Efficient k-Nearest Neighbor Searching in Non-Ordered Discrete Data Spaces

DASHIELL KOLBE
Michigan State University
QIANG ZHU
University of Michigan
SAKTI PRAMANIK
Michigan State University

Numerous techniques have been proposed in the past for supporting efficient k-nearest neighbor (k-NN) queries in continuous data spaces. Limited work has been reported in the literature for k-NN queries in a non-ordered discrete data space (NDDS). Performing k-NN queries in an NDDS raises new challenges. The Hamming distance is usually used to measure the distance between two vectors (objects) in an NDDS. Due to the coarse granularity of the Hamming distance, a k-NN query in an NDDS may lead to a high degree of non-determinism for the query result. We propose a new distance measure, called Granularity-Enhanced Hamming (GEH) distance, that effectively reduces the number of candidate solutions for a query. We have also implemented k-NN queries using multidimensional database indexing in NDDSs. Further, we use the properties of our multidimensional NDDS index to derive the probability of encountering new neighbors within specific regions of the index. This probability is used to develop a new search ordering heuristic. Our experiments on synthetic and genomic data sets demonstrate that our index-based k-NN algorithm is efficient in finding k-NNs in both uniform and non-uniform data sets in NDDSs and that our heuristics are effective in improving the performance of such queries.

Categories and Subject Descriptors: H.4.0 [**Information Systems Applications**]: General

General Terms: Similarity Search, Spatial Indexing

Additional Key Words and Phrases: Database, Non-Ordered Discrete Data Space, Distance Measurement, Nearest Neighbor

1. INTRODUCTION

There is an increasing demand for similarity searches in applications such as geographical information systems [Roussopoulos et al. 1995], multimedia databases [Seidl and Kriegel 1997], molecular biology [Badel et al. 1992], and genome sequence databases [Qian et al. 2003; Qian 2004]. The two most common types of similarity searches are range searches/queries and k-nearest neighbor (k-NN) searches/queries. The former is to find data objects that are within a tolerant distance from a given query point/object, while the latter is to retrieve k-nearest neighbors to the query point. An example of a range query is "find the words in a document that differ from word 'near' by at most two letters". An example of a k-NN query is "find two service stations that are closest to the current one".

Numerous techniques have been proposed in the literature to support efficient similarity searches in (ordered) continuous data spaces (CDS). A majority of them

This work was supported by the National Science Foundation (under grants #IIS-0414576 and #IIS-0414594), Michigan State University and The University of Michigan.

utilize a multidimensional index structure such as the R-tree [Guttman 1988], the R*-tree [Beckmann et al. 1990], the X-tree [Berchtold et al. 1996], the K-D-B tree [Robinson 1981], and the LSD^h-tree [Henrich 1998]. These techniques rely on some essential geometric properties/concepts such as bounding rectangles in CDSs. Much work has centered around a filter and refinement process. Roussopoulos et al. presented a branch-and-bound algorithm for finding k-NNs to a query point. Korn et al. furthered this work by presenting a multi-step k-NN searching algorithm, which was then optimized by Seidl and Kriegel [Seidl and Kriegel 1998]. In [Kolahdouzan and Shahabi 2004], a Voronoi based approach was presented to address k-NN searching in spatial network databases.

Little work has been reported on supporting efficient similarity searches in so-called non-ordered discrete data spaces (NDDS). A d-dimensional NDDS is a Cartesian product of d domains/alphabets consisting of finite non-ordered elements/letters. For example, when searching genome DNA sequences, consisting of letters 'a', 'g', 't', 'c', each sequence is often divided into intervals/strings of a fixed length d (q-gram). These intervals can be considered as vectors from a d-dimensional NDDS with alphabet $\{a, g, t, c\}$ for each dimension. Other examples of non-ordered discrete dimensions are color, gender and profession. Application areas that demand similarity searches in NDDSs include bioinformatics, E-commerce, biometrics and data mining.

Limited existing work on index-based similarity searches in NDDSs has utilized either metric trees such as the M-tree [Ciaccia et al. 1997] or the ND-tree and NSP-tree recently proposed by Qian et al. [Qian et al. 2003; Qian et al. 2006a; Qian et al. 2006b]. Unlike the M-tree, the ND-tree and the NSP-tree indexing techniques were designed specifically for NDDSs. It has been shown that these two techniques outperform the linear scan and typical metric trees such as the M-tree for range queries in NDDSs. Metric trees generally do not perform well in NDDSs because they are too generic and do not take the special characteristics of an NDDS into consideration. On the other hand, Qian et al.'s work in [Qian et al. 2003; Qian et al. 2006a; Qian et al. 2006b] primarily focused on handling range queries. Although a procedure for finding the nearest neighbor (i.e., 1-NN) to a query point was outlined in [Qian et al. 2006a], no empirical evaluation was given.

The issue of k-NN searching in NDDSs is in fact not a trivial extension of earlier work. NDDSs raise new challenges for this problem. First, we observe that unlike a k-NN query in a CDS, a k-NN query in an NDDS based on the conventional Hamming distance [Hamming 1950], often has a large number of alternative solution sets, making the results of the k-NN query non-deterministic. This non-determinism is mainly due to the coarse granularity of the Hamming distance and can sharply reduce the clarity/usefulness of the query results. Second, existing index-based k-NN searching algorithms for CDSs cannot be directly applied to an NDDS due to a lack of relevant geometric concepts/measures. On the other hand, the algorithms using metric trees for a CDS are suboptimal because of their generic nature and ignorance of special characteristics of an NDDS. Third, the information maintained by an NDDS index structure may become very misleading for traditional CDS search ordering strategies, such as those presented by Roussopoulos et al. [Roussopoulos et al. 1995]. This scenario can occur as the distribution of data within the index structure shifts over time.

To tackle the first challenge, we introduce a new extended Hamming distance, called the Granularity-Enhanced Hamming (GEH) distance. The GEH distance improves the semantics of k-NN searching in NDDS by greatly increasing the determinism of the results. To address the second challenge, we propose a k-NN searching algorithm utilizing the ND-tree. Our algorithm extends the notion of incremental range based search [Roussopoulos et al. 1995] (generalized for metric space by Hjaltason and Samet [Hjaltason and Samet 2000]) to NDDSs by introducing suitable pruning metrics and relevant searching heuristics based on our new distance measure and the characteristics of NDDSs. Some preliminary results for uniformly distributed datasets were presented in [Kolbe et al. 2007]. Our study shows that the new GEH distance provides a greatly improved semantic discriminating power that is needed for k-NN searching in NDDSs, and that our searching algorithm is very efficient in supporting k-NN searches in NDDSs. In this paper, we demonstrate through additional experiments that our k-NN searching algorithm is efficient in both uniformly distributed data sets and non-uniformly distributed data sets using zipf distributions as an example. Further, we present a theoretical performance model and demonstrate that the performance of our algorithm matches very closely to what is predicted by this model. To address the third issue, we introduce a method for determining the probability of a vector's existence within any sub-structure of an ND-tree. We demonstrate this probability information can be used to provide a new search ordering strategy that significantly increases the performance of our search algorithm when the information maintained by the index structure is misleading.

The rest of this paper is organized as follows. Section 2 formally defines the problem of k-NN searching, derives the probability of a vector existing within an ND-tree, introduces the new GEH distance in NDDSs, and discusses its properties. Section 3 presents our index-based k-NN searching algorithm for NDDSs, including its pruning metrics and heuristics and theoretical performance model. Section 4 discusses experimental results. Section 5 summarizes our conclusions and gives some future research directions.

2. *K*-NEAREST NEIGHBORS IN NDDS

In this section, we first review some concepts related to NDDSs, including the ND-tree that our searching algorithm is based on. We then formally define a k-NN search/query and identify a major problem associated with k-NN searches in NDDSs. To overcome the problem, we propose a new extended Hamming distance and discuss its properties. Additionally, we introduce a method for determining the probability of a vector/record's existence in any particular subtree of an ND-tree, based upon the properties of NDDSs and the index tree.

2.1 The ND-Tree

The ND-tree has some similarities, in structure and function, to the R-tree [Guttman 1988] and its variants (R*-tree [Beckmann et al. 1990] in particular). As such, the ND-tree is a balanced tree with leaf nodes containing the indexed vectors. The vectors are reached by traversing a set of branches starting at the root and becoming more refined as one traverses toward the leaves.

The ND-tree is built using strategies similar to those for the R-tree. Each vector is inserted into the tree after an appropriate position is found in the tree. The relevant minimum bounding rectangle may need to be split to accommodate the insertion.

A key difference between the ND-tree and its continuous cousins is the way in which a minimum bounding rectangle is defined and utilized. In the ND-tree, minimum bounding rectangles are discrete. A *discrete minimum bounding rectangle* (DMBR) for a set $G = \{R_1, R_2, \ldots, R_n\}$ of discrete rectangles $R_i = \{S_{i1} \times S_{i2} \times \ldots \times S_{id}\}$ $(1 \leq i \leq n)$ is defined as follows:

$$DMBR(G) = (\cup_{i=1}^{n} S_{i1}) \times (\cup_{i=1}^{n} S_{i2}) \times \ldots \times (\cup_{i=1}^{n} S_{id}), \tag{1}$$

where S_{ij} $(1 \leq j \leq d)$ is a set of elements/letters from the alphabet of the j-th dimension of the given d-dimensional NDDS. Such a DMBR allows the ND-tree to utilize a non-Euclidean method of measurement for calculating the distance between a vector $\alpha = (\alpha[1], \alpha[2], \ldots, \alpha[d])$ and DMBR $R = \{S_1 \times S_2 \times \ldots \times S_d\}$:

$$dist(\alpha, R) = \sum_{i=1}^{d} \left\{ \begin{array}{l} 0 \text{ if } \alpha[i] \in S_i \\ 1 \text{ otherwise} \end{array} \right\}. \tag{2}$$

This distance can be interpreted as saying that, for each dimension in a query vector α, if the element represented therein occurs anywhere in the subtree associated with DMBR R, then nothing will be added to the current distance, otherwise a 1 will be added.

2.2 Definition of k-NN in NDDS

When considering a query in an NDDS, the data set may be depicted as a set of concentric spheres with the query point located at the center (as shown in Figure 2.2). Each sphere contains all data points that have r mismatching dimensions with the query point, where r represents the layer/radius of the particular sphere. A straight forward approach to determining the minimal radius r, where a sphere of this radius will contain at least k data points/neighbors, is to find r such that a sphere of radius r contains at least k neighbors while a sphere of radius $r - 1$ contains less than k neighbors. In general, the solution set of k-nearest neighbors for a given query point may not be unique due to multiple objects having the same distance to the query point. Thus, there may be multiple candidate solution sets for a given query point and k value. We define a candidate solution set of k-nearest neighbors for a query point as follows:

Definition 2.2.1. **Candidate *k*-Nearest-Neighbors** Let the universe of discourse for variables A_i and B_i $(1 \leq i \leq k)$ be the set of all objects in the database. Let kNNS denote a candidate solution set of k-nearest neighbors in the database for a query point q and $D(x, y)$ denote the distance between the objects x and y. Then

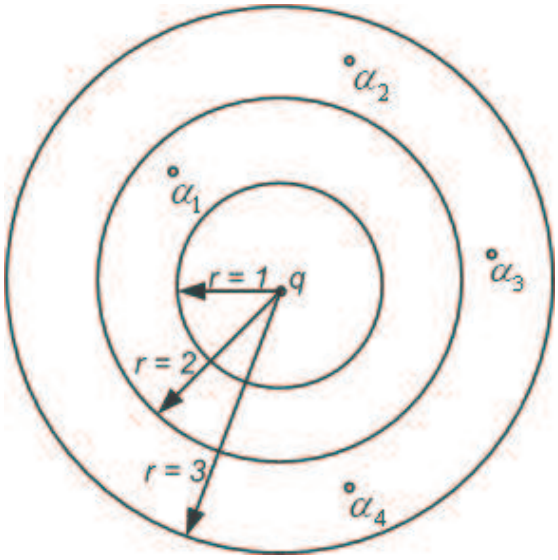

Fig. 1. NDDS data points distributed by distance example

*k*NNS is defined as follows:

$$kNNS \in \left\{ \{A_1, A_2, ..., A_k\} : \begin{array}{l} \forall_{B_1,B_2,...,B_k} \left[\Sigma_{i=1}^{k} D(q, A_i) \leq \Sigma_{j=1}^{k} D(q, B_j)\right] \wedge \\ \forall_{m,n \in \{1,2,...,k\}} \left[(m \neq n) \rightarrow (A_m \neq A_n)\right] \end{array} \right\}. \tag{3}$$

Equation 3 essentially says that k objects/neighbors $A_1, A_2, \ldots, A_k$ in *k*NNS have the minimum total distance to q out of all possible sets of k objects in the database. This definition is in fact valid for both continuous and discrete data spaces. Consider Figure 2.2, if $k = 3$, there are three possible sets of neighbors that satisfy Equation 3: $\{\alpha_1, \alpha_2, \alpha_3\}$, $\{\alpha_1, \alpha_2, \alpha_4\}$, & $\{\alpha_1, \alpha_3, \alpha_4\}$. Each candidate solution set is found within a range of r when a range of $r-1$ would yield less than k neighbors (here, $r = 3$). Thus, each candidate solution set is a subset of the set of neighbors that would be found using the straight forward search approach described above.

Since *k*NNS is a set of neighbors, there is no ordering implied among the neighbors. In the following recursive definition we provide a procedural semantic of a candidate k^{th}-nearest neighbor, which is based on an ordered ranking of the neighbors in the database for a query point q.

Definition 2.2.2. **Candidate k^{th}-Nearest-Neighbor:** Let the universe of discourse for variables A and B be the set of all objects in the database. Let A_k denote a candidate k^{th}-nearest neighbor in the database for a query point q. We

recursively define A_k as follows:

$$A_1 \in \{A : \forall_B (D(q, A) \leq D(q, B))\},$$
$$A_k \in \left\{ A : \forall_B \left(\begin{array}{l} B \notin \{A_1, A_2, \ldots, A_{k-1}\} \wedge \\ A \notin \{A_1, A_2, \ldots, A_{k-1}\} \rightarrow \\ D(q, A) \leq D(q, B) \end{array} \right) \right\} \text{ for } k \geq 2. \tag{4}$$

Definition 2.2.2 can be used to produce all the candidate *k*NNSs given by Definition 2.2.1, as stated in the following proposition.

PROPOSITION 2.2.3. *The set of candidate kNNSs given by Definition 2.2.1 can be produced by Definition 2.2.2.*

PROOF. See Appendix A.

From the above definitions (and Figure 2.2), we can see that there may be multiple possible *k*NNSs for a given query. Therefore, *k*NNS is generally not unique. The non-uniqueness/non-determinism of *k*NNS has an impact on the semantics of the k-nearest neighbors. We define the degree of non-determinism of k-nearest neighbors by the number of possible *k*NNSs that exist for the query. This degree of non-determinism is computed by the following proposition.

PROPOSITION 2.2.4. *The number Δk of candidate kNNSs is given by*

$$\Delta k = \frac{N^{'}!}{t!(N^{'} - t)!}, \tag{5}$$

where t is defined by $D(q, A_{k-t}) \neq D(q, A_{k-t+1}) = D(q, A_{k-t+2}) = \ldots = D(q, A_k)$; $A_j (1 \leq j \leq k)$ denotes the j^{th}-nearest neighbor; $N^{'}$ is the number of nearest neighbors in the database that have the same distance as $D(q, A_k)$.

PROOF. See Appendix B.

Note that t denotes the number of objects with distance $D(q, A_k)$ that have to be included in a *k*NNS: if $t = k$, all the neighbors in a *k*NNS are of the same distance as $D(q, A_k)$. The values of $N^{'}$ and t depend on parameters such as the dimensionality, database size, and the query point.

For a k-NN query on a database in a continuous data space based on the Euclidean distance, *k*NNS is typically unique (i.e. $\Delta k = 1$) since the chance for two objects having the same distance to the query point is usually very small. As a result, the non-determinism is usually not an issue for k-NN searches in continuous data spaces.

However, non-determinism is a common occurrence in an NDDS. As pointed out in [Qian et al. 2003; Qian et al. 2006a], the Hamming distance is typically used for NDDSs. Due to the insufficient semantic discrimination between objects provided by the Hamming distance and the limited number of elements available for each

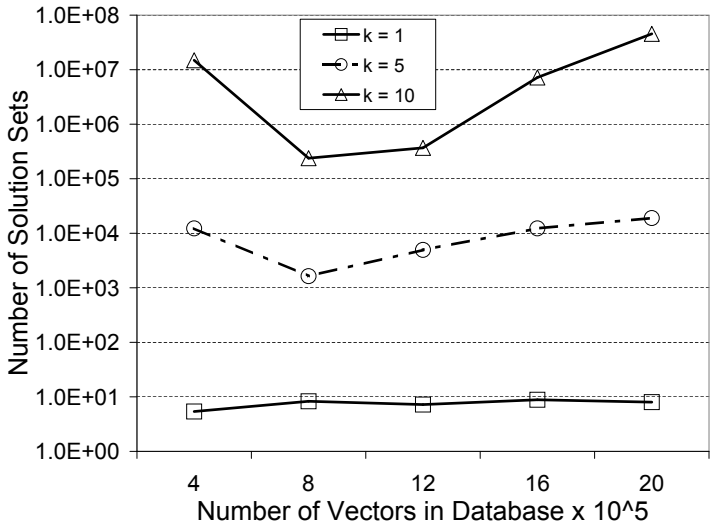

Fig. 2. Comparison of Δk values for the Hamming distance

dimension in an NDDS, Δk for a k-NN query in an NDDS is usually large. For example, for a data set of $2M$ vectors in a 10-dimensional NDDS with uniform distribution, the average Δk values for 100 random k-NN queries with $k = 1, 5, 10$ are about $8.0, 19.0K, 45.5M$ respectively, as shown in Figure 1. This demonstrates a high degree of non-determinism for k-NN searches based on the Hamming distance in an NDDS, especially for large k values. To mitigate the problem, we extend the Hamming distance to provide more semantic discrimination between the neighbors of a k-NN query point in an NDDS.

2.3 Extended Hamming Distance

The Hamming distance (generalized in [Bookstein et al. 2002]) between two vectors $\alpha = (\alpha[1], \alpha[2], \ldots, \alpha[d])$ and $\beta = (\beta[1], \beta[2], \ldots, \beta[d])$ is defined as follows:

$$D_{Hamm}(\alpha, \beta) = \sum_{i=1}^{d} \left\{ \begin{array}{l} 1 \text{ if } \alpha[i] \neq \beta[i] \\ 0 \text{ otherwise} \end{array} \right\}. \tag{6}$$

Intuitively, the Hamming distance indicates the number of dimensions on which the corresponding components of α and β differ.

Although the Hamming distance is very useful for exact matches and range queries in NDDSs, it does not provide an effective semantic for k-NN queries in NDDSs due to the high degree of non-determinism, as mentioned previously. We notice that the Hamming distance does not distinguish equalities for different elements. For example, it treats element $a = a$ as the same as element $b = b$ by assigning 0 to the distance measure in both cases. In many applications such as genome sequence searches, some matches (equalities) may be considered to be more

important than others. Based on this observation, we extend the Hamming distance to capture the semantics of different equalities in the distance measure.

Several constraints have to be considered for such an extension. First, the extended distance should enhance the granularity level of the Hamming distance so that its semantic discriminating power is increased. Second, the semantic of the traditional Hamming distance needs to be preserved. For example, from a given distance value, one should be able to tell how many dimensions are distinct (and how many dimensions are equal) between two vectors. Third, the extended distance should possess a triangular property so that pruning during an index-based search is possible.

We observe that matching two vectors on a dimension with a frequently-occurred element is usually more important than matching the two vectors on the dimension with an uncommon (infrequent) element. Based on this observation, we utilize the frequencies of the elements to extend the Hamming distance as follows:

$$D_{GEH}(\alpha, \beta) = \sum_{i=1}^{d} \left\{ \begin{array}{ll} 1 & \text{if } \alpha[i] \neq \beta[i] \\ \frac{1}{d} f(\alpha[i]) & \text{otherwise} \end{array} \right\}, \tag{7}$$

where

$$f(\alpha[i]) = 1 - frequency(\alpha[i]).$$

This extension starts with the traditional Hamming distance; adding one to the total distance for each dimension that does not match between the two vectors. The difference is that, when the two vectors match on a particular dimension, the frequency of the common element (i.e. $\alpha[i] = \beta[i]$) occurring in the underlying database on the dimension is obtained from a lookup table generated by performing an initial scan of the dataset. This frequency value is then subtracted from one and then added to the distance measure. Thus, the more frequently an element occurs, the more it will subtract from one and thus the less it will add to the distance measure, thereby indicating that the two vectors are closer than if they had matched on a very uncommon element.

The factor of $\frac{1}{d}$ is used to ensure that the frequency-based adjustments to the distance measure do not end up becoming more significant than the original Hamming distance. This is a key factor in maintaining the triangular distance property. Additionally, this guarantees that the solution set (kNNS) returned using this distance will be a subset of the solution set returned if the Hamming distance were used instead.

From the distance definition, we can see that, if $m \leq D_{GEH}(\alpha, \beta) < m + 1$ ($m = 0, 1, \ldots, d$), then vectors α and β mis-match on m dimensions (i.e., match on $d - m$ dimensions). Within each interval, the smaller the distance value, the larger the frequency(ies) of the element(s) shared by α and β on the matching dimension(s). Clearly, unlike the traditional Hamming distance, which has at most $d + 1$ (integer) values – resulting in a quite coarse granularity, this new extended distance allows many more possible values – leading to a refined granularity. We call this extended Hamming distance the *Granularity-Enhanced Hamming (GEH)* distance. Due to its enhanced granularity, the GEH distance can dramatically

reduce Δk in Proposition 2.2.4, leading to more deterministic k-NN searches in NDDSs. For example for the aforementioned data set of $2M$ vectors in a 10-dimensional NDDS under the uniform distribution, the average Δk values for 100 random k-NN queries with $k = 1, 5, 10$ are about 1.09, 1.11, 1.06, respectively (see Figure 3 in Section 4.1).

In fact, the Euclidean distance measurement can be considered to have the finest (continuous) granularity at one end, while the Hamming distance measurement has a very coarse (discrete integers) granularity at the other end. The GEH distance measurement provides an advantage in bringing discrete and continuous distance measurements closer to each other. Additionally, the GEH distance may be used in the pruning step in index-based searches due to its possession of the triangular property. Proof of this property is presented in Appendix C.

2.4 Probability of New Neighbors

In many scenarios, it is useful to know the probability/likelihood of encountering vectors within an index structure that are within the current search radius to a given query vector. For the purposes of our discussion we label each such encountered vector as a new neighbor α; where $\forall_\alpha d(q, \alpha) < r$, where q is the query vector and r is the current search radius. To derive this probability, we first consider the Hamming distance and then extend our solutions to benefit from the enhancements provided by the GEH distance.

For an initial case, we can assume that our index structure has maintained a relatively uniform distribution of elements within its substructures. In a well balanced tree (ND-Tree, M-Tree, etc...), this may prove to be a very reasonable assumption, as most indexing methods will attempt to evenly distribute elements within their substructures. When we consider an ND-Tree as our indexing method, the probability that accessing a subtree with associated DMBR $R = \{S_1 \times S_2 \times \ldots \times S_d\}$ will yield a particular element a in any dimension may be estimated as the reciprocal of the magnitude of the alphabet set on that dimension represented by R. Therefore, the probability of a specific element a occurring in dimension i is estimated as:

$$p\left(a\right)_{R,i} = \frac{1}{|S_i|}.$$

This calculation proves to be fairly accurate so long as the assumption of uniform distribution holds. The accuracy, and therefore effectiveness, of this calculation begins to degrade as the distribution of elements per dimension within a subtree becomes non-uniform.

The true probability may be estimated far more accurately by determining the local ratio of element a within a subtree, with associated DMBR R, on dimension i as follows:

$$p\left(a\right)_{R,i} = f_l\left(a\right)_{R,i}, \tag{8}$$

where

$$f_l(a)_{R,i} = \frac{\#\text{ of vectors } \alpha \text{ in } R\text{'s subtree where } \alpha[i] = a}{\text{total } \#\text{ of vectors in } R\text{'s subtree}}.$$

This method is not reliant upon the indexing method to provide an even distribution: Equation 8 remains accurate even in indexes with heavily skewed distributions.

The probability of encountering new neighbors when examining any particular subtree of an ND-tree is analogous to the probability of such neighbors existing in that subtree. Each dimension in an NDDS is assumed to be independent, therefore, the probability value of encountering specific elements over all dimensions may be determined by the product of the probability values of encountering a specific element in each dimension. Thus the probability of selecting any particular vector $\alpha = (\alpha[1], \alpha[2], \ldots, \alpha[d])$, at random from the subtree with associated DMBR R is the following:

$$PE\left(\alpha, R\right) = \prod_{i=1}^{d} p\left(\alpha[i]\right)_{R,i}. \tag{9}$$

As defined in Section 2.3, the Hamming distance represents the number of non-matching dimensions between any two vectors. The probability of a subtree containing a vector α where $D_{Hamm}(q, \alpha) = 0$ may be determined using Equation 9. However, because at most, only one vector within an ND-Tree will satisfy $D_{Hamm}(q, \alpha) = 0$, we must also consider the probability of a subtree containing vectors $\beta = (\beta[q], \beta[2], \ldots, \beta[d])$, where $D_{Hamm}(q, \beta) = z : z \in \{1, 2, \ldots, d\}$.

PROPOSITION 2.4.1. *Let Y represent the set of dimensions where $\beta[Y_j] \neq q[Y_j]$ and let X represent the set of dimensions where $\beta[X_j] = q[X_j]$. We then define the probability of selecting a particular vector β from the subtree with associated DMBR R, where $D_{Hamm}(q, \beta) = z, (0 \leq z \leq d)$, at random as the following (note that $z = |Y|$):*

$$PNE(\beta, R) = \prod_{j=1}^{|X|} p(\beta[X_j])_{R,X_j} * \prod_{j=1}^{|Y|} (1 - p(\beta[Y_j])_{R,Y_j}). \tag{10}$$

PROOF: As described in Equation 9, the probability of a specific vector existing in a subtree, represented by R, is the product of the probabilities of each element of the vector in the corresponding dimension of the subtree: the probability of the element is defined by $PE\left(\alpha, R\right)$ (Equation 8). The probability of anything except the specified element is $1 - PE\left(\alpha, R\right)$. Thus, the probability of a specific vector, that does not match the query vector in z dimensions, existing in a subtree/R is the product of two terms; the product of $PE\left(\alpha, R\right)$ in the matching dimensions and the product of $1 - PE\left(\alpha, R\right)$ in the non-matching dimensions.

Proposition 2.4.1 describes the method for determining the probability of encountering a vector that matches the query vector on a particular set of dimensions X. An example would be determining the probability of encountering a vector β in a 10-dimensional data set that matched a query vector q in dimensions 1, 3, 8, & 9. In this example, $X = \{1, 3, 8, 9\}$ and $Y = \{0, 2, 4, 5, 6, 7\}$, resulting in $z = D_{Hamm}(q, \beta) = 6$. However, to determine the probability of encountering a vector at a distance z, we are not only interested in this one particular partial-

matching vector, but rather all possible partial-matching vectors that may be found at a distance z from the query vector.

PROPOSITION 2.4.2. *Let B represent all vectors from the set of all objects in the data space where $\beta \in B : [D_{Hamm}(q,\beta) = z]$. The probability that a subtree with associated DMBR $R = \{S_1 \times S_2 \times \ldots \times S_d\}$ will contain a vector β where $D_{Hamm}(q,\beta) = z$ is defined as the following:*

$$PS_z(q,R) = \sum_{\beta \in B} PNE(\beta, R). \tag{11}$$

PROOF: The probability of a subset of independent objects existing in a set, is the summation of the probabilities of each of the individual objects within the subset existing in the set. Thus, the probability of a subset of vectors existing within a subtree, that each have a specified number of dimensions matching a query vector, is the summation of the probabilities of each vector within that subset existing in the subtree.

For example, the probability of selecting a vector β at random from a subtree with associated DMBR R, where $D_{Hamm}(q,\beta) = 1$, is shown as follows:[1]

$$\begin{aligned} PS_1(q,R) = \quad & p(q[1])p(q[2])\ldots p(q[d-1])\left(1 - p\left(q[d]\right)\right) \\ & + p(q[1])p(q[2])\ldots\left(1 - p\left(q[d-1]\right)\right)p(q[d]) \\ & \qquad\qquad \vdots \\ & + \left(1 - p\left(q[1]\right)\right)p(q[2])\ldots p(q[d-1])p(q[d]) \end{aligned}.$$

The probability of a subtree with associated DMBR R containing a vector β where $D_{Hamm}(q,\beta) \leq \lfloor r \rfloor$ is expressed as the summation of the probability of the subtree containing a vector for each integer distance $z : z \in \{0, 1, \ldots, \lfloor r \rfloor\}$. This is expressed formally as follows:

$$PNN_H(q,R) = \sum_{z=0}^{\lfloor r \rfloor} PS_z(q,R). \tag{12}$$

Equation 12 may therefore be used to give an accurate measure as to the likelihood that searching within any subtree will update a solution set when using the Hamming distance. Enhancing the granularity of the Hamming distance leads to an enhancement of the neighbor probability calculated in Equation 12. When using the GEH distance, it is possible for a new neighbor to exist at a distance $\lfloor r \rfloor < D_{GEH}(q,\beta) \leq r$. An adjustment to Equation 11 is needed to properly account for possible neighbors within this range.

PROPOSITION 2.4.3. *Let B' represent all vectors from the set of all objects in the data space where $\beta \in B^{'} : [D_{Hamm}(q,\beta) = \lfloor r \rfloor]$. The probability that a subtree with*

[1] As the dimensionality of the dataset increases, this calculation can become costly as it is on the order of $O(d * \binom{d}{d-z})$. However, much of this cost can be avoided by calculating the vectors in each set B and storing them as binary arrays in a pre-processing step.

DMBR $R = \{S_1 \times S_2 \times \ldots \times S_d\}$ *will contain a record* β *where* $(\lfloor r \rfloor < D_{GEH}(q, \beta) \leq r)$ *is defined as the following:*

$$PR(q, R, r) = \sum_{\beta \in B'} PNE(\beta, R)\delta\left(r - \lfloor r \rfloor, \beta, q\right), \tag{13}$$

where

$$\delta\left(r - \lfloor r \rfloor, \beta, q\right) = \left\{ \begin{array}{cc} 1 & \textit{if } R_{adj}(\beta, q) < r - \lfloor r \rfloor \\ 0 & \textit{otherwise} \end{array} \right\},$$

$$R_{adj}\left(\beta, q\right) = \frac{1}{d}\sum_{j=1}^{d} \left\{ \begin{array}{cc} f(q[j]) & \textit{if } \beta[j] = q[j] \\ 0 & \textit{otherwise} \end{array} \right\}.$$

PROOF: The proof for Proposition 2.4.2 shows that Equation 11 yields the probability of a vector, that has z mismatching dimensions with the query vector, existing in a particular subtree. If $z = \lfloor r \rfloor$, this equation will determine the probability of a vector with $\lfloor r \rfloor$ mismatching dimensions with the query vector existing in the subtree. The set of these vectors is represented by $B^{'}$. Function δ creates a subset of these vectors by removing all vectors v, from the set $B^{'}$, where $D_{GEH}(v, q) > r$. Equation 13 is the summation of each of the remaining vectors $v^{'}$, where $\forall_{v' \in B'} : D_{GEH}(v^{'}, q) < r$. Thus Equation 13 yields the probability of a vector, whose distance to the query vector is between $\lfloor r \rfloor$ and r to the query vector, existing in a particular subtree.

The additional granularity provided by Equation 13 allows us to refine Equation 12 to make use of our GEH distance as follows:

$$PNN_{GEH}\left(q, R, r\right) = PR(q, R, r) + \sum_{z=0}^{\lfloor r \rfloor - 1} PS_z\left(q, R\right). \tag{14}$$

Equation 14 may be used to give an accurate measure of the likelihood that searching within any subtree will update a solution set when our enhanced distance measure is used. We use this measure in section 3.2 to develop an ordering heuristic that provides a conservative assessment of whether or not to visit a particular subtree that is beneficial to search performance in non-uniformly distributed databases.

3. A K-NN ALGORITHM FOR NDDS

To efficiently process k-NN queries in NDDSs, we introduce an index-based k-NN searching algorithm. This algorithm utilizes properties of the ND-tree recently proposed by Qian, et al. [Qian et al. 2003; Qian et al. 2006a] for NDDSs. The basic idea of this algorithm is as follows. It descends the ND-tree from the root following a depth-first search strategy. When k possible neighbors are retrieved, the searching algorithm uses the distance information about the neighbors already collected to start pruning search paths that can be proven to not include any vectors that are closer to the query vector than any of the current neighbors. Our algorithm is based upon earlier incremental ranged based implementations presented

for CDS by Roussopoulos et al. [Roussopoulos et al. 1995] and Hjaltason and Samet [Hjaltason and Samet 2000] (generalized for metric space). Our algorithm extends these implementations to NDDS by introducing metrics and heuristics suitable for such a space. The details of this algorithm are discussed in the following subsections.

3.1 Heuristics

In the worst case scenario, this search would encompass the entire tree structure. However, our extensive experiments have shown that the use of the following heuristics is able to eliminate most search paths before they need to be traversed.

MINDIST Pruning: Similar to [Roussopoulos et al. 1995], we utilize the minimum distance (MINDIST) between a query vector q and a DMBR $R = \{S_1 \times S_2 \times \ldots \times S_d\}$, denoted by $mdist(q, R)$, to prune useless paths. Based on the GEH distance, MINDIST is formally defined as follows:

$$mdist(q, R) = \sum_{i=1}^{d} \left\{ \begin{array}{ll} 1 & \text{if } q[i] \notin S_i \\ \frac{1}{d} f(q[i]) & \text{otherwise} \end{array} \right\} \tag{15}$$

where

$$f(q[i]) = 1 - frequency(q[i]).$$

This calculation is then used with the *Range* of the current k-nearest neighbors (with respect to the query vector) to prune subtrees. Specifically, the heuristic for pruning subtrees is:

H_1: If $mdist(q, R) \geq Range$, then prune the subtree associated with R.

By taking the closest distance between a DMBR and q, we are guaranteeing that no vectors that are included in the DMBR's subtree are closer than the current *Range* and thus need not be included in the continuing search.

MINMAXDIST Pruning: We also utilize the minimum value (MINMAXDIST) of all the maximum distances between a query vector q and a DMBR R along each dimension, denoted by $mmdist(q, R)$, for pruning useless paths. In simple terms, $mmdist(q, R)$ represents the shortest distance from the vector q that can guarantee another vector in R/subtree can be found. For a vector q and DMBR $R = \{S_1 \times S_2 \times \ldots \times S_d\}$, MINMAXDIST is formally defined as follows:

$$mmdist(q, R) = \min_{1 \leq k \leq d} \left\{ f_m\left(q[k], S_k\right) + \sum_{i=1, i \neq k}^{d} f_M\left(q[i], S_i\right) \right\} \tag{16}$$

where

$$f_m\left(q[k], S_k\right) = \left\{ \begin{array}{ll} \frac{1}{d} f(q[k]) & \text{if } q[k] \in S_k \\ 1 & \text{otherwise} \end{array} \right\},$$
$$f_M\left(q[j], S_k\right) = \left\{ \begin{array}{ll} \frac{1}{d} f(q[j]) & \text{if } \{q[j]\} = S_j \\ 1 & \text{otherwise} \end{array} \right\}.$$

where $f()$ on the right hand side of the last two Equations is defined in Equation 15.

In general terms, the summation of f_M determines the number of dimensions where every vector in the associated subtree is guaranteed to have a matching element with the query vector. In these cases, a value of $\frac{1}{d} f(q[j])$ is added to the distance. The value of f_m determines if there is another dimension (not in those checked for f_M) that some subset of the vectors in the associated subtree will match the query vector in that dimension. In this case, a value of $\frac{1}{d} f(q[k])$ is added to the distance. A value of 1 is added for all other cases in f_M and f_m. The summation of these values yields the minimum distance (adjusted for GEH) it can be guaranteed a vector will be located from the query vector in the associated subtree, based upon the information in the DMBR.

To process k-NN searches in our algorithm, $mmdist()$ is calculated for each non-leaf node of the ND-tree using query vector q and all the DMBRs (for subtrees) contained in the current node. Once each of these MINMAXDIST values (for subtrees) have been calculated, they are sorted in ascending order and the k^{th} value is selected as $MINMAXDIST_k$ for the current node.

The k^{th} value is selected to guarantee that *at least* k vectors will be found in searching the current node. This selected $MINMAXDIST_k$ is then used in the following heuristic:

H_2: If $MINMAXDIST_k$(node)< Range, then let Range = $MINMAXDIST_k$(node)

Optimistic Search Ordering: For those subtrees that are not pruned by heuristic H_1 or H_2, we need to decide an order to access them. Two search orders were suggested in [Roussopoulos et al. 1995]: one is based on the ordering of MINDIST values, and the other is based on the ordering of MINMAXDIST values. The MINMAXDIST ordering is too pessimistic to be practically useful. Accessing the subtrees based on such an ordering is almost the same as a random access in NDDSs. From an extensive empirical study, we found that accessing subtrees in the optimistic order of MINDIST values during a k-NN search in an NDDS provided the more promising results. This study was performed with the assumption that the ND-tree is well structured. This access order is shown formally as follows:

H_3: Access subtrees ordered in ascending value of $mdist(q, R)$. In the event of a tie, choose a subtree at random.

Conservative Search Ordering: A problem associated with search ordering heuristic H_3 is that it optimistically assumes that a vector with a distance of the MINDIST value exists in the subtree associated with the relevant DMBR. Typically this is not the case in an NDDS: the set of elements on each dimension from

different vectors often yields a combination that is not an indexed vector in the corresponding subtree. In some instances, the actual distribution of elements per dimension within a subtree may be significantly different than what is expressed in the representing DMBR. As discussed in Section 2.4, this can be estimated by calculating the difference between the assumed uniform distribution, $\frac{1}{|S_i|}$, and the actual distribution, estimated by frequency in Equation 8.

When the difference between the assumed distribution and the actual distribution becomes large for multiple elements or multiple dimensions for a query, the likelihood of a vector with a distance of MINDIST existing in the relevant DMBR greatly decreases. When this occurs, it is more appropriate to order the access of subtrees by the calculated probability of the subtree containing a vector whose distance to the query vector is lessthan the current range, as shown in Equation 14. This access order is given formally as follows:

H_4: Access subtrees in the descending order of the probability of containing a vector α, where $D(q, \alpha) < r$. This probability is calculated by PNN_{GEH}

Heuristic H_4 may be considered as a conservative approach to ordering while heuristic H_3 may be seen as an optimistic approach to ordering.

3.2 Algorithm Description

Our k-NN algorithm adopts a depth first traversal of the ND-tree and applies the aforementioned heuristics to prune non-productive subtrees and determine the access order of the subtrees. The description of the algorithm is given in the following subsections.

3.2.1 *k-NN Query Algorithm.* Given an ND-tree with root node T, Algorithm 3.2.1 finds a set of k-nearest neighbors to query vector q that satisfies Equation 3 in Definition 2.2.1. It invokes two functions: $ChooseSubtree$ and $RetrieveNeighbors$. The former chooses a subtree of a non-leaf node to descend, while the latter updates a list of k-nearest neighbors using vectors in a leaf node.
In the algorithm, step 1 initializes relevant variables. Steps 2 - 15 traverse the ND-tree. Steps 3 - 10 deal with non-leaf nodes by either invoking $ChooseSubtree$ to decide a descending path or backtracking to the ancestors when no more subtree to explore. Steps 11 - 14 deal with leaf nodes by invoking $RetrieveNeighbors$ to update the list of current k-nearest neighbors. Step 16 returns the result. Note that $ChooseSubtree$ not only returns a chosen subtree but also may update $Range$ using heuristic H_2. If no more subtree to choose, it returns $NULL$ for NN at step 4. Similarly, $RetrieveNeighbors$ not only updates kNNS but also may update $Range$ if a closer neighbor(s) is found.

3.2.2 *Function ChooseSubtree.* The effective use of pruning is the most efficient way to reduce the I/O cost for finding a set of k-nearest neighbors. To this end, the heuristics discussed in Section 3.1 are employed in function $ChooseSubtree$.
In this function, steps 1 - 4 handle the case in which the non-leaf is visited for the first time. In this case, the function applies heuristics H_1 - H_4 to update $Range$, prune useless subtrees, and order the remaining subtrees (their root nodes)

Algorithm 3.2.1 k**-NNQuery**
Input: (1) query vector q**; (2) the desired number** k **of nearest neighbors; (3) an ND-tree with root node** T **for a given database.**
Output: a set $kNNS$ **of** k**-nearest neighbors to query vector** q**.**

```
1: let kNNS = ∅, N = T, Range = ∞, Parent = NULL
2: while N ≠ NULL do
3:   if N is a non-leaf node then
4:     [NN, Range] = ChooseSubtree(N, q, k, Range)
5:     if NN ≠ NULL then
6:       Parent = N
7:       N = NN
8:     else
9:       N = Parent
10:    end if
11:  else
12:    [kNNS, Range] = RetrieveNeighbors(N, q, k, Range, kNNS)
13:    N = Parent
14:  end if
15: end while
16: return kNNS
```

Function 3.2.2 $ChooseSubtree(N, q, k, Range)$

```
1: if list L for not yet visited subtrees of N not exist then
2:   use heuristic H2 to update Range
3:   use heuristic H1 to prune subtrees of N
4:   use heuristic H3 or H4 based upon user criteria to sort the remaining subtrees
     not pruned by H1 and create a list L to hold them
5: else
6:   use heuristic H1 to prune subtrees from list L
7: end if
8: if L ≠ ∅ then
9:   remove the 1st subtree NN from L
10:  return [NN, Range]
11: else
12:  return [NULL, Range]
13: end if
```

in a list.[2] The list, L, is maintained in memory for each node N until the search is terminated. Step 6 applies heuristic H_1 and current *Range* to prune useless subtrees for a non-leaf node that was visited before. Steps 8 - 12 return a chosen subtree (if any) and the updated *Range*.

3.2.3 *Function RetreiveNeighbors.* The main task of *RetrieveNeighbor* is to examine the vectors in a given leaf node and update the current k-nearest neighbors

[2]The use of H_3 or H_4 is set at the integrators discretion, based upon their knowledge of the data set. Our criteria for selection is discussed in Section 4.4.

and *Range*.

Function 3.2.3 $RetrieveNeighbors(N, q, k, Range, kNNS)$

```
1: for each vector v in N do
2:   if D_GEH(q, v) < Range then
3:     kNNS = kNNS ∪ {v}
4:     if |kNNS| > k then
5:       remove vector v' from kNNS such that v' has the largest D_GEH(q, v') in
         kNNS
6:       Range = D_GEH(q, v'') such that v'' has the largest D_GEH(q, v'') in kNNS
7:     end if
8:   end if
9: end for
10: return [kNNS, Range]
```

A vector is collected in *k*NNS only if its distance to the query vector is smaller than current *Range* (steps 2 - 3). A vector has to be removed from *k*NNS if it has more than k neighbors after a new one is added (steps 4 - 7). The vector to be removed has the largest distance to the query vector. If there is a tie, a random furthest vector is chosen.

3.3 Performance Model

To analyze the performance of our k-NN search algorithm, presented in Section 3.2, we conducted both empirical and theoretical studies. The results of our empirical studies are presented in Section 4. In this section, we present a theoretical model for estimating the performance of our search algorithm. For our presentation, we assume that our algorithm employs both heuristics H_1 and H_2. We also assume an optimistic ND-tree structure, where a subtree's associated DMBR is reasonably representative of the vectors within the subtree; that is $\forall_{a,i\in\{1,2,\ldots,d\}} : f_l(a)_{R,i} \sim \frac{1}{|S_i|}$. With this assumption, our search algorithm employs H_3 as its search ordering heuristic.[3]

Because of the unique properties of an NDDS, there is no defined center for the data space. This may also be interpreted as any point may be considered to be at the center. Thus, we can define a bounding hyper-sphere around any point within the data space and determine the likely number of objects contained within.

Definition 3.3.1. The area within a distance z from point p in a d-dimensional NDDS with alphabet set A for each dimension is the total number of possible unique points contained within the hyper sphere of radius z centered at point p. This value is formally calculated as the summation of the number of points existing in spherical layers as follows:

[3]The assumption of a reasonably optimistic tree structure covers the majority of ND-Tree's generated in our empirical studies. Non optimistic tree structures, where H_4 would provide a more beneficial ordering method, are considered empirically in Section 4.4.

$$Area(z) = \sum_{i=0}^{z} \binom{d}{i} (|A| - 1)^i. \tag{17}$$

Note that $Area(z)$ is independent of point p under the uniform distribution assumption. Equation 17 may be used to calculate the total area of the data space by setting $z = d$. However, this value may be calculated directly as follows:

$$Area(d) = |A|^d. \tag{18}$$

The probability of an object existing within a distance of z from any point p is the quotient of Equations 17 and 18, as follows:

$$P_{exists}(z) = \frac{Area(z)}{Area(d)}. \tag{19}$$

Definition 3.3.2. The number of likely points contained within a distance z from any point p is the product of the number of points within the data set N and the probability of a point existing within a distance of z from p. This is shown formally as follows:

$$L(z) = P_{exists}(z) * N. \tag{20}$$

The lower/optimal search bound for our performance model is determined as a reasonable distance to assure a specific number of objects. It is reasonable to assume that a minimum distance that needs to be searched is one that is likely to yield at least k neighbors. Thus a lower bound $d_l = z$, is found by solving Equation 20 such that $L(d_l) \geq k$ and $L(d_l - 1) < k$. The lower bound for performance I/O may then be estimated as the number of pages that are touched by a range query of radius d_l. The range query performance is derived similarly to the model provided by Qian et el, [Qian et al. 2006a].

$$IO_r = 1 + \sum_{i=0}^{H-1} (n_i * P_{i,z}). \tag{21}$$

Where n_i represents the estimated number of nodes within the ND-tree at a height of i, $P_{i,z}$ represents the probability a node at height i will be accessed in the ND-tree with a search range of z, H denotes the height of the index tree.

However, because a k-NN query generally begins with a search range equal to the theoretical upper search bound, an adjustment must be made to account for the I/O incurred while searching with a non-optimal search range. We have estimated this value as the number of nodes within each level of the ND-tree raised to a power inversely proportional to their height:

$$Adj = \sum_{i=1}^{H-1} n_{i-1}^{\left(\frac{1}{H-1}\right)}. \tag{22}$$

Adding this adjustment to the range query performance model yields the following:

$$IO_{NN} = 1 + (n_0 * P_{0,z}) + \sum_{i=1}^{H-1} \left[(n_i * P_{i,z}) + n_{i-1}^{\left(\frac{1}{H-1}\right)} \right]. \quad (23)$$

The performance of our search algorithm can be estimated by using Equation 23 setting z to d_l.

4. EXPERIMENTAL RESULTS

To evaluate the effectiveness of our GEH distance and the efficiency of our k-NN searching algorithm, we conducted extensive experiments. The experimental results are presented in this section. Our k-NN searching algorithm was implemented on an ND-Tree in the C++ programming language. For comparison purposes, we also implemented the algorithm on an M-Tree in the C++ programming language for one set of experiments. All experiments were ran on a PC under OS Windows XP. The I/O block size was set at $4K$ bytes for both trees. Both synthetic and genomic data sets were used in our experiments. The synthetic data sets consist of uniformly distributed 10-dimensional vectors with values in each dimension of a vector drawn from an alphabet of size 6; other special case synthetic data sets are listed in the following sub sections. The genomic data sets were created from the Ecoli DNA data (with alphabet: $\{a, g, t, c\}$) extracted from the GenBank. Each experimental data reported here is the average over 100 random queries.

4.1 Effectiveness of GEH Distance

The first set of experiments was conducted to show the effectiveness of the GEH distance over the Hamming distance, by comparing their values of Δk as defined in Proposition 2.2.4 in Section 2.2.

Figure 3 gives the relationship between Δk and database sizes for both the GEH and Hamming distances, when k=1, 5 and 10. The figure shows a significant decrease in the values of Δk using the GEH distance over those using the Hamming distance. This significant improvement in performance for the GEH distance is observed for all the database sizes and k values considered. Figure 3 shows that when the GEH distance is used, Δk values are very close to 1, indicating a promising behavior close to that in CDSs.

4.2 Efficiency of k-NN Algorithm on Uniform Data

One set of experiments was conducted to examine the effects of heuristics H_1 - H_3, presented in Section 3.1, on the performance of our k-NN searching algorithm presented in Section 3.2 on uniform data. We considered the following three versions of our pruning strategies in the experiments.

Version $V1$: only heuristic H_1 is used.

Version $V2$: heuristics H_1 and H_2 are used.

Version $V3$: three heuristics H_1, H_2, and H_3 are used.

Figure 4 shows that $V2$ provides a little improvement in the number of disk accesses over $V1$. However, $V2$ does make good performance improvements over $V1$ for some of the queries. Thus, we have included H_2 in version $V3$. As seen from the figure, $V3$ provides the best performance improvement among the three

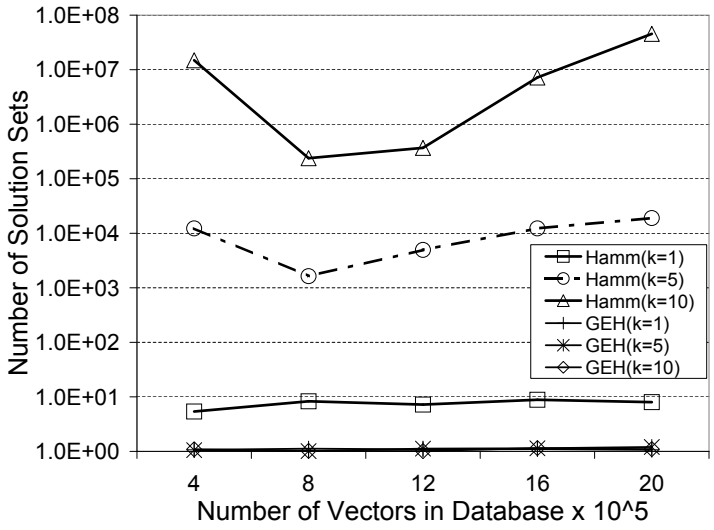

Fig. 3. Comparison of Δk values for the GEH and Hamming distances (ND-Tree)

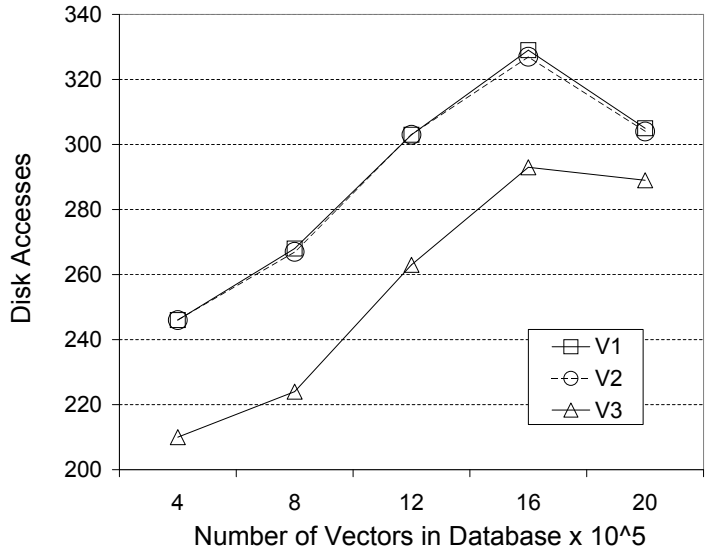

Fig. 4. Effects of heuristics in the k-NN algorithm with $k = 10$ (ND-Tree)

versions for all database sizes tested. Hence $V3$ is adopted in our k-NN searching algorithm and used in all the remaining experiments, except where noted.

Another set of experiments was conducted to compare the disk I/O performance of our index-based k-NN searching algorithm, implemented on both an ND-Tree

and an M-Tree, with that of the linear scan for databases of various sizes. Figure 5 shows the performance comparison for the three methods on synthetic data sets. From the figure, we can see a significant reduction in the number of disk accesses for our k-NN searching algorithm over the linear scan for both tree implementations. Additionally, the results for our ND-Tree implementation agree with the relationship between performance and intrinsic dimensionality [Chávez et al. 2001], whereby as the number of points in the dataset grows larger, the performance improvement gets larger as well. This is due to the likely increase in variance of distances between points in the dataset.[4] Figure 5 also shows that, for all database sizes tested, our algorithm, implemented upon an ND-Tree, always used less than 25% (10% for database sizes greater than $1M$ vectors) of the disk accesses than the linear scan. Our algorithm implemented upon an M-Tree always used less than 10% of the disk accesses of a linear scan. Further, we note that although our M-Tree implementation shows strong performance for smaller database sizes, our ND-Tree implementation performs less disk accesses as the number of vectors in the database increases; $800K$ for $k = 1$, $1.6M$ for $k = 5$, and $2.0M$ for $k = 10$.

Figure 6 shows the performance comparison for the two methods on genomic data sets. Since a genome sequence is typically divided into intervals of length (i.e., the number of dimensions) 11 or 15, both scenarios are included in the figure (for k=10). This figure demonstrates that the performance behavior of our k-NN searching algorithm on real-world genomic data sets is comparable with that we observed for the synthetic data sets.

To observe the performance improvement of our k-NN searching algorithm over various dimensions, we ran random k-NN queries (with $k = 10$) on two series of genomic data sets: one contains 250K vectors for each set and the other contains 1 million vectors for each set. As seen from Figure 7, the performance gain of our algorithm over the linear scan is quite significant for lower dimensions. However, the amount of this improvement decreases with an increasing number of dimensions. This phenomenon of deteriorating performance with an increasing number of dimensions is also true in continuous data spaces due to the well-known dimensionality curse problem. Additionally, we have observed the performance improvement of our k-NN searching algorithm over various alphabet sizes. We performed random k-NN queries (with $k = 10$ and $d = 10$) on databases of $2M$ vectors. Figure 8 shows that the affects of increasing alphabet size are similar to those seen when increasing the dimensionality.

Further, we have compared the disk I/O performance of the k-NN searching algorithm using the GEH distance with that for the k-NN searching algorithm using the Hamming distance. Figure 9 shows the percentage I/Os for the GEH distance versus the Hamming distance for various database sizes and k values. From the figure, we can see that the number of disk accesses decreases for all test cases when the GEH distance is used as opposed to the Hamming distance. In fact, the algorithm using the GEH distance needs only $50\% \sim 70\%$ of I/Os that the algorithm using the Hamming distance needs for all test cases. We feel this is due

[4] As the variance in distances between points in the dataset increases, the pruning power of H_1 is likely to increase due to the growing percentage of datapoints whose distance from the query point falls outside the current search radius.

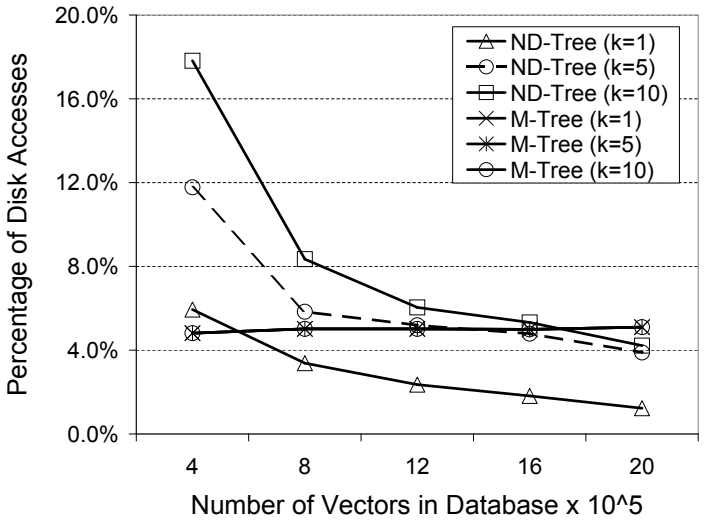

Fig. 5. Performance of the k-NN algorithm vs. the linear scan on synthetic data sets with various sizes

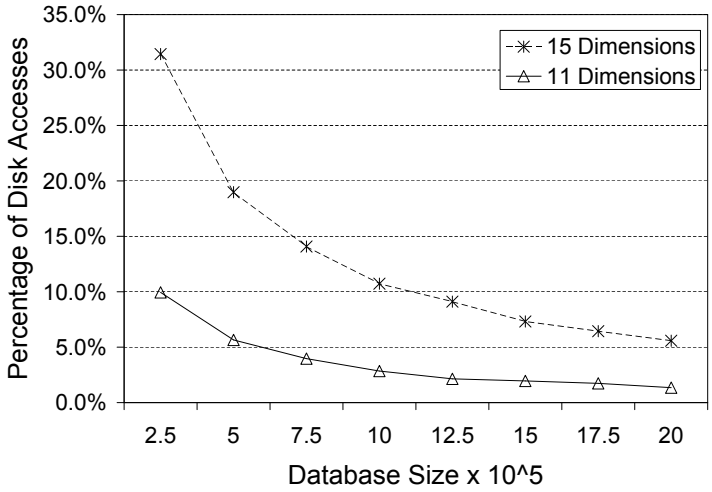

Fig. 6. Performance of the k-NN algorithm vs. the linear scan on genomic data sets with various sizes for k=10 (ND-Tree)

to an increase in the pruning power of heuristic H_1 for the GEH distance. These results indicate that the use of the GEH distance will cost less in disk accesses while

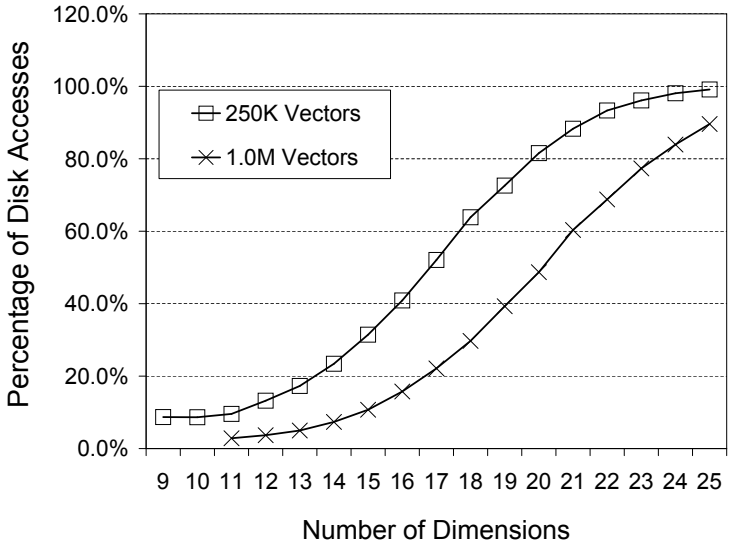

Fig. 7. Performance of the k-NN algorithm vs. the linear scan on genomic data sets with various dimensions for k=10 (ND-Tree)

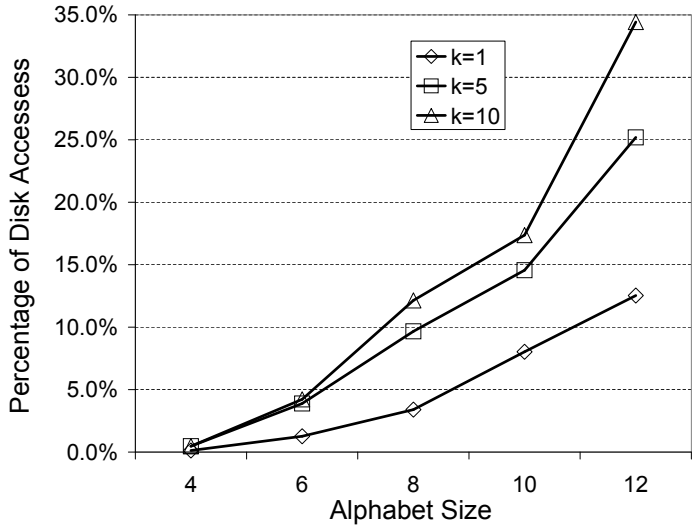

Fig. 8. Performance of the k-NN algorithm vs. the linear scan on synthetic data sets with various dimensions for k=10 and $d = 10$ (ND-Tree)

providing a far more deterministic result than that using the Hamming distance for a k-NN query.

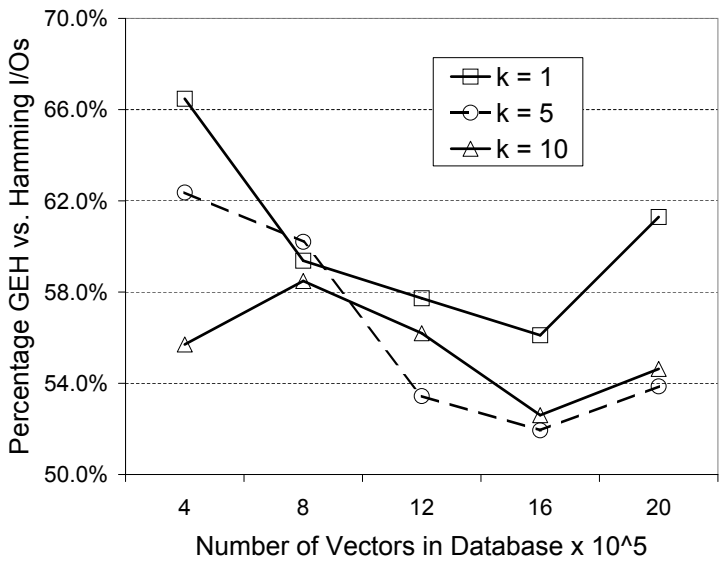

Fig. 9. Performance comparison for the GEH and Hamming distances (ND-Tree)

4.3 Efficiency of k-NN Algorithm on Skewed Data

Experiments also were conducted to examine the I/O performance of our algorithm upon data sets of varying levels of skewness as compared to that of a linear scan. We applied our algorithm, with heuristic version V3 from Section 4.2, to ND-trees constructed from data sets with zipf distributions of 0.0, 0.5, 1.0, and 1.5.

Figure 10 shows significant reduction in the number of disk accesses for our k-NN searching algorithm over the linear scan for all database sizes tested. Similar to the performance gains for uniform data (see section 4.2), our k-NN searching algorithm provides an increased reduction of disk accesses as the database size increases. Figure 10 also shows that our k-NN searching algorithm provides increased performance gains as the level of skewness increases (i.e. the zipf distribution level increases). These results indicate that our searching heuristics (see Section 3.1) are able to identify and prune more useless search paths as the data becomes more skewed.

4.4 Efficiency of k-NN Algorithm on Non-Homogeneous Data

Experiments were conducted to show the effectiveness of our heuristic using the probability equations presented in Section 2.4. We compared the I/O performance between the k-NN algorithm using our probability-based subtree ordering heuristic H_4 against the k-NN algorithm using our traditional MINDIST subtree ordering heuristic H_3. We observed that, although the two heuristics often yield a comparable performance, there are cases in which our probability-based heuristic significantly outperformed the MINDIST one. These cases can occur when the distribution of the dataset shifts over time. For instance, dimensions that are highly relevant to the partitioning of records into subtrees early in the construction of

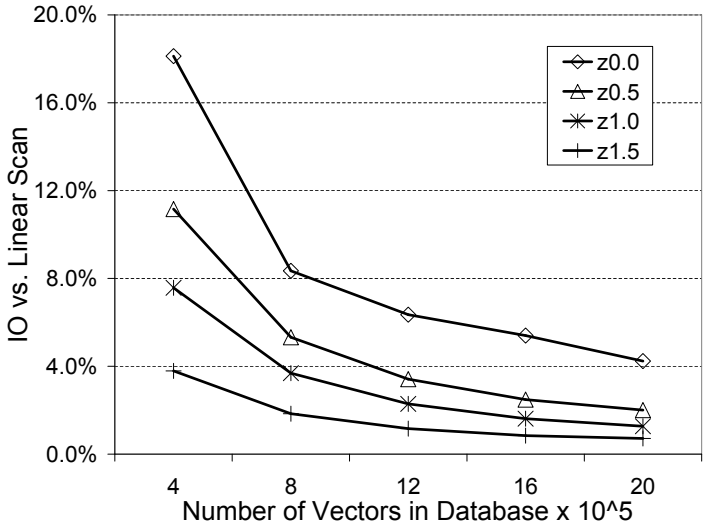

Fig. 10. Performance of the k-NN algorithm vs. the linear scan on synthetic data sets with various sizes and zipf distributions

an ND-tree, may no longer be relevant at later stages of the construction. These dimensions may become misleading when searching for the records inserted into the tree during these later stages. Figures 11, 12, and 13 show our results when searching for 1, 5, and 10 neighbors respectively. Each search was performed on an ND-tree containing 5M vectors using each of the following heuristic combinations:

Version S1: heuristics H_1, H_2, and H_3 are used;

Version S2: heuristics H_1, H_2, and H_4 are used.

The ND-trees constructed from these data sets are known to contain misleading DMBRs in regards to what vectors are present in the relevant subtrees. For our selection of heuristic H_4, we compared the values of $\frac{1}{|S_i|}$ and $f_1(a)_{R,i}$ for each node one level below the root node and labeled a misleading dimension as one in which there was a discrepancy greater than $3:1$ between the two values compared. The number of misleading dimensions indicates the known number of dimensions in each of the DMBRs one level below the root node that meet this criteria; that is, for a particular dimension i, $\frac{1}{|S_i|} > 3 * f_l(a)_{R,i}$.

The results in Figure 11 show that the use of heuristic version S2 provides benefits for most cases tested when searching for only a single neighbor. The cases where the number of misleading dimensions was either very large or very small still show better I/O performance when using heuristic version S1. The results in Figure 12 show that the reduction of I/O when heuristic version S2 is used is much larger for all cases tested when searching for five neighbors rather than a single neighbor. The results in Figure 13 show that the reduction of I/O continues to grow when searching for ten neighbors when using heuristic version S2. These results show that in general, as the number of neighbors being searched for increases, the performance

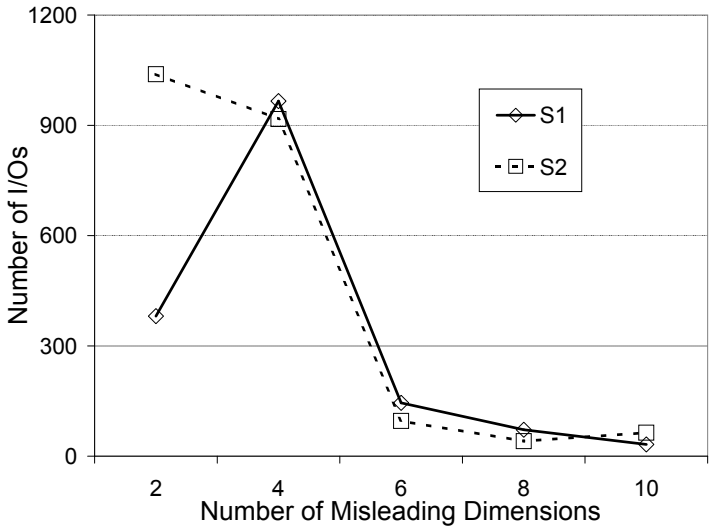

Fig. 11. Performance of the k-NN algorithm on data sets with various misleading dimensions ($k = 1$)

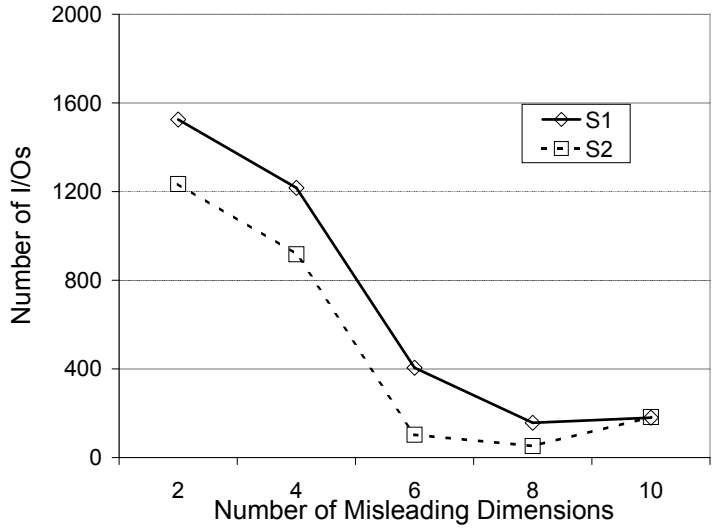

Fig. 12. Performance of the k-NN algorithm on data sets with various misleading dimensions ($k = 5$)

benefits when using heuristic version S2 increase as well.

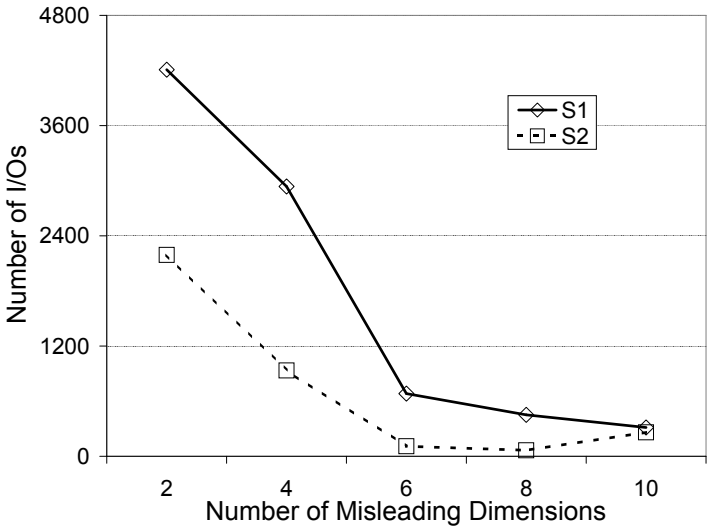

Fig. 13. Performance of the k-NN algorithm on data sets with various misleading dimensions ($k = 10$)

4.5 Verification of Performance Model

Our theoretical performance estimation model was also verified against our uniform synthetic experimental data. We conducted experiments using 10 dimensional data with an alphabet size of 6. The minimum leaf node utilization of our ND-tree was set at 0.4 and the minimum non-leaf node utilization was set at 0.3. We compared our theoretical values to the observed ND-tree performances for databases varying in size from $400K$ vectors to $2.0M$ vectors in $400K$ increments. We also varied the value of k to observe its effects upon the results.

Figures 14 through 16 show the estimated number of I/Os predicted by our performance model, with the actual I/O. The results indicate that our model is quite accurate, estimating the performance within 2% of the actual performance for most test cases. The greatest disparity between estimated and actual performance values occurs in the test cases with small datasets, particularly when searching for only a single neighbor. However, as the size of the dataset increases or as the number of neighbors searched for increases, the performance estimation becomes increasingly accurate.

The above results show that, for both synthetic and genomic uniform data, our k-NN searching algorithm based on the GEH distance far outperforms the linear scan. Additionally, our algorithm outperforms the linear scan for synthetic skewed data. Only when the dimensionality of the underlying NDDS begins to grow excessively, does the benefits of our algorithm start to become less significant. This is a result of the well-known dimensionality curse problem. Further, our performance model provides an accurate estimation of the number of I/Os incurred while performing a k-NN search.

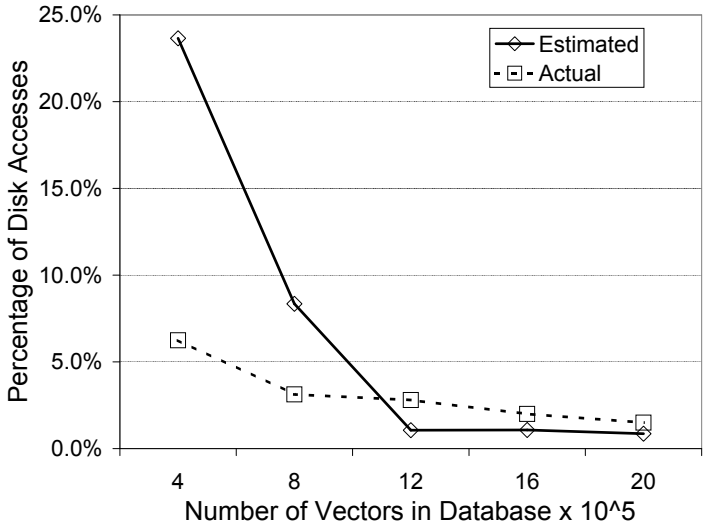

Fig. 14. Estimated and Actual performance of the k-NN algorithm vs. the linear scan on synthetic data sets with various sizes ($k = 1$)

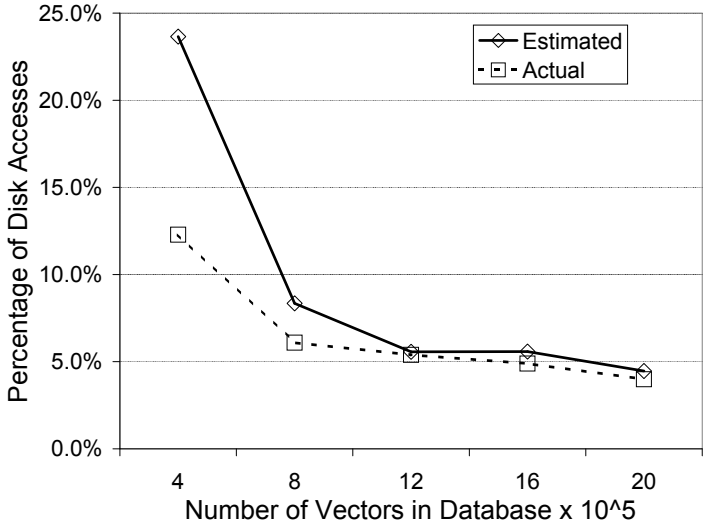

Fig. 15. Estimated and Actual performance of the k-NN algorithm vs. the linear scan on synthetic data sets with various sizes ($k = 5$)

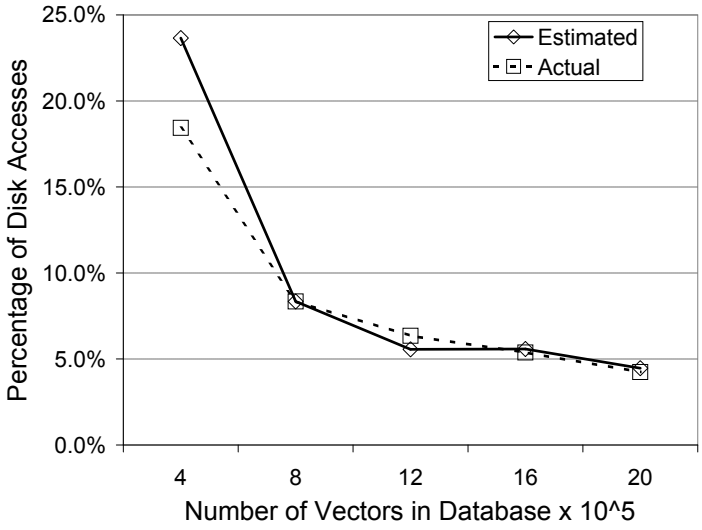

Fig. 16. Estimated and Actual performance of the k-NN algorithm vs. the linear scan on synthetic data sets with various sizes ($k = 10$)

5. CONCLUSIONS

We observe that the issue of k-NN searching in NDDSs is not a simple extension of its counterpart in CDSs and is still an open issue. A major problem with a k-NN search in NDDSs using the conventional Hamming distance is the non-determinism of its solution. That is, there usually is a large number of candidate solutions available. This is mainly caused by the coarse granularity of measurement offered by the Hamming distance. To tackle this problem, we introduce a new extended Hamming distance, i.e., the GEH distance. This new distance takes the semantics of matching scenarios into account, resulting in an enhanced granularity for its measurement. Further, it is proven that the GEH distance possesses the triangular property and therefore may be used in index based pruning heuristics.

To support efficient k-NN searches in NDDSs, we propose a searching algorithm utilizing the ND-tree [Qian et al. 2003; Qian et al. 2006a]. Based on the characteristics of NDDSs, three effective searching heuristics are incorporated into the algorithm. A fourth heuristic is provided that implements a new strategy for probability based search ordering in conservative search scenarios. Further, we provide a performance model to predict the number of I/Os incurred during a k-NN search, using our algorithm, that is based upon the number of neighbors desired and the dimensionality and alphabet size of the data set.

Our extensive experiments demonstrate that our GEH distance measure provides an effective semantic discriminating power among the vectors to mitigate the non-determinism for k-NN searches in NDDSs. Experiments also show that the k-NN searching algorithm is efficient in finding k-NNs in NDDSs, compared to the linear scan method. An implementation of our algorithm, utilizing an M-Tree,

also performed well compared to the linear scan method, but was outperformed by our ND-Tree implementation as the database size grew larger. The algorithm is scalable with respect to the database size and also performs well over non-uniform data distributions. However, when the number of dimensions is high, our algorithm seems to suffer the same dimensionality curse problem as the similar techniques in continuous data spaces.

Our future work involves further performance enhancements for k-NN searches in NDDSs. This includes investigating other useful extensions of the Hamming distance, such as capturing the semantics of mismatching scenarios and applying dynamic semantics to matching scenarios. Additionally, we will continue to investigate the underlying characteristics of NDDSs, such as alphabet size and dimensionality, that can be used in future search heuristics and explore how these techniques may be applied to k-NN searches in hybrid data spaces, which include both continuous and discrete dimensions.

ACKNOWLEDGMENTS

The authors of this paper wish to extend their gratitude towards Gang Qian for his help in developing experimental programs. The preliminary work of this paper was presented at the 23rd International Conference on Data Engineering {ICDE'07}, Istanbul, Turkey, April 15-20, 2007 [Kolbe et al. 2007].

REFERENCES

Badel, A., Mornon, J., and Hazout, S. 1992. Searching for geometric molecular shape complementarity using bidimensional surface profiles. *Journal of Molecular Graphics 10,* 4, 205–211.

Beckmann, N., Kriegel, H.-P., Schneider, R., and Seeger, B. 1990. The r*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990*, H. Garcia-Molina and H. V. Jagadish, Eds. ACM Press, Atlantic City, NJ, U.S.A, 322–331.

Berchtold, S., Keim, D. A., and Kriegel, H.-P. 1996. The X-tree: An index structure for high-dimensional data. In *Proceedings of the 22nd International Conference on Very Large Databases*, T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, Eds. Morgan Kaufmann Publishers, San Francisco, U.S.A., 28–39.

Bookstein, A., Kulyukin, V. A., and Raita, T. 2002. Generalized hamming distance. *Information Retrieval 5,* 4, 353–375.

Chávez, E., Navarro, B., Baeza-Yates, R., and Marroquín, J. L. 2001. Searching in metric spaces. *ACM Computing Surveys 33,* 3, 273–321.

Ciaccia, P., Patella, M., and Zezula, P. 1997. M-tree: An efficient access method for similarity search in metric spaces. In *VLDB '97: Proceedings of the 23rd International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 426–435.

Guttman, A. 1988. *R-trees: a dynamic index structure for spatial searching.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Hamming, R. 1950. Error-detecting and error-correcting codes. *Bell System Technical Journal 29,* 2, 147–160.

Henrich, A. 1998. The lsdh-tree: An access structure for feature vectors. In *ICDE '98: Proceedings of the Fourteenth International Conference on Data Engineering*. IEEE Computer Society, Washington, DC, USA, 362–369.

Hjaltason, G. and Samet, H. 2000. Incremental similarity search in multimedia databases.

Kolahdouzan, M. and Shahabi, C. 2004. Voronoi-based k nearest neighbor search for spatial network databases. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*. VLDB Endowment, Toronto, Canada, 840–851.

Kolbe, D., Zhu, Q., and Pramanik, S. 2007. On k-nearest neighbor searching in non-ordered discrete data spaces. In *ICDE*. IEEE, Istanbul, Turkey, 426–435.

Qian, G. 2004. Principles and applications for supporting similarity queries in non-ordered-discrete and continuous data spaces. Ph.D. thesis, Michigan State University, East Lansing, Michigan, United States.

Qian, G., Zhu, Q., Xue, Q., and Pramanik, S. 2003. The nd-tree: a dynamic indexing technique for multidimensional non-ordered discrete data spaces. In *vldb'2003: Proceedings of the 29th international conference on Very large data bases*. VLDB Endowment, Berlin, Germany, 620–631.

Qian, G., Zhu, Q., Xue, Q., and Pramanik, S. 2006a. Dynamic indexing for multidimensional non-ordered discrete data spaces using a data-partitioning approach. *ACM Trans. Database Syst. 31,* 2, 439–484.

Qian, G., Zhu, Q., Xue, Q., and Pramanik, S. 2006b. A space-partitioning-based indexing method for multidimensional non-ordered discrete data spaces. *ACM Trans. Inf. Syst. 24,* 1, 79–110.

Robinson, J. T. 1981. The k-d-b-tree: a search structure for large multidimensional dynamic indexes. In *SIGMOD '81: Proceedings of the 1981 ACM SIGMOD international conference on Management of data*. ACM, New York, NY, USA, 10–18.

Roussopoulos, N., Kelley, S., and Vincent, F. 1995. Nearest neighbor queries. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995*, M. J. Carey and D. A. Schneider, Eds. ACM Press, San Jose, California, U.S.A., 71–79.

Seidl, T. and Kriegel, H.-P. 1997. Efficient user-adaptable similarity search in large multimedia databases. In *VLDB '97: Proceedings of the 23rd International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 506–515.

Seidl, T. and Kriegel, H.-P. 1998. Optimal multi-step k-nearest neighbor search. *SIGMOD Rec. 27,* 2, 154–165.

A. RECURSIVE GENERATION OF A SOLUTION SET

Proposition 2.2.3 states that the set of candidate *k*NNSs given by Definition 2.2.1 can be produced by Definition 2.2.2. This leads to the hypothesis that if a solution set $NN = \{A_1, A_2, \ldots, A_{k-1}\}$ satisfies Equation 3 for $k-1$ objects, then Equation 4 will select a k^{th} neighbor A_k such that $NN \cup A_k$ will satisfy Equation 3 for k objects and thus be consistent with Definition 2.2.1.

We first consider a base case where $k = 1$. Equation 4 yields the following neighbor A_1:

$$A_1 \in \{A : \forall_B \left(D(q, A) \leq D(q, B)\right)\}.$$

The solution set $\{A_1\}$ satisfies Equation 3 for $k = 1$ and thus is consistent with Definition 2.2.1. Equation 4 may then be used again to yield neighbor A_k:

$$A_k \in \left\{ A : \forall_B \left(\begin{array}{l} D(q, A) \leq D(q, B) \wedge \\ B \notin \{A_1, A_2, \ldots, A_{k-1}\} \wedge \\ A \notin \{A_1, A_2, \ldots, A_{k-1}\} \end{array} \right) \right\} \text{ for } k \geq 2\,.$$

The above function returns the object in the dataset that has a minimum distance from the query object out of all objects in the dataset not currently included in the solution set. Thus the addition of a k^{th} neighbor for $k \geq 2$ will result in a minimal distance to the query point for k objects if the set of neighbors A_1 - A_{k-1} has a minimal distance for $k-1$ objects. Our base case shows that this is true for 1 object, thus the hypothesis is true for all values of k.

B. DERIVATION OF THE NUMBER OF CANDIDATE SOLUTION SETS

This section provides the derivation of the number Δk of candidate *k*NNSs. This may be interpreted as the number of possible solution sets from a dataset that satisfy Equation 3 for k objects. The value of Δk is largely influenced by the number of objects within a given solution set that have the same distance to the query object as the k^{th} neighbor. This value, represented by t, is formally defined as follows:

$$D(q, A_{k-t}) \neq D(q, A_{k-t+1}) = D(q, A_{k-t+2}) = \ldots = D(q, A_k).$$

Each neighbor A_{k-t} - A_k may be replaced by any other potential neighbor from the dataset α, where $D(q, \alpha) = D(q, A_k)$, and the solution set will still satisfy Equation 3. We denote the set of all potential neighbors as $N^{'}$. Thus, Δk is the number of t-element subsets that can be composed from the set of $N^{'}$. This can be represented as the binomial coefficient of t and $N^{'}$ which decomposes into Equation 5:

$$\Delta k = \binom{N^{'}}{t} = \frac{N^{'}!}{t!(N^{'} - t)!}.$$

C. TRIANGULAR PROPERTY OF GEH

Section 2.3 proposed that the GEH distance possessed the triangular inequality property. The proof of this proposition is divided into two steps. First, a base case is established where the property holds. Second, the base case is evolved into the generic case where the property still holds.

Step 1: Assume that $D_{GEH}(V_A, V_C)$ is maximum (i.e. $\forall x : D_{GEH}(V_A, V_C) \geq D_{GEH}(V_A, V_x)$, thus every element in V_A and V_C must be different. From Equation 7, we have:

$$D_{GEH}(V_A, V_C) = d.$$

where d is the number of dimensions of the underlying NDDS. Now assume that $Dist(V_A, V_B)$ is minimal (i.e. $\forall x : D_{GEH}(V_A, V_C) \leq D_{GEH}(V_A, V_x)$, thus every element in V_B must equal the corresponding element in V_A, i.e. $V_A = V_B$. From Equation 7, we have:

$$D_{GEH}(V_A, V_B) = x, \text{ where } 0 \leq x < 1.$$

The term x represents the summation of each of the frequency values obtained using Equation 7 and then dividing by d. Since $V_A = V_B$ and $D_{GEH}(V_A, V_C) = d$, we have:

$$D_{GEH}(V_B, V_C) = d.$$

Since $x + d \geq d$, we have the following inequality:

$$D_{GEH}(V_A, B_V) + D_{GEH}(V_B, V_C) \geq D_{GEH}(V_A, V_C).$$

Step 2: The second step involves three sub-steps; one for each vector that needs to become generic.

Step 2.1: The first step is to evolve V_B into a generic vector, starting with the initial vectors from Step 1: $D_{GEH}(V_A, V_B) = x$ where $x \leq x < 1$; $D_{GEH}(V_B, V_C) = d$; $D_{GEH}(V_A, V_C) = d$.

To make V_B generic, we apply n changes. Each change is represented by switching an element in V_B away from its original element. After these n have been done, we are left with the following distances:

$$\begin{aligned} D_{GEH}(V_A, V_B) &= x + n - c_1, \\ D_{GEH}(V_B, V_C) &= d - n_1^* + c_2, \\ D_{GEH}(V_A, V_C) &= d, \end{aligned}$$

where c_1 represents the culmination of adjustment values from each of the n elements switched; n_1^* represents the number of elements switched that now equal their corresponding element in V_C; c_2 represents the culmination of the adjustment values to be added due to these newly matching elements.

Here, we know that $n \geq n_1^*$, $x \geq c_1$, and $c_2 \geq 0$. Using these inequalities it can be shown that the distance measures still satisfy the triangular inequality property.

Step 2.2: The next step is to evolve V_C into a generic vector, starting with the final vectors from Step 2.1 and applying j changes to V_C. This leaves us with the following distances:

$$\begin{aligned} D_{GEH}(V_A, V_B) &= x + n - c_1, \\ D_{GEH}(V_B, V_C) &= d - n_1^* + c_2 + (j_1^* - c_3) - (j_2^* - c_4), \\ D_{GEH}(V_A, V_C) &= d - j_3^* + c_5, \end{aligned}$$

where j_1^* and j_2^* represent the integer values that $D_{GEH}(V_B, V_C)$ increases and decreases, respectively, as elements are switched; c_3 and c_4 represent the adjustment values due to those changes; j_3^* and c_5 represent the integer and adjustment changes to $D_{GEH}(V_A, V_C)$ due to the element changes. Note that every time j_2^* is incremented there are two possibilities: either the element being switched in V_C becomes a value in V_B that still matches V_A, in which case j_3^* is incremented by one and both c_4 and c_5 are incremented by the same amount; or it becomes a value in V_B that does not match V_A, which means that $n \geq n_1^* - 1$. This leaves us to note that $j_2^* + n_1^* \leq j_3^* + n$ and that $c_4 \geq c_5$. Finally, with $j_1^* \geq c_3$, it can be shown that these distance measures still satisfy the triangular inequality property.

Step 2.3: The final step is to evolve V_A into a generic vector. This is actually a trivial step, because V_A was only defined in relation to the original vectors V_C and V_B, and because V_C and V_B can be transformed into any general vectors from their starting points, we can start V_A as any vector we wish. Thus V_A is a generic vector and the triangular inequality property holds true for any three vectors V_A, V_B, and V_C.