chiueh, *Content based image indexing*, *Proc. VLDB*, 582-593, 1995

[20] P. Ciaccia, M. Patella, and P. Zezula. *M-Trees: An efficient access method for similarity search in metric space*, *Proc. VLDB* pp. 426-435, 1997.

[21] Raphael A. Finkel, Jon Louis Bentley, *Quad Trees: A Data Structure for Retrieval on Composite Keys. Acta Informatica* 4: 1-9 (1974)

[22] Gionis, Indyk and Motwani, *Similarity Search in High Dimensions via Hashing*, *Proc. Intl. Conference on Very Large Data Bases*, 1999.

[23] L. Gravano, P.G. Ipeirotis, H.V. Jagadish, N. Koudas, S. Muthukrishnan, D. Srivastava, *Approximate string joins in a database (almost) for free Proc. VLDB*, 2001.

[24] J. Goldstein and R. Ramakrishnan, *Contrast Plots and p-sphere trees: space vs. time in nearest neighbor searches*, *Proc. VLDB*, pp. 429-440, 2000.

[25] D. Gusfield, *Algorithms on Strings, Trees and Sequences*, *Cambridge University Press*, 1997.

[26] A. Guttman, *R-Trees: A dynamic index structure for spatial searching*, *proc. SIGMOD*, 1984, pp.47-57.

[27] P. Indyk and R. Motwani. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. *Proc. ACM Symp. on Theory of Computing*, 1998, 604–613.

[28] Jackson, Strachan, Dover, *Human Genome Evolution*, *Bios Scientific Publishers*, 1996.

[29] Y. Ji, E. E. Eichler, S. Schwartz, R. D. Nicholls, *Structure of Chromosomal Duplications and their Role in Mediating Human Genomic Disorders*, *Genome Research* 10, 2000.

[30] N. Katayama, S. Satoh, *The SR-tree: An index structure for high dimensional nearest neighbor queries*, *Proc. SIGMOD*, pp. 369-380, 1997.

[31] T. Kahveci, A.K. Singh, *Efficient Index Structures for String Databases*, *Proc. VLDB*, pp. 351-360, 2001.

[32] S. Kurtz, G. Myers *Estimating the Probability of Approximate Matches*, *Proc. Combinatorial Pattern Matching*, LNCS 1264, 1997.

[33] E. Kushilevitz, R. Ostrovsky and Y. Rabani, *Efficient search for approximate nearest neighbor in high dimensional spaces*, *Proc. ACM Symposium on Theory of Computing*, 1998, 614–623.

[34] V. I. Levenshtein, *Binary codes capable of correcting deletions, insertions and reversals*, *Cybernetics and Control Theory*, 10(8):707-710, 1966.

[35] E. S. Lander et al., *Initial sequencing and analysis of the human genome*, *Nature*, 15:409, Feb 2001.

[36] D. Lopresti and A. Tomkins. *Block edit models for approximate string matching*, *Theoretical Computer Science*, 1996.

[37] M. Li, J. H. Badger, C. Xin, S. Kwong, P. Kearney, H. Zhang, *An information based sequence distance and its application to whole mitochondrial genome phylogeny*, *Bioinformatics*, 17, 2001

[38] Ming Li, Xin Chen, Xin Li, Bin Ma, Paul Vitanyi, *The Similarity Metric*, *Proceedings of ACM-SIAM SODA*, Baltimore MD, 2003 (to appear).

[39] S. Muthukrishnan and S. C. Sahinalp, *Approximate nearest neighbors and sequence comparison with block operations* *Proc. ACM Symposium on Theory of Computing*, 2000.

[40] Philip Ball *Algorithm makes tongue tree*, *Nature*, Science update, Jan 22, 2002.

[41] Needleman SB, Wunsch CD, *A general method applicable to the search for similarities in the amino acid sequence of two proteins.* *J Mol Biol.* 1970 Mar;48(3):443-53.

[42] Michael Rodeh, Vaughan R. Pratt, Shimon Even, *Linear Algorithm for Data Compression via String Matching*. *JACM* 28(1): 16-24 (1981).

[43] Y. Sakurai, M. Yoshikawa, S. Uemura, H. Kojima, *The A-tree: An index structure for high dimensional spaces using relative approximation* *VLDB*, pp. 516-526, 2000.

[44] T. Sellis, N. Roussopoulos and C. Faloutsos. *Multidimensional Access Methods: Trees Have Grown Everywhere*, *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB)*, 1997, pp. 13–15.

[45] George P. Smith *Evolution of Repeated DNA Sequences by Unequal Crossover*, *Science*, vol 191, pp 528–535 (1976)

[46] T. F. Smith, M. S. Waterman, *Identification of Common Molecular Subsequences*, *Journal of Molecular Biology* 147:195-197 (1981)

[47] J.A. Storer, *Data compression: methods and theory*, *Computer Science Press*, 1988.

[48] J.K. Uhlmann, *Satisfying general proximity/similarity queries with metric trees*, *IPL*, (4):175-179, 1991.

[49] J. S. Varre, J. P. Delahaye, E. Rivals, *The Transformation Distance: A Dissimilarity Measure Based on Movements of Segments*, *Bioinformatics*, 15:3, 194-202, 1999.

[50] C. Venter et. al., *The sequence of the human genome*, *Science*, 16:291, Feb 2001.

[51] R. Weber, H. Schek, and S. Blott, *A quantitative analysis and performance study for similarity search methods in high dimensional spaces*, *Proc. VLDB*, pp. 194-205, 1998.

[52] Peter N. Yianilos, *Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces*, *Proc. ACM-SIAM Symposium on Discrete Algorithms*, pp. 311-321, 1993.

[53] C. Yu, B.C. Ooi, K. Tan, H.V. Jagadish, *Indexing the Distance, An Efficient Method to KNN Processing*, *Proc. VLDB*, 2001.