Dynamic Indexing for Multidimensional Non-ordered Discrete Data Spaces Using a Data-Partitioning Approach

GANG QIAN
University of Central Oklahoma
QIANG ZHU
The University of Michigan - Dearborn
QIANG XUE and SAKTI PRAMANIK
Michigan State University

Similarity searches in multidimensional Non-ordered Discrete Data Spaces (NDDS) are becoming increasingly important for application areas such as bioinformatics, biometrics, data mining and E-commerce. Efficient similarity searches require robust indexing techniques. Unfortunately, existing indexing methods developed for multidimensional (ordered) Continuous Data Spaces (CDS) such as the R-tree cannot be directly applied to an NDDS. This is because some essential geometric concepts/properties such as the minimum bounding region and the area of a region in a CDS are no longer valid in an NDDS. Other indexing methods based on metric spaces such as the M-tree and the Slim-trees are too general to effectively utilize the special characteristics of NDDSs, resulting in non-optimized performance. In this paper, we propose a new dynamic data-partitioning-based indexing technique, called the ND-tree, to support efficient similarity searches in an NDDS. The key idea is to extend the relevant geometric concepts as well as some indexing strategies used in CDSs to NDDSs. Efficient algorithms for ND-tree construction and techniques to solve relevant issues such as handling dimensions with different alphabets in an NDDS are presented. Our experimental results on synthetic data and real genome sequence data demonstrate that the ND-tree outperforms the linear scan, the M-tree and the Slim-trees for similarity searches in multidimensional NDDSs. A theoretical model is also developed to predict the performance of the ND-tree for random data.

Categories and Subject Descriptors: H.2.2 [**Database Management**]: Physical Design—*Access methods*
General Terms: Algorithms, Performance, Theory
Additional Key Words and Phrases: Non-ordered discrete data spaces, Hamming distance, similarity search, multidimensional index tree

This is a preliminary release of an article accepted by ACM Transactions on Database Systems. The definitive version is currently in production at ACM and, when released, will supersede this version.
Research supported by the US National Science Foundation (under grants # IIS-0414576 and # IIS-0414594), Michigan State University and The University of Michigan.
Authors' current addresses: G. Qian, Department of Computer Science, University of Central Oklahoma, Edmond, OK 73034, USA; email: gqian@ucok.edu; Q. Zhu, Department of Computer and Information Science, The University of Michigan, Dearborn, MI 48128, USA; email: qzhu@umich.edu; Q. Xue and S. Pramanik, Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824, USA; email: {xueqiang,pramanik}@cse.msu.edu.

1. INTRODUCTION

There is an increasing demand for similarity searches in multidimensional Non-ordered Discrete Data Spaces (NDDS) from application areas such as bioinformatics, biometrics, data mining and E-commerce. For example, in genome sequence databases, sequences with alphabet $A = \{a, g, t, c\}$ are broken into substrings (also called intervals) of some fixed-length d for similarity searches [Kent 2002; Xue et al. 2004]. Each interval can be considered as a vector in a d-dimensional data space. For example, interval "*aggcggtgatctgggccaatactga*" 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. The main characteristic of such a data space is that the data values in each dimension are discrete and have no ordering. Other examples of non-ordered discrete values in a dimension of an NDDS are discrete data types such as gender, complexion, profession and user-defined enumerated types. The databases that require searching information in an NDDS can be very large (e.g., the well-known genome sequence database, GenBank, contains over 80 GB genomic data). To support efficient similarity searches in such databases, robust indexing techniques are needed.

Many multidimensional indexing methods have been proposed for Continuous Data Spaces (CDS), where data values in each dimension are continuous and can be ordered along an axis. These techniques can be classified into two categories: data-partitioning-based and space-partitioning-based. The techniques in the first category such as the R-tree [Guttman 1984], the R*-tree [Beckmann et al. 1990], the SS-tree [White and Jain 1996], the SR-tree [Katayama and Satoh 1997] and the X-tree [Berchtold et al. 1996] split an overflow node by partitioning the set of its indexed data objects (according to their distribution). The techniques in the second category such as the K-D-B tree [Robinson 1981] and the LSD^h-tree [Henrich 1998], on the other hand, split an overflow node by partitioning its representing data space (typically via a splitting point in a dimension). The Hybrid-tree that incorporates the strengths of indexing methods in both categories was proposed in [Chakrabarti and Mehrotra 1999]. However, all the above techniques rely on a crucial property of a CDS; that is, the data values in each dimension can be ordered and labeled on an axis. Some essential geometric concepts such as rectangle, sphere, region area, and so on are no longer valid in an NDDS, where data values in each dimension cannot even be labeled on an (ordered) axis. Hence the above techniques cannot be directly applied to an NDDS.

If the alphabet for every dimension in an NDDS is the same, a vector in such a space can be considered as a string over the alphabet. In this case, traditional string indexing methods, such as the Tries [Knuth 1973; Clement et al. 2001], the Prefix B-tree [Bayer and Unterauer 1977] and the String B-tree [Ferragina and Grossi 1999], can be utilized. However, most of these string indexing methods like the Prefix B-trees and the String B-trees were designed for exact searches rather than similarity searches. The Tries does support similarity searches, but its memory-based feature makes it difficult to apply to large databases. Moreover, if the alphabets for different dimensions in an NDDS are different, vectors in such a space can no longer be considered as strings over an alphabet. The string indexing methods are inapplicable in this case.

A number of so-called metric trees have been introduced in recent years [Uhlmann 1991; Chiueh 1994; Brin 1995; Bozkaya and Ozsoyoglu 1997; Ciaccia et al. 1997; Chavez et al. 2001; Traina et al. 2002; Dohnal et al. 2003; Zhou et al. 2003; Skopal et al. 2004]. These trees only consider relative distances of data objects to organize and partition the search space and apply the triangle inequality property of distances to prune the search space. These techniques, in fact, could be applied to support similarity searches in an NDDS. However, most of such trees are static and require costly reorganizations to prevent performance degradation in case of insertions and deletions [Uhlmann 1991; Chiueh 1994; Brin 1995; Bozkaya and Ozsoyoglu 1997]. On the other hand, these techniques are very generic with respect to the underlying data spaces. They only assume the knowledge of relative distances of data objects and do not effectively utilize the special characteristics, such as occurrences and distributions of dimension values, of data objects in a specific data space. Hence, even for dynamic indexing techniques of this type, such as the M-tree [Ciaccia et al. 1997] and its variants [Traina et al. 2002; Zhou et al. 2003; Skopal et al. 2004], their retrieval performance is not optimized.

To support efficient similarity searches in an NDDS, we propose a new indexing technique, called the ND-tree. The key idea is to extend the essential geometric concepts (e.g., minimum bounding rectangle and area of a region) as well as some effective indexing strategies (e.g., node splitting heuristics in the R*-tree) in CDSs to NDDSs. There are several technical challenges for developing an indexing method for an NDDS. They are due to: (1) no ordering of values on each dimension in an NDDS; (2) non-applicability of popular continuous distance measures such as Euclidean distance and Manhattan distance to an NDDS; (3) high probability of vectors to have the same value on a particular dimension in an NDDS; and (4) the limited choices of splitting points on each dimension. The ND-tree is developed in such a way that these difficulties are properly addressed. Our extensive experiments and theoretical analysis demonstrate that the ND-tree can support efficient searches in NDDSs.

The rest of this paper is organized as follows. Section 2 introduces the essential concepts and notation for the ND-tree. Section 3 discusses the details of the ND-tree including the tree structure, its associated algorithms and a performance estimation model. Section 4 presents our experimental results. Section 5 gives the conclusions and future work.

2. CONCEPTS AND NOTATION

As mentioned above, to develop the ND-tree, some essential geometric concepts in CDSs need to be extended to NDDSs. These extended concepts are introduced in this section.

Let A_i ($1 \leq i \leq d$) be an *alphabet* consisting of a finite number of *letters/elements*. It is assumed that there is no ordering among letters in A_i. A *d-dimensional non-ordered discrete data space* (*NDDS*) Ω_d is defined as the Cartesian product of d alphabets: $\Omega_d = A_1 \times A_2 \times ... \times A_d$. A_i is called the *i-th dimension* alphabet of Ω_d. The *area* (*or size*) *of space* Ω_d is defined as: $area(\Omega_d) = |A_1| * |A_2| * ... * |A_d|$, which in fact indicates the number of vectors in the space. Note that, in general, A_i's may be different for different dimensions. For simplicity, we assume that the alphabets for all the dimensions are the same unless stated otherwise. However, as we will see in Section 3.2.7, our discussion can be easily extended to handle the situation where alphabets of the dimensions in an NDDS are different. If all the dimensions of space Ω_d have the same alphabet A, we simply term A as the alphabet of Ω_d.

Let $a_i \in A_i$ $(1 \le i \le d)$. The tuple $\alpha = (a_1, a_2, ..., a_d)$ (or simply "$a_1a_2...a_d$") is called a vector in Ω_d. Let $S_i \subseteq A_i$ $(1 \le i \le d)$. A *discrete rectangle* R in Ω_d is defined as the Cartesian product: $R = S_1 \times S_2 \times ... \times S_d$. S_i is called the *i-th* (*dimension*) *component set* of R. The *length* of the i-th dimension edge of R is $length(R, i) = |S_i|$. The *area* of R is defined as: $area(R) = |S_1| * |S_2| * ... * |S_d|$.

Let $R = S_1 \times S_2 \times ... \times S_d$ and $R' = S'_1 \times S'_2 \times ... \times S'_d$ be two discrete rectangles in Ω_d. The *overlap* $R \cap R'$ of R and R' is the Cartesian product: $R \cap R' = (S_1 \cap S'_1) \times (S_2 \cap S'_2) \times ... \times (S_d \cap S'_d)$. Clearly, $area(R \cap R') = |S_1 \cap S'_1| * |S_2 \cap S'_2| * ... * |S_d \cap S'_d|$. If $R = R \cap R'$ (i.e., $S_i \subseteq S'_i$ for $1 \le i \le d$), R is said to *be contained* in (or *covered by*) R'. Based on this containment relationship, the *discrete minimum bounding rectangle* (*DMBR*) of a set of given discrete rectangles $G = \{ R_1, R_2, ..., R_n \}$ can be defined as follows:

$$DMBR(G) = min\{ R \mid R \; is \; a \; discrete \; rectangle \; containing \; every \; R_i \in G \; (1 \le i \le n) \} \tag{1}$$

where

$$min\{R', R''\} = \begin{cases} R' & if \; area(R') \le area(R'') \\ R'' & otherwise \end{cases}$$

for discrete rectangles R' and R''. In fact, if $R_i = S_{i1} \times S_{i2} \times ... \times S_{id}$, the DMBR of set G can be calculated as follows:

$$DMBR(G) = (\cup_{i=1}^{n} S_{i1}) \times (\cup_{i=1}^{n} S_{i2}) \times ... \times (\cup_{i=1}^{n} S_{id}). \tag{2}$$

The distance measure between two vectors in a data space is important for building a multidimensional index tree. Unfortunately, those widely-used continuous distance measures such as the Euclidean distance cannot be applied to an NDDS. One might think that a simple solution to this problem is to map the letters in the alphabet for each dimension to a set of (ordered) numerical values, and then apply the Euclidean distance. For example, one could map 'a', 'g', 't' and 'c' in the alphabet for a genome sequence database to numerical values 1, 2, 3 and 4, respectively. However, this approach would change the semantics of the elements in the alphabet. For example, the above mapping for the genomic bases (letters) would make the distance between 'a' and 'g' closer than that between 'a' and 'c', which is not the original semantics of the genomic bases. Hence, unless it is for exact match, such a transformation approach is not a proper solution.

One suitable distance measure for NDDSs is the Hamming distance. That is, the distance $dist(\alpha_1, \alpha_2)$ between two vectors α_1 and α_2 in an NDDS is the number of dimensions on which the corresponding components of α_1 and α_2 are different. From the Hamming distance, the (minimum) distance between a vector $\alpha = (a_1, a_2, ..., a_d)$ and a discrete rectangle $R = S_1 \times S_2 \times ... \times S_d$ can be defined as:

$$dist(\alpha, R) = \sum_{i=1}^{d} f(a_i, S_i) \tag{3}$$

where

$$f(a_i, S_i) = \begin{cases} 0 & if\ a_i \in S_i \\ 1 & otherwise. \end{cases}$$

This distance measures how many components of vector α are not contained in the corresponding component sets of rectangle R.

Using the Hamming distance, a range query $range(\alpha_q, r_q)$ can be defined as $\{\ \alpha\ \mid\ \alpha\ is\ a\ vector\ in\ the\ underlying\ NDDS\ and\ dist(\alpha_q, \alpha) \leq r_q\ \}$, where α_q and r_q are the given query vector and search distance (range), respectively. An exact query is a special case of a range query when $r_q = 0$. A range query with $r_q > 0$ gives a similarity search, i.e., retrieving the vectors with at most r_q different components from the query vector. Distance formula (3) is used to check if the DMBR of a tree node may contain vectors that are within the search range of a given query vector.

EXAMPLE 1. Consider a genome sequence database. Assume that the sequences in the database are broken into intervals of length 25 for similarity searches. As we mentioned before, each interval can be considered as a vector in a 25-dimensional NDDS Ω_{25}. The alphabets for all dimensions in Ω_{25} are the same, i.e., $A_i = A = \{a, g, t, c\} (1 \leq i \leq 25)$. The space size: $area(\Omega_{25}) = 4^{25} \approx 1.126 \times 10^{15}$. $R = \{g, t, c\} \times \{g, t\} \times ... \times \{t, c\}$ and $R' = \{a, t, c\} \times \{a, c\} \times ... \times \{c\}$ are two discrete rectangles in Ω_{25}, with areas $3 * 2 * ... * 2$ and $3 * 2 * ... * 1$, respectively. The overlap of R and R' is: $R \cap R' = \{t, c\} \times \emptyset \times ... \times \{c\}$, where $\emptyset$ denotes the empty set. Given vector $\alpha =$ "*aggcggtgatctgggccaatactga*" in Ω_{25}, range query $range(\alpha, 2)$ retrieves all vectors that differ from α on at most 2 dimensions from the database. The distance $dist(\alpha, R)$ between α and R is: 0+1+...+0. □

Note that, although the editor distance has been used for searching genome sequence databases, lately the Hamming distance is also used for searching large genome sequence databases in a filtering step [Kent 2002; Xue et al. 2004].

3. THE ND-TREE

The ND-tree is designed for NDDSs. It is inspired by some popular multidimensional indexing techniques including the R-tree and its variants (the R*-tree in particular). Hence it has some similarities to the R-tree and the R*-tree. It is a data-partitioning-based indexing technique. The distinctive feature of the ND-tree is that it is based on the NDDS concepts such as discrete rectangles and their areas and overlaps defined in Section 2. Furthermore, its development takes some special characteristics of NDDSs into consideration as we will see.

3.1 The Tree Structure

Assume that the keys to be indexed for a database are the vectors in an NDDS Ω_d over an alphabet A. A leaf node in an ND-tree contains an array of entries of the form (*op*, *key*), where *key* is a vector in Ω_d and *op* is a pointer to the object represented by *key* in the database. A non-leaf node N in an ND-tree contains an array of entries of the form (*cp*, $DMBR$), where *cp* is a pointer to a child node N' of N in the tree and $DMBR$ is the discrete minimum bounding rectangle of N'. The DMBR of a leaf node N'' is the DMBR of vectors indexed in N''; that is, each component set of the DMBR consists of all elements of the vectors on the corresponding dimension. The DMBR of a non-leaf node N' is the DMBR of the set of DMBRs of the child nodes of N', following Formula (2).

Let M and m ($2 \leq m \leq \lceil M/2 \rceil$) be the maximum number and the minimum number of entries allowed in each node of an ND-tree, respectively. Note that, since the spaces required for an entry in a leaf node and an entry in a non-leaf node are usually different while the space allocated to each node (i.e., block size) is assumed to be the same, in practice, the maximum number M_l (the minimum number m_l) for a leaf node can be different from the maximum number M_n (the minimum number m_n) for a non-leaf node. To simplify the description of the tree construction algorithms, we use the same M (m) for both leaf and non-leaf nodes. However, our discussions are still valid if M (m) is assumed to be M_l (m_l) when a leaf node is considered and M_n (m_n) when a non-leaf node is considered.

An ND-tree is a balanced tree satisfying the following conditions: (1) the root has at least two children unless it is a leaf, and it has at most M children; (2) every non-leaf node has between m and M children unless it is the root; (3) every leaf node contains at least m entries unless it is the root, and it contains at most M entries; (4) all leaves appear at the same level. Figure 1 shows an example of the ND-tree for a genome sequence database.

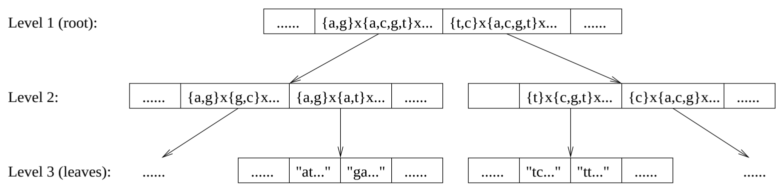


Fig. 1. An example of the ND-tree

3.2 Building the ND-Tree

To build an ND-tree, algorithms to insert/delete/update a data object (vector in Ω_d) into/from/in the tree are needed. In this paper, our discussion is focused on the insertion issues and its related algorithms. However, for completeness, a deletion algorithm for the ND-tree is also described. The update operation can be implemented by a deletion followed by an insertion.

3.2.1 *Insertion Procedure.* The task of the insertion procedure is to insert a new vector α into a given ND-tree. It determines the most suitable leaf node for accommodating α by invoking Algorithm **ChooseLeaf**. If the chosen leaf node overflows after accommodating α, Algorithm **SplitNode** is invoked to split it into two new nodes. The split propagates up the ND-tree if the splitting of the current node causes the parent node to overflow. If the root overflows, a new root is created to accommodate the two nodes resulting from the splitting of the old root. The DMBRs of all affected nodes are adjusted in a bottom-up fashion accordingly.

As a dynamic indexing method, the two algorithms **ChooseLeaf** and **SplitNode** invoked in the above insertion procedure are very important. The strategies used in these algorithms determine the data organization in the tree and are crucial to the performance of the tree. The details of these algorithms will be discussed in the following subsections.

3.2.2 *Choosing Leaf Node.* The purpose of Algorithm **ChooseLeaf** is to find an appropriate leaf node to accommodate a new vector. It starts from the root node and follows a path to the identified leaf node. At each non-leaf node, it has to decide which child node to follow. We have applied several heuristics for choosing a child node in order to obtain a tree with good performance.

Let E_1, E_2, ..., E_p be the entries in the current non-leaf node N, where $m \leq p \leq M$. The overlap of an entry E_k $(1 \leq k \leq p)$ with other entries is defined as:

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

One major problem in high dimensional indexing methods for CDSs is that as the number of dimensions becomes larger, the amount of overlapping among the bounding regions in the tree structure increases significantly, leading to a dramatic degradation of the retrieval performance of the tree [Berchtold et al. 1996; Li 2001]. Our experiments (see Section 4) have shown that NDDSs also have a similar problem. Hence we give the highest priority to the following heuristic:

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

Unlike a multidimensional index tree in a CDS, possible values for the overlap of an entry in the ND-tree (for an NDDS) are limited, which implies that ties may occur frequently. Therefore, other heuristics should be applied to resolve ties. Based on our experiments (see Section 4), we have found that the following heuristics, which were used in some existing multidimensional indexing techniques [Beckmann et al. 1990], are also effective in improving the performance of an ND-tree:

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

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

Using the above heuristics, Algorithm **ChooseLeaf** is given as follows:

Algorithm 3.1. : **ChooseLeaf**
Input: (1) vector α to be inserted; (2) root node T of an ND-tree.
Output: leaf node N chosen for accommodating α.
Method:

```
1. let $N = T$;
2. while $N$ is not a leaf node do
3.    let $S_1$ be the set of child nodes of $N$ determined by $IH_1$;
4.    if $|S_1| = 1$ then
5.       let $N$ be the unique child node in $S_1$;
6.    else if $|S_2| = 1$ where $S_2 \subseteq S_1$ determined by $IH_2$ then
7.       let $N$ be the unique child node in $S_2$;
```

```
8.     else if $|S_3| = 1$ where $S_3 \subseteq S_2$ determined by $IH_3$ then
9.         let $N$ be the unique child node in $S_3$;
10.    else let $N$ be a child node randomly chosen from $S_3$;
11.    end if;
12. end while;
13. return $N$.
```

3.2.3 *Splitting Overflow Node.* Let N be an overflow node with a set of $M + 1$ entries $ES = \{E_1, E_2, ..., E_{M+1}\}$. A 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) $m \leq |ES_1|$, $m \leq |ES_2|$. Let $ES_1.DMBR$ and $ES_2.DMBR$ be the two DMBRs for the DMBRs of the entries in ES_1 and ES_2, respectively. If $area(overlap(P)) = area(ES_1.DMBR \cap ES_2.DMBR) = 0$, P is said to be overlap-free.

Algorithm **SplitNode** takes an overflow node N as the input and splits it into two new nodes N_1 and N_2 whose entry sets are from a partition defined above. Since there are usually many possible partitions for a given overflow node, a good partition that leads to an efficient ND-tree should be chosen for splitting the overflow node. The node splitting procedure is described as follows:

Algorithm 3.2. : **SplitNode**
Input: overflow node N of an ND-tree.
Output: two new nodes N_1 and N_2.
Method:

```
1. invoke Algorithm ChoosePartitionSet on $N$ to find a set $\Delta$ of candidate
       partitions for $N$;
2. invoke Algorithm ChooseBestPartition to choose a best partition $BP$
       from $\Delta$;
3. generate two new nodes $N_1$ and $N_2$ that contain the two entry sets of $BP$,
       respectively;
4. return $N_1$ and $N_2$.
```

In the above procedure, Algorithm **ChoosePartitionSet** determines a set of candidate partitions to consider, while Algorithm **ChooseBestPartition** chooses the best partition from the candidates based on several heuristics. The details of these two algorithms are given below.

3.2.4 *Choosing Candidate Partitions.* To find a good partition for splitting an overflow node N, we need to consider a set of candidate partitions. It is worth noting that, in a CDS, some partitions can be easily ruled out as good candidates based on the ordering and continuous properties of the space. However, in an NDDS, a similar judgment cannot be made due to lack of the corresponding properties. For example, assume that we want to split a leaf node with 4 entries/vectors (the associated database object pointers are omitted here): $\alpha_1 = (x_1, y_1)$, $\alpha_2 = (x_2, y_2)$, $\alpha_3 = (x_3, y_3)$ and $\alpha_4 = (x_4, y_4)$ in a 2-dimensional space H_2 into two nodes with 2 entries in each (assuming the minimum space utilization is 35%). If H_2 is a CDS and $x_1 < x_2 < x_3 < x_4$; $y_1 < y_2 < y_3 < y_4$ (see Figure 2), one can easily determine that the only overlap-free (good) partition is $P = \{\{\alpha_1, \alpha_2\}, \{\alpha_3, \alpha_4\}\}$ and no other partition needs to be considered, based on the ordering of the given values in each dimension. However, if H_2 is an NDDS and the component values (i.e., x_i $'s$

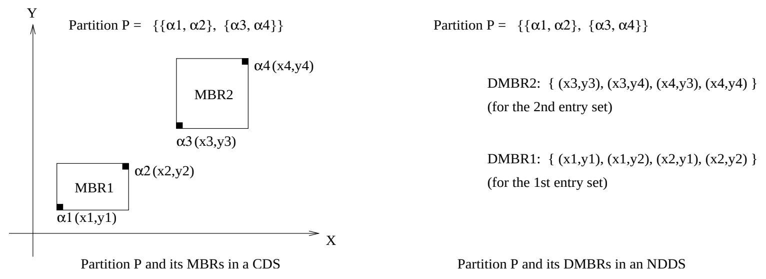

Partition P = {{α1, α2}, {α3, α4}}
Y
X
MBR1
MBR2
α1 (x1,y1)
α2 (x2,y2)
α3 (x3,y3)
α4 (x4,y4)
Partition P and its MBRs in a CDS
Partition P = {{α1, α2}, {α3, α4}}
DMBR2: { (x3,y3), (x3,y4), (x4,y3), (x4,y4) }
(for the 2nd entry set)
DMBR1: { (x1,y1), (x1,y2), (x2,y1), (x2,y2) }
(for the 1st entry set)
Partition P and its DMBRs in an NDDS

Fig. 2. Comparison of partitions in CDS and NDDS

and y_j $'s$) of the 4 given vectors are distinct in each dimension, any partition with 2 entries in each entry set is overlap-free. In other words, entries (vectors) can be grouped in any way without causing overlap in this case. Hence an NDDS provides less information to reduce the number of candidate partitions for consideration, leading to a harder splitting task in general.

One exhaustive way to generate all possible candidate partitions for an overflow node N is as follows. For each permutation of the $M+1$ entries in N, first j ($m \leq j \leq M-m+1$) entries are put in the first entry set of a partition P_j, and the remaining entries are put in the second entry set of P_j. Even for a small M, say 50, this approach would have to consider $51! \approx 1.6 \times 10^{66}$ permutations of the entries in N. Although this approach is guaranteed to find an optimal partition, it is not feasible in practice. A more efficient method to generate a (smaller) set of candidate partitions is required.

We notice that the size of alphabet A for an NDDS is usually small. For example, $|A| = 4$ for a genome sequence database. Let $l_1, l_2, ..., l_{|A|}$ be the letters of alphabet A. A permutation of A is a (ordered) list of letters in A: $< l_{i_1}, l_{i_2}, ..., l_{i_{|A|}} >$ where $l_{i_k} \in A$ and $1 \leq k \leq |A|$. For example, for $A = \{a, g, t, c\}$ for a genome sequence database, $< g, c, a, t >$ and $< t, a, c, g >$ are two permutations of A. Since $|A| = 4$, there are only $4! = 24$ permutations of A. Based on this observation, we have developed the following more efficient algorithm for generating candidate partitions.

Algorithm 3.3. : **ChoosePartitionSet I**
Input: overflow node N of an ND-tree for an NDDS Ω_d over alphabet A.
Output: a set Δ of candidate partitions.
Method:

```
1.  let $\Delta = \emptyset$;
2.  **for** dimension $D = 1$ to $d$ **do**
3.    **for** each permutation $\beta$: $< l_1, l_2, ..., l_{|A|} >$ of $A$ **do**
4.      set up an array of buckets (lists): $bucket[1 .. 4 * |A|]$;
          // $bucket[(i-1)*4+1]$, ..., $bucket[(i-1)*4+4]$ are for letter $l_i$
          // $(1 \leq i \leq |A|)$
5.      **for** each entry $E$ in $N$ **do**
6.        let $l_i$ be the foremost letter in $\beta$ that the $D$-th component set ($S_D$) of
              DMBR of $E$ has;
7.        **if** $S_D$ contains only $l_i$ **then**
```

```
8.                put $E$ into $bucket[(i-1)*4+1]$;
9.             else if $S_D$ contains only $l_i$ and $l_{i+1}$ then
10.               put $E$ into $bucket[(i-1)*4+4]$;
11.            else if $S_D$ contains both $l_i$ and $l_{i+1}$ together with at least one other
                  letter then
12.               put $E$ into $bucket[(i-1)*4+3]$;
13.            else put $E$ into $bucket[(i-1)*4+2]$;
                  // $S_D$ has $l_i$ and at least one non-$l_{i+1}$ letter
14.            end if;
15.         end for;
16.         sort entries within each bucket alphabetically by $\beta$ based on their
               $D$-th component sets;
17.         concatenate $bucket[1]$, ..., $bucket[4*|A|]$ into one list $PN$:
               $< E_1, E_2, ..., E_{M+1} >$;
18.         for $j = m$ to $M - m + 1$ do
19.            generate a partition $P$ from $PN$ with entry sets: $ES_1 = \{E_1, ..., E_j\}$
                  and $ES_2 = \{E_{j+1}, ..., E_{M+1}\}$;
20.            let $\Delta = \Delta \cup \{P\}$;
21.         end for;
22.      end for;
23.   end for;
24.   return $\Delta$.
```

For each dimension (Step 2), Algorithm 3.3 determines one ordering of entries in the overflow node (Steps 4 - 17) for each permutation of alphabet A (Step 3). Each ordering of entries generates $M-2m+2$ candidate partitions (Steps 18 - 21). Hence a total number of $d*(M-2m+2)*(|A|!)$ candidate partitions are considered by the algorithm. Since $|A|$ is usually small, this algorithm is much more efficient than the previous exhaustive approach. For example, if $d = 25$, $M = 50$, $m = 10$ and $|A| = 4$, Algorithm 3.3 considers only 1.92×10^4 partitions. In fact, only half of all permutations of A need to be considered since a permutation and its reverse will yield the same set of candidate partitions by the algorithm. Using this fact, the efficiency of the algorithm can be further improved.

Given a dimension D, to determine the ordering of entries in the overflow node based on a permutation $\beta :< l_1, l_2, ..., l_{|A|} >$ of A, Algorithm 3.3 employs a bucket ordering technique (Steps 4 - 17). The goal is to choose an ordering of entries that has a better chance to generate good partitions (i.e., small overlap). Greedy strategies are adopted here to achieve this goal. Essentially, the algorithm groups the entries according to their foremost (based on β) letters in their D-th component sets. The entries in a group sharing a foremost letter l_i are placed before the entries in a group sharing a foremost letter l_j if $i < j$. In this way, if the splitting point of a partition is at the boundary between two groups, it is guaranteed that the D-th component sets of entries in the second entry set ES_2 of the partition do not have the foremost letters in the D-th component sets of entries in the first entry set ES_1. Furthermore, each group is divided into four subgroups (buckets) according to the rules implemented by Steps 7 - 14. The greedy strategy used here is to (1) put entries from the current group that contain the foremost letter of the next group as close to the next group as possible, and (2) put entries from the current group that contain only its foremost letter close to the previous group. In this way, a partition with the splitting point at the boundary between two buckets in a group is locally optimized with respect to the current as well as its neighboring groups. The alphabetical ordering (based on the given permutation) is then used to sort entries in each bucket based on their D-th component sets. Note that the last and the second last groups have at most one and two non-empty subgroups (buckets), respectively. Considering all permutations for a dimension increases the chance to obtain a good partition of entries based on that dimension, while examining all dimensions increases the chance to obtain a good partition of entries across multiple dimensions.

For the comparison purpose, we also tested the approach to use the alphabetical ordering to sort all entries directly and found that it usually also yields a satisfactory performance. However, there are cases in which the bucket ordering is more effective.

Example 2. Consider an ND-tree for a genomic data set in the 25-dimensional NDDS with alphabet $A = \{a, g, t, c\}$. The maximum and minimum numbers of entries allowed in a tree node are 10 and 3, respectively. Assume that, for a given overflow node N with 11 entries $E_1, E_2, ..., E_{11}$, Algorithm 3.3 is checking the 5th dimension (Step 2) at the current time. The 5th component sets of the DMBRs of the 11 entries are listed as follows, respectively:

$$\{t\}, \{gc\}, \{c\}, \{ac\}, \{c\}, \{agc\}, \{t\}, \{at\}, \{a\}, \{c\}, \{a\}$$

The total number of permutations of alphabet A is $|A|!$=24. As mentioned before, only half of all the permutations need to be considered. Assume that the algorithm is checking one of the 12 permutations, say $< c, a, t, g >$ (Step 3). The non-empty buckets obtained from Steps 4 - 15 are:

$$\begin{array}{lll} bucket[1] = \{E_3, E_5, E_{10}\}, & bucket[2] = \{E_2\}, & bucket[3] = \{E_6\}, \\ bucket[4] = \{E_4\}, & bucket[5] = \{E_9, E_{11}\}, & bucket[8] = \{E_8\}, \\ bucket[9] = \{E_1, E_7\}, & unlisted\ buckets = \emptyset. & \end{array}$$

Thus the entry list obtained at Step 17 is shown in Figure 3. Based on the en-

$$\begin{array}{ccccccccccccccccccccccc} < & E_3 & & E_5 & & E_{10} & & E_2 & & E_6 & & E_4 & & E_9 & & E_{11} & & E_8 & & E_1 & & E_7 & > \\ & & & & & & \uparrow & & \uparrow & & \uparrow & & \uparrow & & \uparrow & & \uparrow & & & & & & \\ & & & & & & P_1 & & P_2 & & P_3 & & P_4 & & P_5 & & P_6 & & & & & & \end{array}$$

Fig. 3. Entry list and partitions

try list, Steps 18 - 21 generate candidate partitions $P_1 \sim P_6$ whose splitting points are also illustrated in Figure 3. For example, partition P_2 consists of $ES_1 = \{E_3, E_5, E_{10}, E_2\}$ and $ES_2 = \{E_6, E_4, E_9, E_{11}, E_8, E_1, E_7\}$. These partitions comprise part of result set Δ returned by Algorithm 3.3. Note that if we replace the 5th component set $\{at\}$ of E_8 with $\{t\}$, P_6 would be an overlap-free partition.

One may notice that the sorting of entries based on a particular permutation of the alphabet may not treat all entries symmetrically. For example, if we have two more entries with the 5th component sets $\{ct\}$ and $\{ag\}$ in the given tree node, we can see that entries with $\{ct\}$ and $\{cg\}$ would be put into the same bucket (i.e., $bucket[2]$); while entries with $\{at\}$ and $\{ag\}$ would be put into two different buckets (i.e., $bucket[8]$ and $bucket[6]$, respectively). Intuitively, one would think that the degree of difference ("distance") between $\{ct\}$ and $\{cg\}$ is similar to that between $\{at\}$ and $\{ag\}$. The reason why the above two pairs are treated differently is that a local greedy strategy to minimize overlapping is applied in our sorting, which separates groups of component sets with different degrees of involvement of two neighboring letters. For the given permutation $< c, a, t, g >$, neither 't' nor 'g' is next to 'c'. On the other hand, 't' is next to 'a', while 'g' is not. Hence $\{ct\}$ and $\{cg\}$ are considered to be more similar to each other than $\{at\}$ and $\{ag\}$ when this permutation is considered. To mitigate the bias, Algorithm 3.3 considers all permutations (Step 3). When permutation $< a, c, t, g >$ is considered, the case in which $\{at\}$ and $\{ag\}$ are put into the same bucket is considered. □

Note that Algorithm 3.3 not only is efficient but also possesses a desirable property, which is stated as follows:

PROPOSITION 3.1. *If there exists at least one overlap-free partition for the overflow node, Algorithm 3.3 will find such a partition.*

PROOF. Based on the assumption, there exists an overlap-free partition $PN = \{ES_1, ES_2\}$. Let $ES_1.DMBR = S_{11} \times S_{12} \times ... \times S_{1d}$ and $ES_2.DMBR = S_{21} \times S_{22} \times ... \times S_{2d}$. Since $area(ES_1.DMBR \cap ES_2.DMBR) = 0$, there exists a dimension D ($1 \leq D \leq d$) such that $S_{1D} \cap S_{2D} = \emptyset$. Since Algorithm 3.3 examines every dimension, dimension D will be checked. Without loss of generality, assume $S_{1D} \cup S_{2D} = A$, where A is the alphabet for the underlying NDDS.

Consider the following permutation of A: $PA =< l_{11}, ..., l_{1s}, l_{21}, ..., l_{2t} >$ where $l_{1i} \in S_{1D}$ ($1 \leq i \leq s$), $l_{2j} \in S_{2D}$ ($1 \leq j \leq t$), and $s + t = |A|$. Enumerate all entries of the overflow node based on PA in the way described in Steps 4 - 17 of Algorithm 3.3. We have the entry list $EL =< E_1, E_2, ..., E_{M+1} >$ shown in Figure 4. Since $S_{1D} \cap S_{2D} = \emptyset$, all entries in Part 1 do not contain letters in S_{2D} on the

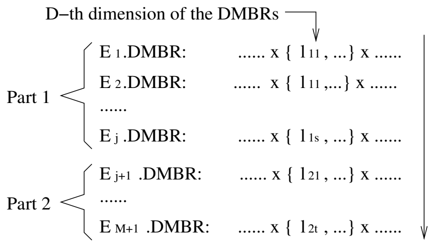

D–th dimension of the DMBRs
Part 1
E 1.DMBR: ...... x { l 11 , ...} x ......
E 2.DMBR: ...... x { l 11 ,...} x ......
......
E j .DMBR: ...... x { l 1s , ...} x ......
Part 2
E j+1 .DMBR: ...... x { l 21 , ...} x ......
......
E M+1 .DMBR: ...... x { l 2t , ...} x ......

Fig. 4. A permutation of entries ($1 \leq j \leq M + 1$)

D-th dimension, and all entries in Part 2 do not contain letters in S_{1D} on the D-th dimension. In fact, $Part1 = ES_1$ and $Part2 = ES_2$, which yields the partition PN. Since the algorithm examines all permutations of A, such a partition will be put into the set of candidate partitions. □

The time complexity of Algorithm 3.3 can be obtained from the following analysis. In the worst case, Steps 5 - 15 require a time complexity of $O((M+1) * |A|)$; Step 16 requires a time complexity of $O((M+1) * log(M+1))$; Step 17 requires a time complexity of $O(4 * |A|)$; and Steps 18 - 21 require a time complexity of $O(M-2m+2)$. Hence the worst-case time complexity of Algorithm 3.3 is: $O(d * (|A|!) * [(M+1) * |A| + (M+1) * log(M+1) + 4 * |A| + (M-2m+2)])$, namely,

$$Time(ChoosePartitionSet\ I) = O(d * (|A|!) * M * (|A| + logM)). \qquad (4)$$

The main space required by Algorithm 3.3 is the array of buckets. Note that, although finding a set of candidate partitions and choosing a best partition from the candidate set are presented as two steps in Algorithm 3.2 for conceptual simplicity, they can be implemented in a pipelining fashion. Thus we need only one array of buckets (i.e., the space for one candidate partition) in Algorithm 3.3. The array has $4 * |A|$ items (bucket heads). Each bucket (list) can have up to $M+1$ entries. However, the total number of entries in all the buckets remains $M+1$. Besides, the buckets only need to store the current dimension D and the relevant entry number Eid for each entry. Using D and Eid, the corresponding DMBR and its D-th component set can be found in the given overflow node of the ND-tree (outside the algorithm). Therefore, the space complexity of Algorithm 3.3 is: $O(4 * |A| + 2 * (M+1))$, namely,

$$Space(ChoosePartitionSet\ I) = O(|A| + M). \qquad (5)$$

It is possible that alphabet A for some NDDS is large. In this case, the number of possible permutations of A may be too large to be efficiently used in Algorithm 3.3. We have, therefore, developed another algorithm to efficiently generate candidate partitions in such a case. The key idea is to use some strategies to intelligently determine one ordering of entries in the overflow node for each dimension rather than consider $|A|!$ orderings determined by all permutations of A for each dimension. This algorithm is described as follows:

ALGORITHM 3.4. : **ChoosePartitionSet II**
Input: overflow node N of an ND-tree for an NDDS Ω_d over alphabet A.
Output: a set Δ of candidate partitions
Method:

```
1. let $\Delta = \emptyset$;
2. for dimension $D = 1$ to $d$ do
3.    auxiliary tree $T = build\_aux\_tree(N, D)$;
4.    $D$-th component sets list $CS = sort\_csets(T)$;
5.    replace each component set in $CS$ with its associated entries to get entry
          list $PN$;
6.    for $j = m$ to $M - m + 1$ do
7.       generate a partition $P$ from $PN$ with entry sets: $ES_1 = \{E_1, ..., E_j\}$ and
             $ES_2 = \{E_{j+1}, ..., E_{M+1}\}$;
8.       let $\Delta = \Delta \cup \{P\}$;
9.    end for;
10. end for;
11. return $\Delta$.
```

For each dimension (Step 2), Algorithm 3.4 first builds an auxiliary tree by invoking Function *build_aux_tree* (Step 3) and then uses the tree to sort the D-th component sets of the entries by invoking Function *sort_csets* (Step 4). The position of each entry in the ordering is determined by its D-th component set in the sorted list *CS* (Step 5). Note that, if multiple entries have the same D-th component set, a random order is chosen among them. Using the resulting entry list, the algorithm generates $M - 2m + 2$ candidate partitions. Hence the total number of candidate partitions considered by the algorithm is $d * (M - 2m + 2)$.

The algorithm also possesses a desirable property; that is, it generates an overlap-free partition if there exists one. This property is achieved by building an auxiliary tree in Function *build_aux_tree*. Each node T in the auxiliary tree has three data fields:

—$T.sets$ — the group (set) of the D-th component sets represented by the subtree rooted at T,
—$T.freq$ — the total frequency of sets in $T.sets$, where the frequency of a (D-th component) set is defined as the number of entries having the set,
—$T.letters$ — the set of letters appearing in any set in $T.sets$.

The D-th component set groups represented by the subtrees at the same level are disjoint in the sense that a component set in one group does not share any letter with any component set in another group. Hence, if a root T has subtrees T_1, ..., T_n ($n > 1$) and $T.sets = T_1.sets \cup ... \cup T_n.sets$, then we find the disjoint groups $T_1.sets$, ..., $T_n.sets$ of all D-th component sets. By placing the entries with the component sets in the same group together, an overlap-free partition can be obtained by using a splitting point at the boundary between two groups. The auxiliary tree is obtained by repeatedly merging the component sets that directly or indirectly intersect with each other, as described as follows:

```
FUNCTION 1. $auxiliary\_tree = build\_aux\_tree(N, D)$
 1. find set $L$ of letters appearing in at least one $D$-th component set;
 2. initialize forest $F$ with single-node trees, one tree $T$ for each $l \in L$ and
       set $T.letters = \{l\}$, $T.sets = \emptyset$, $T.freq = 0$;
 3. sort all $D$-th component sets by size in ascending order and break ties by
       frequency in descending order into set list $SL$;
 4. for each set $S$ in $SL$ do
 5.    if there is only one tree $T$ in $F$ such that $T.letters \cap S \neq \emptyset$ then
 6.       let $T.letters = T.letters \cup S$,  $T.sets = T.sets \cup \{S\}$,
              $T.freq = T.freq + frequency\ of\ S$;
 7.    else let $T_1$, ..., $T_n$ ($n > 1$) be trees in $F$ whose $T_i.letters \cap S \neq \emptyset$ ($1 \leq i \leq n$);
 8.       create a new root $T$ with each $T_i$ as a subtree;
 9.       let $T.letters = (\cup_{i=1}^{n} T_i.letters) \cup S$,  $T.sets = (\cup_{i=1}^{n} T_i.sets) \cup \{S\}$,
              $T.freq = (\sum_{i=1}^{n} T_i.freq) + frequency\ of\ S$;
10.       replace $T_1$, ..., $T_n$ by $T$ in $F$;
11.    end if;
12. end for;
13. if $F$ has 2 or more trees $T_1$, ..., $T_n$ ($n > 1$) then
14.    create a new root $T$ with each $T_i$ as a subtree;
```

```
15.    let $T.letters = \cup_{i=1}^{n} T_i.letters,\ T.sets = \cup_{i=1}^{n} T_i.sets,$
           $T.freq = \sum_{i=1}^{n} T_i.freq;$
16. else let $T$ be the unique tree in $F$;
17. end if;
18. return $T$.
```

The time complexity of Function *build_aux_tree* can be obtained from the following analysis. In the worst case, Step 1 requires a time complexity of $O((M+1)*|A|)$; Step 2 requires a time complexity of $O(|A|)$; Step 3 requires a time complexity of $O((M+1)*log(M+1))$; Steps 5 and 7 require a time complexity of $O(|A|^2)$; Step 6 requires a time complexity of $O(|A|+(M+1)+1)$; Step 8 requires a time complexity of $O(|A|)$; Step 9 requires a time complexity of $O(|A|*(|A|+1)+(M+1)*(|A|+1)+(|A|+1))$; Step 10 requires a time complexity of $O(|A|)$; Steps 4 - 12 require a time complexity of $O((M+1)*[|A|^2+|A|*(|A|+1)+(M+1)*(|A|+1)])$; Step 13 requires a time complexity of $O(|A|)$; Step 14 requires a time complexity of $O(|A|)$; Step 15 requires a time complexity of $O(|A|*(|A|+M+2))$; Step 16 requires a time complexity of $O(1)$. Hence the worst-case time complexity of Function *build_aux_tree* is $O((M+1)*[|A|^2+|A|*(|A|+1)+(M+1)*(|A|+1)])$, namely,

$$Time(build\_aux\_tree) = O(M*|A|*(|A|+M)). \tag{6}$$

The main space required by Function *build_aux_tree* is for the auxiliary tree that it builds. For each auxiliary tree node N, in the worst case, $N.letters$ requires a space complexity of $O(|A|)$; $N.freq$ requires a space complexity of $O(1)$; $N.sets$ requires a space complexity of $O(M+1)$ assuming every D-th component set is represented by the current dimension D and the corresponding entry number Eid as in Algorithm 3.3. Hence the worst-case space required for each auxiliary tree node is $O(|A|+M)$. From Step 2, the maximum number $Max\_\#\_leaf(T)$ of leaf nodes in an auxiliary tree T is:

$$Max\_\#\_leaf(T) = O(|A|). \tag{7}$$

From Steps 4, 8 and 14, the maximum number $Max\_\#\_nonleaf(T)$ of non-leaf nodes in an auxiliary tree T is $O(M+2)$, namely,

$$Max\_\#\_nonleaf(T) = O(M). \tag{8}$$

Therefore, the worst-case space complexity of Function *build_aux_tree* is

$$Space(build\_aux\_tree) = O((|A|+M)^2). \tag{9}$$

Using the auxiliary tree generated by Function *build_aux_tree*, Algorithm 3.4 invokes Function *sort_csets* to determine the ordering of all D-th component sets. To do that, starting from the root node T, *sort_csets* first determines the ordering of the component set groups represented by all subtrees of T and put them into a list *ml* with each group as an element. The ordering decision is based on the frequencies of the groups/subtrees. The principle is to put the groups with smaller frequencies in the middle of *ml* to increase the chance to obtain more diverse candidate partitions. For example, assume that the auxiliary tree identifies 4 disjoint groups G_1, ..., G_4 of all component sets with frequencies 2, 6, 6, 2, respectively, and the minimum space requirement for the ND-tree is $m = 3$. If list $< G_1, G_2, G_3, G_4 >$ is used, we can obtain only one overlap-free partition (with the splitting point at the boundary between G_2 and G_3). If list $< G_2, G_1, G_4, G_3 >$ is used, we can have three overlap-free partitions (with splitting points at the boundaries between G_2 and G_1, G_1 and G_4, and G_4 and G_3, respectively).

There may be some component sets in $T.sets$ that are not represented by any of its subtrees (since they may contain letters in more than one subtree). Such a component set is called a crossing set. If current list ml has n elements (after removing empty group elements if any), there are $n + 1$ possible positions for a crossing set e. After e is put at one of the positions, there are $n+2$ gaps/boundaries between two consecutive elements in the list. However, the leftmost and rightmost boundaries have all the elements on one of their two sides. For the remaining n boundaries, placing a splitting point at such a boundary will result in a non-trivial partition. For each of these n partitions, we can calculate the number of common letters (i.e., intersection on the D-th dimension) shared between the left component sets and the right component sets. We place e at a position with the minimal sum of the sizes of above D-th intersections at the n boundaries.

Each group element in ml is represented by a subtree. To determine the ordering among the component sets in the group, the above procedure is recursively applied to the subtree until the height of a subtree is 1. In that case, the corresponding (component set) group element in ml is directly replaced by the component set (if any) in the group. Once the component sets within every group element in ml are determined, the ordering among all component sets is obtained.

```
FUNCTION 2. $set\_list = sort\_csets(T)$
 1. **if** height of tree $T = 1$ **then**
 2.    **if** $T.sets \neq \emptyset$ **then**
 3.       put the sets in $T.sets$ into list $set\_list$;
 4.    **else** set $set\_list$ to null;
 5.    **end if**;
 6. **else** set lists $L_1 = L_2 = \emptyset$;
 7.    let $weight_1 = weight_2 = 0$;
 8.    **while** there is an unconsidered subtree of $T$ **do**
 9.       get such subtree $T'$ with highest frequency;
10.       **if** $weight_1 \leq weight_2$ **then**
11.          let $weight_1 = weight_1 + T'.freq$;
12.          add $T'.sets$ to the end of $L_1$;
13.       **else** let $weight_2 = weight_2 + T'.freq$;
14.          add $T'.sets$ to the beginning of $L_2$;
15.       **end if**;
16.    **end while**;
17.    concatenate $L_1$ and $L_2$ into $ml$;
18.    let $S$ be the set of crossing sets in $T.sets$;
19.    **for** each set $e$ in $S$ **do**
20.       insert $e$ into a position in $ml$ with the minimal sum of the sizes of all
              $D$-th intersections;
21.    **end for**;
```

```
22.    for each subtree $T'$ of $T$, do
23.        $set\_list' = sort\_csets(T')$;
24.        replace group $T'.sets$ in $ml$ with $set\_list'$;
25.    end for;
26.    $set\_list = ml$;
27. end if;
28. return $set\_list$.
```

Since the above merge-and-sort procedure allows Algorithm 3.4 to make an intelligent choice of candidate partitions, our experiments demonstrate that the performance of an ND-tree obtained from this algorithm is comparable to that of an ND-tree obtained from Algorithm 3.3 (see Section 4).

The time complexity of Function *sort_csets* can be obtained from the following analysis. We notice that the algorithm segment Steps 1 - 5 is applied to each leaf node of the input auxiliary tree. In the worst case, Steps 1 - 2 require a time complexity of $O(1)$; Step 3 requires a time complexity of $O(M+1)$; Steps 4 - 5 require a time complexity of $O(1)$. Hence, Steps 1 - 5 require a time complexity of $O(M+1)$, which is the worst-case time needed for each leaf node. From Formula (7), the worst-case time needed for all leaf nodes of the input auxiliary tree is:

$$Time\_for\_all\_leaves = O(|A| * M). \qquad (10)$$

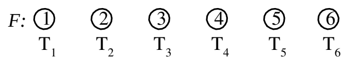
On the other hand, the algorithm segment Steps 6 - 27 (except 23) is applied to each non-leaf node of the given auxiliary tree. In the worst case, Steps 6 - 7 require a time complexity of $O(1)$; Step 9 requires a time complexity of $O(|A|)$; Steps 10 - 15 require a time complexity of $O(1)$; Steps 8 - 16 require a time complexity of $O(|A|^2)$; Step 17 requires a time complexity of $O(1)$; Step 18 requires a time complexity of $O((M+1)^2 * |A|)$; Step 20 requires a time complexity of $O((|A| + M + 2) * [(|A| + M + 1)^2 * |A| + (|A| + M + 1) + 1])$ because there are no more than $O(|A| + M + 1)$ elements in list *ml*; Steps 19 - 21 require a time complexity of $O((M+1)*(|A|+M+2)*[(|A|+M+1)^2*|A|+(|A|+M+1)+1])$; Step 24 requires a time complexity of $O(|A| * (|A| + M + 1))$; Steps 22 - 25 (except 23) require a time complexity of $O(|A|^2 * (|A| + M + 1))$; Step 26 requires a time complexity of $O(1)$. Note that Step 23 is the place where the function extends its computation to the child nodes of the current non-leaf node. Hence the worst-case time needed for each non-leaf node (excluding its children) is $O(((M+1) * (|A| + M + 2) * [(|A| + M + 1)^2 * |A| + (|A| + M + 1) + 1])$, namely, $O(M * (|A| + M)^3 * |A|)$. From Formula (8), the worst-case time needed for all non-leaf nodes of the input auxiliary tree is:

$$Time\_for\_all\_nonleaves = O(M^2 * (|A| + M)^3 * |A|) \qquad (11)$$

From Formulas (10) and (11), the worst-case time complexity of Function *sort_csets* is

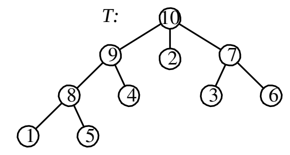
$$Time(sort\_csets) = O(M^2 * (|A| + M)^3 * |A|). \qquad (12)$$

The main space required by Function *sort_csets* is the list *ml*, which has no more than $O(|A| + M + 1)$ elements. Each element has no more than $O(M+1)$ component sets. As before, assume that each component set is represented by the current dimension D and the corresponding entry number *Eid*. Hence the worst-



1.*letters*: {*a*}, 1.*freq*: 0, 1.*sets*: ∅
2.*letters*: {*b*}, 2.*freq*: 0, 2.*sets*: ∅
...
6.*letters*: {*f*}, 6.*freq*: 0, 6.*sets*: ∅

Fig. 5. Initial forest F



1.*letters*: {*a*},	1.*freq*: 1,	1.*sets*: **{**{*a*}**}**
2.*letters*: {*b*},	2.*freq*: 1,	2.*sets*: **{**{*b*}**}**
3.*letters*: {*c*},	3.*freq*: 1,	3.*sets*: **{**{*c*}**}**
4.*letters*: {*d*},	4.*freq*: 0,	4.*sets*: ∅
5.*letters*: {*e*},	5.*freq*: 2,	5.*sets*: **{**{*e*}**}**
6.*letters*: {*f*},	6.*freq*: 1,	6.*sets*: **{**{*f*}**}**
7.*letters*: {*cf*},	7.*freq*: 4,	7.*sets*: **{**{*c*},{*f*},{*cf*}**}**
8.*letters*: {*ae*},	8.*freq*: 4,	8.*sets*: **{**{*a*},{*e*},{*ae*}**}**
9.*letters*: {*ade*},	9.*freq*: 6,	9.*sets*: **{**{*a*},{*e*},{*ae*},{*de*},{*ade*}**}**
10.*letters*: {*abcdef*},	10.*freq*: 11,	10.*sets*: all sets that appears

Fig. 6. Final auxiliary tree T

case space complexity of Function *sort_csets* is:

$$Space(sort\_csets) = O(M * (|A| + M)). \tag{13}$$

Example 3. Consider an ND-tree with alphabet $A = \{a, b, c, d, e, f\}$ for a 20-dimensional NDDS. The maximum and minimum numbers of entries allowed in a tree node are 10 and 3, respectively. Assume that, for a given overflow node N with 11 entries $E_1, E_2, ..., E_{11}$, Algorithm 3.4 is checking the 3rd dimension (Step 2) at the current time. The 3rd component sets of the DMBRs of the 11 entries are listed as follows, respectively:

$$\{c\}, \{ade\}, \{b\}, \{ae\}, \{f\}, \{e\}, \{cf\}, \{de\}, \{e\}, \{cf\}, \{a\}$$

The initial forest F generated at Step 2 of Function *build_aux_tree* is illustrated in Figure 5. The auxiliary tree T obtained by the function is illustrated in Figure 6. Note that non-leaf node of T is numbered according to its order of merging.

Using auxiliary tree T, recursive Function *sort_csets* is invoked to sort the component sets. List ml in Function *sort_csets* evolves as follows:

$<\boldsymbol{\{}\{a\}, \{e\}, \{ae\}, \{de\}, \{ade\}\boldsymbol{\}}, \boldsymbol{\{}\{b\}\boldsymbol{\}}, \boldsymbol{\{}\{c\}, \{f\}, \{cf\}\boldsymbol{\}}>;$
$< \{de\}, \{ade\}, \{e\}, \{ae\}, \{a\}, \boldsymbol{\{}\{b\}\boldsymbol{\}}, \boldsymbol{\{}\{c\}, \{f\}, \{cf\}\boldsymbol{\}}>;$
$< \{de\}, \{ade\}, \{e\}, \{ae\}, \{a\}, \{b\}, \boldsymbol{\{}\{c\}, \{f\}, \{cf\}\boldsymbol{\}}>;$
$< \{de\}, \{ade\}, \{e\}, \{ae\}, \{a\}, \{b\}, \{c\}, \{cf\}, \{f\} >.$

Based on the set list returned by Function *sort_csets*, Step 5 in Algorithm 3.4 produces the following sorted entry list *PN*: $< E_8$, E_2, E_6, E_9, E_4, E_{11}, E_3, E_1, E_7, E_{10}, $E_5 >$.

Based on *PN*, Algorithm 3.4 generates candidate partitions in the same way as Example 2, which comprise part of result set Δ returned by Algorithm 3.4. Note that the two partitions with splitting points at the boundary between E_{11} and E_3 and the boundary between E_3 and E_1 are overlap-free partitions. □

The time complexity of Algorithm 3.4 can be obtained from the following analysis. In the worst case, Step 5 requires $O(M + 1)$; Steps 6 - 9 require $O(M - 2m + 2)$.

From Formulas (6) and (12), the worst-case time complexity of Algorithm 3.4 is:

$$Time(ChoosePartitionSet\ II) = O(d * M^2 * |A| * (|A| + M)^3). \qquad (14)$$

Unlike Formula (4), the above time complexity does not contain the factor ($|A|!$). Algorithm 3.4 is more efficient for a large alphabet.

The main space required by Algorithm 3.4 is for the auxiliary tree generated by Function *build_aux_tree*, the component sets list returned by Function *sort_csets* and a entry list (of length $O(M + 1)$) used in the main body of the algorithm. Therefore, the worst-case space complexity of the algorithm is:

$$Space(ChoosePartitionSet\ II) = O((|A| + M)^2). \qquad (15)$$

Comparing to Formula (5), the space requirement of Algorithm 3.4 is higher than that of Algorithm 3.3.

3.2.5 *Choosing the Best Partition.* Once a set of candidate partitions are generated, we need to select the best one from them based on some heuristics. As mentioned before, due to the limited size of an NDDS, many ties may occur for one heuristic. Hence multiple heuristics are required. After evaluating heuristics in some popular indexing methods (such as the R*-tree, the X-tree and the Hybrid-tree), we have identified the following effective heuristics for choosing a partition (i.e., a split) of an overflow node of an ND-tree in an NDDS:

SH_1 : Choose a partition that generates a minimum overlap of the DMBRs of the two new nodes after splitting ("minimize overlap").

SH_2 : Choose a partition that splits on the dimension where the edge length of the DMBR of the overflow node is the largest ("maximize span").

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 ("center split").

SH_4 : Choose a partition that minimizes the total area of the DMBRs of the two new nodes after splitting ("minimize area").

From our experiments (see Section 4), we observed that heuristic SH_1 is the most effective one in an NDDS, but many ties may occur as expected. Heuristics SH_2 and SH_3 can effectively resolve ties in such cases. Heuristic SH_4 is also effective. However, it is expensive to use since it has to examine all dimensions of a DMBR. In contrast, heuristics SH_1 - SH_3 can be met without examining all dimensions. For example, SH_1 is met as long as one dimension is found to have no overlap between the corresponding component sets of the new DMBRs; and SH_2 is met as long as the splitting dimension is found to have the maximum edge length $|A|$ for the current DMBR. Hence the first three heuristics are suggested to be used in choosing the best partition for an ND-tree as follows:

Algorithm 3.5. : **ChooseBestPartition**

```
Input: set $\Delta$ of candidate partitions for overflow node $N$ of an ND-tree in an
        NDDS over alphabet $A$.
Output: chosen partition $BP$ of overflow node $N$.
Method:
 1. let $BP = \{ES_1, ES_2\}$ be any partition in $\Delta$ with splitting dimension $D$;
```

```
2.  let $BP\_overlap = area(ES_1.DMBR \cap ES_2.DMBR)$;
3.  let $BP\_span = length(BP.DMBR, D)$;
4.  let $BP\_balance = abs(length(ES_1.DMBR, D) - length(ES_2.DMBR, D))$;
5.  let $\Delta = \Delta - \{BP\}$;
6.  while $\Delta$ is not empty and not($BP\_overlap = 0$ and $BP\_span = |A|$ and
        $BP\_balance = 0$) do
7.    let $CP = \{ES_1, ES_2\}$ be any partition in $\Delta$ with splitting dimension $D$;
8.    let $CP\_overlap = area(ES_1.DMBR \cap ES_2.DMBR)$;
9.    let $CP\_span = length(BP.DMBR, D)$;
10.   let $CP\_balance = abs(length(ES_1.DMBR, D) - length(ES_2.DMBR, D))$;
11.   let $\Delta = \Delta - \{CP\}$;
12.   if $CP\_overlap < BP\_overlap$ then
13.     let $BP = CP$;
14.     let $BP\_overlap = CP\_overlap$;
15.     let $BP\_span = CP\_span$;
16.     let $BP\_balance = CP\_balance$;
17.   else if $CP\_overlap = BP\_overlap$ then
18.     if $CP\_span < BP\_span$ then
19.       let $BP = CP$;
20.       let $BP\_span = CP\_span$;
21.       let $BP\_balance = CP\_balance$;
22.     else if $CP\_span = BP\_span$ then
23.       if $CP\_balance < BP\_balance$ then
24.         let $BP = CP$;
25.         let $BP\_balance = CP\_balance$;
26.       end if;
27.     end if;
28.   end if;
29. end while;
30. return $BP$.
```

As mentioned before, the computation of $area(ES_1.DMBR \cap ES_2.DMBR)$ at Steps 2 and 8 ends once the overlap on one dimension is found to be zero. The second condition at Step 6 is also used to improve the algorithm performance; that is, if the current best partition meet all three heuristics, no need to check other partitions.

3.2.6 *Deletion Procedure.* The deletion algorithm for the ND-tree adopts a reinsertion strategy similar to that of the R-tree [Guttman 1984]. If the removal of an entry does not cause any underflow of its corresponding node, the entry is simply removed from the tree. Otherwise, the underflow node will be removed from the tree and all its remaining entries (subtrees) are reinserted. Like the insertion procedure, the deletion operation may propagate up to the root of the tree. The affected DMBRs of the deletion operation are adjusted accordingly. The deletion algorithm is described as follows:

ALGORITHM 3.6. : **Deletion**
Input: (1) vector α to be deleted; (2) an ND-tree with root node N for the

```
        underlying database.
Output: modified ND-tree (may be empty) with $\alpha$ deleted.
Method:
  1. starting from the root $N$, locate the leaf $CN$ containing $\alpha$;
  2. if $CN$ does not exist then
  3.    return;
  4. end if;
  5. remove the entry of $\alpha$ from $CN$;
  6. initialize set reinst_buf to hold entries to be reinserted;
  7. while $CN$ underflows and $CN$ is not the root do
  8.    put remaining entries in $CN$ into reinst_buf;
  9.    let $P$ be the parent of $CN$;
  10.   remove the entry for $CN$ from $P$;
  11.   $CN = P$;
  12. end while;
  13. if $CN$ is not the root then
  14.   adjust the DMBRs up to the root as necessary;
  15. end if;
  16. if reinst_buf is not empty then
  17.   reinsert all entries from reinst_buf into the ND-tree at their proper levels;
  18. end if;
  19. if the root of the adjusted tree has only a single child $SN$ then
  20.   let $SN$ be the new root of the tree;
  21. end if;
  22. return.
```

In the worst case, the removal of entries in Algorithm 3.6 can involve all levels of the tree. The reinserted entries that are collected in *reinst_buf* may be either leaf node entries or non-leaf node entries, which must be inserted at the appropriate levels in the tree. With some minor modifications, the insertion procedure discussed previously can directly insert a non-leaf node entry (subtree) into an ND-tree at an appropriate level. Specifically, using the same set of heuristics, Algorithm **Choose-Leaf** can be revised to locate a non-leaf node for accommodating a given non-leaf node entry. Directly inserting non-leaf entries into a tree improves the performance of the deletion procedure, comparing to the way inserting leaf node entries (vectors) all the time. The reinsertion strategy allows the ND-tree to have an opportunity to improve its performance through reconstruction.

Note that the deletion procedure is not unique. An alternative way to handle underflow is to merge sibling nodes. Unlike a CDS, the non-ordering property of an NDDS makes it possible to merge nodes that are at the same level but not necessary to be directly next to each other, which increase the chance for performance improvement. However, the tree reorganization through merging is relatively local, comparing to the reconstruction through reinsertion. The performance of the tree obtained by the merging strategy is usually not as good as the reinsertion strategy. On the other hand, the deletion procedure adopting the merging strategy is more efficient since it requires less efforts.

Similar to many indexing techniques in CDSs, choosing effective and efficient deletion strategies for an index tree in NDDSs deserves further studies.

3.2.7 *Handling NDDSs with Different Alphabets on Dimensions.* In the previous discussion, the alphabets for all dimensions in an NDDS are assumed to be the same. In this subsection, we discuss how to generalize the ND-tree to handle an NDDS with various alphabets on different dimensions.

Let $\Omega_d = A_1 \times A_2 \times ... \times A_d$ be an NDDS, where alphabets A_i's $(1 \leq i \leq d)$ may be different from each other. If alphabet sizes $|A_1| = |A_2| = ... = |A_d|$, no change is needed for the ND-tree building algorithms except that the corresponding alphabet A_i should be considered when we process the i-th component element/set of a vector/DMBR. However, if $|A_i| \neq |A_j|$ for some $i \neq j$, a new issue arises – that is, how to properly calculate the corresponding discrete geometrical measures such as the length (therefore, the area and the overlap).

A straightforward way to handle this case is to still apply the concepts defined in Section 2 directly in the ND-tree. For example, for a discrete rectangle $R = S_1 \times S_2 \times ... \times S_d$ in Ω_d, where $S_i \subseteq A_i$ $(1 \leq i \leq d)$, the length of the i-th dimension edge of R is defined as $length(R, i) = |S_i|$ and, hence, the area of R is defined as $area(R) = |S_1| * |S_2| * ... * |S_d|$, as discussed in Section 2.

However, as the alphabet sizes are different, the above edge length definition may be improper. For example, assume $|A_1| = 50$, $|A_2| = 4$ and $d = 2$. Let discrete rectangle $R' = S_1 \times S_2$ be the DMBR for an overflow node in an ND-tree, where $S_1 = \{a, b, c, d, e\} \subseteq A_1$ and $S_2 = \{1, 2, 3, 4\} = A_2$. From the above edge length definition, $length(R', 1) = 5 > length(R', 2) = 4$. Using the "maximize span" heuristic for splitting the DMBR of an overflow node (i.e., SH_2), we need to choose the 1st dimension to split R', i.e., splitting its 1st component set S_1 into two smaller component sets S_{11} and S_{12}, resulting in two smaller DMBRs $R_{11} = S_{11} \times S_2$ and $R_{12} = S_{12} \times S_2$. Notice that S_1 contains only 10% of the elements in A_1 (i.e., already having a strong pruning power), while S_2 contains 100% of the elements in A_2 (i.e., having no pruning power at all). To gain a better overall pruning power, the ND-tree would prefer to split the DMBR on the 2nd dimension. For instance, if we split S_2 into two smaller component sets $S_{21} = \{1, 2\}$ and $S_{22} = \{3, 4\}$, we would achieve a 50% pruning power (instead of none) on the 2nd dimension for resulting DMBRs $R_{21} = S_1 \times S_{21}$ and $R_{22} = S_1 \times S_{22}$. If we want to search vectors within Hamming distance 1 from a query vector $(g, 2)$, for example, we can prune the subtree with DMBR R_{22} obtained from the 2nd split, while we cannot prune any subtree with DMBR R_{11} or R_{12} obtained from the 1st split. This example shows that an element from A_1 should not receive the same weight as that for an element in A_2 for the length measure due to the various alphabet sizes for the different dimensions. Hence the above edge length definition may not allow us to make a fair comparison during constructing an index tree. In general, this direct approach has the problem to create an unsolicited bias among different dimensions when their alphabet sizes are different.

One way to solve the problem is to normalize the length measure of a discrete rectangle for each dimension with the corresponding alphabet size. In other words, we define the length of the i-th dimension edge of discrete rectangle R as $length(R, i) = |S_i|/|A_i|$. The area and overlap are then calculated based on the normalized length values. For example, the area of R can be defined as $area(R) = (|S_1|/|A_1|) * (|S_2|/|A_2|) * ... * (|S_d|/|A_d|)$. Using the normalization approach, length measures for different dimensions are comparable – leading to a fair comparison during the tree construction. For example, for the aforementioned DMBR R', since $length(R', 1) = 5/50 < length(R', 2) = 4/4$ based on the new length measure, the ND-tree will split R' on the 2nd dimension as desired. In fact, as we will see from experimental results in Section 4, the normalization approach is generally better than the direct one. The degree of improvement depends on the difference among the alphabet sizes. It is observed that the more the difference (variance of alphabet sizes), the better is the normalization approach.

A more general way to solve the problem is as follows. Assume that the distribution of data elements in each dimension for a given application is known. In other words, each element a in alphabet A_i for the i-th dimension has a probability $p(a)$ to occur as the i-th component of a vector in the application. For each dimension i,

$$\sum_{a \in A_i} p(a) = 1.$$

For a given discrete rectangle $R = S_1 \times S_2 \times ... \times S_d$ in Ω_d, where $S_i \subseteq A_i$ $(1 \leq i \leq d)$, the length of the i-th dimension edge of R can be defined as:

$$length(R, i) = \sum_{a \in S_i} p(a)$$

and the area of R is defined as $|R| = length(R, 1) * length(R, 2) * ... * length(R, d)$. The advantage of this approach is that it takes the data distributions into consideration. A component set containing elements with higher occurring probabilities should have a larger length and, therefore, contribute more to the area measure of the corresponding discrete rectangle. This approach is especially suitable for those NDDSs with large alphabets of skewed element distributions. In such a case, this new length definition should be used even for an NDDS with alphabets of the same size for all dimensions.

One disadvantage of this approach is that the distribution of data elements on each dimension may not be known in practice. In this case, the frequencies of sample data may be employed in the above formulas as the estimates of their probabilities.

In fact, the previous normalization approach is a special case of this probability-based approach, in which the distribution of data elements is uniform, i.e., assuming a probability of $1/|A_i|$ for each element in alphabet A_i $(1 \leq i \leq d)$. This simple approach can be applied when the distribution of data elements is uniform or unknown.

3.3 Range Query Processing

After an ND-tree is created for a database in an NDDS, a range query $range(\alpha_q, r_q)$ can be efficiently evaluated using the tree. The main idea is to start from the root node and prune away the nodes whose DMBRs are out of the query/search range until the leaf nodes containing the desired vectors are found. The search algorithm is given as follows:

Algorithm 3.7. : **RangeQuery**

Input: (1) range query $range(\alpha_q, r_q)$; (2) an ND-tree with root node N for the underlying database.

Output: set VS of vectors within the query range.

Method:

```
1.  let $VS = \emptyset$;
2.  push $N$ into a stack $NStack$ of nodes;
3.  while $NStack \neq \emptyset$ do
4.     let $CN = pop(NStack)$;
5.     if $CN$ is a leaf node then
6.        for each vector $v$ in $CN$ do
7.           if $dist(\alpha_q, v) \leq r_q$ then let $VS = VS \cup \{v\}$;
8.        end for;
9.     else
10.       for each entry $E$ in $CN$ do
11.          if $dist(\alpha_q, E.DMBR) \leq r_q$ then
               push each child node pointed to by $E$ into $NStack$;
12.          end if;
13.       end for;
14.    end if;
15. end while;
16. return $VS$.
```

3.4 Performance Model

To analyze the performance of the ND-tree, we conducted both empirical and theoretical studies. The results of the empirical study will be reported in Section 4. In this subsection, we present a theoretical model for estimating the performance of the ND-tree. With this model, we can predict the performance behavior of the ND-tree for different input parameter values.

Let Ω_d be a given NDDS, T be an ND-tree built for a set V of vectors in Ω_d, and Q be a similarity (range) query to search for qualified vectors from V. For simplicity, we assume that: (1) vectors in V are uniformly distributed in Ω_d (i.e., random data); (2) there is no correlation among different dimensions for vectors in V; and (3) the same alphabet A is assumed for all the dimensions of Ω_d. The input parameters for our performance estimation model are given in Table I.

Table I. Input Parameters of a Performance Estimation Model for ND-tree

Symbol	Description
$\lvert A\rvert$	the size of alphabet A
d	the number of dimensions of Ω_d
$\lvert V\rvert$	the total number of vectors indexed in ND-tree T
M_l	the maximum number of vectors allowed in a leaf node of T
M_n	the maximum number of non-leaf entries allowed in a non-leaf node of T
h	the Hamming distance used for query Q

PROPOSITION 3.2. *For given parameters $|A|$, d, $|V|$, M_l, M_n and h listed in Table I, the expected total number of disk I/O's for using ND-tree T to perform similarity query Q on a set V of vectors in space Ω_d can be estimated as:*

$$IO = 1 + \sum_{i=0}^{H-1} (n_i \cdot P_{i,h}), \tag{16}$$

where

$$n_i = \begin{cases} 2^{\lceil log_2 \lceil \frac{|V|}{M_l} \rceil \rceil} & for \quad i = 0, \\ 2^{\lceil log_2 \lceil \frac{n_{i-1}}{M_n} \rceil \rceil} & for \quad 1 \le i \le H, \end{cases}$$

$$H = \lceil log_b \; n_0 \rceil,$$
$$b = 2^{\lfloor log_2 \; M_n \rfloor},$$

$$P_{i,h} = \begin{cases} (B'_i)^{d'_i} \cdot (B''_i)^{d''_i} & for \quad h = 0, \\ \sum_{k=0}^{h} [C(d'_i, k) \cdot C(d''_i, h-k) \cdot (B'_i)^{d'_i - k} \cdot (1 - B'_i)^k \cdot (B''_i)^{d''_i + k - h} & \\ \quad \cdot (1 - B''_i)^{h-k}] + P_{i,h-1} & for \quad h \ge 1, \end{cases}$$

$$d'_i = d - d''_i,$$
$$d''_i = \lfloor (log_2 \; n_i) \; mod \; d \rfloor,$$
$$B'_i = s'_i / |A|,$$
$$s'_i = \begin{cases} s_i \;, & if \; (log_2 \; n_i)/d \; < \; 1, \\ \frac{s_H}{2^{\lfloor \frac{log_2 \; n_i}{d} \rfloor}} \;, & otherwise, \end{cases}$$
$$B''_i = s''_i / |A|,$$
$$s''_i = \frac{s_H}{2^{\lceil \frac{log_2 \; n_i}{d} \rceil}},$$
$$s_i = \sum_{j=1}^{|A|} j \cdot T_{i,j},$$

$$T_{i,j} = \begin{cases} 1/(|A|)^{w_i - 1} & for \quad j = 1, \\ C(|A|, j) \cdot [j^{w_i} - \sum_{k=1}^{j-1} (C(j,k) \cdot \frac{(|A|)^{w_i}}{C(|A|,k)} \cdot T_{i,k})]/(|A|)^{w_i} & for \quad 2 \le j \le |A|, \end{cases}$$

$$w_i = \lceil |V| / n_i \rceil.$$

Here $C(m, n)$ denotes the number of n-combinations of a set of m distinct elements.

PROOF. See Appendix. □

As we will see in Section 4, experimental results agree well with theoretical estimates obtained from this model.

3.5 Nearest Neighbor Query Processing

Nearest neighbor (NN) queries are another type of similarity queries. Given a query vector α_q, an NN query returns the vector(s) that is closest to α_q in the data set. For an NDDS, due to its discrete nature, a query vector is very likely to have more than one NN in a database. Although the focus of this paper is to use the ND-tree for range queries, the ND-tree also supports NN queries. For completeness, an NN query algorithm is described in this section.

Our NN query algorithm is based on a similar idea found in the branch-and-bound algorithm proposed for the R-tree in [Roussopoulos et al. 1995]. The two distance metrics, namely, MINDIST and MINMAXDIST, in [Roussopoulos et al. 1995] are extended from a CDS to an NDDS. MINDIST provides the minimum possible distance between a query vector α_q and a vector in a DMBR R, which has been defined as $dist(\alpha_q, R)$ by Formula (3) in Section 2. MINMAXDIST, on the other hand, gives the minimum value of all the maximum distances between a query vector α_q and vectors on each of the dimensions of a DMBR R respectively. For a d-dimensional NDDS, we define MINMAXDIST ($mmdist$) between a vector $\alpha = a_1a_2...a_d$ and a DMBR $R = S_1 \times S_2 \times ... \times S_d$ as:

$$mmdist(\alpha, R) \;=\; \min_{1 \le k \le d} \{f_m(a_k, S_k) + \sum_{i=1, i \ne k}^{d} f_M(a_i, S_i)\} \tag{17}$$

where

$$f_m(a_k, S_k) = \begin{cases} 0 & if\ a_k \in S_k \\ 1 & otherwise \end{cases} \qquad and$$

$$f_M(a_i, S_i) = \begin{cases} 0 & if\ \{a_i\} = S_i \\ 1 & otherwise. \end{cases}$$

Similar to its counterpart for a CDS in [Roussopoulos et al. 1995], MINDIST $dist(\alpha_q, R)$ between α_q and R has the following property: $\forall \alpha \in R$, $dist(\alpha_q, R) \le dist(\alpha_q, \alpha)$. We also have the property that MINMAXDIST $mmdist(\alpha_q, R)$ between α_q and R is the minimum guaranteed distance such that: $\exists \alpha \in R$, $dist(\alpha_q, \alpha) \le mmdist(\alpha_q, R)$. This property can be proven in a similar way to that in [Roussopoulos et al. 1995]. The above two properties of MINDIST and MINMAXDIST are exploited in our NN query algorithm to prune unnecessary branches in the underlying tree as follows:

ALGORITHM 3.8. : **NNQuery**

Input: (1) query vector α_q; (2) an ND-tree with root node N for the underlying database.

Output: set VS of vectors that are the NN of α_q.

Method:

```
1. let $VS = \emptyset$;
2. let $NQueue$ be a priority queue of tree nodes, which is sorted first by node
      level number in the descending order and then by $dist(\alpha_q, R)$ in the
      ascending order, where $R$ denotes the DMBR of the corresponding node in
      $NQueue$;
3. put root $N$ into $NQueue$;
4. let $NN\_dist = \infty$;
5. while $NQueue \ne \emptyset$ do
6.    dequeue the first node $CN$ in $NQueue$;
7.    let $R_{CN}$ be the DMBR of $CN$;
```

```
8.    if $dist(\alpha_q, R_{CN}) > NN\_dist$ then // current node should be pruned
9.       continue; // try next node in the queue
10.   end if;
11.   if $CN$ is a leaf node then
12.      for each vector $v$ in $CN$ do
13.         if $dist(\alpha_q, v) < NN\_dist$ then
14.            let $VS = \{v\}$;
15.            let $NN\_dist = dist(\alpha_q, v)$;
16.         else if $dist(\alpha_q, v) == NN\_dist$ then
17.            let $VS = VS \cup \{v\}$;
18.         end if;
19.      end for;
20.   else
21.      for each entry $E$ in $CN$ do
22.         if $dist(\alpha_q, E.DMBR) \leq NN\_dist$ then put the corresponding child
               into $NQueue$;
23.         if $mmdist(\alpha_q, E.DMBR) < NN\_dist$ then
24.            let $VS = \emptyset$;
25.            let $NN\_dist = mmdist(\alpha_q, E.DMBR)$;
26.         end if;
27.      end for;
28.   end if;
29. end while;
30. return $VS$.
```

Algorithm 3.8 utilizes a priority queue to determine the access order of the tree nodes (Step 2). A node at a lower tree level is given a higher priority. Among those nodes at the same tree level, the one with a smaller MINDIST $dist(\alpha_q, DMBR)$ is given a higher priority. The search starts from the root node; i.e., the priority queue has the root node first (Step 3). For each node dequeued from the priority queue, its children are added into the queue (Steps 21 and 22). Based on the property of MINDIST, any node N with $dist(\alpha_q, DMBR\_of\_N)$ greater than the current NN distance NN_dist is pruned (Steps 8 and 22). Based on the property of MINMAXDIST, NN_dist is updated if a smaller $mmdist(\alpha_q, DMBR)$ is found (Step 23). NN_dist is also updated if a smaller distance between query vector α_q and an indexed vector in a leaf node is found (Steps 13 - 15). The reason why a higher priority is given to a node at a lower tree level is to use the indexed vectors in a leaf node to reduce NN_dist as early as possible. The resulting NN vector(s) for α_q is found at Steps 14 and 17. The algorithm terminates when the priority queue is empty.

Algorithm 3.8 represents one approach for NN queries in an NDDS. Our studies on NN query processing in NDDSs are ongoing. The related issues that are under investigation are alternative strategies for NN queries, k-nearest neighbor (k-NN) query processing, approximate k-NN query processing, and alternative distance measures for NDDSs.

4. EXPERIMENTAL RESULTS

To determine effective heuristics for building an ND-tree and evaluate its performance for various NDDSs, we conducted extensive experiments using both real data (bacteria genome sequences extracted from the GenBank of National Center for Biotechnology Information) and synthetic data (generated with the uniform distribution). The experimental programs were implemented in C++ on a PC with a Pentium 4 (2.26 GHz) CPU and 256 MB memory under operating system (OS) Linux Kernel 2.4.20. To save space for an ND-tree, we employed a bitmap compression scheme to compress DMBRs in non-leaf nodes. No separate buffering strategy was employed in the experimental programs besides the buffering mechanism provided by the underlying OS. Each node of an index tree is saved in one disk block/page.

4.1 Evaluation of Strategies for Tree Construction Algorithms

One set of experiments were conducted to determine effective strategies for building an efficient ND-tree. Typical experimental results are reported in Tables II $\sim$ V. The performance data shown in the tables is based on the average number (io) of disk I/O's (blocks) for executing 100 random test queries. r_q denotes the Hamming distance range for the test queries. $key\#$ indicates the number of database vectors indexed by the ND-tree. The maximum number M of entries allowed for each node in a tree was set to 50. The minimum number m of entries required for each node was set to 15 (i.e., the minimum space utilization was set to 30%). A 25-dimensional genome sequence data set was used in Tables II $\sim$ IV. 10-dimensional random data sets were used in Table V.

Table II shows the performance comparison among the following three versions

Table II. Performance effect of heuristics for choosing insertion leaf node

$key\#$	$r_q = 1$			$r_q = 2$			$r_q = 3$		
	io V_a	io V_b	io V_c	io V_a	io V_b	io V_c	io V_a	io V_b	io V_c
13927	14	18	22	48	57	68	115	129	148
29957	17	32	52	67	108	160	183	254	342
45088	18	47	80	75	161	241	215	383	515
56963	21	54	103	86	191	308	252	458	652
59961	21	56	108	87	198	323	258	475	685

of algorithms for choosing a leaf node for insertion, based on different combinations of heuristics in the given order to break ties:

—Version V_a: using IH_1, IH_2, IH_3;

—Version V_b: using IH_2, IH_3;

—Version V_c: using IH_2

From the table, we can see that all the heuristics are effective. In particular, heuristic IH_1 can significantly improve query performance (see the performance difference between V_a (with IH_1) and V_b (without IH_1)). In other words, the increased overlap in an ND-tree may greatly degrade the performance. Hence we should keep the overlap in an ND-tree as small as possible. It is also noted that the larger the database size, the more improved is the query performance.

Table III shows the performance comparison between Algorithm 3.3 (permutation

Table III. Performance comparison between permutation and merge-and-sort approaches

key#	$r_q = 1$		$r_q = 2$		$r_q = 3$	
	io permu.	*io* m&s	*io* permu.	*io* m&s	*io* permu.	*io* m&s
29957	16	16	63	63	171	172
45088	18	18	73	73	209	208
56963	20	21	82	83	240	242
59961	21	21	84	85	247	250
68717	21	22	88	89	264	266
77341	21	22	90	90	271	274

approach) and Algorithm 3.4 (merge-and-sort approach) to choose candidate partitions for splitting an overflow node. From the table, we can see that the performance of the permutation approach is slightly better than that of the merge-and-sort approach since the former takes more partitions into consideration. However, the performance of the latter is not much inferior and, hence, can be used for an NDDS with a large alphabet size when the computational overhead of the former is large.

Table IV shows the performance comparison among the following five versions

Table IV. Performance effect of heuristics for choosing best partition for $r_q = 3$

key#	*io* V_1	*io* V_2	*io* V_3	*io* V_4	*io* V_5
13927	181	116	119	119	105
29957	315	194	185	182	171
45088	401	243	224	217	209
56963	461	276	254	245	240
59961	477	288	260	255	247

of algorithms for choosing the best partition for Algorithm **SplitNode** based on different combinations of heuristics with their correspondent ordering to break ties:

—Version V_1: using SH_1;

—Version V_2: using SH_1, SH_4;

—Version V_3: using SH_1, SH_2;

—Version V_4: using SH_1, SH_2, SH_3;

—Version V_5: using SH_1, SH_2, SH_3, SH_4.

Since the overlap in an ND-tree may greatly degrade the performance, as seen from the previous experiments, heuristic SH_1 ("minimize overlap") is applied in all the versions. Due to the space limitation, only the results for $r_q = 3$ are reported here. From the table we can see that heuristics $SH_2 \sim SH_4$ are all effective in improving performance. Although version V_5 is most effective, it may not be feasible in practice since heuristic SH_4 has a lot of overhead as we mentioned in Section 3.2.5. Hence the best practical version is V_4, which is not only very effective but also efficient.

Table V shows the performance comparison between the direct and normalization

Table V. Performance comparison between direct and normalization approaches for $r_q = 3$

key#	*var* = 1.1		*var* = 36.7		*var* = 90.0	
	io direct	*io* norm	*io* direct	*io* norm	*io* direct	*io* norm
20000	144	143	211	135	265	130
40000	205	203	311	201	423	196
60000	291	296	416	284	606	272
80000	315	314	468	309	726	301
100000	357	355	516	346	890	342

approaches for NDDSs with different alphabets on each dimension. Experimental results based on three 10-dimensional data sets with different variances of alphabet sizes are presented. Alphabet sizes used range from 2 to 20. From Table V, we can see that when the difference (variance) among dimensions is small, the normalization approach performs similarly to the direct approach. As the difference increases, the normalization approach becomes increasingly better than the direct approach. From the experimental results, we conclude that, to index NDDSs with various alphabet sizes on different dimensions, the normalized values of geometrical measures, such as length, area and overlap, are better to be used in the construction algorithms of the ND-tree.

4.2 Performance Analysis of the ND-tree

We also conducted another set of experiments to evaluate the overall performance of the ND-tree for data sets in different NDDSs. Both genomic data and synthetic data were used in the experiments. The effects of the dimensionality and alphabet size of an NDDS on the performance of an ND-tree were examined. As before, the query performance was measured based on the average number of I/Os for executing 100 random test queries for each case. We compared the performance of the ND-tree with that of the linear scan as well as several dynamic metric trees. The disk block size was assumed to be 4 KB for all the access methods, which, in turn, determines the maximum number of entries allowed in a node (block) when the considered access method is an index tree. The minimum space utilization of a node for the ND-tree was set to 30% as before.

4.2.1 *Performance Comparison with Linear Scan.* To perform range queries on a database in an NDDS, a straightforward method is to employ the linear scan. We compared the performance of our ND-tree with that of the linear scan. To make a fair comparison, we assume that the linear scan is well-tuned with data being placed on disk sequentially without fragments, which boosts its performance by a factor of 10. In other words, the performance of the linear scan for executing a query is assumed to be only 10% of the number of disk I/O's for scanning all the disk blocks of the data file. This benchmark was also used in [Chakrabarti and Mehrotra 1999; Weber et al. 1998]. We will refer to this benchmark as the 10% linear scan in the following discussion.

Figure 7 shows the performance comparison of the two search methods for the

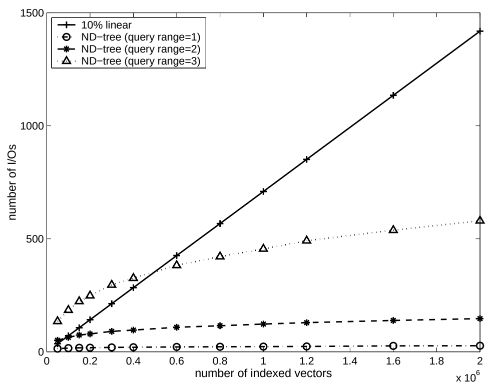


Fig. 7. Performance comparison between ND-tree and 10% linear scan for genomic data

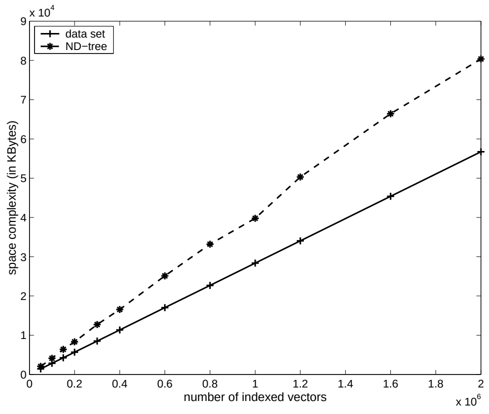


Fig. 8. Space complexity of ND-tree

bacteria genomic data set in an NDDS with 25 dimensions. From the figure, we can see that the performance of the ND-tree is usually better than that of the 10% linear scan. For a query with a large range on a small database, the ND-tree may not outperform the 10% linear scan, which is normal since an indexing method is usually not superior to the linear scan for a small database, and the larger the selectivity a query has, the less the benefit an index tree could gain. As the database size becomes larger, the ND-tree is increasingly more efficient than the 10% linear scan as shown in the figure. In fact, the ND-tree scales well with the size of the database. For example, the ND-tree, on the average, is about 2.4 times more efficient than the 10% linear scan for queries with range 3 on a genomic data set of 2M (million) vectors. Figure 8 shows the space complexity of the ND-tree. From the figure, we can see that the size of the tree is almost always about 1.46 times more than the size of the data set, which indicates an average disk space utilization of 68.5%.

We also observed the (average) execution time, which reflects not only the I/O costs but also the CPU and buffering effects, for the two search methods. Although the execution time measure depends heavily on a particular underlying environment, it shows the actual performance behaviors of the access methods in the given environment and may help to identify some important factors. We notice that, for a given database size, the execution time of the linear scan increases gradually, rather than staying unchanged (like its I/O cost), as the query range increases. For example, for a genomic data set of 1M vectors, the execution times of the linear scan for queries with ranges 1, 2 and 3 are 0.140 sec., 0.163 sec. and 0.186 sec, respectively. This is because the CPU overhead increases when larger search ranges are evaluated based on the Hamming distance. The execution times of the ND-tree for queries with ranges 1, 2 and 3 on the same data set are 0.005 sec., 0.024 sec. and 0.080 sec., respectively. The increase of the execution time of the ND-tree for larger query ranges is mainly caused by the increase of its I/O cost although its CPU overhead is also increasing. When the number of disk blocks accessed by the ND-tree becomes relatively large compared with the size of the underlying data set, the linear scan can surpass the ND-tree in performance. For example, for the above data set of 1M vectors, the execution time of the linear scan becomes better than that of the ND-tree (0.258 sec. vs 0.337 sec.) when the query range reaches 5. In general, the larger the data set, the larger the query range at which the number of disk blocks accessed by the ND-tree becomes relatively large.

4.2.2 *Performance Comparison with the M-tree and the Slim-trees.* As mentioned in Section 1, a dynamic metric tree, such as the popular M-tree [Ciaccia et al. 1997], can also be used to perform range queries in an NDDS. We compared the performance of our ND-tree with that of the M-tree as well as one of its recent variants – the Slim-tree [Traina et al. 2002]. The generalized hyperplane version (mM_RAD_2) of the M-tree and the min-max version of the Slim-tree, which were reported to have the best performance in [Ciaccia et al. 1997] and [Traina et al. 2002] respectively, were used in our experiments. Besides, the Slim-down algorithm [Traina et al. 2002], which post-processes the Slim-tree to further enhance its query performance, was also included in our comparisons.

Figures 9 and 10 show the comparisons of performance, in terms of disk I/Os, among the ND-tree, the M-tree, the Slim-tree I (without slimming-down) and the

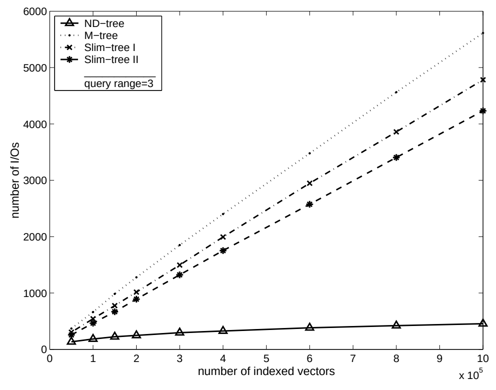


Fig. 9. Performance comparison between ND-tree and metric trees for genomic data (disk I/Os vs. various DB sizes)

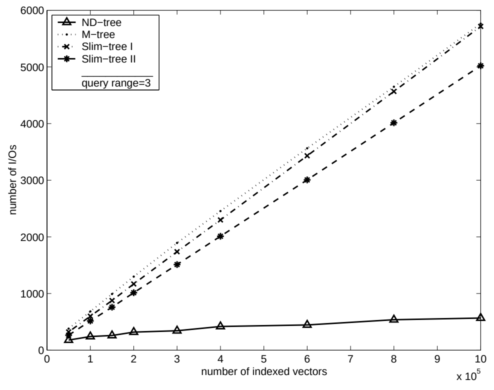


Fig. 10. Performance comparison between ND-tree and metric trees for binary data (disk I/Os vs. various DB sizes)

Slim-tree II (with slimming-down) for range queries (range = 3) with various data set sizes on a 25-dimensional genomic data set as well as a 40-dimensional binary data set (i.e., alphabet = {0, 1}). From the figures, we can see that the ND-tree significantly outperforms the M-tree and the Slim-trees. For example, the ND-tree, on the average, is about 12.3, 10.5 and 9.3 times faster than the M-tree, the Slim-tree I and the Slim-tree II, respectively, in terms of disk I/Os on a genomic data set of 1M vectors. Furthermore, the larger the data set, the more is the performance improvement achieved by the ND-tree.

To examine the CPU and buffering effects on query performance of the index trees, the average execution time of the same set of 100 random test queries for each data set size was also measured. The results are shown in Figures 11 and 12, which demonstrate again that the ND-tree significantly outperforms the M-tree and

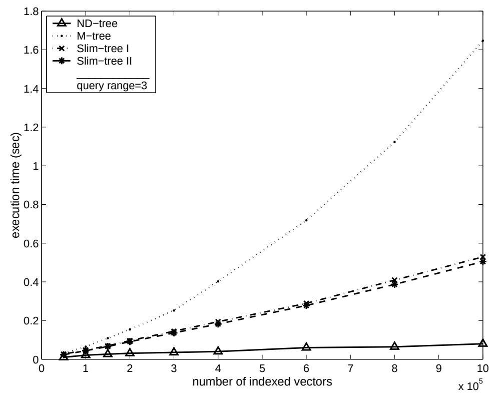


Fig. 11. Performance comparison between ND-tree and metric trees for genomic data (execution time vs. various DB sizes)

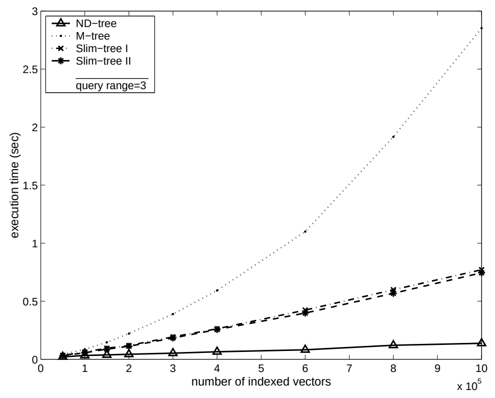


Fig. 12. Performance comparison between ND-tree and metric trees for binary data (execution time vs. various DB sizes)

the Slim-trees. For example, the ND-tree, on the average, is about 20.6, 6.6 and 6.3 times faster than the M-tree, the Slim-tree I and the Slim-tree II, respectively, in terms of execution time on a genomic data set of 1M vectors. However, it is also observed that the relative difference of execution time between a Slim-tree and an ND-tree is smaller, comparing to their relative difference of I/O counts. This could be because: (1) the Slim-trees usually have a better space utilization (e.g., for the genomic data set of 1M vectors, the space utilization of the Slim-tree I is about 83.3%, comparing to 72.0% for the ND-tree), which leads to a more effective use of the underlying OS buffering mechanism; (2) the Slim-trees store some pre-computed distances to allow some subtrees to be pruned without much CPU cost. On the other hand, although the M-tree also makes use of pre-computed distances to save some CPU cost, its space utilization is low (e.g., about 21.7% for the genomic data set of 1M vectors) and progressively deteriorates with the increase of the database size, leading to a large and fast-growing file for the tree on the disk (comparing to other trees). Since reading (random) blocks from a large data file is slower than reading blocks from a small data file due to larger disk seek time and a smaller success rate of pre-fetching hits, the execution time of an M-tree deteriorates rapidly as the database size increases.

Figures 13 and 14 show the performance comparisons among the ND-tree, the M-tree and the Slim-trees for queries with various ranges on the same 25-dimensional genomic data set of 1M vectors. From the figures, we can see that the ND-tree always outperforms the M-tree, and it outperforms the Slim-trees for query ranges within a certain Hamming distance (e.g., ≤ 8). As the query range gets larger, the performance of every index tree quickly degrades to a saturated point where the entire tree has to be scanned when processing such a query. In that case, the Slim-trees require a slightly less number of I/Os than the ND-tree since the Slim-trees usually have a better space utilization. However, in such a case, the pre-computed distances stored in the Slim-trees usually cannot help in pruning subtrees, i.e., the distances between the query vector and the centers of subtrees almost always have to be calculated during the search. Furthermore, in such a case, checking useless

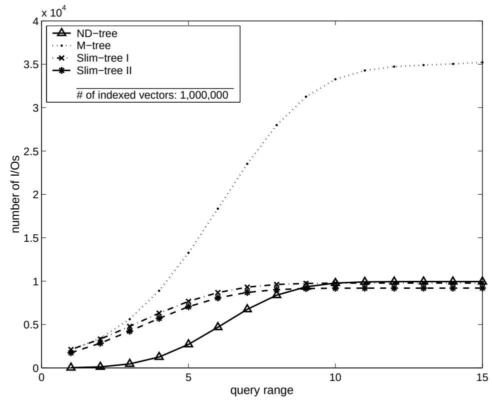


Fig. 13. Performance comparison between ND-tree and metric trees for genomic data (disk I/Os, various query ranges)

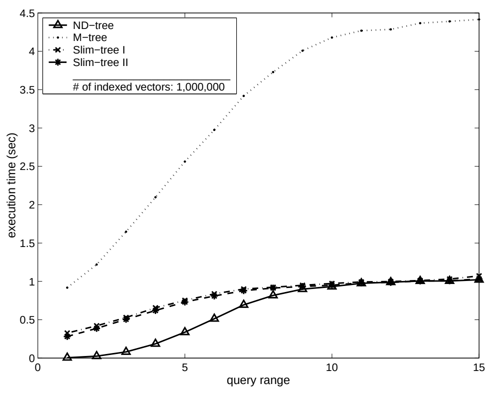


Fig. 14. Performance comparison between ND-tree and metric trees for genomic data (execution time, various query ranges)

pre-computed distances incurs extra CPU overhead. As a result of balancing CPU and I/O costs, the execution time of a Slim-tree is comparable to that of the ND-tree after the saturated point. From Figure 14, we can see that Figure 11 actually demonstrates one of the optimal cases (i.e., queries of range 3) favoring the ND-tree over the Slim-trees.

As pointed out earlier, the ND-tree is more efficient than other trees primarily because it makes use of more geometric information of an NDDS for optimization to gain more pruning power for searches. Although the M-tree and the Slim-trees demonstrated a less competitive performance for range queries in NDDSs, they were designed for a more general purpose and can be applied to more applications.

4.2.3 *Scalability of the ND-tree for Dimensionality and Alphabet Size.* To analyze the scalability of the ND-tree for the dimensionality and alphabet size, we conducted experiments using synthetic data sets with various parameter values for an NDDS. Figures 15 and 16 show experimental results for varying dimensionalities and alphabet sizes. From the figures, we see that the ND-tree scales well with both the dimensionality and the alphabet size. For a fixed alphabet size and data set size, increasing the number of dimensions for an NDDS does not significantly affect the performance of the ND-tree for range queries. This is due to the effectiveness of the overlap-reducing heuristics used in our tree construction. However, the performance of the 10% linear scan degrades significantly since a larger dimensionality implies larger vectors and hence more disk blocks. For a fixed dimensionality and data set size, increasing the alphabet size for an NDDS has some effect on the performance of the ND-tree. As mentioned before, our implementation of the ND-tree employed a bitmap compression scheme to compress non-leaf nodes, which makes the fan-out (number of entries) of each non-leaf node decreasing as the alphabet size increases. Decreasing the fan-out of a non-leaf node causes the performance of the ND-tree to degrade. On the other hand, since a larger alphabet size provides more choices for splitting an overflow node, a better tree can be constructed. This positive impact and the previous negative impact from increasing the alphabet size

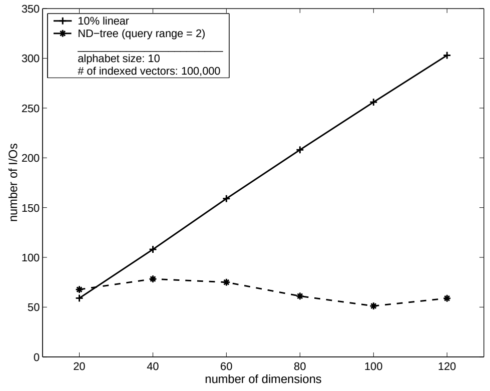


Fig. 15. Scalability of ND-tree on dimensionality

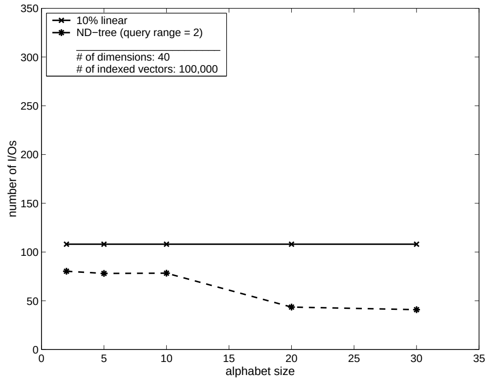


Fig. 16. Scalability of ND-tree on alphabet size

are balanced initially. But the former dominates the performance of the tree as the alphabet size becomes large, as shown in Figure 16. Note that our C++ implementation of the ND-tree assumes that the input database is not compressed and, hence, nor are its vectors indexed in the leaf nodes of an ND-tree, which makes the number of vectors in a leaf node not affected by the change of the alphabet size. Under this assumption, the performance of the 10% linear scan stay the same as the alphabet size increases.

4.3 Verification of the Performance Model

We also verified our theoretical performance estimation model using the experimental data. The experimental setup was assumed to be the same as in Section 4.2. Figures 17 and 18 show the comparisons between the theoretical and experimental

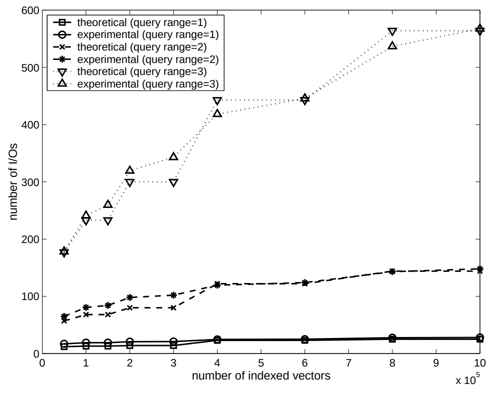


Fig. 17. Comparison between theoretical and experimental results for binary data

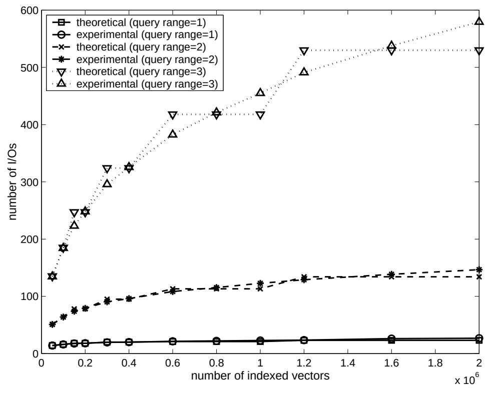


Fig. 18. Comparison between theoretical and experimental results for genomic data

results for the 40-dimensional synthetic binary data set and the 25-dimensional real genomic data set, respectively. Figures 19 and 20 show the comparisons between the theoretical and experimental results for varying dimensionalities and alphabet sizes, respectively, on synthetic data sets. From the figures, we can see that the predicted numbers of disk I/O's from the theoretical model are very close to those from the experimental results, which corroborates the correctness of the model. Note that although the genomic data set does not strictly follow the uniformly distribution, the theoretical model can still predict the query performance of the ND-tree well as shown in Figure 18.

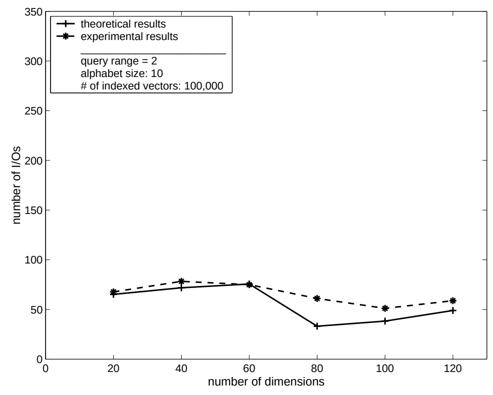


Fig. 19. Comparison between theoretical and experimental results for varying dimensionalities

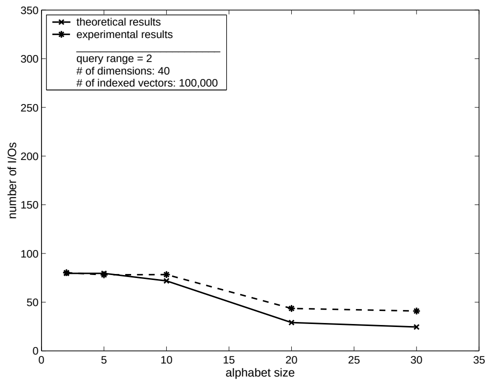


Fig. 20. Comparison between theoretical and experimental results for varying alphabet sizes

5. CONCLUSIONS

There is an increasing demand for supporting efficient similarity searches in NDDSs from applications such as genome sequence databases. Unfortunately, existing indexing methods either cannot be directly applied to an NDDS (e.g., the R-tree, the K-D-B-tree and the Hybrid-tree) due to lack of essential geometric concepts/properties or have suboptimal performance (e.g., the metric trees) due to their generic nature. We have proposed a new dynamic indexing method, i.e., the ND-tree, to address these challenges.

The ND-tree is inspired by several popular multidimensional indexing methods including the R*-tree, the X-tree and the Hybrid tree. However, it is based on some essential geometric concepts/properties that we extend from a CDS to an NDDS. Development of the ND-tree takes into consideration some characteristics, such as limited alphabet sizes and data distributions, of an NDDS. As a result, strategies such as the permutation and the merge-and-sort approaches to generating candidate partitions for an overflow node, the multiple heuristics to break frequently-occurring ties, the efficient implementation of some heuristics, the normalization and probability-based approaches to handling varying alphabet sizes for different dimensions, and the space compression scheme for tree nodes are incorporated into the ND-tree construction. In particular, it has been shown that both the permutation and the merge-and-sort approaches can guarantee generation of an overlap-free partition if there exists one. Our complexity analysis also shows that the permutation approach is more suitable for NDDSs with small alphabets, while the merge-and-sort is more suitable for NDDSs with large alphabets.

A set of heuristics that are effective for indexing in an NDDS are identified and integrated into the tree construction algorithms. This is done after carefully evaluating the heuristics in existing multidimensional indexing methods via extensive experiments. For example, minimizing overlap (enlargement) is found to be the most effective heuristic to achieve an efficient ND-tree, which is similar to the case for index trees in a CDS. On the other hand, minimizing area is found to be an expensive heuristic for an NDDS although it is also effective.

Our extensive experiments on synthetic and genome sequence data have demonstrated that:

—The ND-tree outperforms the linear scan, the M-tree and the Slim-trees for executing queries with reasonable ranges in an NDDS. In fact, the larger the data set, the more is the improvement in performance.

—The ND-tree scales well with the database size, the alphabet size as well as the dimension for an NDDS.

We have also developed a performance estimation model to predict the performance behavior of the ND-tree for random data. It can be used to estimate the performance of an ND-tree for various input parameters. Experiments on synthetic and real data sets have demonstrated that this model is quite accurate. In addition, for completeness, we have presented a deletion algorithm and a nearest-neighbor query processing algorithm for the ND-tree.

In summary, our study provides a foundation for developing indexing techniques for large databases with non-ordered discrete data. However, our work is just the beginning of the research to support efficient similarity searches in NDDSs. In future work, we plan to further explore more heuristics/strategies for building and maintaining an index tree in NDDSs and further study techniques/issues for supporting efficient NN searches in NDDSs. It is worth noting that, after this work on the ND-tree, we recently proposed another NDDS indexing technique, called the NSP-tree [Qian et al. 2006]. In contrast to the ND-tree, which is a data-partitioning-based indexing technique, the NSP-tree employs a space-partitioning-based scheme. The work on the NSP-tree is an effort to explore the possibility of adopting a guaranteed overlap-free indexing structure to improve search efficiency in NDDSs. It has been shown that the NSP-tree has a better search performance for skewed data sets, while the ND-tree is advantageous in guaranteeing the minimum space utilization. We also plan to extend our indexing techniques to support queries in hybrid data spaces that include both continuous and non-ordered discrete dimensions.

APPENDIX

In this appendix, we sketch the derivation of the performance estimation model given in Proposition 3.2. In the following discussion, if a node N of an ND-tree is at level j $(1 \leq j \leq H+1)$, we also say N is at layer $(H-j+1)$, where H is the height of the tree. In other words, the leaf nodes are at layer 0, the parent nodes of the leaves are at layer 1, ..., and the root node is at layer H. We still use the symbols defined in Table I.

Based on the uniform distribution assumption and the tree building heuristics, an ND-tree grows in the following way. Before a leaf node splits due to overflowing, all leaf nodes contain about the same number of indexed vectors. They are getting filled up at about the same pace. During this period (before any leaf node overflows), we say that the leaf nodes are in the accumulating stage. When one leaf node overflows and splits (into two), the other leaf nodes will also start to overflow and split one by one in quick succession until all leaf nodes have split. We say the leaf nodes are in the splitting transition in this period. Comparing to the accumulating stage, the splitting transition is relatively short. Each new leaf node is about half full right after the splitting transition. The ND-tree grows by repeating the above process.

Let n be the number of leaf nodes in an ND-tree. From the above observation, we have:

$$2^{\lceil log_2 \lceil \frac{|V|}{M_l} \rceil \rceil - 1} < n \leq 2^{\lceil log_2 \lceil \frac{|V|}{M_l} \rceil \rceil} .$$

When n equals the right end, the leaf nodes of the ND-tree are in the accumulating stage. Otherwise, the tree is in the splitting transition. Since the splitting transition is relatively short, we can consider the accumulating stage only for our performance estimation model. Hence the following formula is used to estimate the number of leaf nodes in the ND-tree:

$$n_0 = 2^{\lceil log_2 \lceil \frac{|V|}{M_l} \rceil \rceil} . \tag{18}$$

A similar analysis can be applied to the parent nodes of the leaves (i.e., nodes at layer 1). In other words, when the parent nodes are in the accumulating stage, their number can be estimated as:

$$n_1 = 2^{\lceil log_2 \lceil \frac{n_0}{M_n} \rceil \rceil} . \tag{19}$$

Since the splitting transition is relatively short, Formula (19) can be used to estimate the number of non-leaf nodes of the ND-tree at layer 1. In general, the number of non-leaf nodes of the ND-tree at layer i can be estimated as follows:

$$n_i = 2^{\lceil log_2 \lceil \frac{n_{i-1}}{M_n} \rceil \rceil} , \qquad (1 \leq i \leq H) , \tag{20}$$

where $H = \lceil log_b \ n_0 \rceil$ and $b = 2^{\lfloor log_2 M_n \rfloor}$.

From (18) $\sim$ (20), the expected number of vectors indexed in a node (subtree) at layer i of the ND-tree is:

$$w_i = \lceil |V|/n_i \rceil , \qquad (0 \leq i \leq H) . \tag{21}$$

During the growth of an ND-tree, some dimensions are chosen to be split and there could also exist other dimensions that have never been split yet. We call such dimensions as unsplit dimensions and the edges on those unsplit dimensions of a DMBR are called unsplit edges. Under the assumptions of the uniform distribution and the independence among dimensions, the expected lengths of unsplit edges of the DMBR of a node of the ND-tree are about the same. Note that we still consider the dominant accumulating stage. The probability for the length of an unsplit edge of the DMBR of a node at layer i to be 1 is:

$$T_{i,1} = |A|/(|A|)^{w_i} = 1/(|A|)^{w_i - 1} . \tag{22}$$

Based on Equation (22), the probability for the length of an unsplit edge to be j $(2 \leq j \leq |A|)$ can be computed as:

$$T_{i,j} = C(|A|, j) \cdot [j^{w_i} - \sum_{k=1}^{j-1} (C(j,k) \cdot \frac{(|A|)^{w_i}}{C(|A|,k)} \cdot T_k)]/(|A|)^{w_i} \ , \quad (2 \leq j \leq |A|) \ . \tag{23}$$

Hence the expected length of an unsplit edge of the DMBR of a node at layer i can be computed as:

$$s_i = \sum_{j=1}^{|A|} j \cdot T_{i,j} \ . \tag{24}$$

In particular, $s_{_H}$ is the expected length of each edge (since no split) of the DMBR of the root node of the ND-tree.

Based on the uniform distribution assumption and heuristics SH_2 ("maximize span") and SH_3 ("center split"), we can assume that a sequence of n node splits will split on the 1st dimension, the 2nd dimension, ..., the last (d-th) dimension, then back to the 1st dimension, ..., until n splits are done. Each split will divide the component set of a DMBR on the relevant dimension into two equal-sized component subsets.

To obtain n_i $(0 \leq i \leq H)$ nodes at layer i of the ND-tree (in the accumulating stage), the expected number of splits needed is $log_2 \ n_i$ (starting from splitting the root node). Let

$$d_i'' = \lfloor (log_2 \ n_i) \ mod \ d \rfloor \ , \tag{25}$$

$$d_i' = d - d_i'' \ . \tag{26}$$

The DMBR of a node N at layer i has the following expected edge length:

$$s_i'' = \frac{s_{_H}}{2^{\lceil \frac{log_2 \ n_i}{d} \rceil}} \tag{27}$$

on d_i'' dimensions. It has the following expected edge length:

$$s_i' = \begin{cases} s_i \ , & if \ (log_2 \ n_i)/d \ < \ 1 \ , \\ \frac{s_{_H}}{2^{\lfloor \frac{log_2 \ n_i}{d} \rfloor}} \ , & otherwise \end{cases} \tag{28}$$

on d_i' dimensions. Note that the first case in (28) represents the situation when the d_i' dimensions of the DMBR have never been split yet so that Equation (24) can be applied.

For node N, the probability for a component of a query vector α_q to be covered by the corresponding component set of the DMBR of N is given as:

$$B_i' = s_i'/|A| \ , \tag{29}$$

or

$$B_i'' = s_i''/|A| \tag{30}$$

depending on the relevant dimension. Hence the probability for a node N at layer i to be accessed by range query $range(\alpha_q, 0)$ for Hamming distance 0 can be calculated as:

$$P_{i,0} = (B_i')^{d_i'} \cdot (B_i'')^{d_i''} \ . \tag{31}$$

Using (31), the probability for node N to be accessed by range query $range(\alpha_q, h)$ for Hamming distance h can be evaluated recursively as:

$$P_{i,h} = \sum_{k=0}^{h} [C(d'_i, k) \cdot C(d''_i, h-k) \cdot (B'_i)^{d'_i - k} \cdot (1 - B'_i)^k \cdot (B''_i)^{d''_i + k - h} \cdot (1 - B''_i)^{h-k}] + P_{i,h-1} \quad . \tag{32}$$

Therefore, the expected total number of disk I/O's for using the ND-tree to perform range query $range(\alpha_q, h)$ for Hamming distance h can be estimated as:

$$IO = 1 + \sum_{i=0}^{H-1} (n_i * P_{i,h}) \quad . \tag{33}$$

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable comments and constructive suggestions for improving the paper. The preliminary work of this paper was presented at the 29th International Conference on Very Large Data Bases (VLDB'03), Berlin, Germany, September 9-12, 2003 [Qian et al. 2003].

REFERENCES

Bayer, R. and Unterauer, K. 1977. Prefix B-trees. *ACM Trans. on Database Syst. 2,* 1, 11–26.

Beckmann, N., Kriegel, H., Schneider, R., and Seeger, B. 1990. The R*-tree: an efficient and robust access method for points and rectangles. In *Proc. of SIGMOD.* 322–331.

Berchtold, S., Keim, D. A., and Kriegel, H.-P. 1996. The X-tree: an index structure for high-dimensional data. In *Proc. of VLDB.* 28–39.

Bozkaya, T. and Ozsoyoglu, M. 1997. Distance-based indexing for high-dimensional metric spaces. In *Proc. of SIGMOD.* 357–368.

Brin, S. 1995. Near neighbor search in large metric spaces. In *Proc. of VLDB.* 574–584.

Chakrabarti, K. and Mehrotra, S. 1999. The hybrid tree: an index structure for high dimensional feature spaces. In *Proc. of ICDE.* 440–447.

Chavez, E., Navarro, G., Baez-Yates, R., and Marroquin, J. L. 2001. Searching in metric spaces. *ACM Computing Surveys 33,* 3, 273–321.

Chiueh, T. 1994. Content-based image indexing. In *Proc. of VLDB.* 582–593.

Ciaccia, P., Patella, M., and Zezula, P. 1997. M-tree: an efficient access method for similarity search in metric spaces. In *Proc. of VLDB.* 426–435.

Clement, J., Flajolet, P., and Vallee, B. 2001. Dynamic sources in information theory: a general analysis of trie structures. *Algorithm 29,* 1/2, 307–369.

Dohnal, V., Gennaro, C., Savino, P., and Zezula, P. 2003. D-index: Distance searching index for metric data sets. *Multimedia Tools Appl 21,* 1, 9–33.

Ferragina, P. and Grossi, R. 1999. The String B-tree: a new data structure for string search in external memory and its applications. *J. ACM 46,* 2, 236–280.

Guttman, A. 1984. R-trees: a dynamic index structure for spatial searching. In *Proc. of SIGMOD.* 47–57.

Henrich, A. 1998. The LSD^h-tree: an access structure for feature vectors. In *Proc. of ICDE.* 362–369.

Katayama, N. and Satoh, S. 1997. The SR-tree: an index structure for high-dimensional nearest neighbor queries. In *Proc. of SIGMOD.* 369–380.

Kent, W. J. 2002. BLAT — the BLAST-like aligment tool. *Genome Research 12*, 656–664.

Knuth, D. E. 1973. *The Art of Computer Programming, Vol. 3.* Addison-Wesley, Reading, Mass.

LI, J. 2001. Efficient similarity search based on data distribution properties in high dimension. In *Ph.D. Dissertation, Michigan State University.*

QIAN, G., ZHU, Q., XUE, Q., AND PRAMANIK, S. 2003. The ND-tree: a dynamic indexing technique for multidimensional non-ordered discrete data spaces. In *Proc. of VLDB*. 620–631.

QIAN, G., ZHU, Q., XUE, Q., AND PRAMANIK, S. 2006. A space-partitioning-based indexing method for multidimensional non-ordered discrete data spaces. *ACM Trans. on Information Syst. 23*, (to appear).

ROBINSON, J. T. 1981. The K-D-B-tree: a search structure for large multidimensional dynamic indexes. In *Proc. of SIGMOD*. 10–18.

ROUSSOPOULOS, N., KELLEY, S., AND VINCENT, F. 1995. Nearest neighbor queries. In *Proc. of SIGMOD*. 71–79.

SKOPAL, T., POKORNY, J., AND SNASEL, V. 2004. PM-tree: pivoting metric tree for similarity search in multimedia databases. In *Proc. of Dateso 2004 Annual Int'l Workshop on DAtabases, TExts, Specifications and Objects (DATESO'04)*. 27–37.

TRAINA, C., TRAINA, A., FALOUTSOS, C., AND SEEGER, B. 2002. Fast indexing and visualization of metric data sets using slim-trees. *IEEE Trans. on Knowl. and Data Eng. 14,* 2, 244–260.

UHLMANN, J. K. 1991. Satisfying general proximity/similarity queries with metric trees. *Inf. Proc. Lett. 40,* 4, 175–179.

WEBER, R., SCHEK, H.-J., AND BLOTT, S. 1998. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proc. of VLDB*. 357–367.

WHITE, D. AND JAIN, R. 1996. Similarity indexing with the SS-tree. In *Proc. of ICDE*. 516–523.

XUE, Q., QIAN, G., COLE, J. R., AND PRAMANIK, S. 2004. Investigation on approximate q-gram matching in genome sequence databases. In *Tech. Report, Michigan State Univ.*

ZHOU, X., WANG, G., YU, J. X., AND YU, G. 2003. M^+-tree: a new dynamical multidimensional index for metric spaces. In *Proc. of the 14th Australasian Database Conf.* 161–168.

Received December 2004; revised August 2005; accepted November 2005