This is to certify that the
dissertation entitled

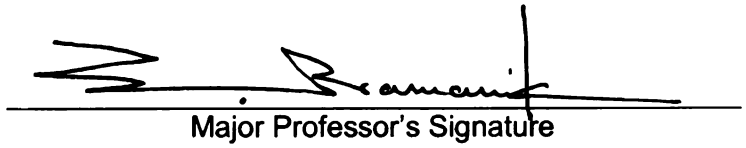
Techniques for Efficient *k*-Nearest Neighbor Searching in Non-Ordered Discrete and Hybrid Data Spaces

presented by

Dashiell Matthews Kolbe

has been accepted towards fulfillment
of the requirements for the

Doctoral degree in Computer Science



Major Professor's Signature

5/12/2010

Date

TECHNIQUES FOR EFFICIENT K-NEAREST NEIGHBOR SEARCHING IN NON-ORDERED DISCRETE AND HYBRID DATA SPACES

By

Dashiell Matthews Kolbe

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Computer Science

2010

ABSTRACT

TECHNIQUES FOR EFFICIENT K-NEAREST NEIGHBOR SEARCHING IN NON-ORDERED DISCRETE AND HYBRID DATA SPACES

By

Dashiell Matthews Kolbe

Similarity searches/queries in Non-Ordered Discrete Data Spaces (NDDS) and Hybrid Data Spaces (HDS) are becoming increasingly useful in areas such as bioinformatics, multimedia, text retrieval, audio and video compression, data-mining, and E-commerce. The objective of this dissertation is to develop and analyze novel methods to support similarity searches in such spaces.

In this dissertation, we first discuss performing k-Nearest Neighbor (k-NN) searches in NDDSs. Performing such searches in NDDSs raises new challenges. Due to the coarse granularity of the commonly used Hamming distance measure, a nearest neighbor query in an NDDS may lead to a large set of candidate solutions, creating a high degree of non-determinism. We propose a new distance measure that reduces the number of candidate solutions for a query while preserving the essential properties of Hamming distance. We have also implemented nearest neighbor queries using multidimensional database indexing in NDDSs. 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 demonstrate that our nearest neighbor algorithm is efficient in finding nearest neighbors in NDDSs and that our heuristics are effective in improving the performance of such queries.

We then discuss our work on providing a generalization of our GEH distance. This generalized form allows our distance measure to be applied to a broad range of applications. Of these, we discuss a new rank based implementation well suited to applications with heavily skewed data distributions. Our experiments demonstrate the benefits of an adaptable distance metric by presenting scenarios that demonstrate performance changes depending upon the distance measure used.

Finally, we discuss extending k-NN searching to HDS. We consider the challenges of exploiting both the CDS and NDDS properties of HDS for optimizing potential search algorithms. In particular we consider how key search information is maintained in HDS data structures and what rules must be observed to guarantee the correctness of search results in such spaces. Further, the concept of search execution stages is introduced to develop efficient k-NN search algorithms for HDS. Lastly, a theoretical performance model is developed for HDS searching to validate our experimental results.

To my parents, who always believed in me.

ACKNOWLEDGMENTS

I would like to first acknowledge my thesis advisor, Dr. Sakti Pramanik, who has provided a strong guiding hand throughout my graduate career at Michigan State University. My growth as a researcher would not have been possible without Dr. Pramanik.

I would also like to provide special acknowledgement for Dr. Qiang Zhu of the University of Michigan. Many of the ideas presented in this thesis are the result of discussions with Dr. Zhu and Dr. Pramanik. The level of collaboration that was achieved in these discussions is something that I continue to strive for in my daily life.

I also extend my sincere gratitude to my thesis committee, Dr. Mark Dykman, Dr. James Cole, and Dr. Rong Jin. They provided both their time and expertise to improve both the depth and breadth of this thesis.

Lastly, I would like to thank my family and friends for being my village. My parents, Nancy, Chris, and Ted, recieve my deepest gratitude for their undending love and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

Nearest neighbor searches/queries in Non-Ordered Discrete Data Spaces (NDDS) and Hybrid Data Spaces (HDS) are becoming increasingly useful in areas such as bioinformatics, multimedia, text retrieval, audio and video compression, information security, data-mining, and E-commerce. This form of searching may be stated as the following problem: given a set S of n data points in an a dataset X (with $|X| \geq k$) and a query point $q \in X$, return a set of $k > 0$ objects $A \subseteq X$ where $\forall_{u \in A, v \in S-A} : D(q,u) \leq D(q,v)$ and $|A| = k$. Examples of such queries are "finding the k closest restaurants to the intersection of Fifth and Main," and "finding the k fixed length words that differ the least from the word 'near'."

Numerous techniques have been proposed in the literature to support efficient nearest neighbor queries in continuous (ordered) data spaces (CDS) and general metric spaces. Excellent surveys of both the theoretical and practical aspects of

nearest neighbor searching in such spaces have been presented by Chavez et al. [14] and Hjaltason and Samet [27] respectively. Little work has been reported on supporting efficient similarity searches in either NDDSs or HDSs. 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. A d-dimensional HDS is a Cartesian product of $m (m \leq d)$ dimensions/alphabets consisting of finite non-ordered elements/letters and $n = d - m$ continuous dimensions. For example, consider sports related data containing match win statistics and match locations. Match win statistics could be considered continuous data while match locations could be considered non-ordered discrete data.

The remainder of this section is comprised as follows: Section 1.1 discusses motivating applications; Section 1.2 introduces space partitioning concepts considered throughout this dissertation; Section 1.3 provides an overview of the research presented in this dissertation.

1.1 Motivating Applications

This section presents a sample of applications that rely upon performing efficient k-NN similarity searches in CDSs, NDDSs, and HDSs. Both fixed length and variable length data applications are presented for completeness.

1.1.1 Multimedia Objects

Facial recognition algorithms are used by a wide variety of applications such as law enforcement and security to identify an object by a live or still image. This is achieved by comparing a selected set of features from the image against similar feature sets stored in a database. This is equivalent to creating a feature vector composed of HDS objects.

Finger print matching and voice recognition are both handled in a similar fashion. For each, a feature vector is established, containing values representing key structures within the object, such as whorls in a fingerprint, which is then compared with an indexed set of feature vectors to determine the closest matches. Here, it is common practice to try to determine the single closest match, ideally considered an exact match. However, due to approximation involved in creating the feature vectors, nearest matching is a much more practical and efficient approach.

1.1.2 Computational Biology

Biological data, particularly genome sequences, may be represented as a d dimensional vector α, with each dimension i ($1 \leq i \leq d$) containing a value from the set of possible elements A_i for the i^{th} dimension. In genome sequencing, the alphabet over all dimensions is the same: $A = \{a, g, t, c\}$, such that a 25-dimensional segment of a genome strand could be represented in an NDDS Ω_{25} as $\alpha =$ "$agtcaagtcaaatccagtcaatcca$".

Initially, searching in genome sequence databases employed editor distance metrics or substitution matrices, such as BLOSUM, to determine the similarity between genome sequences. Lately, however, the Hamming distance (discussed in Section 1.2) has become more popular for searching in large genome sequence databases [29], [54]. Unfortunately, the Hamming distance has a poor level of granularity in measurement which can lead to unclear results. We consider this issue in depth in Chapter 3.

1.1.3 Text Retrieval

Text objects are typically represented as strings of varying length. Several index methods have been proposed to support efficient searches over such data. Examples of this method are Tries [30], [18], the Prefix B-tree [5], and the String B-tree [19].

Most indexing methods in this category, such as Prefix B-tree and String B-tree, are designed specifically for exact searches rather than similarity searches. The Tries model does support similarity searches, but is difficult to apply to large databases due to its memory-based feature.

Text retrieval operations are however fundamentally different from those in other applications listed in this section. Datasets of previously listed applications, such as genome sequences, are composed of objects each represented by a string/vector of fixed length/size. Due to this, geometric properties such as area and range can be applied to the dataset. Text retrieval operations however, cannot take advantage of these geometric properties due to text objects being composed of varying length strings. This paper focuses upon searching in datasets of fixed dimensionality and thus does not consider applications towards efficient text retrieval. The issue of text retrieval is covered in detail in [45].

1.1.4 Document Retrieval

Document retrieval is used extensively in World Wide Web and database applications. Traditionally, a test document d_i is classified by creating a feature vector containing the percent composition of certain key words through-out the document. For example, if a document is being classified by only three key words *'river'*, *'bank'*, and *'sediment'*, a feature vector $V = \{0.5, 0.3, 0.2\}$ might be generated to identify

that out of the key words found in the document, 50% are *'river'*, 30% are *'bank'*, and 20% are *'sediment'*. This vector would then be compared to a dataset U_T composed of similar feature vectors generated from a training/representative set of documents.

Most work in this field has focused upon improving construction of feature vectors [36] and choosing an optimal number of neighbors to search for [4]. However, [4] showed how the current standard method of cosine similarity measurements sometimes provides an intuitively incorrect similarity match between feature vectors. This phenomenon results from the feature vectors themselves being only an approximation of key word occurrences within a document and pays little attention to how those words are related to each other. A possible solution to this problem is to calculate the similarity between two documents based upon a discrete representation of their key words.

1.2 Space Partitioning Concepts

This section discusses common data space environments that k-NN searches are performed in. Continuous Data Space, in the form of General Metric Space, is discussed first. This is followed by a discussion of Non-Ordered Discrete Data Space. Next, the combination of NDDS and CDS is presented as Hybrid Data Space. Finally, the concept of vector space distance measurement is introduced.

1.2.1 Metric Space

Let U denote the universe of valid objects being searched. The function $D : U \times U \rightarrow R$ denotes a measure of distance between objects. To be used in search and indexing applications, such distance functions generally must satisfy the following properties.

Positiveness:	$\forall x, y \in U, D(x, y) \geq 0$
Symmetry:	$\forall x, y \in U, D(x, y) = d(y, x)$
Reflexivity:	$\forall x \in U, D(x, x) = 0$
Strict Positiveness:	$\forall x, y \in U, x \neq y \Rightarrow D(x, y) > 0$
Triangular Inequality:	$\forall x, y, z \in U, D(x, y) \leq d(x, z) + D(z, y)$

If a distance measurement D satisfies the Triangular property , then the pair (U, D) is called a *metric space*. If D does not satisfy the Strict Positiveness property, the space is called a *pseudo-metric space*. If the Symmetry property is not fulfilled, the space is generally described as a *quasi-metric space*; the case of a city map containing one way streets is a common example of such a space.

From the above definitions, it can be inferred that most spaces are in fact specialized implementations of the general metric space. Specifically, if the elements in the metric space are tuples of real numbers, then the space may be described as a *finite-dimensional vector space*[40]. Such spaces generally make use of geometric distance measures; such as Euclidean or Manhattan. When such tuples represent

numbers from a continuous (ordered) dataset, the pair (U, D) is considered a CDS.

1.2.2 Non-Ordered Discrete Space

Discrete space is based upon the concept of all objects in the universe of discourse U being discrete and non-ordered along each dimension. Ordered discrete objects typically demonstrate the same properties as CDS objects and thus, are not considered in this section. Examples of such non-ordered discrete data are multimedia objects, profession, gender, bioinformatics, and user-defined types. Each of these examples may be represented as a feature vector in a d-dimensional space. Consider, in genome sequence databases such as GenBank, sequences with alphabet $A = \{a, g, t, c\}$ are broken into substrings of fixed-length d for similarity searches [29], [54]. Each substring can be considered as a vector in d-dimensional space. For example, substring *aggctttgcaaggctttgcagcact* is a vector in the 25-dimensional data space, where the i^{th} character is a letter chosen from alphabet A in the i^{th} dimension. In this example, a is no closer to c than it is to t and so forth. Thus, mapping discrete space into continuous space by applying a form of ordering changes the semantics of the dataset. A formal definition of a NDDS is as follows.

Let A_i, $(1 \leq i \leq d)$ be an alphabet consisting of a finite number of non-ordered elements. A d-dimensional NDDS Ω_d, is defined as the Cartesian product of d alphabets: $\Omega_d = \{A_1 \times A_2 \times \ldots \times A_d\}$. A_i represents the i^{th} dimension alphabet of

Ω_d. The area of space Ω_d is defined as:

$$area(\Omega_d) = \prod_{i=1}^{d} |A_i|.$$

This formula also indicates the number of possible unique vectors in Ω_d. A vector α is defined by the tuple $\alpha = (\alpha[1], \alpha[2], \ldots, \alpha[d_i])$, where $\alpha[j] \in A_i$. A discrete rectangle R in Ω_d is defined as the Cartesian product: $R = \{S_1 \times S_2 \ldots \times S_d\}$, where S_i is a set of elements/letters from the alphabet of the j-th dimension of the given d-dimensional NDDS. The length of the i^{th} dimension edge of R is defined as:

$$length(R, i) = |S_i|.$$

The area of R follows the formula for the area of the data space as:

$$area(R) = \prod_{i=1}^{d} |S_i|.$$

1.2.3 Hybrid Space

Hybrid space is composed of a combination of NDDS elements and CDS elements. Consider domain $D_i (1 \leq i \leq d)$. If D_i is comprised of non-ordered discrete elements,

it corresponds to an alphabet A_i. If D_i is comprised of continuous elements, it corresponds to some continuous span. For the purposes of our discussion, we will consider this span to be normalized within $[0, 1]$. A d-dimensional hybrid data space (HDS) Ω_d is defined as the Cartesian product of d such domains:

$$\Omega_d = D_1 \times D_2 \times \ldots \times D_d.$$

As described by Qian [41], a vector in Ω_d is comprised of the tuple $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_d)$, where $\alpha_i \in D_i (1 \leq i \leq d)$. Subdomains within D_i are defined as $S_i \subseteq D_i (1 \leq i \leq d)$, where $S \subseteq A_i = D_i$ if D_i is a non-ordered discrete domain, or S_i is a range $[\min_i, \max_i]$ if D_i is a continuous domain. A hybrid rectangle is then defined as the Cartesian product of domains S_i thus that:

$$R = S_1 \times S_2 \times \ldots \times S_d.$$

The length of the i^{th} dimension edge of R is defined as:

$$length(R_i) = \left\{ \begin{array}{ll} \frac{|S_i|}{|A_i|} & \text{if dimension } i \text{ is non-ordered discrete} \\ \max_i - \min_i & \text{otherwise} \end{array} \right\}.$$

The area of R is defined as:

$$area(R) = \prod_{i=1}^{d} length(R_i).$$

1.2.4 Vector Space Distance Measurements

Determining an efficient distance measurement is still an open problem. As discussed by Qian [41], the vector model is widely used to support similarity queries. Additional models, such as the Boolean model and probabilistic model [2], have also been proposed to support similarity queries. However, as discussed by Baeza et. al. [2], the vector model is considered to be more effective for similarity searches.

To perform searches, each object in the database, as well as the query object are represented as fixed length vectors. For example, consider security applications tracking intrusion attempts. Each attempt in the database can be transformed into a vector based upon time, frequency, intruder ID, intrusion method, and other intrusion characteristics. Each intrusion can now be considered a point in a multidimensional vector space (either CDS, NDDS, or HDS dependent upon the feature generation methods). The distance values between each pair of vector objects may now be calculated utilizing a distance metric most suitable for such a space. Typically, a pair of objects with a minimal distance value between them are considered more similar than a pair of objects with a maximal distance value between them.

The focus of our research is on supporting efficient similarity queries using the vector space model. It should be noted that not all applications have objects that may be efficiently represented as natural vectors. Qian [41] notes that some forms of multimedia data, such as video clips, are often stored in their native format due to a loss of precision when generating a feature vector to represent them. Although not considered in detail here, the issue of designing an effective feature generation algorithm is still an open problem. Excellent surveys concerning feature generation for multimedia objects are presented in [1] and [49].

1.3 Overview of the dissertation

The remainder of this dissertation is organized as follows: Chapter 2 discusses previous work performed in this and related fields, including multidimensional indexing and similarity search algorithm development; Chapter 3 presents our research in developing novel similarity search methods for NDDSs. Chapter 4 presents our work in developing a non-application specific distance measure for NDDSs. Chapter 5 presents our research in extending our work in similarity searches in NDDSs to HDSs. Chapter 6 summarizes the contributions of this dissertation and provides directions for future work.

CHAPTER 2

Previous Work

This chapter presents previous research work related to this dissertation. We first discuss the evolution of common similarity searching algorithms. We then present an overview of popular index structures used to support efficient similarity searches in vector spaces. These index structures are used to maintain data in CDSs, NDDSs, as well as recently proposed work to support HDSs. Lastly, we present work concerning distance measure comparisons.

2.1 Nearest Neighbor Algorithms

Most similarity searching algorithms may be distilled to a simple formula applicable to CDS, NDDS, or HDS index structures. Further, range queries may be viewed as a special case of nearest neighbor queries where the final search radius is known at the start of the search. As such, this section focuses upon nearest neighbor search

algorithms. The following algorithms perform similarity searches for the nearest neighbor to a query based on an index structure of fixed dimensionality. Generally, the search for a single nearest neighbor may be expanded to find k nearest neighbors by maintaining a list of neighbors found and using the distance between the neighbor that occupies the k^{th} distance related slot and the query point as a search range/radius value to search within. Each of the index structures described in the following sections may generally be used in conjuncture with one of the following search methods. However, developers will typically modify the algorithm to better suit the applicable structure.

2.1.1 Exact Searching Methods

The most basic method to perform a k-NN similarity search involves using a range search algorithm. Begin with radius $r = \alpha : (\alpha > 0)$ centered at query point q. Increase α until at least k elements lie within the radius. The cost, in terms of page accesses, of this algorithm is similar to that of performing a range search. This cost however is greatly affected by the amount the value α is adjusted by every time the desired number of elements is not yet found. Too small, the performance cost will quickly grow; too large, the number of points returned will far exceed the desired number thus decreasing the usefulness of the solution.

A more elegant approach was proposed for both general metric spaces and CDSs

[12, 3, 47] by beginning the search on any data structure using $r = \infty$. The search begins at the anchor/root of the data structure. Each time the query point q is compared to some element p, the search radius is updated such that $r = min(r, D(q, p))$. The search then backtracks to the nearest split in the index where a new path has not yet been traversed and continues down the new path using the newly calculated radius.

As the search continues, the possibility increases that an entire path of the index structure may not need to be searched due to the decreasing size of the radius r. Pruning heuristics are employed to determine if a certain route is no longer necessary to search. Roussopoulos et al. [47] provided two heuristics for CDS based index structures, namely R-trees, named MINDIST and MINMAXDIST that invoke this pruning. MINMAXDIST is used in the breadth search of the covering rectangle of the current search path by determining the minimum maximum distance of all covering rectangles contained in the current one. This distance is used to update the current search radius such that $r = min(r$,MINMAXDIST). MINDIST is then used in the depth traversal to determine if any point of a covering rectangle associated with a new search path is within the search radius. If no point is within the search radius, that search path is pruned from the remainder of the search and thus the effective size of the database to search is decreased.

This range reduction method may be improved further by attaining a smaller radius value earlier in the search. Several techniques have been used in both CDSs and general metric spaces [51, 28, 16]. The underlying idea of each technique is that certain paths may be identified by their high level statistics that will yield a closer point to the query point q earlier in the search. The most common application of this idea is to order potential search paths by either their MINDIST or MINMAXDIST values to q. The MINDIST ordering gives the optimistic approach that a lower MINDIST value is caused by a relatively closer object in the index structure. This may not always prove true in spatial index structures. Commonly, some point of a search path only exists at the top most layers. At higher levels within an index structure, points may actually be the result of the intersection of lines drawn from several points lower in the index structure. When this technique appears to suffer from this problem, the pessimistic approach using MINMAXDIST may be used instead. Here, search paths are ordered by the increasing value of their furthest point. Thus a search may correctly assume that it will at least not encounter any points further away than the MINMAXDIST.

2.1.2 Approximate Searching Methods

Relaxing the precision of the query results may lead to even further reductions in time complexity. This is a reasonable procedure for many applications due to some

approximation in the modalization of feature vectors for both general metric and CDS indexes. In addition to the query itself, a user specifies some query parameter ϵ to control how far away from the query point the search may progress. In this manner, the algorithm avoids the costly initial stages of a similarity search. On subsequent searches of similar databases, ϵ may decrease to approach zero. As ϵ decreases, the time complexity, along with the precision of the result decreases as well.

A probabilistic algorithm was given for vector spaces by Yianilos et al. [55], using a method described as *aggressive pruning* to improve the performance. Here, the idea is to increase the number of branches that are pruned at the expense of possible points in the set of nearest neighbors. This process is controlled such that the probability of success is always known. Unfortunately, the data structure used is only useful in a very limited radius; in databases or with searches that could result in neighbors with distances beyond the possible radius, the algorithm is not able to guarantee a result of the true nearest neighbors. This form of similarity searching is described as *Approximate Similarity Searching*. This topic is not covered in detail in this paper, but is mentioned here for completeness. An in depth coverage may be found in [53].

2.1.3 Unique Searching Methods

The techniques described thus far cover universal proposals for performing k-NN similarity searches. There are however examples of k-NN search algorithms developed for specific indexes that are inapplicable in a generic sense. Such algorithms depend upon the structure developed to support them and are unable to be incorporated with common indexing techniques. Clarkson [17] proposes a method that alleviates the need to perform extensive backtracking by creating a GNAT-like data structure where points are inserted into multiple subtrees. The tree is constructed by first selecting representatives for the root(s) and then inserting each element u into not only the subtree of its closest representative p, but also into the subtree of any other representative p' such that $D(u, p') \leq 3 * D(u, p)$. During a query on object α_q, the search enters all subtrees such that $D(\alpha_q, p') \leq 3 * D(\alpha_q, p)$. As shown by Clarkson, this is enough to guarantee the retrieval of the nearest neighbor and could be extended to determine the set of k-NN.

2.2 General Metric Space and CDS Index Structures

Numerous data indexing structures have been proposed in the literature that support efficient similarity searches in CDSs and general metric spaces. Most CDS data

index structures may be classified into two categories: data-partitioning and space-partitioning. Examples of the former split an overflow node by partitioning the set of its indexed data objects. The later, split an overflow node by partitioning its representative data space (typically via a splitting point in a dimension). Most of these methods are not applicable in either NDDS or HDS due to some essential geometric concepts such as a bounding box no longer being valid in such spaces. An in depth discussion of continuous multidimensional indexing structures may be found in [21].

Metric trees represent a wholly different approach to indexing data. Here, data points are not represented in any type of dimensional space, rather metric trees implement structures using only the distance information between data points and some center origin/root. Such non-geometric trees are generally not optimized toward NDDSs or HDSs, such as CDS implementations are toward CDSs. However, they do provide a prevalent counter solution to implementing a vector space model for either an NDDS or HDS. An in depth discussion of searching in metric space may be found in [40].

This section presents an overview of commonly employed index structures for both CDS and general metric space. Chapter 5 discusses some of the issues that arise when applying CDS based index structures to an HDS dataset.

2.2.1 KD-Tree

The KD-tree [8, 7], was one of the first proposed d-dimensional data index structures. Structured as a binary tree, the KD-tree recursively subdivides a dataset along $(d-1)$-dimensional hyperplanes whose direction alternates among the d possibilities. A simple example is for $d = 3$, the splitting hyperplanes would be perpendicular to the x, y, and z axes. Each hyperplane must contain at least one point where interior nodes have one or two descendants. Searching and insertion of new points are simple procedures that use the interior nodes as guide posts. Deletion however, is much more complicated and invariably results in a costly reorganization of the tree structure.

The KD-tree structure is highly sensitive to the order in which the points are inserted. A poor insertion order generally results in data points being scattered all over the tree. A solution to this problem was presented by Bentley et al. as the adaptive KD-tree [9]. By relaxing the requirements that hyperplanes have to contain a data point as well as being strictly alternating, the adaptive KD-tree was able to choose splits that resulted in a much more even data distribution on both sides of a particular branch. Conceptually, this resulted in pushing all data points out to the leaf nodes leaving interior nodes to only contain dimensional information. The adaptive KD-tree is unfortunately very static in nature; frequent insertions and deletions require costly reorganizations to keep the tree balanced. However, the

adaptive KD-tree does perform well when the dataset is known prior to construction of the tree.

2.2.2 LSD-Tree

The Local Split Decision (LSD) Tree [24] is organized as an adaptive KD-tree, partitioning a dataset into disjoint cells of various sizes. Better adapted to the distribution of a dataset than fixed binary partitioning, the LSD-tree uses a special paging algorithm to preserve the external balancing property; i.e., the heights of its external subtrees differ by at most one. This is accomplished by paging out subtrees when the structure becomes too large to fit in main memory. While this strategy increases the efficiency of the tree structure it prevents the LSD-tree from being used in general-purpose database systems; where such extensive paging is not available.

The split strategy of the LSD-tree tries to accommodate skewed data by combining data-dependent as well as distribution-dependent split strategies; SP_1 and SP_2 respectively. The former attempts to achieve the most balanced structure by trying to keep an equal number of data objects on both sides of the split. The later is performed at a fixed dimension and position where the underlying data is in a known distribution pattern. Determining the split position SP, is the linear combination of applying one of the strategies: $SP = \alpha SP_1 + (1 - \alpha)SP_2$, where $\alpha = (0, 1)$. The factor α may vary as objects are inserted or deleted from the tree. This property

increases the efficiency of the LSD tree, but makes integration with generic database systems of unknown data distributions difficult.

2.2.3 R-Tree and R*-tree

The R-Tree [22] is a height-balanced tree with index records/pointers in its leaf nodes, similar to the adaptive KD-tree. Each node v represents a disk page along a d-dimensional interval $I^d(v)$. If the node is an interior node, all descendants v_i of v are contained in the interval $I^d(v)$. If the node is a leaf node, then $I^d(v)$ represents the d-dimensional bounding box of all the objects stored in the node. Nodes at the same level may overlap area. Each node contains between m and M entries unless it is the root. The lower bound m, is used to prevent the degeneration of trees and ensure efficient storage utilization. If the number of entries in a node drops below m, the node is deleted and the descendants of the node are reinserted into the tree, (tree condensation). The upper bound M is used to maintain each node's size to that of one disk page. Being a height-balanced tree, all of the leaves are at the same level and the height is at most $\lceil log_m(N) \rceil$ for $N : (N > 1)$ index records.

Objects in an R-Tree are represented by their Minimum Bounding Interval $I^d(o)$. To insert an object, the tree begins at the root and traverses a single depth first path to a leaf node. At each depth layer, heuristics are used to determine which descendant path to follow. The first is a calculation of which path would require

the least enlargement of area to the interval $I^d(v_i)$ representing it. If there are multiple paths that satisfy this criterion, Guttman et al. [22] proposed selecting the descendant associated with the smallest interval. This process continues until a leaf node is reached and the object pointer is placed within. If this results in an expansion of the leaf's interval, the interval change propagates upwards toward the root. If insertion results in the number of objects in the leaf node exceeding M, the leaf node is split, distributing the entries between the old and new disk pages. This change also propagates upwards toward the root.

Deletion is handled in a similar manner to insertion. First an exact match search is performed to determine if the selected element for deletion is contained in the database. If so, the element is deleted, and the containing leaf node is checked to see if the area interval may be reduced. If the area is reduced, the change is propagated upwards toward the root node. If this deletion causes the number of elements to drop below the lower bound m, the remaining elements in the leaf node are copied to temporary storage and then deleted from the database, the changes caused by this deletion are again propagated upwards which generally results in the adjustment of several intermediate nodes. Once this is completed, the elements in temporary storage are inserted back into the index structure following the insert method described above.

Searching an R-Tree may be performed in a similar manner to the first stages of insertion. At each index node v, all index entries are checked to see if they intersect with the search interval. If v is an interior node, the search continues to all the descendant nodes v_i who were found to intersect the search interval. If v is a leaf node, the search adds all the entries that intersected the search interval to the solution set. Due to interior nodes overlapping area, it is common for a search to include multiple descendants from interior nodes. In the worst case scenario, this will lead to every single node in the tree having to be accessed.

Several weaknesses in the original tree construction algorithms were identified through careful analysis of R-tree behavior under different data distributions. It was identified that the insertion phase was a critical step in building an R-tree towards good search performance [48]. The result of this study was the R*-tree. The R*-tree provides a more controlled insertion performance by introduced a policy called *forced reinsert*: if a node overflows, it is not split immediately, rather p entries are removed from the node and reinserted into the tree. Through testing, it was proposed in [6] that p should be about 30% of the maximum number of entries per node, M.

Additionally, the R*-tree further addresses the issue of node-splitting by adding more heuristics to avoid making random choices. In Guttman et al.'s original R-tree algorithm, node-splitting policy was based solely on trying to minimize the

area covered by the two new nodes. This lead to many possible splits, whereby a random split was selected. The R*-tree breaks these ties by taking three more factors into account: overlap, perimeter values, and storage utilization of the two new nodes. The reduction of the overlap between two nodes reduces the probability that a search will have to follow multiple paths. Reduction in the perimeter of a bounding box increases the density of descendants, which in turn increases the storage utilization of the selected node. Increased storage utilization allows for a greater area to be created before a split is necessary, thereby decreasing the probability of a search needing to follow multiple paths. These additions lead to a marked improvement in performance of the R*-tree over the R-tree [48].

2.2.4 Burkhard-Keller Tree

The Burkhard-Keller Tree (BKT) [12] is considered one of the first metric trees and may be seen as the basis for many of the metric trees proposed after. The BKT is defined as follows. Let U represent the universe of discourse for all valid objects within the database. An arbitrary element $p \in U$ is selected as the root of the tree. For each distance $i > 0$, define $U_i = \{u \in U, d(u, p) = i\}$ as the set of all elements at distance i from the root p. For all nonempty U_i, build child p_i, hereafter labeled a *pivot*, where the BKT is recursively built for U_i. This process is repeated until no more elements are left for processing or there are only b elements left which may

then be placed within a *bucket* of size b.

The BKT supports range queries in discrete space effectively. Given a query q and a distance r, the search begins at the root and enters all children i such that $d(p,q) - r \leq i \leq d(p,q) + r$, and proceeds recursively. Whenever a leaf/bucket is encountered, the items are compared sequentially. This guarantees an accurate solution due to the BKT satisfying the Triangular Inequality Property.

2.2.5 Fixed Query Tree

The Fixed Query Tree (FQT) [3], is an extension of the BKT. Unlike the BKT, the FQT stores all elements at the leaves, leaving all internal nodes as pivot points. This construction allows for fast backtracking by allowing the effective storage of a search path in temporary memory. If a search visits many nodes of the same level, only one comparison is needed because all of the pivots at that level are the same. Baeza-Yates et al. demonstrated that FQTs performed fewer distance evaluations at query time than the original BKT. This improvement is at the expense of a taller tree, where unlike the BKT, it is not true that a different element is placed in each node of the tree. The Fixed Height FQT (FHQT), originally proposed in [3], was discussed as a variant of the FQT in [2]. Here, all leaves are at the same height h, regardless of the bucket size (similar to many CDS implementations). This has the affect of making some leaves deeper than necessary. However, because the search path is

maintained in temporary memory, this does not represent a significant detriment to the performance of the FHQT.

2.2.6 Fixed Queries Array

The Fixed Queries Array (FQA) [14], is described as a compact representation of the FHQT. No longer described as a tree structure, the FQA is an array representation of the elements stored in the leaves of an FHQT seen left to right. For each element in the array, h numbers representing the branches to traverse in the tree to reach the element from the root are computed. These numbers are coded in b bits and concatenated in a single number where the higher levels of the tree are the most significant digits. The FQA is then sorted by these numbers. As such, each subtree in the FHQT corresponds to an interval in the FQA. Updates are accomplished using two binary searches through the FQA. The FQA improves the efficiency of the FHQT by being able to include more pivot points within the same amount of memory.

2.2.7 M-Tree

The M-tree [16] is a metric tree designed to provide efficient dynamic organization and strong I/O performance in searching. Similar in structure to that of a GNAT [11], the M-tree chooses a set of representatives at each node and the elements closest to each of these representatives are grouped into subtrees rooted by the representative.

Insertions are handled in similar methods to that of an R-tree. Upon an insertion, an element is inserted in the "best" node, defined as that causing the subtree covering radius to expand the least. In the result of a tie, the closest representative is chosen. Upon reaching a leaf, the insertion of the element may cause overflow, (i.e., the number of elements equals $M + 1$). In such a case, the node is split in two and the elements are partitioned between the resulting nodes. One node element is promoted upwards to become a representative; this change propagates to the root of the tree. Searches are performed by comparing the search radius r_s with each representative's covering radius r_c in a node. For all representative in the node where $r_s < r_c$, the search continues recursively through the subtrees of the effective representative. As shown by Ciaccia et al. [16], the M-tree shows impressive performance results against CDS geometric indexes.

2.3 NDDS Models in Vector Space

Currently, NDDS indexing techniques utilizing vector space representations of data points are fairly limited. An exhaustive search of the literature yields only two methods specifically applicable towards indexing such a space representation; the ND-tree and the NSP-tree. This work appears to provide the most significant results toward performing k-nearest neighbor queries within an NDDS. As such, both methods are discussed in detail in the following subsections.

2.3.1 The ND-tree

The ND-tree [43] is inspired by popular CDS multidimensional indexing techniques such as R-tree and its variants (the R*-tree in particular). Like many of the techniques that inspired it, the ND-tree is a balanced tree satisfying three conditions: (1) the root has at least two children, unless it is a leaf, and at most M children; (2) every non leaf and leaf node has between m and M children or entries respectively, unless it is the root; (3) all leaves appear at the same level. Here, M and m represent the upper and lower bounds set on the number of children/entries, where $2 \leq m \leq \lceil M/2 \rceil$.

Unlike previous examples of balanced trees, the ND-tree is designed specifically for NDDSs and as such is based upon the NDDS concepts such as discrete rectangles and their areas of overlap defined in Section 1.2.2. Due to this design consideration, the ND-tree is able to take special characteristics of NDDSs into consideration that *metric trees* are unable to utilize.

The ND-tree is a structure of indexed vectors from an NDDS Ω_d over an alphabet A, where A_i represents the alphabet of the i^{th} dimension. Leaf nodes consist of tuples of the form (*op*, *key*), where *key* is a vector from Ω_d representing an object, pointed to by *op*, in the database. A non leaf node N also consists of tuples, of the form (*cp*, *DMBR*), where *cp* is a pointer to a child node N' of N and *DMBR*

represents the discrete minimum bounding rectangle, described in section 1.2.2, of N'. The $DMBR$ of a leaf node N'', consists of all the vectors indexed in N''. The $DMBR$ of a non leaf node N' is the set of all $DMBR$s of the child nodes of N'.

To build an ND-tree, the algorithm **ChooseLeaf** is implemented to determine the most suitable leaf node for inserting a new vector α into the tree. Starting at the root and progressing to the appropriate leaf, the algorithm **ChooseLeaf** must determine which descendant path to follow at each non leaf node it encounters. Decisions are based upon three heuristics used in ascending order for tie breaks, such that if IH_1 results in two or more possible paths, IH_2 is used to narrow the field further, and so on until a child must be chosen at random. These heuristics are presented as follows:

IH_1: Choose a child node corresponding to the entry with the least enlargement of $overlap(E_k.DMBR)$ after the insertion. [43]

IH_1 chooses a child node E_k from entries $E_{1,2}, \ldots, E_p$, $m \leq p \leq M$ and $1 \leq k \leq p$, such that the insertion of vector α results in the least enlargement of $overlap(E_k.DMBR)$, defined as:

$$overlap(E_k.DMBR) = \sum_{i=1, i \neq k}^{p} area(E_k.DMBR \cap E_i.DMBR) \qquad (2.1)$$

IH_1 results from the observation of the ND-tree experiencing similar retrieval performance degradation due to the increasing amount of overlap between bounding regions as seen for CDSs [10], [38]. This increase in overlap is a major concern for high dimensional indexing methods and as such has been described as *the high dimensionality curse*. Unlike multidimensional index trees in CDS, the number of possible values for the overlap of an entry in an NDDS are limited, implying ties may arise frequently. As such IH2 and IH3, two area based heuristics, are given to break possible ties:

IH_2: Choose a child node corresponding to the entry E_k with the least enlargement of $area(E_k.DMBR)$ after the insertion.[43]

IH_3: Choose a child node corresponding to the entry E_k with the minimum $area(E_k.DMBR)$.[43]

Once any node contains more entries than the maximum allowable value, the overflowing node is split into two new nodes N_1 and N_2 whose entry sets are from a partition defined as follows: let N be an overflow node containing $M + 1$ entries $ES = \{E_1, E_2, \ldots, E_{M+1}\}$. Partition P of N is a pair of entry sets $P = \{ES_1, ES_2\}$

such that: (1) $ES_1 \cup ES_2 = ES$; (2) $ES_1 \cap ES_2 = \emptyset$; and (3) $m \leq |ES_1|, m \leq |ES_2|$. Partition P is determined through the algorithm **SplitNode**, which takes an overflow node N as the input and splits it into two new nodes N_1 and N_2 whose entry sets come from a partition as defined above. This is a critical step in the creation of an ND-tree as many split possibilities exist and a good partition may lead to an efficient tree. Qian et al. [43] proposes handling this in two phases: (1)determine the set of all possible partitions; (2) select the partition most likely to lead to an efficient tree structure. The exhaustive approach to implementing this process is very costly. As shown by Qian et al., even for relatively small values of M, for example 50, an exhaustive approach would have to consider a result so large as to make the operation impractical: here, $51! \approx 1.6 \times 10^{66}$ permutations. Thus, a more efficient method of generating a (smaller) set of candidate partitions is required.

A more efficient method of generating candidate partition sets stems from the property that the size of an alphabet A for an NDDS is usually small; i.e., in genome sequence examples $|A| = 4$. Let $l_1, l_2, \ldots, l_{|A|}$ be the elements of A, in this case $\{a, g, t, c\}$. The number of permutations on the elements of an alphabet is thus also relatively small, here $4! = 24$. For example, $< a, c, g, t >$ and $< g, c, a, t >$ are both permutations of the set A. Using these observations Qian et al. proposes an algorithm to choose a set of candidate partitions consisting of $d * (M - 2m + 2) * (|A|!)$ candidates. For example, if $d = 25$, $M = 50$, $m = 10$, and $|A| = 4$, the

alphabet permutation based algorithm only considers 1.92×10^4. Further, because a permutation and its reverse yield the same set of candidate partitions [43], only half of the aforementioned candidates need be considered; a significant improvement over the exhaustive method.

It is possible that alphabet A for some NDDS is large. In this case, the aforementioned method no longer provides as significant an improvement over the exhaustive method. If such a case arises, Qian et al. propose determining one ordering of entries in the overflow node for each dimension rather than consider $|A|!$ orderings on each dimension. This is accomplished by creating an auxiliary tree for each dimension T_i and sorting the component sets generated from T_i.

Once a candidate set of partitions has been generated, **SplitNode** selects an appropriate partition based upon four heuristics, as follows:

SH_1: Choose a partition that generates a minimum overlap of the DMBRs of the two new nodes after splitting.[43]

SH_2: Choose a partition that splits on the dimension where the edge length of the DMBR of the overflow node is the largest.[43]

SH_3: Choose a partition that has the closest edge lengths of the DMBRs of the two new nodes on the splitting dimension after splitting.[43]

SH_4: Choose a partition that minimizes the total area of the DMBRs of the two new nodes after splitting.[43]

Heuristics SH_1 through SH_4 are applied in ascending order until there are no ties present. If a tie still exists after the application of SH_4, a partition is chosen at random.

Searching an ND-tree is performed similarly to searching an R-tree. Starting at the root, each child nodes DMBR is compared to the search radius to determine if there is an intersection. If the node intersects the search continues recursively.

2.3.2 NSP-Tree

A common problem among data partitioning based index structures, such as R*-tree in CDS and ND-tree in NDDS, is that of regional overlap. For such index structures an overflow node is split by grouping its indexed vectors into two sets DS_1 and DS_2 for two new tree nodes such that the new nodes meet a minimum space utilization requirement. Commonly, as the dimensionality of such a space grows, the probability of large overlapping regions increases dramatically. This overlap causes a drastic reduction in the efficiency of searching within such a structure. A solution to this problem was proposed for CDSs in the form of space-partitioning data indexes. The LSD tree discussed earlier is an example of such methods, where an overflow node is

split by partitioning its corresponding data space into two non overlapping subspaces for two new tree nodes. The indexed vectors are then distributed among the new nodes based upon which subspace they belong. While this method does lead to high search performances in such structures, the nodes in the tree generally no longer guarantee a minimum space utilization.

Unfortunately, as was the case for the ND-tree, the methods used in creating CDS implementations of a space-partitioning index structure do not directly apply to an NDDS. Thus, a new structure, labeled the NSP-tree [44], was proposed. The NSP-tree utilizes many of the concepts described for the ND-tree. For example, the methods used to represent the universe of discourse and minimum bounding rectangles remain the same between the two tree structures.

The key difference between an ND-tree and an NSP-tree lies in the method of partitioning the data points into smaller subspaces. The NSP-tree splits an overflow node based upon the frequency distribution of the vectors indexed within the node, such that an even distribution is seen between the two new tree nodes. This distribution method differs from CDS models, where a split is performed upon a chosen dimension and the data points are distributed in relation to the split point. This method no longer applies in a space where no ordering exists among the data points. In an NDDS it is impossible to describe some point x_d being less than or greater

than some point y_d without violating the semantics of the dataset.

To increase search efficiency, the NSP-tree utilizes multiple bounding boxes within each subspace to help eliminate the amount of dead space that is included within the subspace; dead space is any area covered that does not contain any indexed vectors. This is similar to techniques used in CDS, however an interesting property of an NDDS is able to exploited by the NSP-tree. Consider two 2-dimensional points $P_1(x_1, y_1)$ and $P_2(x_2, y_2)$. The MBR necessary to cover such points in a CDS would be a rectangle with points P_1 and P_2 representing corners along one of the diagonals. Such a representation includes a rather large portion of dead space, (roughly all the area is dead space). In an NDDS however, the MBR may be represented as the Cartesian product of the two points $\{x_1, x_2\} \times \{y_1, y_2\}$ which contains a very small dead space $\{(x_1, y_2), (x_2, y_1)\}$. As shown by Qian et al., the NSP-tree shows favorable performance comparisons with the ND-tree, particularly for skewed data [44].

2.4 HDS Models in Vector Space

This section describes currently available methods for vector space indexing of HDS objects. Similar to the previous section describing NDDS indexing methods, there exists a limited amount of HDS indexing methods. Although it should be noted that metric space models such as M-tree could also be used to index such objects,

research by Qian et. al [43] and Chen et. al [15] suggest that this is not as efficient as indexing methods prioritized toward HDSs. In this section we focus upon two more recently proposed HDS indexing methods, the ND^h-tree and the CND-tree.

2.4.1 ND^h-tree

The ND^h-tree, as proposed by Qian [41], is an extension of the ND-tree. The key difference is that instead of discrete minimum bounding rectangles, the ND^h-tree utilizes hybrid minimum bounding rectangles. Initial results reported by Qian [41] demonstrate the effectiveness of utilizing an index structure for HDS objects specifically designed for such a space. The ND^h-tree serves as an inspiration for the CND-tree introduced by Chen et al. [15]. In this dissertation we focus upon the more recent contribution and will describe key differences as necessary.

2.4.2 CND-tree

The CND-tree [15] is similar in structure and function to the R*-tree and the ND-tree. As such, the CND-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. Each vector is inserted into the tree after an appropriate position is found. The relevant minimum bounding rectangle may be either enlarged or split to accommodate the insertion.

The key difference between the CND-tree and related non-hybrid trees (R*-tree for continuous space and ND-tree for discrete space) is the way in which a minimum bounding rectangle is defined and utilized. In the CND-tree, a minimum bounding rectangle contains both continuous and non-ordered discrete dimensions. A hybrid minimum bounding rectangle (HMBR), with d_D discrete dimensions and d_C continuous dimensions, for a set of hybrid rectangles $G = R_1 \times R_2 \times \ldots \times R_n$ with discrete sub-rectangles $R_i^D = S_{i,1} \times \ldots \times S_{i,d_D}$ and continuous sub-rectangles $R_i^C = S_{i,d_D+1} \times \ldots \times S_{i,d_D+d_C}$ (i.e. $R_i = R_i^D \times R_i^C$) is defined as follows:

$$\begin{aligned} HMBR(G) = \; & \left\{\cup_{i=1}^{n} S_{i,1}\right\} \times \ldots \times \left\{\cup_{i=1}^{n} S_{i,d_D}\right\} \times \\ & \left(\min S_{i,d_D+1}, \max S_{i,d_D+1}\right) \times \ldots \times \\ & \left(\min S_{i,d_D+d_C}, \max S_{i,d_D+d_C}\right), \end{aligned} \tag{2.2}$$

where $S_{i,j}$ $(1 \leq j \leq d_D)$ is a set of elements/letters from the alphabet of the j-th dimension and $S_{i,k}(d_D + 1 \leq k \leq d_D + d_C)$ is an interval from the k^{th} dimension.

2.5 Determining Distance

An integral part of any similarity search algorithm is the distance measure employed. Because we are interested in how "close" one object is to another, the selection of a distance measure provides the semantic definition of what "close" means for the current applications.

For NDDSs, the inability to be ordered along an axis renders standard forms of distance measurement, such as Euclidean or Manhattan, inapplicable. In turn, a common method of calculating the distance between two discrete objects is to apply the Hamming measurement. Essentially, this measurement represents the number of dimensions between two d-dimensional vectors that contain different elements. This is described formally as follows:

$$D_{Hamm}(V_1, V_2) = \sum_{i=1}^{d} \left\{ \begin{array}{ll} 0 & \text{if } V_1[i] = V_2[i] \\ 1 & \text{otherwise} \end{array} \right\}. \tag{2.3}$$

This distance is useful in discrete spaces due to its non-reliance upon dataset semantics, particularly for equality measurements. However, its usefulness declines rapidly when applied to other operations, such as grouping, due to its limited cardinality. The cardinality of a NDDS, $Card_{D_H()}(U)$, for a d-dimensional dataset U with an alphabet size $|A_i|$ for each dimension i in d, is calculated as the product of the alphabet sizes from each dimension, as follows:

$$Card_{D_H()}(U) = \prod_{i=1}^{d} |A_i|.$$

Using the aforementioned genome sequence example, a 25-dimensional dataset with an alphabet size of 4 for each dimension would have a cardinality of

1,125,889,906,842,624: that is, there are over 10^{15} possible distinct elements in the dataset. However, if the Hamming distance formula is used to calculate the distance between the objects, there are only $d + 1$ (26) different possible distances between any two objects.

For HDSs, Chen et al. [15] utilized a non-Euclidean measurement for calculating the distance between a vector $\alpha = (\alpha[1], \alpha[2], \ldots, \alpha[d_D + d_C])$ and an HMBR $R = \{S_1 \times S_2 \times \ldots \times S_{d_D+d_C}\}$ to perform range queries:

$$dist(R, \alpha) = \sum_{i=1}^{n} f(s_i, \alpha[i]) \tag{2.4}$$

where

$$f(S_i, \alpha[i]) = \left\{ \begin{array}{ll} 0 & \text{if } i \text{ is a discrete dimension and } d_i \in S_i \\ & \text{or } i \text{ is a continuous dimension and} \\ & \min(S_i) - \delta t \leq d_i \leq \max(S_i) + \delta t \\ 1 & \text{otherwise} \end{array} \right\}$$

Equation 2.4 essentially discretizes the data of the continuous dimensions utilizing a tolerance value of δt determined by an application expert. This method is similar to that used in several machine learning techniques[13, 20, 39].

CHAPTER 3

k-Nearest Neighbor Searching in NDDS

In this chapter, we consider k-Nearest Neighbor (k-NN) searching in NDDSs. Searching in NDDSs presents several new challenges that we discuss. Additionally, we present a formal definition of a k-NN query/search and introduce a new distance measure suitable for searches in NDDSs. A generalized form of this distance measure (and the benefits inherited from this generalized form) is presented in Chapter 4.

3.1 Motivations and Challenges

Numerous techniques have been proposed in the literature to support efficient similarity searches in (ordered) continuous data spaces. A majority of them utilize a multidimensional index structure such as the R-tree [22], the R*-tree [6], the X-tree

[10], the K-D-B tree [46], and the LSD^h-tree [25]. 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 [35], which was then optimized by Seidl and Kriegel [50]. In [31], 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 non-ordered discrete data spaces. Limited existing work on index-based similarity searches in NDDSs has utilized either metric trees such as the M-tree [16] or the ND-tree and the NSP-tree recently proposed by Qian et al. [42, 43, 44]. 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 [42, 43, 44] 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 [43], 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 [23], 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 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. [47]. 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 [32]. 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 dissertation, we demonstrate that our k-NN searching algorithm is efficient in both uniformly distributed datasets and non-uniformly distributed datasets 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-tree 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 chapter is organized as follows. Section 3.2 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 3.3 presents our index-based k-NN searching algorithm for NDDSs, including

its pruning metrics and heuristics and theoretical performance model. Section 3.4 discusses experimental results.

3.2 *k*-Nearest Neighbors in NDDS

In this section, we 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 (see Chapter 2), based upon the properties of NDDSs and the index tree.

3.2.1 Definition of *k*-NN in NDDS

When considering a query in an NDDS, the dataset may be depicted as a set of concentric spheres with the query point located at the center (as shown in Figure 1). Each sphere contains all data points that have r or less mismatching dimensions with the query point, where r represents the layer/radius of the particular sphere. 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. One way to resolve the non-uniqueness/non-determinism of a k-NN

search is to find the minimum radius r such that a sphere of r will contain at least k data points/neighbors, while a sphere of radius $r-1$ contains less than k neighbors. Although such an approach indeed resolves the non-determinism problem, the solution to such a k-NN search may not be what the users are expecting, since they usually not only want to know the minimum distance/radius for a k-NN search but also want to know the actual k-NN neighbors. Note that the non-determinism also theoretically exists for a k-NN search in a continuous data space although it is not as prominent as it is in an NDDS since the chance for two data points having the same distance to a query point in a continuous data space is usually very small. In this dissertation, we adopt the traditional approach to finding the actual neighbors for a k-NN search in an NDDS and resolve the non-determinism problem in another way. Note that, once the k-nearest neighbors are found, the above minimum radius r is also found.

We define a candidate solution set of k-nearest neighbors for a query point as follows:

Definition 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*

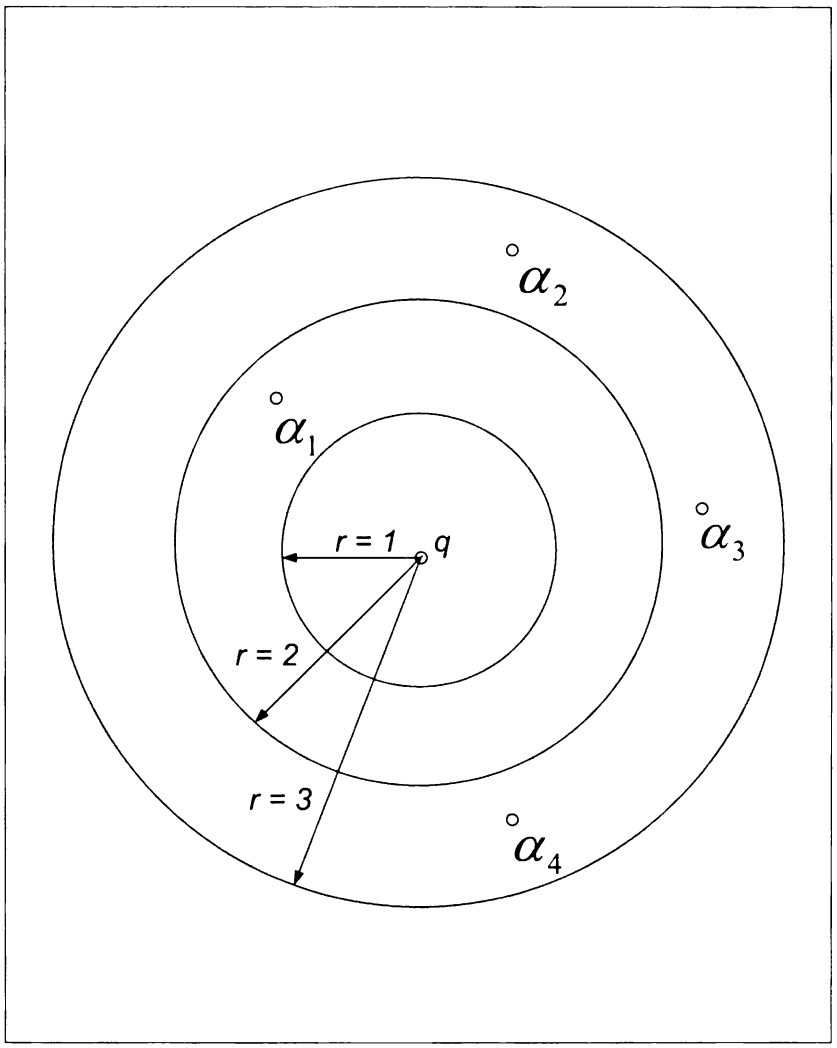


Figure 3.1. Example of NDDS data points distributed by distance

point q and $D(x, y)$ denote the distance between the objects x and y. Then kNNS is defined as follows:

$$kNNS = \{A_1, A_2, \ldots, A_k\} \Rightarrow \left\{ \begin{array}{l} \forall_{B_1, B_2, \ldots, B_k} \left[\Sigma_{i=1}^{k} D(q, A_i) \leq \Sigma_{j=1}^{k} D(q, B_j)\right] \wedge \\ \forall_{m,n \in \{1,2,\ldots,k\}} \left[(m \neq n) \rightarrow (A_m \neq A_n)\right] \end{array} \right\}. \tag{3.1}$$

Equation 3.1 essentially says that k objects/neighbors $A_1, A_2, \ldots, A_k$ in kNNS 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 1, if $k = 3$, there are three possible sets of neighbors that satisfy Equation 3.1: $\{a_1, a_2, a_3\}$, $\{a_1, a_2, a_4\}$, and $\{a_1, a_3, a_4\}$. 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 size k subset of the set of neighbors that would be found using above minimum distance r.

Since kNNS 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. ***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:*

$$
\begin{aligned}
A_1 = \quad & A \Rightarrow \forall_B \left(D(q, A) \leq D(q, B) \right), \\
A_k = \quad & A \Rightarrow \forall_B \left[\begin{array}{l} B \notin \{A_1, A_2, \ldots, A_{k-1}\} \wedge \\ A \notin \{A_1, A_2, \ldots, A_{k-1}\} \end{array} \right] \left(D(q, A) \leq D(q, B) \right) \ \textit{for } k \geq 2.
\end{aligned}
\tag{3.2}
$$

Definition 2 can be used to produce the candidate *k*NNSs given by Definition 1, as stated in the following proposition.

Lemma 3.2.1. *Each candidate kNNS produced by Definition 2.2.2 is contained in the set of candidate kNNSs given by Definition 2.2.1.*

Proof. Proposition 2.2.3 states that the set of candidate *k*NNSs given by Definition 1 can be produced by Definition 2. This leads to the hypothesis that if a solution set $NN = \{A_1, A_2, \ldots, A_{k-1}\}$ satisfies Equation 3.1 for $k-1$ objects, then Equation

3.2 will select a k^{th} neighbor A_k such that $NN \cup A_k$ will satisfy Equation 3.1 for k objects and thus be consistent with Definition 1.

We first consider a base case where $k = 1$. Equation 3.2 yields the following neighbor A_1:

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

The solution set $\{A_1\}$ satisfies Equation 3.1 for $k = 1$ and thus is consistent with Definition 1. Equation 3.2 may then be used again to yield neighbor A_k:

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

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. □

From the above definitions (and Figure 1), 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.

Lemma 3.2.2. *The number Δk of candidate kNNSs is given by*

$$\Delta k = \frac{N'!}{t!(N'-t)!}, \tag{3.3}$$

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. 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 3.1 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 3.1. 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 3.3:

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

□

Note that t denotes the number of objects with distance $D(q, A_k)$ that have to be included in a kNNS. If $t = k$, all the neighbors in a kNNS are of the same distance as $D(q, A_k)$. In this case A_{k-t} is inapplicable. The values of N' and t depend on parameters such as the dimensionality, the database size, and the query point.

For a k-NN query on a database in a continuous data space based on the Euclidean distance, kNNS 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 [42, 43], 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 dimension in an NDDS, Δk for a k-NN query in an NDDS is usually large. For example, for a dataset 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 2. 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.

3.2.2 Extended Hamming Distance

Intuitively, the Hamming distance indicates the number of dimensions on which the corresponding components of α and β differ. As discussed by Qian et al. [42, 43], the Hamming distance is well suited for search applications in NDDS. In particular, we note that applications with different alphabet sets for different dimensions or with no known similarity matrix (needed for many edit distances) present strong cases for using the Hamming distance when searching. Additionally, recent work has applied the Hamming distance when searching in large genome sequence databases [37, 29].

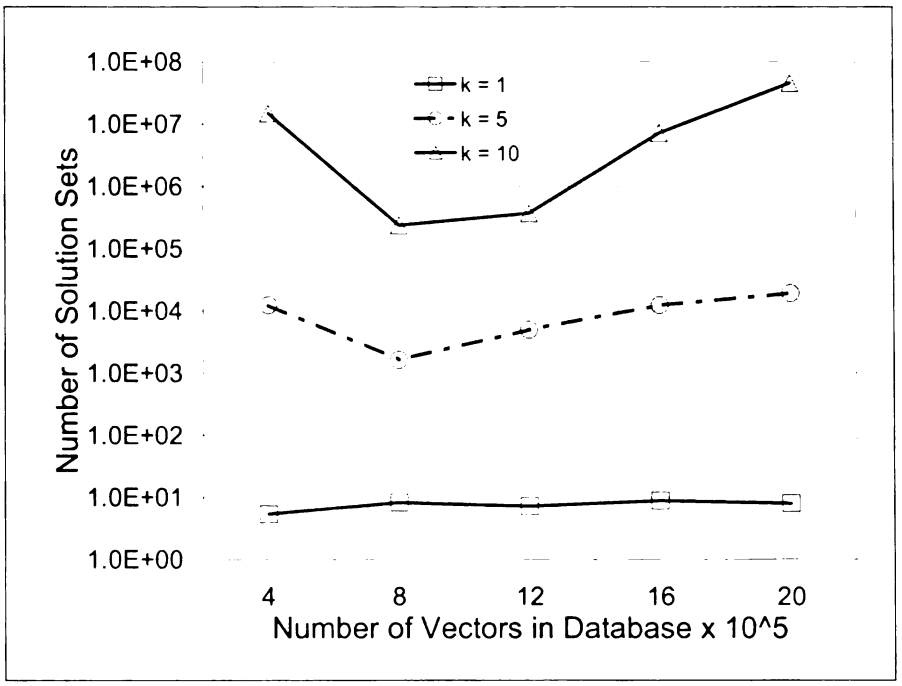


Figure 3.2. Comparison of Δk values for the Hamming distance

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 the 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:

$$D_{GEH}(\alpha, \beta) = \sum_{i=1}^{d} \left\{ \begin{array}{ll} 1 & \text{if } \alpha[i] \neq \beta[i] \\ \frac{1}{d} f(\alpha[i]) & \text{otherwise} \end{array} \right\}, \tag{3.4}$$

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. This frequency based adjustment results in the possibility of fractional distance values rather than just integer distance values (as seen when using the traditional Hamming distance).

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 guarantees that the solution set (kNNS) returned using this distance will be among the candidate solution sets returned if the Hamming distance were used instead. We also note that function $f(\alpha[i])$ is not restricted to the definition given in Equation 3.4. So long as the values of $f(\alpha[i])$ are within the range of $[0, 1)$, the factor of $\frac{1}{d}$ guarantees the semantic of the Hamming distance is maintained. In Chapter 4, we explore this concept more thoroughly and present a generalized form of Equation 3.4.

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). Additionally, the function $f(\alpha[i])$ plays a factor in preserving the triangular property of Equation 3.4, as shown in Appendix B.

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 3.2.2, leading to more deterministic k-NN searches in NDDSs. As an example, consider again Figure 1. If we assume that vectors α_2, α_3, and α_4 each match query vector q in only one dimension such that $\alpha_2[1] = q[1]$, $\alpha_3[2] = q[2]$, and $\alpha_4[3] = q[3]$, and we also assume $f(q[3]) < f(q[4]) < f(q[2])$. The use of the GEH distance resolves the non-determinism seen earlier when $k = 3$. Here, the solution set would be $\{\alpha_1, \alpha_3, \alpha_4\}$ (one of the candidate kNNS when the Hamming distance was used) since $D_{GEH}(q, \alpha_1) < D_{GEH}(q, \alpha_3) < D_{GEH}(q, \alpha_4) < D_{GEH}(q, \alpha_2)$. On a larger scale, for the aforementioned dataset 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 3.3 in Section 3.4.1).

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.

3.2.3 Probability of Valid Neighbors

In many scenarios, it is useful to know the probability/likelihood of encountering vectors within an index tree 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 valid neighbor α; where $\forall_\alpha D(q, \alpha) \leq r$, 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 tree has maintained a relatively uniform distribution of elements within its subtrees. In a well balanced tree (ND-tree, M-tee, etc...), this may prove to be a very reasonable assumption, as most indexing methods will attempt to evenly distribute elements within their subtrees. 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:

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

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 $p(a)_{R,i}$, may be estimated far more accurately by determining the local frequency ratio of element a within a subtree, with associated DMBR R, on dimension i as follows:

$$p(a)_{R,i} = f_l(a)_{R,i}, \tag{3.5}$$

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 3.5 remains accurate even for indexes with heavily skewed distributions.

The probability of encountering valid 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:

$$PE(\alpha, R) = \prod_{i=1}^{d} p(\alpha[i])_{R,i}. \tag{3.6}$$

As defined in Section 3.2.2, 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 3.6. However, because at most one vector within an ND-tree will satisfy $D_{Hamm}(q, \alpha) = 0$, we also have to consider the probability of a subtree containing vectors $\beta = (\beta[1], \beta[2], \ldots, \beta[d])$, where $D_{Hamm}(q, \beta) = z$ $(z \in \{1, 2, \ldots, d\})$.

Lemma 3.2.3. *Let Y represent the set of dimensions where $\beta[Y_j] \neq q[Y_j]$ and let X*

represent the set of dimensions where $\beta[X_j] = q[X_j]$. *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 is given as the following (note that* $z = |Y|$*):*

$$PNE(\beta, R) = \prod_{j=1}^{|X|} p(\beta[X_j])_{R,X_j} * \prod_{j=1}^{|Y|} (1 - p(\beta[Y_j])_{R,Y_j}). \tag{3.7}$$

Proof. As described in Equation 3.6, 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 estimated by $p(\beta[X_j])_{R,X_j}$ (Equation 3.5). The probability of anything except the specified element is $1 - p(\beta[Y_j])_{R,Y_j}$. Thus, the probability of a specific vector, which does not match the query vector in z dimensions, existing in a subtree with R is the product of two terms: the product of $p(\beta[X_j])_{R,X_j}$ in the matching dimensions and the product of $1 - p(\beta[Y_j])_{R,Y_j}$ in the non-matching dimensions. □

Lemma 3.2.3 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 dataset that matched a query vector q in dimensions 1, 3, 8, and 9. In this example, $X = \{1, 3, 8, 9\}$ and $Y = \{2, 4, 5, 6, 7, 10\}$, 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 also all possible partial-matching vectors that may be found at a distance z from the query vector.

Proposition 1. *Let B represent the set of all vectors from the given dataset in an NDDS 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 given as the following:*

$$PS_z(q, R) = \sum_{\beta \in B} PNE(\beta, R). \tag{3.8}$$

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, each of which has a specified number of dimensions matching a query vector, is the summation of the probabilities of all vectors 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 given as follows:

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

We note that as the dimensionality of the dataset increases, this calculation can become costly, i.e. $O(d * \binom{d}{d-z})$. One possible solution is to create a hash structure in a pre-processing step, that stores binary arrays representing the different combinations of matches and non-matches for each particular distance value. Here, the set of keys is the set of integers $(0, 1, \ldots, d)$, and the values are the sets of binary arrays with a corresponding number of 0's. For example, the key "3" would retrieve the set of binary arrays that contains all possible permutations with exactly three 0's. The set B for a particular distance may be determined by retrieving the corresponding set of binary arrays, where for each array, a value of 1 would correspond to a dimension in X and a 0 a dimension in Y (see Proposition 2.4.1). This method can drastically reduce CPU time.

The summation of the probability values given by Equation 3.8 for each integer distance $z \in \{0, 1, \ldots, r\}$ yields the probability of the subtree containing a vector β that is within range r to the query vector (i.e. $D_{Hamm}(q, \beta) \leq r$). This is expressed formally as follows:

$$PNN_H(q, R) = \sum_{z=0}^{r} PS_z(q, R). \tag{3.9}$$

Equation 3.9 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 3.9. When using the GEH distance, r is no longer strictly an integer. Thus, it is possible for a valid neighbor to exist at a distance $\lfloor r \rfloor < D_{GEH}(q, \beta) \leq r$, where r is a real number. An adjustment to Equation 3.8 is needed to properly account for possible neighbors within this range.

Proposition 2. *Let B' represent the set of all vectors from the given dataset in an NDDS where $\beta \in B' : [D_{Hamm}(q, \beta) = \lfloor r \rfloor]$. The probability that a subtree with DMBR $R = S_1 \times S_2 \times \ldots \times S_d$ will contain a record β where $(\lfloor r \rfloor \leq D_{GEH}(q, \beta) \leq r)$ is given as the following:*

$$PR(q, R, r) = \sum_{\beta \in B'} PNE(\beta, R)\delta(r - \lfloor r \rfloor, \beta, q), \tag{3.10}$$

where

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

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

Proof. The proof for Proposition 1 shows that Equation 3.8 yields the probability of a vector, which 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 3.10 is the summation of each of the remaining vectors v', where $\forall_{v' \in B'} : D_{GEH}(v', q) \leq r$. Thus Equation 3.10 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 3.10 allows us to refine Equation 3.9 to make use of our GEH distance as follows:

$$PNN_{GEH}(q, R, r) = PR(q, R, r) + \sum_{z=0}^{\lfloor r \rfloor - 1} PS_z(q, R). \qquad (3.11)$$

Equation 3.11 may be used to give an accurate estimate 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 3.3.2 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.

3.3 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. [42, 43] 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 inspired by earlier incremental ranged based implementations presented for CDS by Roussopoulos et al. [47] and Hjaltason and Samet [26] (generalized for metric space). Our algorithm extends these implementations to NDDSs by introducing metrics and heuristics suitable for such a space. The details of this algorithm are discussed in the following subsections.

3.3.1 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 [47], 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:

$$mdist(q, R) = \sum_{i=1}^{d} \left\{ \begin{array}{ll} 1 & \text{if } q[i] \notin S_i \\ \frac{1}{d} f(q[i]) & \text{otherwise} \end{array} \right\} \tag{3.12}$$

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:

$$mmdist(q, R) = \min_{1 \leq k \leq d} \left\{ f_m(q[k], S_k) + \sum_{i=1, i \neq k}^{d} f_M(q[i], S_i) \right\} \tag{3.13}$$

where

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

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

where $f()$ on the right hand side of the last two Equations is defined in Equation 3.12.

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 (since the component set S_i on the corresponding dimension contains only the corresponding element $q[i]$ in the query vector). In these cases, a value of $\frac{1}{d} f(q[i])$ is added to the distance (i.e. the GEH adjustment for a matching dimension). The value of f_m determines if there is another dimension (not in those checked for f_M) in which at least one vector in the associated subtree will match the query vector. 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) that can be guaranteed a vector will be located from the query vector in the associated subtree, based upon the information in the DMBR. For example, given a query vector $q = (a, b, c)$ and a $DMBR = \{a, d\} \times \{b\} \times \{c, a\}$, we have $f_m(a, \{a, d\}) + f_M(b, \{b\}) + f_M(c, \{c, a\}) = \frac{1}{3} f(a) + \frac{1}{3} f(b) + 1$, which indicates that a vector (i.e. $(a, b, ?)$) matching q on the first two dimensions is guaranteed

to exist in the corresponding subtree. The minimum $mmdist()$ of such distances is sought in Equation 3.13. If $Range \geq mmdist()$, it is guaranteed that at least one valid neighbor can be found in the corresponding subtree.

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 [47]: 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 from what is expressed in the representing DMBR. As discussed in Section 3.2.3, 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 3.5.

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 less than or equal to the current range, as shown in Equation 3.11. 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 $D_{GEH}(q, \alpha) \leq Range$. 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.

3.3.2 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 as follows.

k-NN Query Algorithm: Given an ND-tree with root node T, Algorithm k-NN Query finds a set of k-nearest neighbors, maintained in queue $kNNS$, to query vector q that satisfies Equation 3.1 in Definition 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.

Algorithm 1 k-NN Query
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
```

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 there are no more subtrees 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 ($kNNS$).

Note that *ChooseSubtree* not only returns a chosen subtree but also may update *Range* using heuristic H_2. If there are no more subtrees 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.

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 3.3.1 are employed in function *ChooseSubtree*.

Function 2 *ChooseSubtree*$(N, q, k, Range)$

1: **if** list L for not yet visited subtrees of N not exist **then**
2: use heuristic H_2 to update *Range*
3: use heuristic H_1 to prune subtrees of N
4: use heuristic H_3 or H_4 based upon user criteria to sort the remaining subtrees not pruned by H_1 and create a list L to hold them
5: **else**
6: use heuristic H_1 to prune subtrees from list L
7: **end if**
8: **if** $L \neq \emptyset$ **then**
9: remove the 1st subtree NN from L
10: **return** $[NN, Range]$
11: **else**
12: **return** $[NULL, Range]$
13: **end if**

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) in a list L. The subtrees that are not in this list are those that have already been processed or pruned. 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*. Note that heuristics H_3 and H_4 are suitable for different datasets. Their effects on performance and practical selection guidance will be discussed in Section 3.4.4.

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

Function 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 $kNNS$ only if its distance to the query vector is smaller than current *Range* (steps 2 - 3). A vector has to be removed from $kNNS$ 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.

3.3.3 Performance Model

To analyze the performance of our k-NN search algorithm, presented in Section 3.3.2, we conducted both empirical and theoretical studies. The results of our empirical studies are presented in Section 3.4. 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.[1]

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. *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:*

[1]The assumption of a reasonably optimistic tree structure covers the majority of ND-trees generated in our empirical studies. Non-optimistic tree structures, where our fourth heuristic would provide a more beneficial ordering method, are considered empirically in Section 3.4.4.

$$Area(z) = \sum_{i=0}^{z} \binom{d}{i} (|A| - 1)^i. \tag{3.14}$$

Note that $Area(z)$ is independent of point p under the uniform distribution assumption. Equation 3.14 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:

$$Area(d) = |A|^d. \tag{3.15}$$

The probability of an object existing within a distance of z from any point p is the quotient of Equations 3.14 and 3.15, as follows:

$$P_{exists}(z) = \frac{Area(z)}{Area(d)}. \tag{3.16}$$

Proposition 3. *The number of likely points contained within a distance z from any point p is the product of the number of points within the dataset N and the probability of a point existing within a distance of z from p. This is shown formally as follows:*

$$L(z) = P_{exists}(z) * N. \tag{3.17}$$

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 3.17 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. [43].

$$IO_r = 1 + \sum_{i=0}^{H-1} (n_i * P_{i,z}), \tag{3.18}$$

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, and 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 the height of that level:

$$Adj = \sum_{i=1}^{H-1} n_{i-1}^{\left(\frac{1}{i-1}\right)}. \tag{3.19}$$

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

$$IO_{NN} = 1 + (n_0 * P_{0,z}) + \sum_{i=1}^{H-1} \left[(n_i * P_{i,z}) + n_{i-1}^{\left(\frac{1}{i-1}\right)} \right]. \tag{3.20}$$

The performance of our search algorithm can be estimated by using Equation 3.20 setting z to d_l.

3.4 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 using an ND-tree in the C++ programming language. For comparison purposes, we also implemented the k-NN searching using an M-tree in the C++ programming language for a 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 datasets were used in our experiments. The synthetic datasets 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 datasets are listed in the following subsections. The genomic datasets were created from 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.

3.4.1 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 3.2.2 in Section 3.2.1.

Figure 3.3 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 3.3 shows that when the GEH distance is used, Δk values are very close to 1, indicating a promising behavior close to that in CDSs.

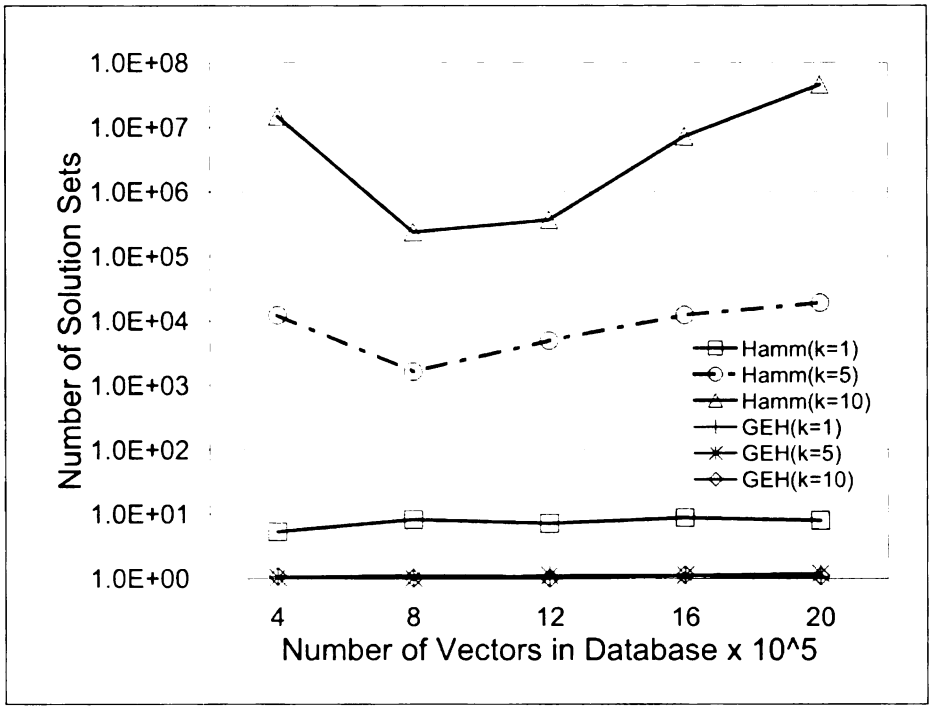


Figure 3.3. Comparison of Δk values for the GEH and Hamming distances

3.4.2 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 3.3.1, on the performance of our k-NN searching algorithm presented in Section 3.3.2 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.

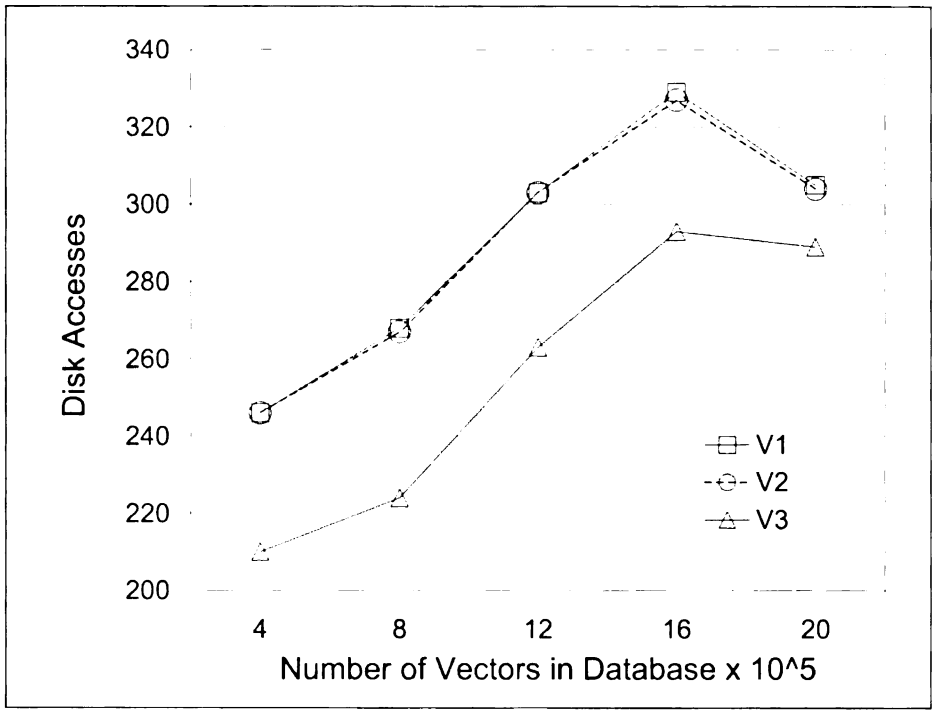


Figure 3.4. Effects of heuristics in the k-NN algorithm using ND-tree with $k = 10$

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 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 performances of our k-NN searching algorithm using an ND-tree, the k-NN searching based on an M-tree, and the linear scan for databases of various sizes. Figure 3.5 shows the performance comparison for our k-NN searching algorithm using an ND-tree and the

linear scan. Figure 3.6 shows the performance comparison in number of disk accesses of our k-NN searching algorithm using an ND-tree and the k-NN searching based on an M-tree. From the figures, we can see a significant reduction in the number of disk accesses for our k-NN searching algorithm using an ND-tree over both the M-tree searching algorithm and the linear scan. Additionally, the results in Figure 3.5 show that the performance gains of our k-NN algorithm increase as the database size increases. As the database grows larger, the density of points within the available data space increases as well. This causes the search range to decrease at a faster rate, due to finding more points at closer distances to the query point, resulting in a greater percentage of subtrees being pruned by H_1.[2] Figure 3.5 also shows that, for all database sizes tested, our algorithm, implemented using an ND-tree, always used less than 25% (10% for database sizes of $1M$ or more vectors) of the disk accesses than the linear scan.

Figure 3.7 shows the performance comparison of our algorithm implemented using an ND-tree and the linear scan method on genomic datasets. 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

[2]It should be noted that this behavior is not unique to the ND-tree. An in-depth discussion of this behavior was presented by Chavez et al. [14]. Although Chavez et al. focus primarily on searching in general metric spaces, the same principles apply here.

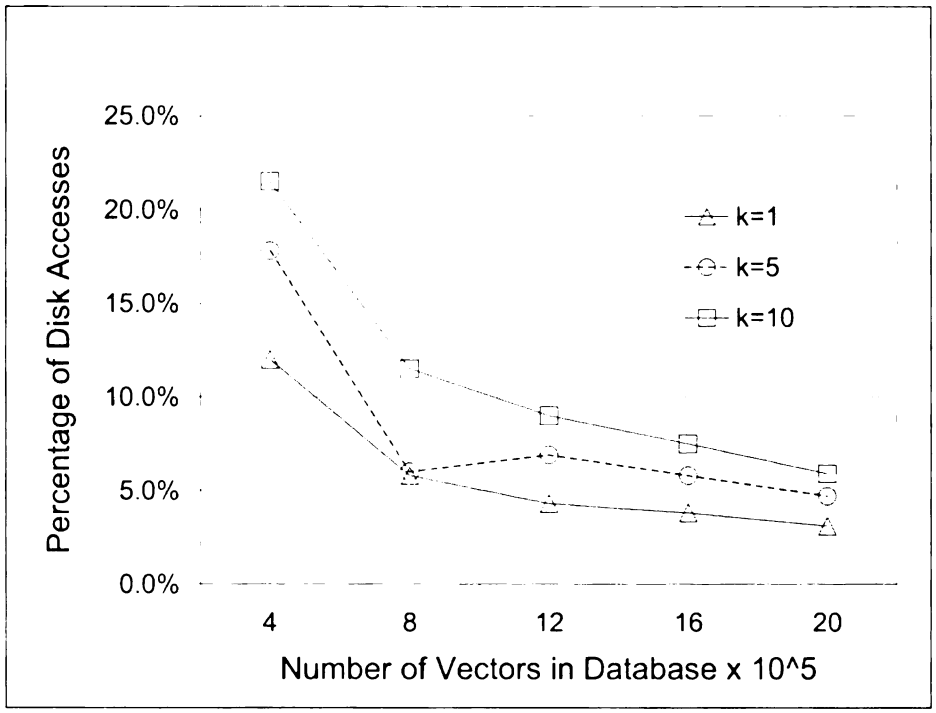


Figure 3.5. Performance of the k-NN algorithm using ND-tree vs. the linear scan on synthetic datasets with various sizes

genomic datasets is comparable with that we observed for the synthetic datasets.

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 datasets: one contains 250K vectors for each set and the other contains 1 million vectors for each set. As seen from Figure 3.8, 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-

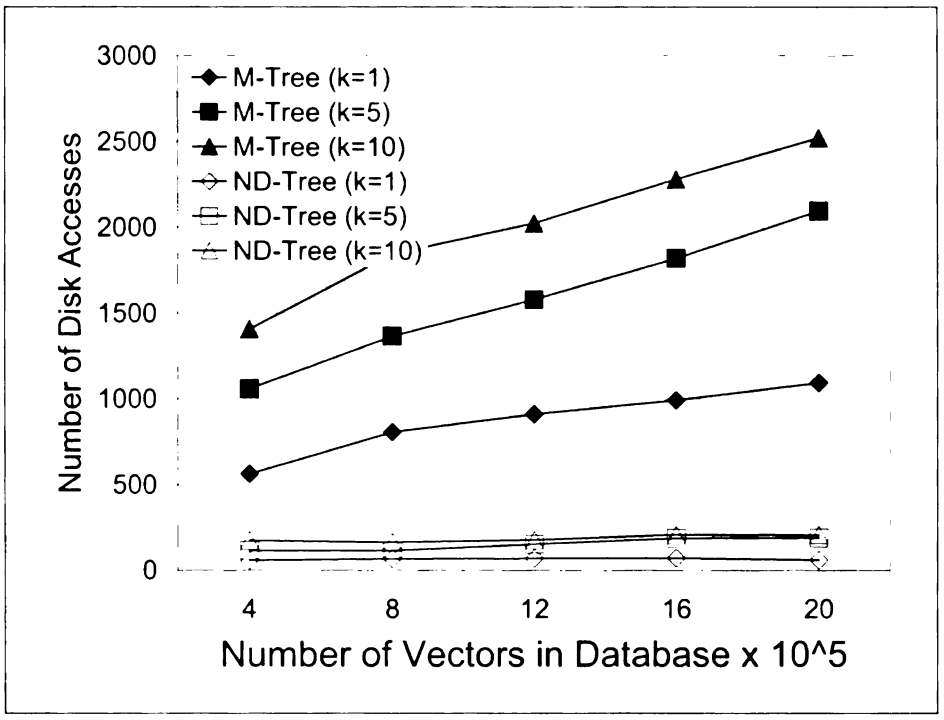


Figure 3.6. Number of Disk Accesses comparison of the k-NN algorithm using ND-tree vs. the k-NN searching based on M-tree

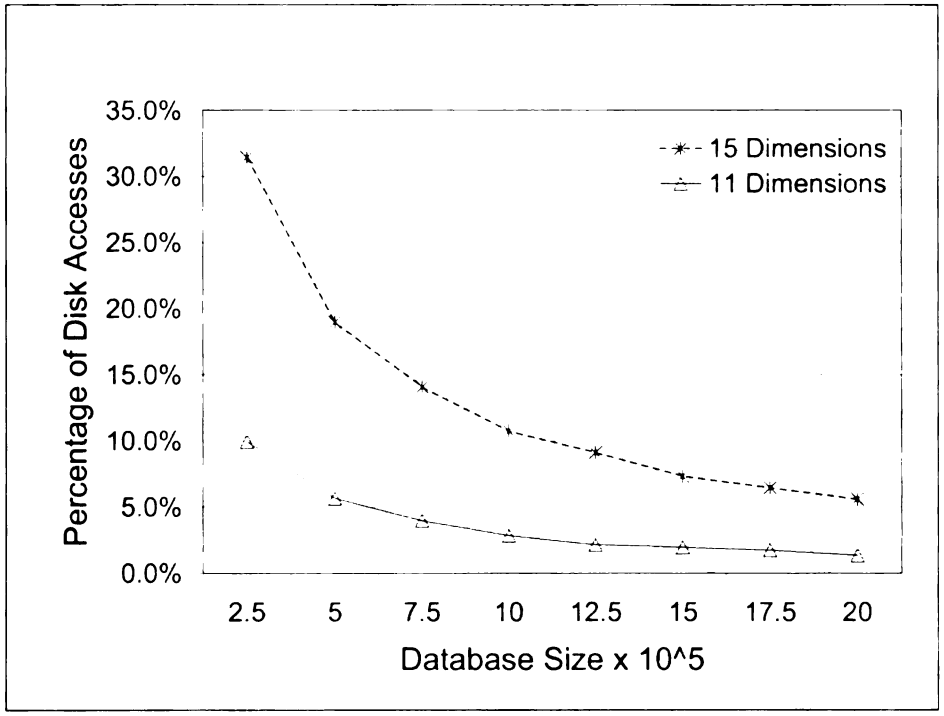


Figure 3.7. Performance of the k-NN algorithm using ND-tree vs. the linear scan on genomic datasets with various sizes for k=10

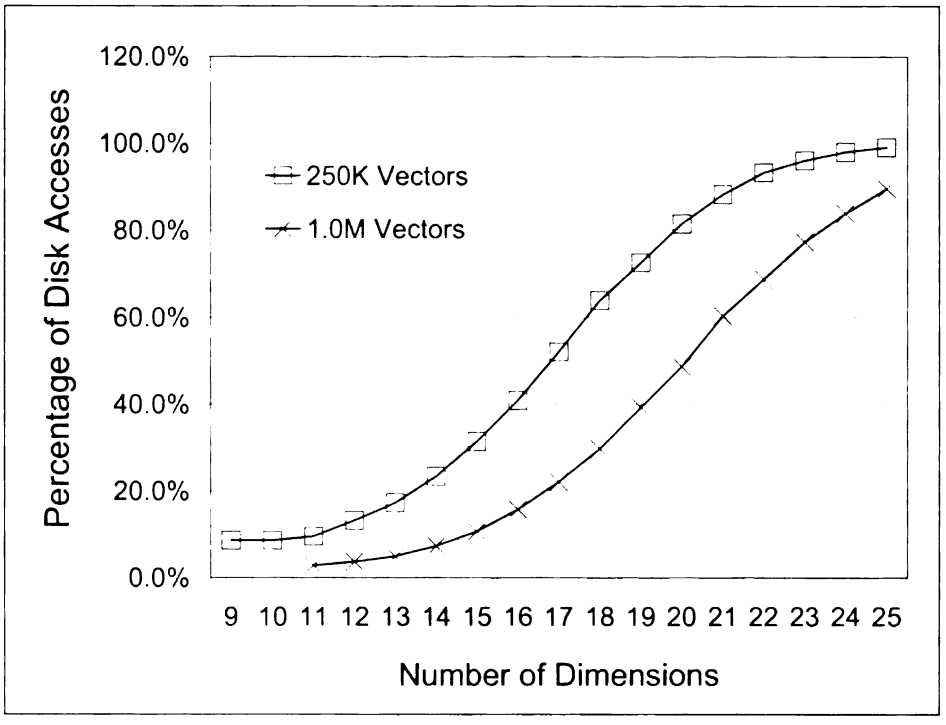


Figure 3.8. Performance of the k-NN algorithm using ND-tree vs. the linear scan on genomic datasets with various dimensions for k=10

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 3.9 shows that the effects of increasing alphabet size are similar to the effects of the dimensionality curse.

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 3.10 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

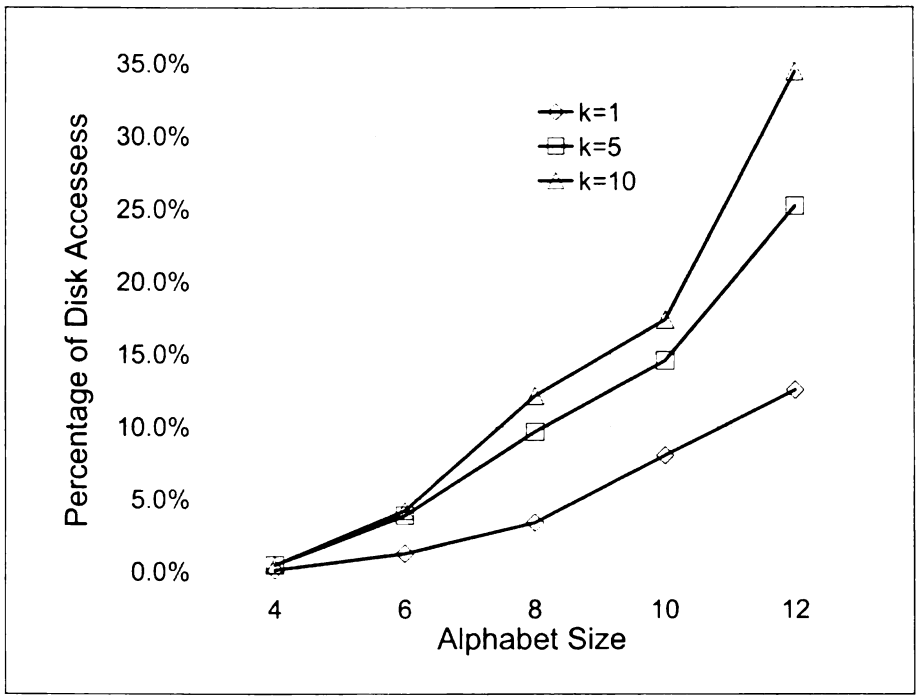


Figure 3.9. Performance of the k-NN algorithm using ND-tree vs. the linear scan on synthetic datasets with various dimensions for k=10 and $d = 10$

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

3.4.3 Efficiency of k-NN Algorithm on Skewed Data

Experiments were also conducted to examine the I/O performance of our algorithm upon datasets of varying levels of skewness as compared to that of a linear scan. We applied our algorithm, with heuristic version V3 from Section 3.4.2, to ND-trees

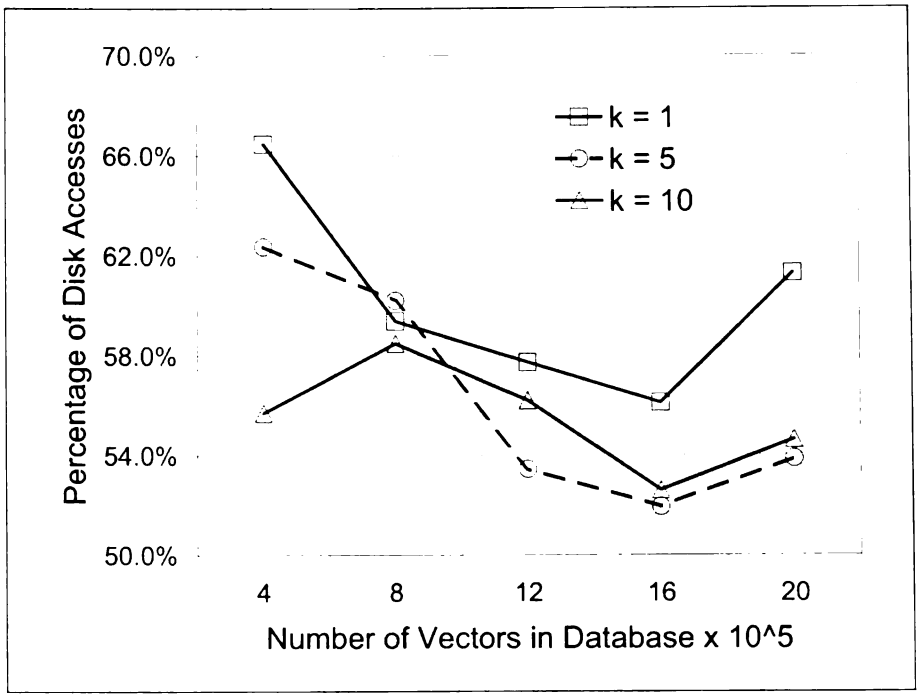


Figure 3.10. Performance comparison for the k-NN searching using ND-tree based on GEH and Hamming distances

constructed from datasets with zipf distributions of 0.0, 0.5, 1.0, and 1.5.

Figure 3.11 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 3.4.2), our k-NN searching algorithm provides an increased reduction of disk accesses as the database size increases. Figure 3.11 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 3.3.1) are able to identify and prune more useless search paths as the data becomes more skewed.

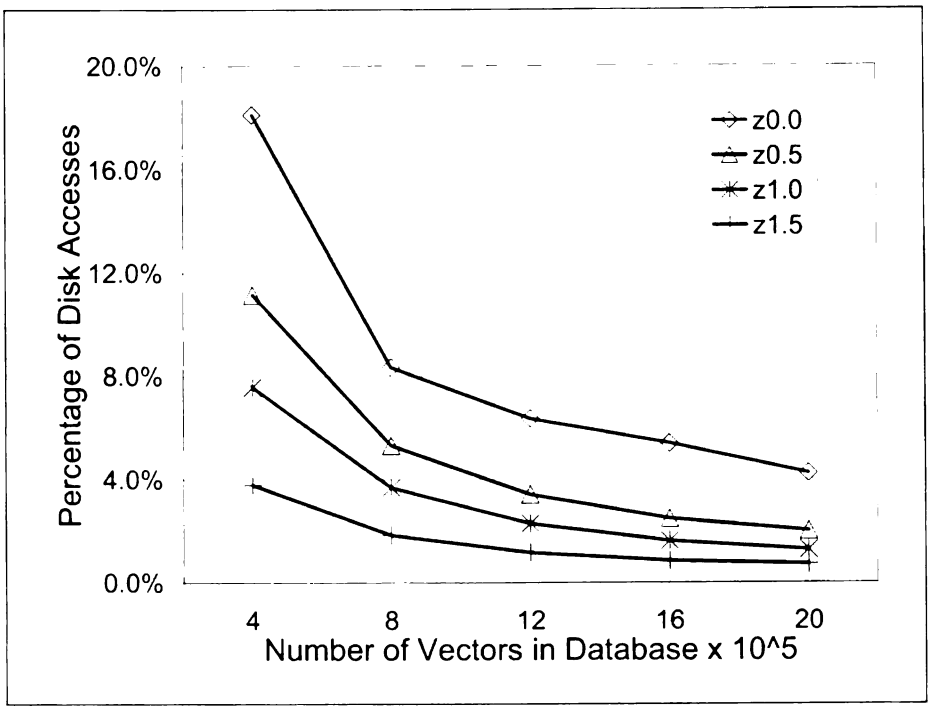


Figure 3.11. Performance of the k-NN algorithm using ND-tree vs. the linear scan on synthetic datasets with various sizes and zipf distributions

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

Experiments were conducted to show the effectiveness of our heuristic using the probability formulas presented in Section 3.2.3. 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, both of which utilize ND-tree. 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

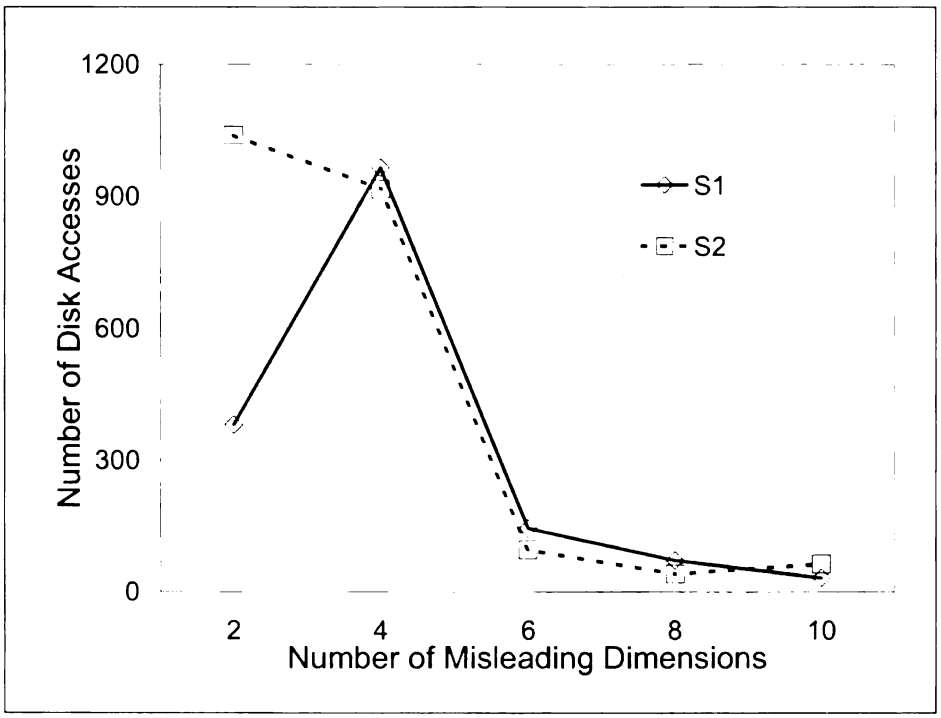


Figure 3.12. Performance of the k-NN algorithm using ND-tree on datasets with various misleading dimensions ($k = 1$)

highly relevant to the partitioning of vectors into subtrees early in the construction of 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 12, 13, and 14 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 datasets are known to contain misleading DM-

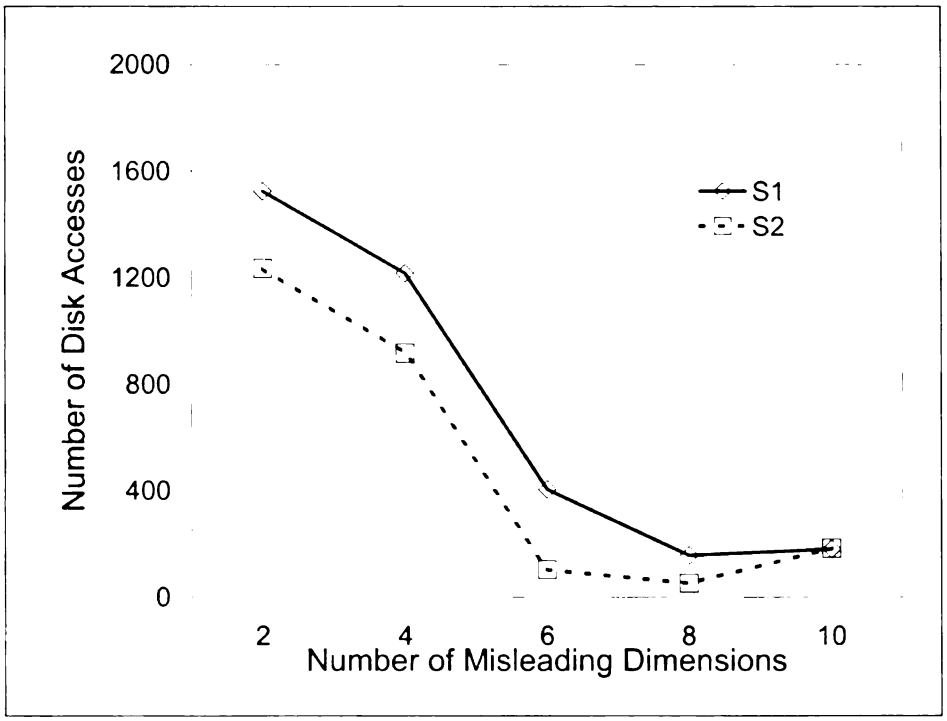


Figure 3.13. Performance of the k-NN algorithm using ND-tree on datasets with various misleading dimensions ($k = 5$)

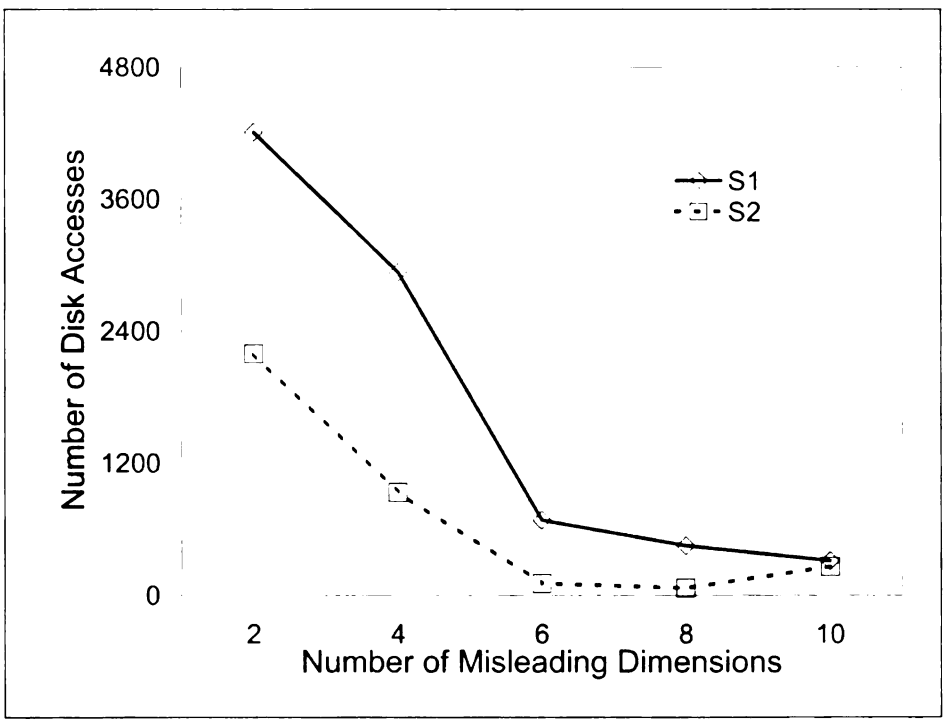


Figure 3.14. Performance of the k-NN algorithm using ND-tree on datasets with various misleading dimensions ($k = 10$)

BRs 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 $f_l(a)_{R,i}$ for each node at 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 at one level below the root node that meet this criterion; that is, for a particular dimension i, $\frac{1}{|S_i|} > 3 * f_l(a)_{R,i} \vee \frac{1}{|S_i|} < \frac{1}{3} * f_l(a)_{R,i}$.

The results in Figure 12 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 13 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 14 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 benefits when using heuristic version S2 increase as well. Additionally, we notice that in Figures 13 and 14, the performance when using heuristic version S2 becomes similar to the performance when using S1 as the number of misleading dimensions approaches the total number of dimensions. This is likely due to the reduction of

non-misleading paths available. As the number of non-misleading paths approaches 0, heuristic version S2 will be forced to choose similar paths to heuristic version S1.

3.4.5 Verification of Performance Model

Our theoretical performance estimation model was also verified using 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 3.15 through 3.17 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.

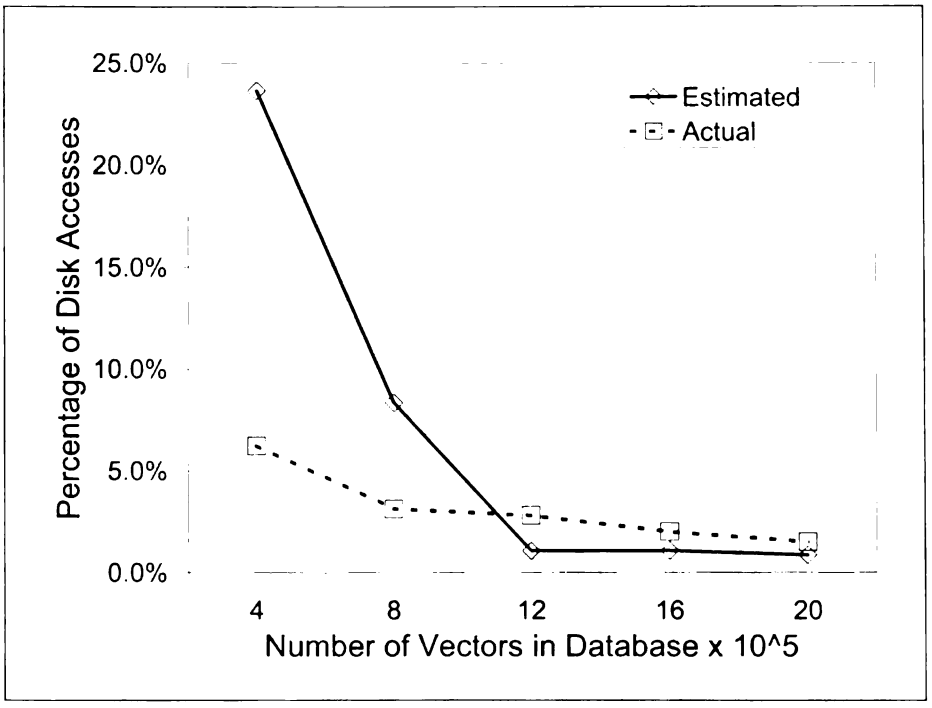


Figure 3.15. Estimated and Actual performance of the k-NN algorithm vs. the linear scan on synthetic datasets with various sizes ($k = 1$)

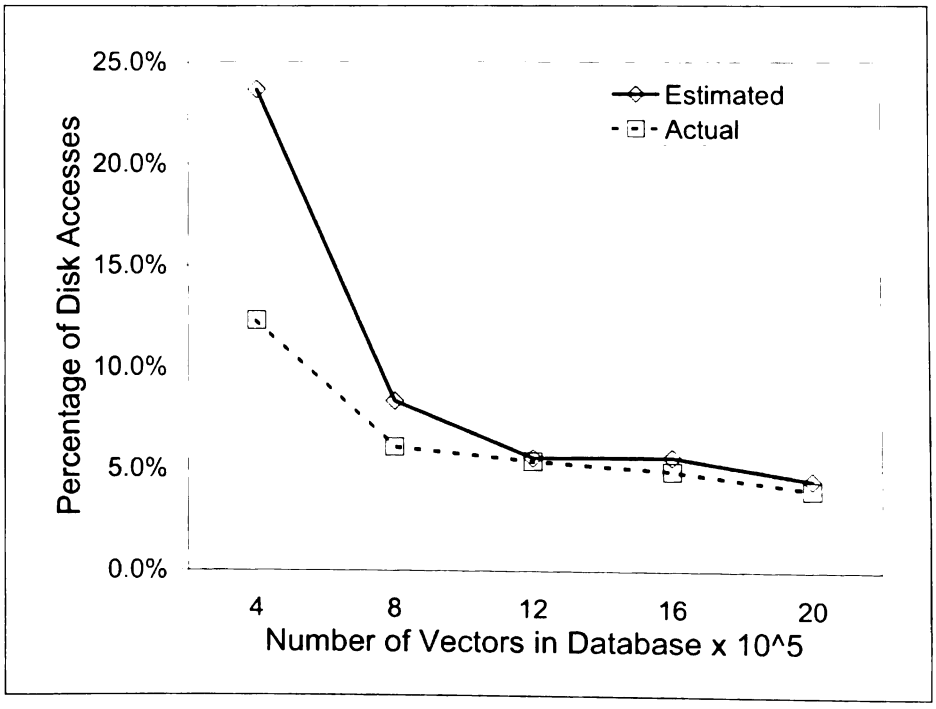


Figure 3.16. Estimated and Actual performance of the k-NN algorithm vs. the linear scan on synthetic datasets with various sizes ($k = 5$)

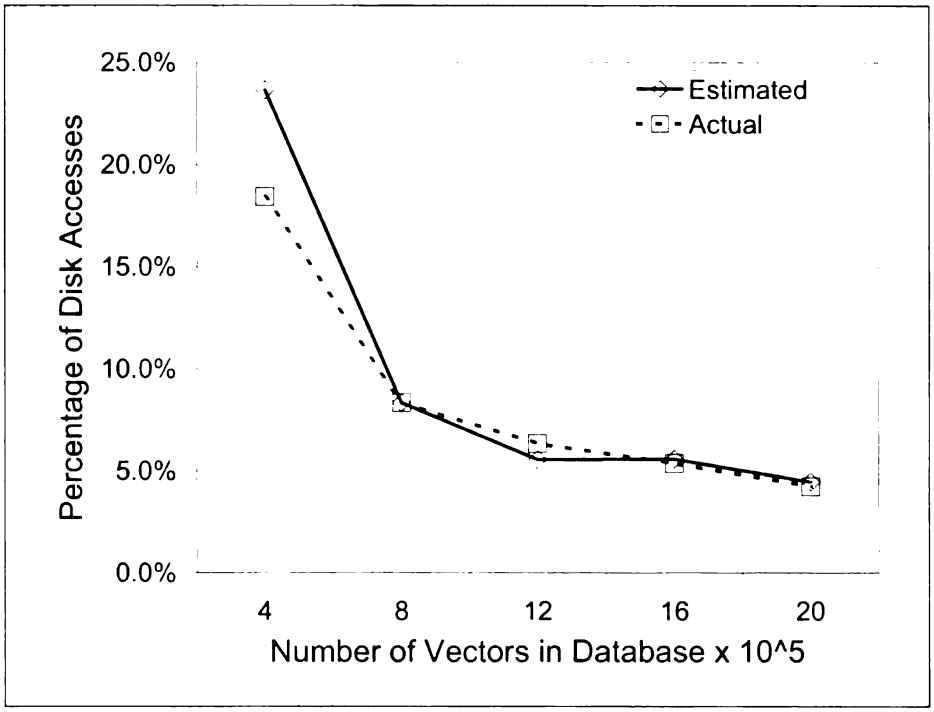


Figure 3.17. Estimated and Actual performance of the k-NN algorithm vs. the linear scan on synthetic datasets with various sizes ($k = 10$)

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 of a large database.

CHAPTER 4

Understanding Distance in NDDS

In this chapter we consider in more detail the issue of distance measurement in Non-Ordered Discrete Data Spaces. To efficiently handle a much broader array of applications than those presented in the previous chapter we present a generalized form of our Granularity Enhanced Hamming (GEH) distance. We then provide a new implementation of this distance.

4.1 Motivations and Challenges

A major problem with k-NN searching in NDDSs. as discussed in Chapter 3 is the non-determinism of the solution. That is, there is usually a large number of candidate solutions available which may obscure the result. This is mainly caused by the coarse granularity of the commonly used distance metric, the Hamming distance. An extension to the Hamming distance, termed the Granularity Enhanced Distance

(GEH) distance, was introduced in [32] as a solution to this problem. We demonstrated that the GEH distance greatly reduced the non-determinism of the solution set, as well as provided performance benefits, while maintaining the semantics of the original Hamming distance [32, 33]. However, the GEH distance introduced in [32] was tied directly to data point frequency values in a manner that may not be ideal in all scenarios. Applications/scenarios with other more relevant dataset characteristics (distribution, clustering, etc...) may not experience the same performance benefits seen in [32].

To address this issue, we introduced a generalized form of the GEH distance in [34]. This form may be optimized to a much broader set of applications than the original GEH distance presented in [32]. Conditions/constraints are presented that maintain the necessary distance metric properties to be used as a pruning metric. We show that the original GEH distance is, in fact, an instantiation of this generalized form. Further, we present a new instantiation of the generalized GEH form that demonstrates the benefits of adapting the generalized form for specific scenarios.

The rest of this chapter is organized as follows. Section 4.2 presents the generalized form of the GEH distance. Section 4.3 introduces a new ranking based GEH instantiation derived from the generalized form.

4.2 Generalized GEH Distance

The GEH distance, originally presented in [32], expanded upon the Hamming distance to provide more granularity while maintaining all of the semantics of the Hamming distance. This was accomplished by adding an adjustment value to the Hamming distance between two vectors based upon their matching elements. This is presented formally as follows:

$$D_{GEH}(\alpha, \beta) = \sum_{i=1}^{d} \left\{ \begin{array}{ll} 1 & \text{if } \alpha[i] \neq \beta[i] \\ \frac{1}{d} f(\alpha[i]) & \text{otherwise} \end{array} \right\}, \tag{4.1}$$

where

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

The value of $f_g(\alpha[i])$ is the number of occurrences of $\alpha[i]$ in the i^{th} dimension of the dataset, divided by the number of vectors in the dataset; essentially, a global frequency value. While this does provide a dramatic increase in the determinism of result sets when used in a similarity search, this distance metric may not provide an ideal distance semantic for all applications. Equation 4.1 is limited to applications where the global frequency of elements has some significance in the dataset.

Applications where other dataset characteristics provide a better semantic may not be able to benefit from using Equation 4.1 to the same degree as the results shown in [32]. To address this issue, we propose a generalization of the GEH distance that may be optimized to a much broader set of applications.

We observe that the Hamming distance assumes that a worst case match (i.e. a non-match) between two elements is represented by a distance of 1, while all other matches are represented by a distance of 0. We expand upon this by adding more granularity to the values assigned to different types of matches. We propose the following generalized form of the GEH distance to accomplish this goal:

$$D_{GEH}(\alpha, \beta) = D_{Hamm}(\alpha, \beta) + \frac{1}{C}\sum_{i=1}^{d} f_{geh}(\alpha[i], \beta[i]), \tag{4.2}$$

where

$$\begin{aligned}
&\text{Constraint 1:} && \forall_{\alpha,\beta} : C \geq d - D_{Hamm}(\alpha, \beta) \\
&\text{Constraint 2:} && \forall_{\alpha[i],\beta[i]} : 0 < f_{geh}(\alpha[i], \beta[i]) < 1 \\
&\text{Constraint 3:} && \forall_{\alpha[i],\beta[i]} : f_{geh}(\alpha[i], \beta[i]) = f_{geh}(\beta[i], \alpha[i]) \\
&\text{Constraint 4:} && \forall_{\alpha[i],\beta[i]} : \alpha[i] \neq \beta[i] \rightarrow f_{geh}(\alpha[i], \beta[i]) = 0.
\end{aligned}$$

Here, f_{geh} represents some function, chosen by an application expert, that will provide an adjustment to the Hamming distance for each dimension. The variable C

is a pseudo-constant[1] used to guarantee the adjustment values of f_{geh} do not become more dominant than the original Hamming distance. Constraint 1 indicates that the value of C must be greater than or equal to the number of matching dimensions between two vectors. Constraint 2 indicates that the result of function f_{geh} for the i^{th} element of the two vectors being considered must be in the range of $(0, 1)$ non-inclusive. Constraint 3 indicates that function f_{geh} must be symmetric. Constraint 4 indicates that the result of f_{geh} for the i^{th} elements of the two vectors being considered must equal 0 if the these two elements do not match.[2]

From Equation 4.2, 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), therefore preserving the original semantics of the Hamming distance.[3] Further, the four provided constraints allow the generalized GEH distance to maintain the metric properties necessary for use as a pruning metric in similarity searches as described in the following lemmas:

Lemma 4.2.1. *The generalized GEH distance maintains the Positiveness property*

[1]The term *pseudo-constant* is used to indicate that C is not strictly a constant, and may vary as long as Constraint 1 of Equation 4.2 is maintained.

[2]Note that both variables α and β are passed to to the adjustment function. This enables the adjustment function to be fully expressed whereby Constraint 4 may be verified.

[3]Many application specific solutions such as BLOSUM, employed in bioinformatics, reduce the non-determinism of solution sets by utilizing a cost matrix as a direct form of distance measurement. This is similar in theory to utilizing the adjustment function as a distance measure directly. Unfortunately, these methods do not preserve the semantics of the original Hamming distance and thus lose a level of portability between application environments. However, solutions such as BLOSUM may be incorporated into Equation 4.2 by utilizing the diagonal of the cost matrix for the adjustment function.

(i.e. $\forall_{x,y} : D_{GEH}(x, y) \geq 0$*).*

Proof. By maintaining the Hamming distance within the GEH distance, we are guaranteed a positive value if any elements between the two vectors do not match. Condition 2 indicates that all values resulting from matching elements will have non-negative values. □

Lemma 4.2.2. *The generalized GEH distance maintains the Strict Positiveness property (i.e.* $\forall_{x,y} : x \neq y \rightarrow D_{GEH}(x, y) > 0$*).*

Proof. This property is inherited by maintaining the Hamming distance within the generalized GEH distance, whereby any two vectors that are not equal will have a distance greater than '0' based upon a '1' being added to the distance for each non-matching dimension. Constraint 2 guarantees that the values added from function f_{geh} will all be non-negative. □

Lemma 4.2.3. *The generalized GEH distance maintains the Symmetry property (i.e.* $\forall_{x,y} : D_{GEH}(x, y) = D_{GEH}(y, x)$*).*

Proof. The Hamming distance is known to maintain symmetry between vectors. In addition, Constraint 3 guarantees that the values provided by the function f_{geh} will maintain symmetry as well. □

Lemma 4.2.4. *The generalized GEH distance maintains the Pseudo-Reflexivity property (i.e.* $\forall_{x,y} : D_{GEH}(x,x) < 1 \wedge x \neq y \rightarrow D_{GEH}(x,y) \geq 1$*).*[4]

Proof. This property is maintained due to Constraints 1 and 2, which stipulate that the additional value added to the Hamming distance will always be in the range of $(0,1)$, non-inclusive. Thus the distance between two vectors that exactly match will have a distance value less than '1'. Any vectors that are different in one or more dimensions will have a distance greater than or equal to '1'. □

Lemma 4.2.5. *The generalized GEH distance possesses the Triangular Inequality Property (i.e.* $\forall_{x,y,z} : D_{GEH}(x,y) + D_{GEH}(y,z) \geq D_{GEH}(x,z)$*).*

Proof. We first consider the Hamming portion of the generalized GEH distance. For any dimension $i \in [1,d]$, if $x_i \neq z_i$ then either $x_i \neq y_i \oplus z_i \neq y_i$ or $x_i \neq y_i \wedge z_i \neq y_i$. Thus for each dimension i where the right side of the inequality (i.e. $D_{GEH}(x,z)$) would be incremented by an integer value of '1', the left side of the inequality (i.e. $D_{GEH}(x,y) + D_{GEH}(y,z)$) would be incremented by an integer value of either '1' or '2', thus maintaining the inequality. Next, we consider the adjustment portion of the GEH distance (i.e. $\frac{1}{C} f_{geh}()$). For any dimension $i \in [1,d]$, if $x_i = z_i$ then either $x_i = y_i \wedge z_i = y_i$ or $x_i \neq y_i \wedge z_i \neq y_i$. Thus, due to constraints 2 and 3,

[4]Note that the traditional property of Reflexivity (i.e. $\forall x : D(x,x) = 0$) is replaced by the property of Pseudo-Reflexivity. This is a reasonable substitution in an NDDS due to two vectors exactly matching each other still being identifiable from all other pairs of vectors based only upon the distance between them.

for all dimensions where this is the case, the left side will either be incremented by twice as much as the right side or be incremented by an integer value of '2' while the right side is incremented by some value less than '1'. Constraint 4 indicates that no additions will be made if the values in the dimension match, leaving the Hamming component to be dominant. Thus the adjustment values maintain the inequality. □

4.3 Ranking Based GEH Instantiation

As described in [14], many search algorithms demonstrate better performance when the distances between data points are distributed evenly throughout the distance range. We note that the original GEH distance, Equation 4.1, is likely to result in a heavily skewed distribution of possible distances. As the alphabet size grows, the likely values of $f_g(\alpha[i])$ trend closer to '0' leading to a clumping of distance values close to the floor value. Additionally, setting $C = d$ results in C having a dominant role in the distance value as the dimensionality of the dataset grows larger. To address these issues, we propose a new GEH distance instantiation:

$$\begin{aligned} f_{geh}(\alpha[i], \beta[i]) &= \frac{\text{rank}_i(\alpha[i])}{|A_i|+1} \\ C &= d - D_{Hamm}(\alpha, \beta) + 1 \end{aligned} . \tag{4.3}$$

Here, the term $\text{rank}_i(\alpha[i])$ indicates the global rank of element $\alpha[i]$ among the alphabet set in dimension i. The ranking mechanism employed should be set by an

Table 4.1. Varying Dimensionality

	Hamm.	Freq.	Rank
d = 5	36	7	6
d = 10	968	472	480
d = 15	5914	4591	4675
d = 20	8294	8228	8232

application expert on the condition that it results in the different elements of the alphabet receiving ranking values of $[1, |A|]$ inclusive. The value of C tracks to the number of matching dimensions between vectors α and β. As an example ranking mechanism, we consider the frequency of elements within a dimension, applying a higher rank (lower integer value) to elements that occur more frequently, and a lower rank (higher integer value) to elements that occur less frequently. For example, if the alphabet set in dimension i consists of $\{a, b, c\}$, where a appears in 20% of the vectors, b appears in 50% of the vectors, and c appears in 30% of the vectors in dimension i, the rank of each of the elements in dimension i would be as follows: $a \rightarrow 3$, $b \rightarrow 1$, and $c \rightarrow 2$. Although an element's frequency within a dimension still plays a role in the determination of the GEH distance (in this example), the ranking mechanism maintains a uniform distribution of distance values over varying alphabet sizes. Additionally, having the value of C track to the number of matching dimensions rather than the dimensionality of the dataset reduces the dominance of C as the dimensionality of the dataset grows larger.

To evaluate the benefits of an adaptable distance metric, we performed a series of k-NN queries utilizing the GEH distance implementations in Equations 4.1 and

Table 4.2. Varying Zipf Distribution

	Hamm.	Freq.	Rank
zipf 0.0	968	472	480
zipf 0.5	693	399	301
zipf 1.0	381	233	126
zipf 1.5	105	73	30

4.3 as well as the Hamming distance. Table 1 shows a comparison of I/O results while searching uniformly distributed datasets of varying dimensionality. These results demonstrate a scenario where the frequency based GEH implementation provides slightly better search performance than the rank based GEH implementation. Further, our results agree with those in [52] linking a decreasing performance with increasing dimensionality.[5] Table 2 shows a comparison of the I/O results while searching 10-dimensional datasets of varying zipf distribution. For this scenario, use of the new ranking based GEH implementation provides a strong performance improvement over the frequency based GEH distance implementation. This is in agreement with the results shown in [14] concerning search performance and distance value distribution. These results highlight scenarios where Equations 4.1 and 4.3 provide search performance improvements specific to each case, thus demonstrating the benefits of an adaptable distance metric.

[5]Note that for the largest dimensionality tested, $d = 20$, the I/O results when using both ranking based and frequency based GEH implementations begin to approach each other. We attribute this to the dimensionality of the dataset playing a less dominant role in Equation 4.3 than in Equation 4.1

CHAPTER 5

k-Nearest Neighbor in Hybrid Data Spaces

In this chapter, we consider k-Nearest Neighbor (k-NN) searching in Hybrid Data Spaces. Searching in HDSs presents several new challenges not presented in either CDS or NDDS searching applications. We examine these issues and discuss methods to resolve them. Further, we extend the theoretical performance model presented in Chapter 3 to HDSs.

5.1 Motivation and Challenges

Nearest neighbor searches/queries in Hybrid Data Spaces (HDS) are becoming increasingly useful in many contemporary applications such as machine learning, information retrieval and security, bioinformatics, multimedia, and data-mining. Consider the following information retrieval task. Given a set of network sites and a range of

times, determine the set of k network intrusions that match a set of query criteria most closely. When examining records of intrusions, the sites in which they occurred could be considered discrete data, in that an intrusion either did or did not occur at that site. The times active in a particular site could be considered continuous data, in that an intrusion may have been active over only a period of time.

Several techniques have been proposed in the literature to support such searches in both continuous (ordered) data spaces and non-ordered discrete data spaces. A majority of these techniques utilize a multidimensional index structure such as R*-tree[6] or the ND-tree[43]. Little work has been reported in the literature on supporting efficient nearest neighbor searches in hybrid data spaces.

Efficient index based nearest neighbor searching is dependent upon the usefulness of information maintained in the search index structure. When searching an index containing hybrid data, a difficult scenario occurs when one set of dimensional data is unknown. This scenario is analogous to using a non-hybrid index, such as R*-tree or ND-tree, to maintain hybrid data, based on their continuous or discrete subspace. If this structure remains unmodified, performing nearest neighbor searches becomes impractical due to the values of the non-native dimensions (discrete for R*-tree, continuous for ND-tree).

To guarantee all valid neighbors are found in these scenarios, additional consid-

erations must be taken into account. First, when examining the current set of objects/records found to determine a search range, it must be assumed that all non-native dimensions that are not maintained in the index structure differ in value from the query vector, i.e.:

$$D(q, NN_k) = D_N(q, NN_k) + d_M, \tag{5.1}$$

where D_N is the distance in native dimensions between the query vector q and the k^{th} neighbor NN_k and d_M is the maximum distance possible between the non-native dimensions in the dataset and q. Second, when comparing a bounding box or possible new neighbor, the opposite assumption must be made, i.e.:

$$D(q, X) = D_N(q, X) + 0, \tag{5.2}$$

where X is either a bounding box or object. This is due to the possibility of some vector within X (or X itself) exactly matching the query vector on all non-native dimensions that are not maintained in the index structure. Examining these facts, it is clear that as the number of non-native dimensions grows, it becomes increasingly difficult to exclude any portion of the index from the search.

To address these issues, we consider performing k-Nearest Neighbor (k-NN)

searches utilizing the CND-tree, a recently proposed multidimensional index for HDS [15]. When considering k-NN searches utilizing an HDS index, we present a best-first searching algorithm that utilizes characteristics of HDS in its heuristics to reduce the I/O cost of determining a valid set of k neighbors. Further, we present a theoretical performance model for this algorithm based on the characteristics of an HDS and the hybrid index

The rest of this chapter is organized as follows. Section 5.2 presents a brief analysis of the different stages of a k-NN search. Section 5.3 presents our best first algorithm, its heuristics, and our theoretical performance model. Section 5.4 discusses experimental results.

5.2 Nearest Neighbor Search Stages

In this section, we present a discussion of the different stages of a k-NN search. By properly categorizing these stages, we are able to develop heuristics that improve search performance with more accuracy. The reader is recommended to review Chapter 2 for an overview of the CND-tree.

When performing k-NN queries using a multi-dimensional index structure, the act of traversing the tree (assuming a minimum distance based ordering) can be broken into three distinct stages: range reduction, overlap, and exhaustive. In this section

we consider the first two stages in more detail, while the exhaustive stage will be revisited in Section 5.4.2.

The range reduction stage occurs while the current search range, r_c, is greater than the final search range value. During this stage, new neighbors/objects that are found are likely to result in reducing the current search range. The order in which nodes are visited has a direct impact upon the I/O cost of this stage.

The overlap stage occurs when the search range has been reduced to its final value, but there still exists nodes R in the search path whose bounding box is such that $D(q, R_i) < r_c$. The amount of overlap between nodes within data organization structures directly affects the I/O cost of this stage. Data organization structures with a minimal amount of overlap within their internal nodes are likely to incur less I/O costs when searching during this stage than data organizational structures with a greater amount of area overlap.

Figures 5.1, 5.2 and 5.3 break down the I/O cost due to the range reduction and overlap stages when searching a CND-tree. Figures 5.1 and 5.2 show the effects when either the number of continuous or discrete dimensions is held constant (at 6) and the number of the other dimensions varies. Figure 5.3 shows the effects when the number of both continuous and discrete dimensions is held constant (6 continuous, 3 discrete) and the number of records in the database varies. As seen from these figures, the I/O

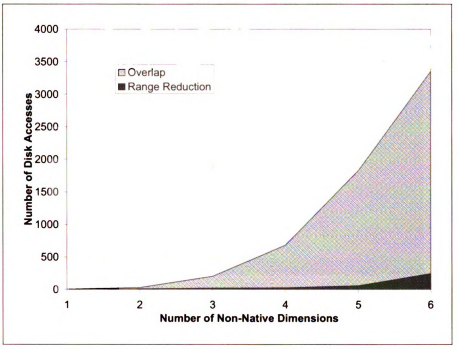


Figure 5.1. Search stage I/O with variable number of continuous dimensions

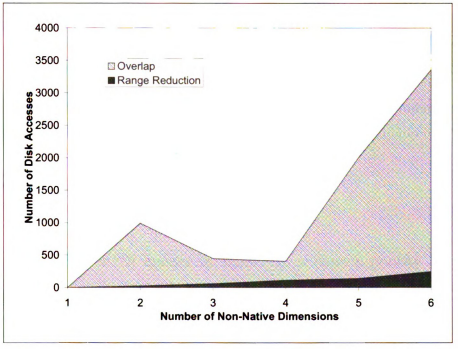


Figure 5.2. Search stage I/O with variable number of discrete dimensions

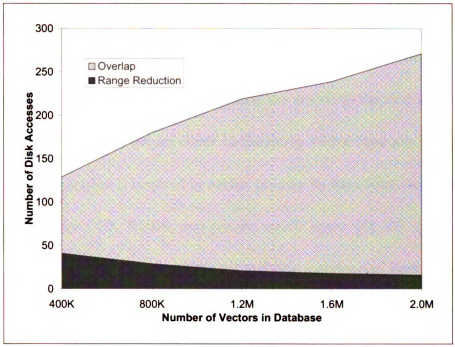


Figure 5.3. Search stage I/O with variable database size

costs of the overlap stage rise dramatically as the number of dimensions increases. This stage has a much less dramatic increase in I/O cost as the size of the database increases. Additionally, the I/O cost of the range reduction stage actually reduces as the database size grows, while increasing as the number of dimensions increases.[1]

5.3 Search Algorithm

To efficiently process k-NN queries in HDSs, we present a priority backtracking index-based searching algorithm that utilizes properties of the CND-tree for HDS. This algorithm initiates a search in the root node and then visits each subsequent

[1]This is a similar phenomenon to what was reported by Chavez et al. [14]. Although Chavez et al. focus primarily on searching in general metric spaces, the same principles apply here.

node based upon a "best-first" criterion that is defined in the heuristics. When k possible neighbors are retrieved, the searching algorithm uses the distance information of the neighbors collected to start pruning the 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 inspired by earlier priority backtracking based implementations presented for CDS, NDDS, and generic metric space [33, 27]. Our algorithm extends these implementations to HDSs by utilizing metrics and heuristics suitable for such a space. In particular, we introduce the notion of using an estimated match likelihood value to help prioritize ordering. Additionally, we present a performance model to accurately estimate the I/O cost of executing our algorithm.

5.3.1 Match Likelihood

Consider a query vector q and a bounding box R. If the total number of dimensions in the dataset is $d_t = d_D + d_C$ and the distance between q and R is $D(q, R) = x$, then it is possible that there exists an object in the subtree associated with R that matches q on $d_t - x$ dimensions (assuming a distance metric similar to Equation 2.4 is utilized). In a multidimensional index structure such as the ND-tree, R*-tree, or CND-tree this is not the most likely scenario. More likely, there exist several objects in the subtree associated with R that match q on a subset of dimensions that are represented in R. The projections of these objects at higher levels in the index tree

can create a misleading picture of the vectors in its associated subtree, similar to the concept discussed in Chapter 3 for NDDS searching.

We may infer from the bounding box of R how likely the information that R contains is likely to represent the vectors in its subtree. For the purposes of this discussion, we will assume that our index tree has maintained a relatively uniform distribution of elements within its subtrees. When we consider the CND-tree as our indexing method, the probability of accessing a subtree with associated bounding box $R = S_{D1} \times \ldots \times S_{Dd} \times S_{C1} \ldots \times S_{Cd}$ will yield a particular element a in dimension i may be estimated as follows (assuming $a \in R_i$):

$$p(a)_{R,i} = \left\{ \begin{array}{ll} \frac{1}{|S_i|} & \text{if } i \text{ is discrete} \\ \frac{\delta t}{\max S_i - \min S_i} & \text{if } i \text{ is continuous} \end{array} \right\}, \tag{5.3}$$

where the match likelihood of encountering a vector $\alpha = \alpha_1, \alpha_2, \ldots, \alpha_t$ in the subtree associated with R may be approximated as follows:

$$P(\alpha)_R = \sum_{d_t} \left\{ \begin{array}{ll} p(\alpha[i])_{R,i} & \text{if } \alpha[i] \in R \\ 0 & \text{otherwise} \end{array} \right\}. \tag{5.4}$$

Equation 5.3 calculates the reciprocal of the magnitude of the alphabet set on dimension i of R for discrete dimensions and the quotient of the threshold value and the range of the set on dimension i of R for continuous dimensions. Equation 5.4

then determines the summation of these values for all elements in vector α that are represented in R. It should be noted that more in depth methods were presented for estimating this likelihood value for NDDS in [33]. However, we are only interested in using this value to break ties in cases where the minimum distance between a subset of the nodes and the query vector is the same. Thus, the generalizations of Equations 5.3 and 5.4 are sufficient.

5.3.2 Algorithm Description

In a worst case scenario, a search would encompass the entire index structure. However, our extensive experiments have shown that utilizing the following heuristics eliminates most search paths before they need to be traversed.

*H*1: *If the minimum distance between the query vector and HMBR R is greater than the current range, then prune the subtree associated with R.*

*H*2: *Access subtrees in the ascending order of the minimum distance between the query vector and the associated HMBR R.*

*H*3: *In the event of a tie between subtrees due to heuristic H*2*, order those subtrees*

that tied in the descending order of the match likelihood (Equation 5.4) between the query vector and the associated HMBR R.

Our k-NN algorithm applies these heuristics to prune non-productive subtrees and determines the access order of the remaining subtrees during a best-first traversal of the CND-tree.

Given a CND-tree for vectors from HDS X_d with root node T, Algorithm Priority k-NN Query finds a set of k-nearest neighbors, NN, to query vector q, where $NN \subseteq X$, $|NN| = k$, and $\forall_{u \in NN, v \in X-NN} D(q,u) \leq D(q,v)$. It utilizes a priority queue, labeled Q, of CND-tree nodes that is sorted based upon heuristics $H2$ and $H3$. It invokes two functions: *FindSubtrees* and *RetrieveNeighbors*. The former finds all subtrees of the current node N that are within *Range* of the query vector and adds their nodes to Q. The latter updates the list of k-nearest neighbors, NN, using vectors in the current leaf node.

In the algorithm, step 1 initializes the range variable. Step 2 starts the search at the root node by inserting it into Q. Steps 3 - 14 traverse the CND-tree, where steps 4 and 5 select the next node to be visited and remove it from Q. Steps 6 - 8 deal with non-leaf nodes. Step 7 invokes *FindSubtrees* to collect all subtrees of the current node that are within *Range*. Step 8 sorts Q according to heuristic $H2$

Algorithm 4 Priority k-NN Query

```
1: Range = d_t
2: Q.Insert( T )
3: while !Q.Empty() do
4:    N = Q.Top()
5:    Q.Pop()
6:    if N.Height > 1 then
7:       FindSubtrees( N, Range, q, Q )
8:       Q.Sort()
9:    else
10:      RetrieveNeighbors( N, Range, q, NN )
11:      Range = NN[k].Dist()
12:      Q.Prune( Range )
13:   end if
14: end while
15: Return NN
```

and $H3$. Steps 9 - 12 deal with leaf nodes. Step 10 invokes *RetrieveNeighbors* to update the list of current k-nearest neighbors. Step 11 updates *Range* to equal the distance of the current k^{th} neighbor from the query vector. Step 12 prunes nodes from Q according to heuristic $H1$. Finally, step 15 returns the result. Note that *FindSubtrees* only updates Q with subtrees that are currently within *Range*. If no subtrees of the current node are within range, no new nodes will be added to Q. The WHILE loop in steps 3 - 14 is terminated when all subtrees that are within the current range have been visited.

5.3.3 Performance Model

To analyze the performance of our k-NN search algorithm, we conducted both empirical and theoretical studies. The results of our empirical studies are presented

in Section 5.4. In this section, we present a theoretical model for estimating the performance of our search algorithm. To accomplish this, we first derive an estimate of the likely search range that will yield k objects/neighbors. We then derive an estimate of how many tree nodes/pages will be accessed by a search of a variable range. For our presentation, we assume that our algorithm employs heuristics $H1$, $H2$, and $H3$ and that our dataset is reasonably uniform. For reference, many of the variables used throughout this section are maintained in Table 1.

The likely distance of the k^{th} neighbor from the query point represents the final search range that can be used to prune nodes from the search path. To determine this distance, we estimate the ratio of area within a specific distance and the total possible area of the dataset, similar to the method employed for NDDSs in [33].

The area within a distance z from a point p in a $d_D + d_C$ dimensional HDS with alphabet A for each discrete dimension, span S for each continuous dimension, and threshold value δt for Equation 2.4 is analogous to the number of unique points within a hyper sphere of radius z centered at point p:

$$Area(z) = \sum_{x=0}^{z} \left[\sum_{y=\max(0,x-d_M)}^{\min(x,d_m)} [f_a(x,y)] \right] , \tag{5.5}$$

where

Table 5.1. Performance Model Variables

n_i	num of nodes at layer i
w_i	num of vectors in a node at layer i
H	height of tree
R	$= \lceil \frac{S}{2*\delta t} \rceil$
d_M	$= \max(d_D, d_C)$
d_m	$= \min(d_D, d_C)$
a_M	$= \left\{ \begin{array}{ll} \lvert A \rvert & \text{if } d_M = d_D \\ R & \text{otherwise} \end{array} \right\}$
a_m	$= \left\{ \begin{array}{ll} \lvert A \rvert & \text{if } d_m = d_D \\ R & \text{otherwise} \end{array} \right\}$
d_{M1}	$= \left\{ \begin{array}{ll} \max(d'_{D,i}, d''_{D,i}) & \text{if } d_M = d_D \\ \max(d'_{C,i}, d''_{C,i}) & \text{otherwise} \end{array} \right\}$
d_{M2}	$= \left\{ \begin{array}{ll} \min(d'_{D,i}, d''_{D,i}) & \text{if } d_M = d_D \\ \min(d'_{C,i}, d''_{C,i}) & \text{otherwise} \end{array} \right\}$
d_{m1}	$= \left\{ \begin{array}{ll} \max(d'_{D,i}, d''_{D,i}) & \text{if } d_m = d_D \\ \max(d'_{C,i}, d''_{C,i}) & \text{otherwise} \end{array} \right\}$
d_{m2}	$= \left\{ \begin{array}{ll} \min(d'_{D,i}, d''_{D,i}) & \text{if } d_m = d_D \\ \min(d'_{C,i}, d''_{C,i}) & \text{otherwise} \end{array} \right\}$
B_{M1}	$= \left\{ \begin{array}{ll} B'_{D,i} & \text{if } d_M = d_D \wedge d_{M1} = d'_{D,i} \\ B''_{D,i} & \text{if } d_M = d_D \wedge d_{M1} = d''_{D,i} \\ B'_{C,i} & \text{if } d_M = d_C \wedge d_{M1} = d'_{C,i} \\ B''_{C,i} & \text{otherwise} \end{array} \right\}$
B_{M2}	$= \left\{ \begin{array}{ll} B'_{D,i} & \text{if } d_M = d_D \wedge d_{M2} = d'_{D,i} \\ B''_{D,i} & \text{if } d_M = d_D \wedge d_{M2} = d''_{D,i} \\ B'_{C,i} & \text{if } d_M = d_C \wedge d_{M2} = d'_{C,i} \\ B''_{C,i} & \text{otherwise} \end{array} \right\}$
B_{m1}	$= \left\{ \begin{array}{ll} B'_{D,i} & \text{if } d_m = d_D \wedge d_{m1} = d'_{D,i} \\ B''_{D,i} & \text{if } d_m = d_D \wedge d_{m1} = d''_{D,i} \\ B'_{C,i} & \text{if } d_m = d_C \wedge d_{m1} = d'_{C,i} \\ B''_{C,i} & \text{otherwise} \end{array} \right\}$
B_{m2}	$= \left\{ \begin{array}{ll} B'_{D,i} & \text{if } d_m = d_D \wedge d_{m2} = d'_{D,i} \\ B''_{D,i} & \text{if } d_m = d_D \wedge d_{m2} = d''_{D,i} \\ B'_{C,i} & \text{if } d_m = d_C \wedge d_{m2} = d'_{C,i} \\ B''_{C,i} & \text{otherwise} \end{array} \right\}$

$$f_a(x,y) \quad = \binom{d_m}{y}(a_m-1)^y\binom{d_M}{x-y}(a_M-1)^{x-y} \ .$$

The probability of an object existing within a distance z from any point p, may be calculated as follows:

$$P_{exists}(z) = \frac{Area(z)}{Area(d_D + d_C)}. \tag{5.6}$$

The product of the number of points N within the dataset and the probability of a point existing within a distance z from any point p yields the number of points likely to exist in that subspace:

$$L(z) = P_{exists}(z) * N. \tag{5.7}$$

It is reasonable to assume that a minimum distance that needs to be searched is one less than the distance that is likely to yield at least k neighbors. Thus a search range $r = z$, is found be solving Equation 5.7 such that $L(r+1) \geq k$ and $L(r) < k$. The reason for this is that a search range that yields more than k neighbors is likely to extend beyond the overlap stage (Section 5.2).

Next, we determine how many pages are likely to be accessed by a search with

a range of r. We derive this value in a similar fashion to the method derived by Qian et al. [43] for NDDS. As new records are inserted into the CND-tree, some dimensions may be split and others may not be. Assuming a reasonably uniform distribution and independence among dimensions, the probability for the length of an unsplit edge of an HMBR to be '1' is as follows[2]:

$$\begin{aligned} T_{D,i,1} &= \frac{1}{|A|^{w_i}-1}, \\ T_{C,i,1} &= \frac{1}{R^{w_i}-1}. \end{aligned} \tag{5.8}$$

Using Equation 5.8, the probability for anthebibliography edge length of an HMBR to be j is as follows:

$$\begin{aligned} T_{D,i,j} &= \frac{\binom{|A|}{j}\left[j^{w_i}-\sum_{k=1}^{j-1}\binom{j}{k}\frac{|A|^{w_i}}{\binom{|A|}{k}}*T_{i,k}\right]}{|A|^{w_i}}, \\ T_{C,i,j} &= \frac{\binom{R}{j}\left[j^{w_i}-\sum_{k=1}^{j-1}\binom{j}{k}\frac{R^{w_i}}{\binom{R}{k}}*T_{i,k}\right]}{R^{w_i}}. \end{aligned}$$

Hence, the expected edge length of an HMBR of a node at layer i is:

$$\begin{aligned} s_{D,i} &= \sum_{j=1}^{|A|} j * T_{D,i,j}, \\ s_{C,i} &= \sum_{j=1}^{R} j * T_{C,i,j}. \end{aligned} \tag{5.9}$$

We can assume that a sequence of n node splits will split reasonably evenly

[2]For clarity, equations with a D or C subscript are relevant to discrete or continuous dimensions only, respectively.

throughout the $d_D + d_C$ dimensions. Each split will divide the component set of the HMBR on the relevant dimension into two equal sized component subsets. To obtain n_i nodes at layer i of the CND-tree, the expected number of splits needed is $\log_2 n_i$. As such, at any given time, it is likely that some nodes will be split one more time than others. This can be determined as follows:

$$\begin{aligned} d_i'' &= \lfloor (\log_2 n_i) \mod (d_D + d_C) \rfloor, \\ d_i' &= (d_D + d_C) - d_i'', \end{aligned} \tag{5.10}$$

where d_i'' represents those dimensions that have been split an extra time.

If we assume these splits have been distributed evenly among continuous and discrete dimensions, we have the following:

$$\begin{aligned} d_{D,i}'' &= \lceil \tfrac{d_D \cdot d_i''}{d_D + d_C} \rceil, \\ d_{C,i}'' &= \lfloor \tfrac{d_C \cdot d_i''}{d_D + d_C} \rfloor, \\ d_{D,i}' &= \lceil \tfrac{d_D \cdot d_i'}{d_D + d_C} \rceil, \\ d_{C,i}' &= \lfloor \tfrac{d_C \cdot d_i'}{d_D + d_C} \rfloor. \end{aligned} \tag{5.11}$$

So, the HMBR of a node at layer i has the following expected edge length on d'' and d' dimensions respectively:

$$
\begin{aligned}
s''_{D,i} &= \frac{S_{D,H}}{2^{\lceil \frac{\log_2 n_i}{d_D+d_C} \rceil}}, \\
s''_{C,i} &= \frac{S_{C,H}}{2^{\lceil \frac{\log_2 n_i}{d_D+d_C} \rceil}}, \\
s'_{D,i} &= \left\{ \begin{array}{ll} s_{D,i} & \text{if } \frac{\log_2 n_i}{d_D+d_C} < 1 \\ \frac{S_{D,H}}{2^{\lceil \frac{\log_2 n_i}{d_D+d_C} \rceil}} & \text{otherwise} \end{array} \right\}, \\
s'_{C,i} &= \left\{ \begin{array}{ll} s_{C,i} & \text{if } \frac{\log_2 n_i}{d_D+d_C} < 1 \\ \frac{S_{C,H}}{2^{\lceil \frac{\log_2 n_i}{d_D+d_C} \rceil}} & \text{otherwise} \end{array} \right\}.
\end{aligned}
\tag{5.12}
$$

Thus, for any node, the probability for a component of a query vector to be covered by the corresponding component of the HMBR of that node for discrete and continuous dimensions, respectively, is given as follows:

$$
\begin{aligned}
B'_{D,i} &= \frac{s'_{D,i}}{|A|}, \\
B''_{D,i} &= \frac{s''_{D,i}}{|A|}, \\
B'_{C,i} &= \frac{s'_{C,i}}{R}, \\
B''_{C,i} &= \frac{s''_{C,i}}{R}.
\end{aligned}
\tag{5.13}
$$

Using Equation 5.13, the probability for a node to be accessed by a query of range/distance h can be evaluated recursively as follows:

$$
P_{i,h} = \sum_{k=0}^{h} \left[\sum_{s=s_0}^{s_f} \left[\sum_{m=m_0}^{m_f} \left[f * \sum_{p=p_0}^{p_f} g \right] \right] \right], \tag{5.14}
$$

where

$$
\begin{aligned}
s_0 &= \max(0, k - d_M), \\
s_f &= \min(k, d_m), \\
m_0 &= \max(0, s - d_{m1}), \\
m_f &= \min(s, d_{m2}), \\
p_0 &= \max(0, k - s - d_{M1}), \\
p_f &= \min(k - s, d_{M2}), \\
f &= \binom{d_{m2}}{m} B_{m2}^{d_{m2}-m}(1 - B_{m2})^m * \binom{d_{m1}}{s-m} B_{m1}^{d_{m1}-s+m}(1 - B_{m1})^{s-m}, \\
g &= \binom{d_{M2}}{p} B_{M2}^{d_{M2}-p}(1 - B_{M2})^p * \binom{d_{M1}}{k-s-p} B_{M1}^{d_{M1}-k+s+p}(1 - B_{M1})^{k-s-p}.
\end{aligned}
$$

Thus, the expected number of node/page accesses for performing a query with search range r can be estimated as:

$$IO = 1 + \sum_{i=0}^{H-1} (n_i * P_{i,r}). \tag{5.15}$$

5.4 Experimental Results

Our k-NN searching algorithm was implemented using a CND-tree, an ND-tree, an R*-tree, and a linear scan. For our experiments, the linear scan is considered twice: once with native non-ordered discrete dimensions and once with native continuous dimensions. Both the ND-tree and the R*-tree were modified to store hybrid data in their leaves. This modification affects the shape of each of these trees but does not

incur a change in either of their insertion/split algorithms.[3] All experiments were ran on a PC under OS Linux Ubuntu. The I/O block size was set at 4K for all trees. Two series of datasets were used. The first consists of 1M vectors with six native dimensions and a variable number of non-native dimensions (1 - 6). The second set has six native dimensions and three non-native dimensions and a variable number of vectors.[4] Each experimental data reported here is the average over 100 random queries with $k = 10$.

5.4.1 Effects of Heuristics and Datasets

The first set of experiments was conducted to show the effects of the dimensionality and database (DB) size on the query performance. Figures 5.4 and 5.5 show this data. As both Figure 5.4 and 5.5 show, the CND-tree provides by far the most promising results in all tests.[5] Due to this, the remainder of the experimental data considers only the CND-tree. It should be noted in Figure 5.5 that the ND-tree

[3]Each leaf node is structured to hold data with native and non-native dimensions. The non-native dimensions play no part in determining the composition of the covering hyper rectangle. However, the extra dimensional information requires more space for each object than what would be needed for native dimensional data only. While this extra space requirement does decreases the amount of objects that a leaf node can contain before overflowing, the extra information maintained negates the need to maintain the search constant d_M (Equation 5.1). However, because no changes have been made to the internal nodes of these trees, Equation 5.2 must still be observed.

[4]This configuration is chosen, as it is likely the amount of non-native dimensional information will be significantly smaller than the amount of native dimensional information in most real world scenarios.

[5]Note that for the linear scan results, "Nat=D" indicates a scan of hybrid data with native non-ordered discrete dimensions and "Nat=C" indicates a scan of hybrid data with native continuous dimensions. The datasets used for each of these experiments are identical to those used for the ND-tree and R*-tree based searching, respectively.

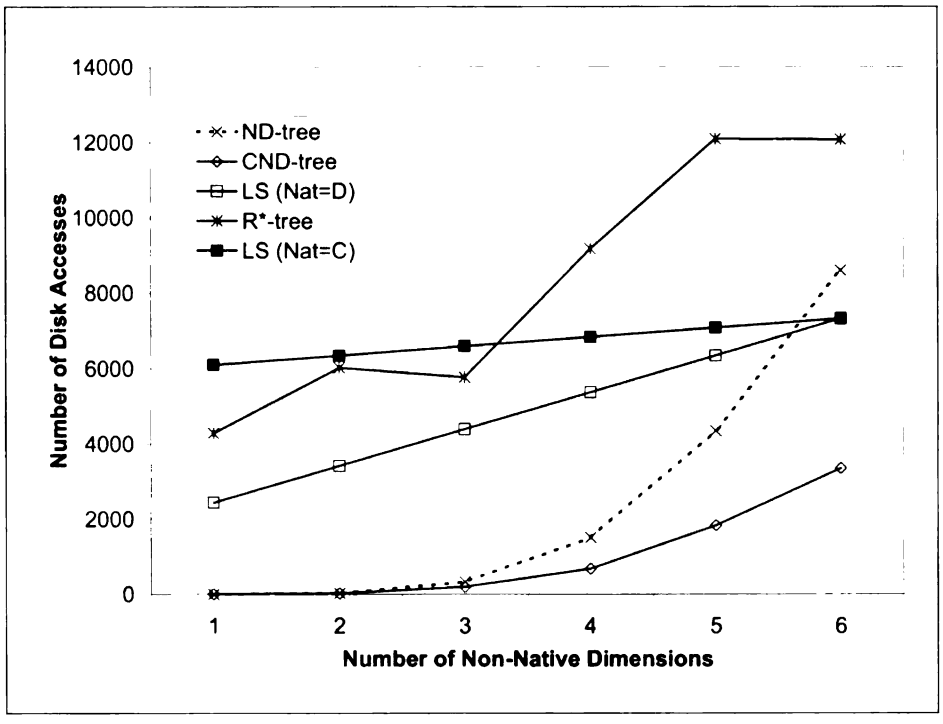


Figure 5.4. Performance I/O with variable number of non-native dimensions

appears to provide somewhat similar results to the CND-tree as the size of the database increases. Figure 5.6 shows that when viewed independantly from other search results, the CND-tree still provides significant performance benefits over the ND-tree as the number of vectors in the database increases.

The second set of experiments was conducted to show the effectiveness of our search heuristics compared to similar heuristics utilized in a non-hybrid space (i.e. a continuous or discrete space). Figure 5.7 shows the I/O comparison of the first search stage when searching the CND-tree with and without the use of heuristic $H3$. It can clearly be seen that using $H3$ decreases the number of I/O incurred in the first stage of searching over all dimensional combinations tested.

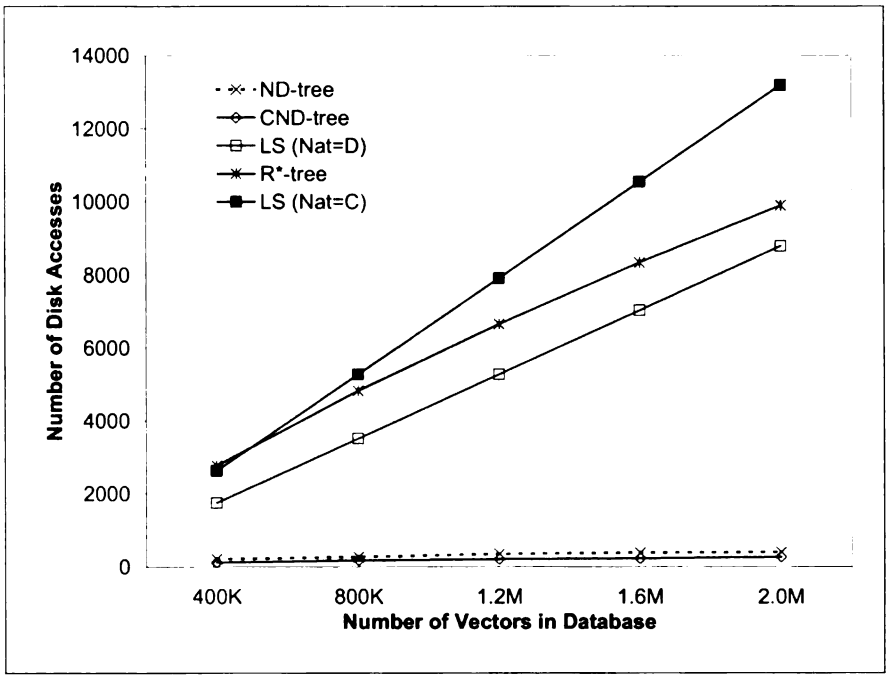


Figure 5.5. Performance I/O with variable database size

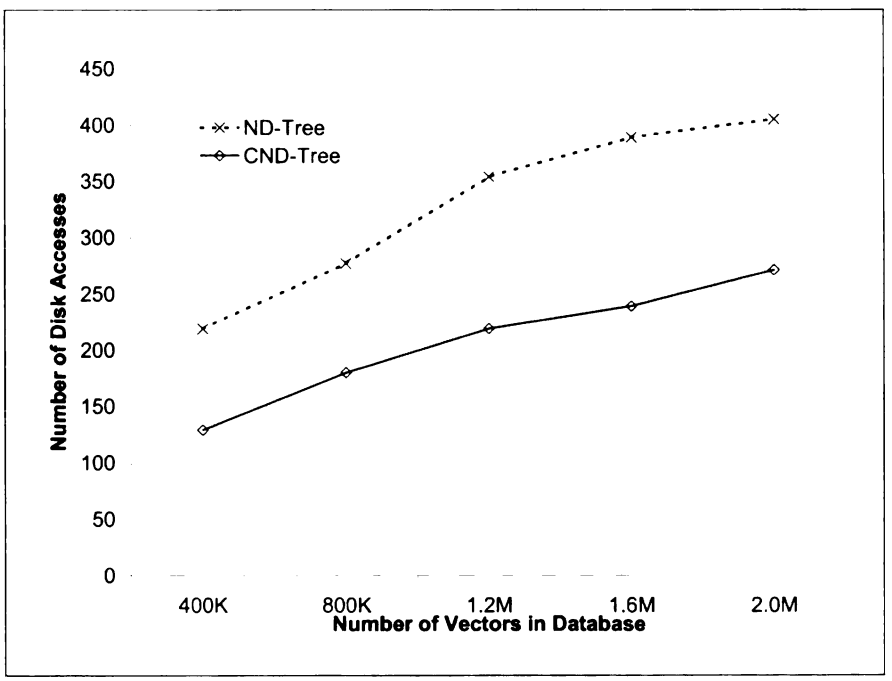


Figure 5.6. Performance I/O with variable database size (CND-tree and ND-tree only)

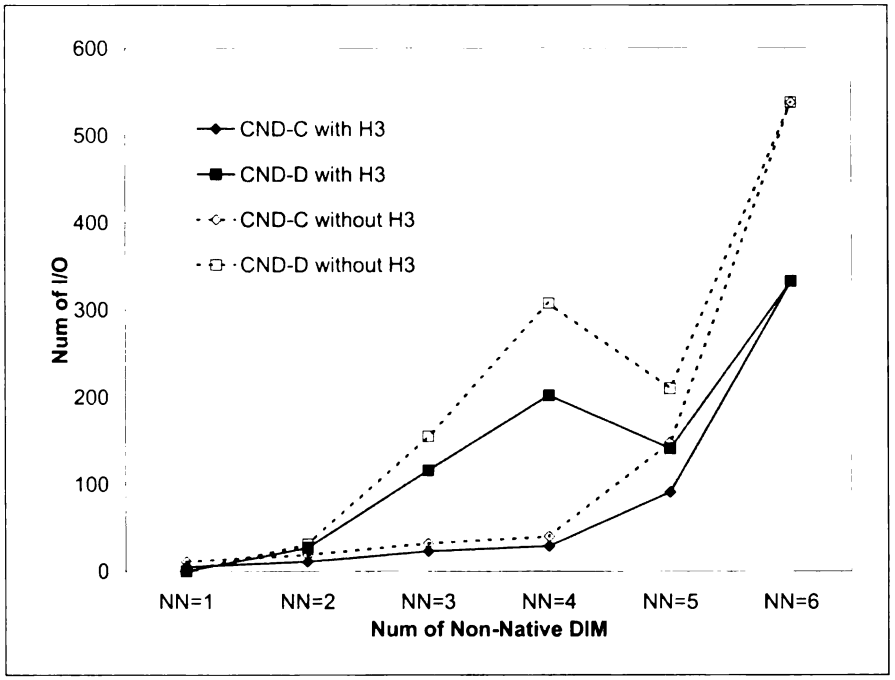


Figure 5.7. Performance I/O comparing ordering methods

5.4.2 Performance Model Verification

Our theoretical performance estimation model was verified against our synthetic experimental data. We conducted experiments using data with $d_D = 6$, $d_C = 3$, $|A| = 6$, and $R = 100$. The maximum number of leaf node objects was set at 227 and the maximum number of non-leaf node objects was set to 127.[6] We compared our theoretical estimates to the observed CND-tree performances for databases varying in size from $400K$ vectors to $2.0M$ vectors in $400K$ increments.

Figure 5.8 shows the estimated number of I/O predicted by our performance model

[6]As described in Qian et. al [43], the values of 127 and 227 were chosen for similar search structures to maximize search performance.

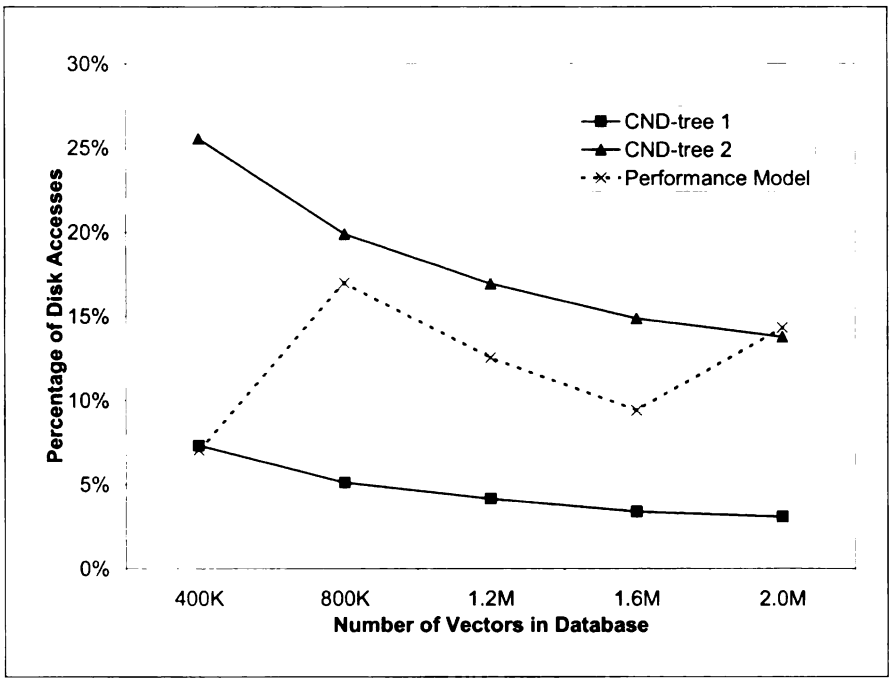


Figure 5.8. Performance model comparison with variable database size

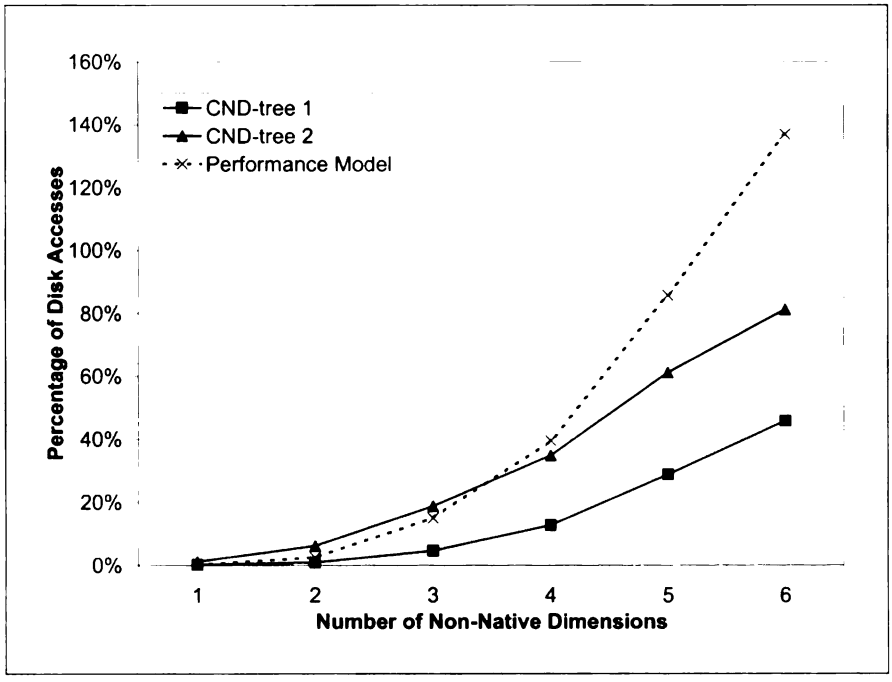


Figure 5.9. Performance model comparison with variable number of non-native dimensions

as well as the actual observed I/O as a percentage of the I/O incurred by a linear scan. Two lines are drawn for the observed I/O when searching the CND-tree. CND-tree 1 represents the percentage of I/O observed when the search is stopped as soon as stage 2 (overlap) has ended. CND-tree 2 represents the percentage of I/O observed when the search is stopped as soon as stage 3 (exhaustive) has ended.[7] The Performance model line represents the predicted I/O percentage when $z = r - 1$. As shown in Figure 5.8, our theoretical performance model does a quite accurate job in predicting the number of I/O that would be incurred by using our algorithm on databases of varying sizes.

We also performed a comparison of our theoretical performance model and the observed I/O when varying the number of continuous dimensions present in the CND-tree. For this set of experiments, $d_D = 6$, $|A| = 6$, $R = 100$, and d_C varies from 1 to 6. The number of vectors in the database is $1M$ and the maximum numbers for the leaf node and non-leaf node objects is again set at 227 and 127 respectively.

Figure 5.9 shows the estimated number of I/O predicted by our performance model as well as the actual observed I/O of these experiments, again as a percentage of the I/O incurred by a linear scan. Again, two lines are drawn for the CND-tree

[7]The exhaustive stage occurs when searching nodes R whose bounding box is the same distance from the query point as the final search range. This stage introduces no closer neighbors than what have already been found, but may be required if multiple sets of k objects form valid solution sets, typical of k-NN searches in discrete databases as discussed in [32, 33].

to represent the number of I/O at the end of the second and third search stages. As shown in Figure 5.9 our performance model does a very good job of predicting the number of I/O for performing a k-NN search when the number of continuous dimensions is less than or equal to the number of discrete dimensions. However, as the number of continuous dimensions grows, the observed CND-tree results begin to outperform the theoretical results predicted by our performance model. We believe this phenomenon may be related to the discretizing of the continuous dimensions by Equation 2.4.

CHAPTER 6

Conclusion

Similarity searches in NDDSs and HDSs are becoming increasingly important in application areas such as bioinformatics, biometrics, E-commerce and data mining. Unfortunately, the prevalent searching techniques based on multidimensional indexes such as the R-tree and the K-D-B tree in CDSs cannot be directly applied to either NDDSs or HDSs. On the other hand, recent work [42, 43, 44, 41, 15] on similarity searching in NDDSs and HDSs focused on range queries. Nearest neighbor searches were not developed to the same extent. In particular, the k-nearest neighbor searching is still an open issue. We observe that the issue of k-NN searching in NDDSs and HDSs is not a simple extension of its counterpart in CDSs.

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 gran-

ularity 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 [42, 43]. 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/O 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 dataset.

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. 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.

We have demonstrated that k-NN searches in HDSs greatly benefit from the use of a multidimensional index developed for such a space. As discussed in Chapter 5, the use of HDS based index eliminates the need for maintaining costly search constants to guarantee correct results. Further, our experimental results confirm that the use of a HDS index vastly outperforms searches utilizing non-hybrid space indexes, sometimes by a factor of 10 to 1, as shown in Section 5.4. We have also shown that the use of our newly introduced searching heuristic provides excellent benefits in reducing the I/O costs associated with the first stage of k-NN searching. Our experimental results in Section 5.4 demonstrate a performance benefit of almost 33%. Additionally, we have presented a theoretical performance model that accurately predicts the I/O cost of performing a k-NN search using our algorithm presented in Section 5.3.

Our future work involves the study of the underlying characteristics of NDDSs and HDSs that may be applied to optimizing data index structures for such spaces. Similar to [52], such a study could provide an optimization in index structure construction by determining the relationship between the dimensionality of a dataset and the estimated performance for data retrieval. Additionally, search performance in

NDDSs and HDSs are also affected by the cardinality of the alphabet of the dataset. While much work been reported on understanding these relationships and optimizing an index structure through that knowledge for CDSs, it remains an open issue for NDDSs and HDSs. Additionally, we will continue to investigate the underlying characteristics of NDDSs and HDSs that can be used in future search heuristics. Finally, our theoretical performance model assumes a uniform node splitting policy of the underlying index structure. We would like to expand upon this to accommodate more potential node split policies.

APPENDIX A

Intrinsic Dimensionality in Non-Ordered Discrete Data Spaces

In this appendix, we present a discussion of the effects of intrinsic dimensionality in NDDSs. We first provide an overview of the concepts of this topic followed by a discussion of the differences in NDDS and CDS dataset distribution. We then discuss the effect of intrinsic dimensionality on search performance when using the GEH distance. This appendix provides additional rationale for the development of the rank based GEH implementation from Chapter 4.

A.1 Overview

In both NDDSs and CDSs, designers of search techniques are hampered by the curse of dimensionality [32, 52]. As discussed by Chavez et al. [14], traditional indexing techniques for vector spaces (e.g. kd-tree, R*-tree) have an exponential dependency

on the representational dimension of the data space. Many recent indexing techniques attempt to avoid the problems associated with this relationship by removing the representational dimension of the space. A common technique is to translate the vector space data to a generic metric space.

Unfortunately, these techniques are unable to completely resolve the curse of dimensionality due to the existence of the intrinsic dimensionality of the dataset. Chavez et al. [14], defined the intrinsic dimensionality ρ of a dataset by the quotient of the mean μ and variance σ^2 of a histogram of distance values between objects in the dataset, as shown below:

$$\rho = \frac{\mu^2}{2\sigma^2}. \tag{A.1}$$

Equation A.1 indicates that a dataset's intrinsic dimensionality grows in line with the mean and inversely with the variance of distance values. The result of this equation may be used to indicate the performance potential of searching within a dataset. As shown in [14], the potential performance for searching a dataset is inversely proportional to the intrinsic dimensionality of the dataset.

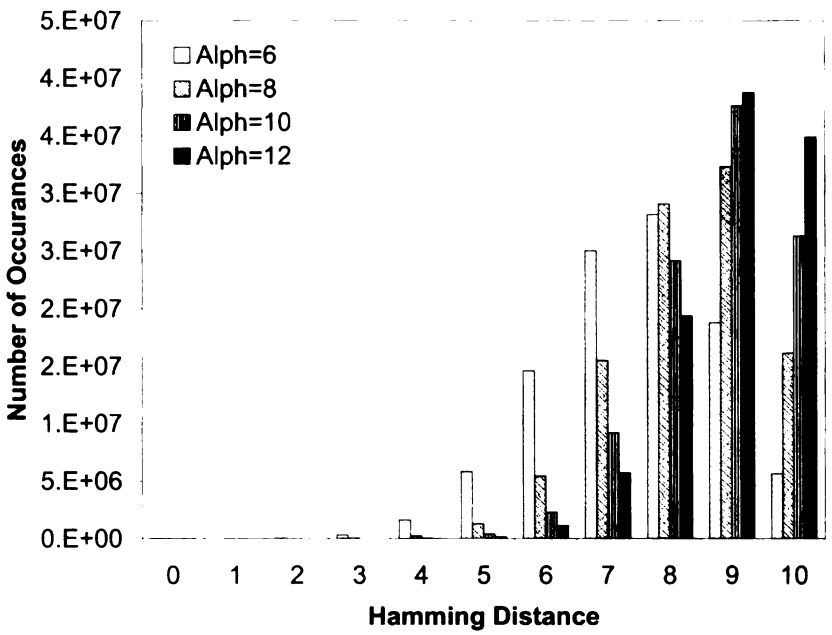


Figure A.1. Histogram of Hamming Distance Values

A.2 Distribution of NDDS Datasets

The histogram of distances between points in either a CDS or a generic metric space will usually result in a semi-continuous curve. As shown in [14], the mean for such histograms in either space is likely to trend equally toward either end of the available range. For an NDDS, this no longer holds. Consider Figure A.1 which shows the histogram of Hamming distance values for a 10-dimensional NDDS of $10K$ vectors with variable alphabet sizes.

We notice three points. First, the mean for each dataset in Figure A.1 appears to be highly dependent upon the cardinality of the alphabet set, such that as the

alphabet size grows larger, so does the mean of distance values. Second, the distance values between points appear to clump disproportionately toward the high end of the distance range. Third, the possible distance values between points is restricted due to the use of the Hamming distance.

Each of these points agrees with our understanding of NDDSs discussed in Chapters 3 and 4. In particular, we note that in an NDDS, there is no defined center of the data space. Thus, available distance metrics, such as Hamming or GEH, have difficulty considering point distance relationships. As shown in [33], the number of points likely to exist at an integer distance z from another point grows exponentially with the value of the alphabet size (this value was described as a hyper-spherical area in [33]), such that:

$$Area(z) = \sum_{i=0}^{z} \binom{d}{i} (|A| - 1)^i. \tag{A.2}$$

Additionally in [33], we showed that the probability of a point existing in an NDDS increases dramatically as the distance z increases:

$$P_{exists}(z) = \frac{\sum_{i=0}^{z} \binom{d}{i} (|A| - 1)^i}{|A|^d}. \tag{A.3}$$

Equation A.2 explains the dependence between the mean of distance values and

the alphabet size of the dataset. Equation A.3 explains the disproportionate amount of distance values between points in an NDDS close to d (as seen in Figure A.1). The third point is easily explained by the discussion of non-granularity of the Hamming distance in Section 4.2.

When we again consider the distances between points in an NDDS, but instead of the Hamming distance metric, we use the GEH distance metric presented in Section 4.2, we are given the histogram shown in Figure A.2 (we have used a scatter plot instead of a column plot to help illustrate the differences with Figure A.1 more clearly). We notice that the restriction on distance values has been greatly reduced. However, we still do not have a continuous curve as would be expected in a CDS or generic metric space. Instead, we see local peaks between each integer distance value with their own local mean and variance.

A.3 Distribution Effects on Search Performance

We define local variance for an integer i as the variance of distance values between $[i, i+1)$. A local mean may be defined in the same manner. To account for these local values, we modify the performance formula given earlier as follows:

$$\rho = \rho_o + \rho_l, \tag{A.4}$$

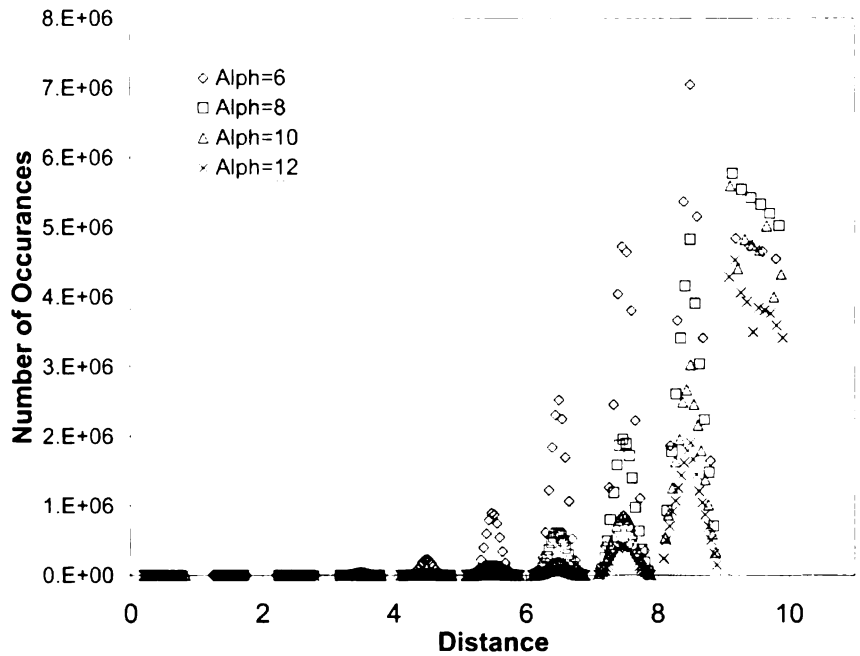


Figure A.2. Histogram of GEH Distance Values

where

$$\begin{aligned} \rho_o &= \frac{\mu^2}{2\sigma^2} \\ \rho_l &= \frac{1}{2d}\sum_{i=0}^{d-1}\left(\frac{\mu_i - i}{\sigma_i}\right)^2. \end{aligned}$$

The value of ρ_l in Equation A.4 captures the expected performance between each integer value. This value may be interpreted as the average performance indicator based on the normalized local means and variances of a dataset. When using an extension to the Hamming distance, such as GEH, this value gives an indication of the likely number of pathways that may be pruned during the refinement stages of a k-NN search. However, because one of the goals when extending the Hamming distance is to retain its original semantics, the value of ρ_l is less dominant in determining the

overall performance measure than the value of ρ_o. This is due to the effects of a Hamming extension typically only becoming a factor when comparing distances in the same integer range.

To help illustrate this point, consider a random query q for a 10 dimensional dataset U. In this case, Figure A.2 can represent the distribution of distances between q and a possible pivot point $p \in U$ (in our case, p is a DMBR), for various alphabet sizes. Assuming our distance metric $D()$ maintains the triangular inequality property, we can eliminate from our search any point u such that $D(p,u) \notin [D(p,q)-r, D(p,q)+r]$, where r is the current search radius [14]. As the variance of distances between integer values increases, more points (search paths) may be discarded when searching within that range. However, this increase in variance has no affect on the amount of points that may be pruned outside the current integer range.

A.4 Experimental Results

We have compared the distance distribution histograms of the ranking based GEH distance implementation from Section 4.2 with that of the frequency based GEH distance implementation from [32] over datasets of various zipf distributions. In these instances, all data is from 10-dimensional datasets of $10K$ vectors with $|A| = 10$. Comparing Figures A.4 and A.3, we see that the ranking based implementation shows

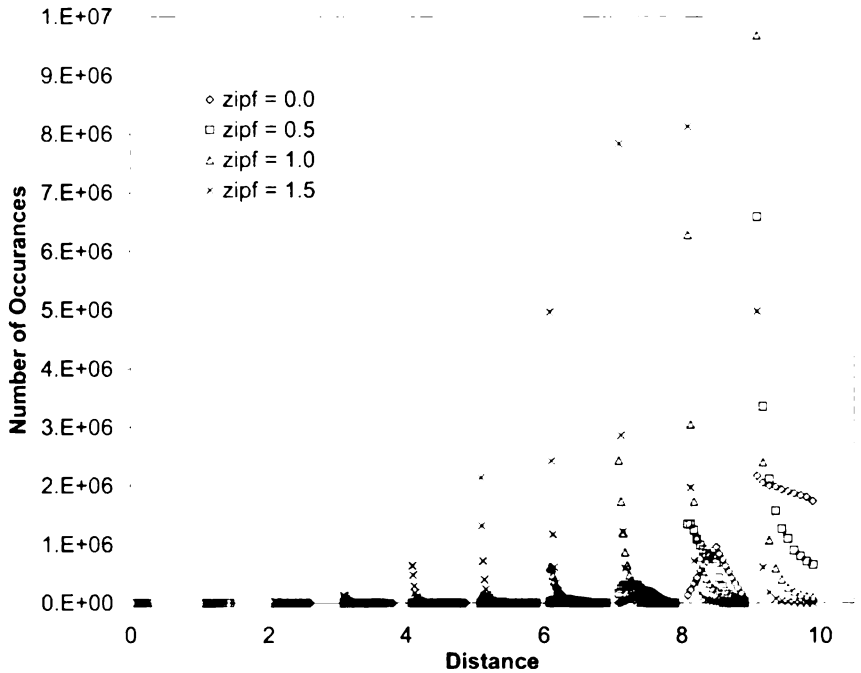


Figure A.3. GEH_{Rank} $zipf = [0.0 - 1.5]$

large improvements, in terms of distribution characteristics, over the frequency based distances as the zipf level increases. This indicates that as the underlying dataset becomes more skewed, the performance benefits of using the ranking based implementation of GEH distance over the frequency based implementation will become greater as well. This agrees with the search performance results from Chapter 4.

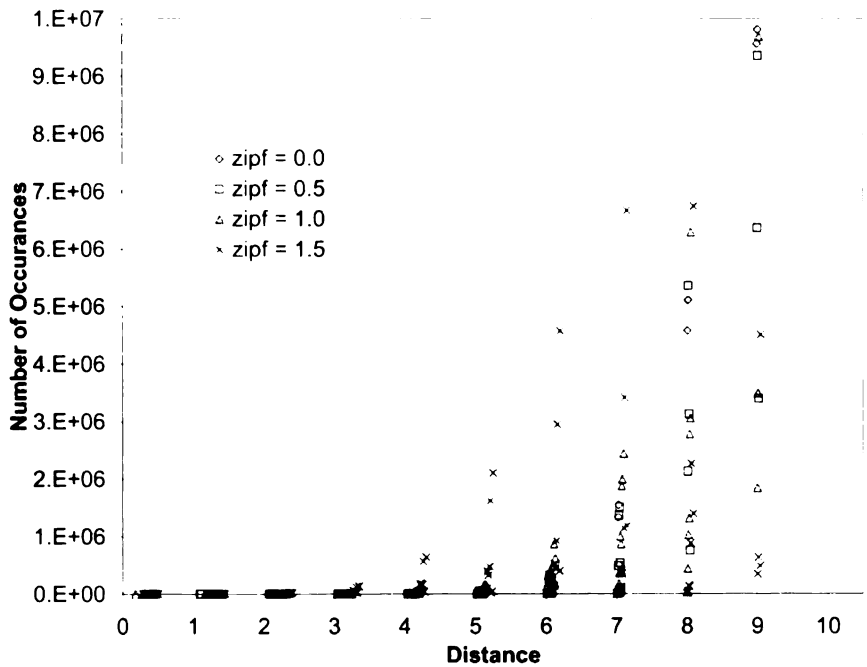


Figure A.4. GEH_{Freq}, $zipf = [0.0 - 1.5]$

APPENDIX B

Triangular Property of GEH - Extended Proof

To maintain the inherent mathematical correctness associated with building and searching index trees in NDDS, our newly introduced GEH distance must maintain the Triangular Property. In Chapter 4, we presented a short proof of this property. In this appendix, we provide a second proof in extended form.

Definition 4. ***Triangular Property for Vectors:*** *For any three vectors V_A, V_B, and V_C, the addition of the distances between any two pairs is greater than or equal to the distance between the third pair, namely:*

$$D(V_A, V_B) + D(V_B, V_C) \geq D(V_A, V_C). \tag{B.1}$$

The long form proof of the GEH distance maintaining this inequality is handled

in two steps: first, a base case is established where the property holds. Second, the base case is evolved into the generic case.

Step 1: Consider three d-dimensional vectors V_A, V_B, and V_C. Assume that $D(V_A, V_C)$ is a maximal value, thus every element in V_A and V_C must be different, or:

$$D(V_A, V_C) = d.$$

Next assume $D(V_A, V_B)$ is a minimal value, where every element in V_B equals the corresponding element in V_A, i.e. $V_A = V_B$. Using the GEH distance yields:

$$D(V_A, V_B) = x,$$

where $0 < x < 1$. The term x represents the adjustment values obtained using some method defined by an application expert (see Chapter 4). Because $V_A = V_B$, the distance between V_B and V_C is the following:

$$D(V_B, V_C) = d.$$

Thus we have the following inequality;

$$D(V_A, V_B) + D(V_B, V_C) > D(V_A, V_C)$$
$$\Rightarrow x + n > n$$

Step 2: The second step is divided into three sub-steps; one for each vector V_A, V_B, and V_C.

Step 2.1: First, we evolve V_B into a generic vector. From Step 1, we have the following distance values:

$$\begin{aligned} D(V_A, V_B) &= x \\ D(V_B, V_C) &= n \\ D(V_A, V_C) &= n, \end{aligned}$$

where $0 < x < 1$.

To make V_B generic, we apply k changes, where each change represents switching an element in V_B away from its original value. After this has been done, we are left with the following distances:

$$\begin{aligned} D(V_A, V_B) &= x + k - c_1 \\ D(V_B, V_C) &= n - k_1^* + c_2 \\ D(V_A, V_C) &= n. \end{aligned}$$

Here, c_1 represents the culmination of adjustment values from each of the k elements switched, k_1^* represents the number of elements switched that now equal their corre-

sponding element in V_C, and c_2 represents the culmination of the adjustment values to be added due to these newly matching values. Because $k \geq k_1^*$, $c \geq c_1$, and $c_2 \geq 0$, the GEH distance between V_A, V_B, and V_C still maintains the inequality from Definition 4.

Step 2.2: Next, we evolve V_C into a generic vector. We start with the final vectors from Step 2.1 and apply j changes to V_C. We now have the following distance values:

$$\begin{aligned} D(V_A, V_B) &= x + k - c_1 \\ D(V_B, V_C) &= n - k_1^* + c_2 + (j_1^* - c_3) - (j_2^* - c_4) \\ D(V_A, V_C) &= n - j_3^* + c_5. \end{aligned}$$

Here, j_1^* and j_2^* represent the integer values that $D(V_B, V_C)$ increases and decreases by respectively as elements are switched; c_3 and c_4 represent the adjustment values due to those changes; j_3^* and $c5$ represent the integer and adjustment changes to $D(V_A, V_C)$ due to element changes. It is important to note that every time j_2^* is incremented there are two possibilities: either the value 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 $k \geq k_1^* - 1$. This leaves us to note that $j_2^* + k_1^* \leq j_3^* + k$ and that $c_4 \geq c_5$. Finally, with $j_1^* \geq c_3$, it can be shown that these distance measures still maintain the Triangular Inequality property from

Definition 4.

Step 2.3: Finally, we evolve V_A into a generic vector. Because of our initial conditions, this is actually a trivial step. Due to V_A only being defined in relation to the original vectors V_C and V_B, and V_C and V_B being able to be manipulated into any general vectors from their starting point, we can start V_A as any vector we wish. Thus V_A is a generic vector and the triangular inequality holds true for any three vectors V_A, V_B, and V_C.

APPENDIX C

MinMaxDistance Discussion

Much of the work presented in this dissertation has focused upon improving search performance in terms of reducing the number of I/O required to perform a search. In this appendix, we prove that many search algorithms may be improved in computational performance by removing the MINMAXDIST heuristic while suffering no loss of I/O performance.

C.1 Overview

For the purposes of this discussion we define the MINDIST for a tree node N and a query point q as follows:

$$MINDIST_N(q) = \min \forall_{N_S} D(q, N_S). \tag{C.1}$$

Where N_S represents a subtree/object of N and D is a valid distance metric for the data space being used. We define the MINMAXDIST in a similar manner:

$$MINMAXDIST_N(q) = \min \forall_{N_S} (\max D(q, N_S)) . \tag{C.2}$$

Using Equations C.1 and C.2, the MINDIST pruning, MINMAXDIST range reduction, and MINDIST ordering equations, hereafter referred to as H_1, H_2, and H_3 respectively, are as follows:

H_1: *For all subtrees N_S of N, remove/prune any subtree whose MINDIST value to q is greater than the current search range.*

H_2: *If the MINMAXDIST of a subtree N_S of node N is less than the current search range, reduce the current search range such that it equals the value of the MINMAXDIST of that subtree.*

H_3: *Order those subtrees N_S not pruned by heuristic H_1 in increasing order of their MINDIST value to q.*

For the remainder of this discussion we will hold the following assumptions to be true:

Assumption 1. *MINDIST node ordering is being used.*

Assumption 2. *MINDIST node pruning is being used.*

Assumption 3. *A depth first search strategy is being employed.*

C.2 Proof

Consider a d-dimensional non-leaf node N, that represents the local root for a branch in an index tree. The search range of the k Nearest Neighbor search is represented by r. If heuristic H_2 is employed, the search range is updated by the MINMAXDIST value for the sub-nodes of N_R as follows:

$$r = \min(r_0, MINMAXDIST_N(q)), \tag{C.3}$$

where r_0 acts as a place holder for the search range before it is updated. For future use we will label the minimal MINMAXDIST value of the subtrees of node N as D_{MM}.

The subtrees of N can be categorized into three groups: N_{S1}, N_{S2}, and N_{S3}, where the following holds true:

$$\begin{aligned} 0 &\leq MINDIST(N_{S1}) \leq r \\ r &< MINDIST(N_{S2}) \leq r_0 \\ r_0 &< MINDIST(N_{S3}) \leq D_{MAX}. \end{aligned} \tag{C.4}$$

where D_{MAX} represents the maximum distance value possible between a sub-tree/object and a query point for the data space being used.

Lemma C.2.1. *Of the three categories of subtrees, a k Nearest Neighbor search will access these groups in the following order: N_{S1} first, N_{S2} second, and N_{S3} third.*

Proof. This is a result of Assumption 1. □

Lemma C.2.2. *The subtree of N with the minimum MINMAXDIST value is contained in the sub-tree group N_{S1}.*

Proof. Equation C.3 indicates that this particular subtree will be used to set the value of r when Heuristic H_2 is employed. This subtree will be in N_{S1} due to its MINDIST value being less than or equal to its MINMAXDIST value. □

Lemma C.2.3. *If Heuristic H_2 is employed, Heuristic H_1 will prune subtree groups N_{S2} and N_{S3}, due to $r = r_{MM}$.*

Proof. Heuristic H_2 and Equation C.3 indicate that the updated search range will be equal to r. Equation C.4 classifies that subtree groups N_{S2} and N_{S3} will have a MINDIST value greater than r and will thus be pruned by Heuristic H_1. □

Lemma C.2.4. *If Heuristic H_2 is not employed, Heuristic H_1 will prune subtree group N_{S3}.*

Proof. Similar to the preceding proof, we are only guaranteed that the current search range will be equal to r_0. Equation C.4 only classifies subtree group N_{S3} as having a MINDIST value greater than r_0 and is thus the only group guaranteed to be pruned by Heuristic H_1. □

Lemma C.2.5. *If Heuristic H_2 is not employed, the value of r will be less than or equal to D_{MM} before visiting any sub-nodes in group N_{S2}.*

Proof. Due to Assumptions 1 and 3, the search algorithm will visit the subtrees of group N_{S1} before returning to the current node N and considering subtrees in the groups N_{S2} and N_{S3}. According to Lemma C.2.2, the subtree with a MINMAXDIST value D_{MM} is contained in subtree group N_{S1}. Thus the search is guaranteed to visit an object with a distance value less than or equal to D_{MM} before returning to N. □

Lemma C.2.6. *Heuristic H_2 provides no I/O benefits when assumptions 1 through 3 are true.*

Proof. Lemma C.2.5 indicates that the value of r will be less than or equal to D_{MM} before the search algorithm considers visiting any subtrees from group N_{S2}. Thus Heuristic H_1 will prune these subtrees before they are visited regardless of Heuristic H_2 being employed. □

BIBLIOGRAPHY

[1] Y. A. Aslandogan and C. T. Yu. Techniques and systems for image and video retrieval. *IEEE TKDE*, 11:56–63, 1999.

[2] Ricardo A. Baeza-Yates. Searching: An algorithmic tour. In *Encyclopedia of Computer Science and Technology Vol. 37*, pages 331–359, New York, New York, 1997. CRC Press.

[3] Ricardo A. Baeza-Yates, Walter Cunto, Udi Manber, and Sun Wu. Proximity matching using fixed-queries trees. In *CPM '94: Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching*, pages 198–212, London, UK, 1994. Springer-Verlag.

[4] L. Baoli, L. Qin, and Y. Shiwen. An adaptive k-nearest neighbor text categorization strategy. *ACM Transactions on Asian Language Information Processing*, 3:215–226, 2004.

[5] R. Bayer and K. Unterauer. Prefix b-trees. *ACM Transactions on Databases Systems*, 2:11–26, 1977.

[6] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In Hector Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990*, pages 322–331, Atlantic City, NJ, U.S.A, 1990. ACM Press.

[7] J. L. Bentley. Multidimensional binary search trees in database applications. *IEEE Trans. Softw. Eng.*, 5(4):333–340, 1979.

[8] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[9] Jon Louis Bentley and Jerome H. Friedman. Data structures for range searching. *ACM Comput. Surv.*, 11(4):397–409, 1979.

[10] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. The X-tree: An index structure for high-dimensional data. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *Proceedings of the 22nd International Conference on Very Large Databases*, pages 28–39, San Francisco, U.S.A., 1996. Morgan Kaufmann Publishers.

[11] Sergey Brin. Near neighbor search in large metric spaces. In *VLDB '95: Proceedings of the 21th International Conference on Very Large Data Bases*, pages 574–584, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[12] W. A. Burkhard and R. M. Keller. Some approaches to best-match file searching. *Commun. ACM*, 16(4):230–236, 1973.

[13] J Catlett. On changing continuous attributes into ordered discrete attributes. In *Proceedings of the European Working Session on Maching Learning*, pages 164–178, 1991.

[14] Edgar Chávez, Bonzalo Navarro, Ricardo Baeza-Yates, and José Luis Marroquín. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, 2001.

[15] Changqing Chen, Sakti Pramanik, Qiang Zhu, Watve Alok, and Gang Qian. The c-nd tree: A multidimensional index for hybrid continuous and non-ordered discrete data spaces. In *Proceedings of EDBT*, 2009.

[16] Paolo Ciaccia, Marco Patella, and Pavel Zezula. 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*, pages 426–435, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[17] Kenneth L. Clarkson. Nearest neighbor queries in metric spaces. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 609–617, New York, NY, USA, 1997. ACM.

[18] J. Clement, P. Flajolet, and B. Vallee. Dynamic sources in information theory: A general analysis of trie structures. *Algorithm*, 29, 2001.

[19] P. Ferragina and R. Grossi. The string b-tree: A new data structure for string search in external memory and its applications. *Journal ACM*, 46:236–280, 1999.

[20] A Freitas. *A Survey of Evolutionary Algorithms for Data Mining and Knowledge Discovery.* ACM, 2003.

[21] Volker Gaede and Oliver Gunther. Multidimensional access methods. *ACM Computing Surveys*, 30:170–231, 1998.

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

[23] R. Hamming. Error-detecting and error-correcting codes. *Bell System Technical Journal*, 29(2):147–160, 1950.

[24] A. Henrich, H. W. Six, and P. Widmayer. The lsd tree: spatial access to multidimensional and non-point objects. In *VLDB '89: Proceedings of the 15th international conference on Very large data bases*, pages 45–53, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

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

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

[27] G Hjaltason and H Samet. Index-driven similarity search in metric spaces. *ACM Transactions on Database Systems*, 28:517–580, 2003.

[28] Gisli R. Hjaltason and Hanan Samet. Ranking in spatial databases. In *SSD '95: Proceedings of the 4th International Symposium on Advances in Spatial Databases*, pages 83–95, London, UK, 1995. Springer-Verlag.

[29] W. J. Kent. Blat–the blast-like alignment tool. *Genome Res*, 12(4):656–664, April 2002.

[30] D. E. Knuth. *The Art of Computer Programming, Vol. 3.* Addison-Wesley, Reading, MA, USA, 1973.

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

[32] Dashiell Kolbe, Qiang Zhu, and Sakti Pramanik. On k-nearest neighbor searching in non-ordered discrete data spaces. In *ICDE*, pages 426–435, Istanbul, Turkey, 2007. IEEE.

[33] Dashiell Kolbe, Qiang Zhu, and Sakti Pramanik. Efficient k-nearest neighbor searching in non-ordered discrete data spaces. *ACM Transactions on Information Systems*, 28, 2010.

[34] Dashiell Kolbe, Qiang Zhu, and Sakti Pramanik. Reducing non-determinism of k-nn searching in non-ordered discrete data spaces. *Information Processing Letters*, 2010.

[35] Flip Korn, Nikolaos Sidiropoulos, Christos Faloutsos, Eliot Siegel, and Zenon Protopapas. Fast nearest neighbor search in medical image databases. In *VLDB '96: Proceedings of the 22th International Conference on Very Large Data Bases*, pages 215–226, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[36] O.W. Kwon and J.H. Lee. Web page classification based on k-nearest neighbor approach. In *Proceedings of the 5th International Workshop Information Retrieval with Asian Languages*, 2000.

[37] F Lewis, Gareth J Hughes, Andrew Rambaut, Anton Pozniak, and Andrew J Leigh Brown. Episodic sexual transmission of HIV revealed by molecular phylodynamics. *PLoS Medicine*, 5(3), 2008.

[38] Jinhua Li. *Efficient Similarity Search Based on Data Distribution Properties in High Dimension.* PhD thesis, Michigan State University, East Lansing, Michigan, United States, 2001.

[39] A Macskassy, H Hirsh, A Banerjee, and A Dayanik. Converting numerical classification into text classification. *Artificial Intelligence*, 143(1):51–77, 2003.

[40] Gonzalo Navarro and Ricardo Baeza-yates. Searching in metric spaces. *ACM Computing Surveys*, 33:273–321, 2001.

[41] Gang Qian. *Principles and applications for supporting similarity queries in non-ordered-discrete and continuous data spaces.* PhD thesis, Michigan State University, East Lansing, Michigan, United States, 2004.

[42] Gang Qian, Qiang Zhu, Qiang Xue, and Sakti Pramanik. 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*, pages 620–631, Berlin, Germany, 2003. VLDB Endowment.

[43] Gang Qian, Qiang Zhu, Qiang Xue, and Sakti Pramanik. Dynamic indexing for multidimensional non-ordered discrete data spaces using a data-partitioning approach. *ACM Trans. Database Syst.*, 31(2):439–484, 2006.

[44] Gang Qian, Qiang Zhu, Qiang Xue, and Sakti Pramanik. A space-partitioning-based indexing method for multidimensional non-ordered discrete data spaces. *ACM Trans. Inf. Syst.*, 24(1):79–110, 2006.

[45] E. Riloff and L. Hollaar. Text databases and information retrieval. *ACM Computing Surveys*, 28, 1996.

[46] John T. Robinson. 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*, pages 10–18, New York, NY, USA, 1981. ACM.

[47] Nick Roussopoulos, Stephen Kelley, and Frédéic Vincent. Nearest neighbor queries. In Michael J. Carey and Donovan A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995*, pages 71–79, San Jose, California, U.S.A., 1995. ACM Press.

[48] Nick Roussopoulos and Daniel Leifker. Direct spatial search on pictorial databases using packed r-trees. *SIGMOD Rec.*, 14(4):17–31, 1985.

[49] Y. Rui, T. S. Huang, and S. Change. Image retrival: Current techniques, promising directions, and open issues. *J. Visual Communication and Image Representation*, 10:39–62, 1999.

[50] Thomas Seidl and Hans-Peter Kriegel. Optimal multi-step k-nearest neighbor search. *SIGMOD Rec.*, 27(2):154–165, 1998.

[51] J. Uhlmann. Implementing metric trees to satisfy general proximity/similarity queries, 1991.

[52] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB '98: Proceedings of the 24rd International Conference on Very Large Data Bases*, pages 194–205, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[53] D. White and R. Jain. Algorithms and strategies for similarity retrieval, 1996.

[54] Q. Xue, G. Qian, J.R. Cole, and S. Pramanik. Investigation on approximate q-gram matching in genome sequence databases, 2004.

[55] Peter N. Yianilos. Locally lifting the curse of dimensionality for nearest neighbor search (extended abstract). In *SODA '00: Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, pages 361–370, Philadelphia, PA, USA, 2000. Society for Industrial and Applied Mathematics.