An Assessment of a Metric Space Database Index to Support Sequence Homology

Rui Mao, Weijia Xu, Neha Singh, Daniel P. Miranker
{rmao, xwj, neha, miranker}@cs.utexas.edu
Department of Computer Sciences, the University of Texas at Austin

Abstract

Global alignment of sequences based on simple-edit distance forms a metric-space. Hierarchical metric-space clustering methods have been commonly used to organize proteomes into taxonomies. Consequently, it is often anticipated that hierarchical clustering can be leveraged as a basis for scalable database index structures capable of managing the hyper-exponential growth of sequence data. M-tree is one such data structure specialized for the management of large data sets on disk.

We explore the application of M-trees to the storage and retrieval of peptide sequence data. Exploiting a technique first suggested by Myers, we organize the database as records of fixed length substrings. Empirical results are promising. However, metric-space indexes are subject to "the curse of dimensionality" and the ultimate performance of an index is sensitive to the quality of the initial construction of the index. We introduce new hierarchical bulk-load algorithm that alternates between top-down and bottom-up clustering to initialize the index. Using the Yeast Proteome, we show that our bi-directional bulk load produces a more effective index than the existing M-tree initialization algorithms.

1. Introduction

The Moore's constant (doubling time) for the size of GenBank has shrunk from 18 months to 15 months [2]. The Moore's constant for GenBank is now smaller than the Moore's constant for the doubling of processor speed. Thus, effective support of sequence driven queries requires the development of scalable database index structures. In particular, it will be insufficient to merely store sequence data in a database and use utilities to scan the database (e.g. BLAST). The sequences must serve as index keys.

We have investigated the use of M-trees as the basis of such an index [10]. An M-tree is a metric-space indexing package from the University of Bologna, Italy. The in-memory processing of nearest-neighbor and range queries for static metric spaces has received extensive attention [7, 20]. To our knowledge, M-trees are the most productive work to date entailing dynamic disk-based data.

Definition 1: A *metric space* is a set of objects, S, and a *[metric] distance function*, d, such that, given any three objects, x, y, z,

(i) $d(x, y) >= 0$ and $d(x, y) = 0$ iff $x = y$. (*Positivity*)

(ii) $d(x, y) = d(y, x)$. (*Symmetry*)
(iii) $d(x, y) + d(y, z) >= d(x, z)$. (*Triangle Inequality*)

The value of a metric-space approach is that it is unnecessary to find a meaning for the data with respect to the axes of a coordinate system. There is a clear connection between algorithms for hierarchical clustering and the construction of metric-space index trees. One can say that a tree-based index structure of a metric space materializes a persistent representation of a hierarchical clustering of the data [7]. The triangle inequality is leveraged both to find naturally occurring data clusters, allocate them to separate sub-trees and to prune the search when retrieving data by traversing the tree.

Global alignment of sequences based on simple edit distance forms a metric(Levenshtein distance)[13]. Hence it is commonly asserted that metric-space indexing should be applicable to biological sequence data. This has been accomplished by Giladi et.al. in the SST algorithm. In their work they derive a mapping of short nucleotide sequences to a binary vector space. There are many protein classification studies entailing the projection of similarity scores into a metric-space and deriving hierarchical clusterings [1, 22, 25, 28].

We are conducting our work in the context of developing a next-generation database management system, MoBIoS, (Molecular Biological Information System), specialized for storage, retrieval and mining of biological data types. MoBIoS comprises a metric-space based storage-manager and a database query language embodying the semantics of genomic and proteomics data. The research addresses a primary challenge facing biological databases: many of the data types are incompatible with the relational data model. Even when object-relational data models (e.g. XML) are used, there are few results comprising scalable index structures. The general interest in metric-space indexing stems from multimedia data types, such as images, video and music [4, 7].

With respect to sequence look-up we expect our complete homology search implementation to follow a general framework first proposed and analyzed by Myers [24]. Let the database comprise a sequence, A of length N. Let the query be a string, W of length P, and a similarity threshold D.

Off-line:

1) Divide a sequence database into small substrings of fixed length, T.
2) Build an index to support constant-time, O(1), look-up of matches.

On-line query:

1) Divide the query string W into substrings, Wi, of length T, $i = 0..\lceil W/T \rceil$.
2) For each Wi, generate all strings that fall within the similarity threshold D. Use the index to determine if each generated string is in the database. If not, discard it.

3) Chain the valid strings together to form solutions to the full query.

Variations on the division (and overlap) of the database and query sequences lead to different chaining algorithms. The reader may recognize that BLAST divides the query sequence, but not the database and in-lieu of building an index off-line, generates the hot-spot index at query time. Contemporary efforts, SST, BLAT and our own, to name a few [15, 21], seek to replace the generate and test phase of on-line step 2, with an index look-up that retrieves directly all substrings within a neighborhood of the query substring.

Giladi et. al. report performance for the SST algorithm as 27 times faster than BLAST when running search alone and 15 times faster than BLAST when considering both building and searching time [15]. In their work strings of nucleotides were mapped to a metric-space using Manhattan distance and a k-tuple encoding of subsequences where the dimension of the vector is S^k, where S is the size of the alphabet of the nucleotides. Using a main-memory index tree they further report average time complexity for building database index is O(nlogn) and O(mlogn) average time complexity for search. Although the authors claim the technique is equally applicable to peptides, the performance, in terms of speed and accuracy will suffer due to significantly increased alphabet size. (20 amino acid vs. 4 nucleotides.)Kent reports slightly faster results with a similar approach comprising a hash-based index, (BLAT)[21].

For these sequence homology efforts, the distance functions do not embody an evolutionary model and the application of the algorithms is limited to sequence assembly and evolutionarily close questions.

"BLAT is a very effective tool for doing nucleotide alignments between mRNA and genomic DNA from the same species" [21].

In both the sequence homology efforts and the protein taxonomy efforts the focus is on main-memory applications. To achieve our goals for MoBIoS we need to resolve two issues. We need to work through the details of a scalable disk-based metric-tree index. We need a direct method for computing an evolutionary distance between subsequences. PAM and related substitution matrices compute similarities, not distances. Similarity matrices reward more similar sequences with higher scores, an intuitively comfortable representation for biologists, but reverse the order in a metric. In a metric more similar objects must have distance closer to zero.

In related work we have addressed the distance issue by reworking the PAM substitution matrices to entail a metric [27]. Intuitively, where the PAM model speaks to the frequency of amino acid substitutions, for a given frequency, we have computed an expected time. Relatively speaking, those substitutions that occur more frequently can be expected to occur in a shorter time. Thus, our matrix,

mPAM, speaks to the requirement of a metric-distance that more similar objects score closer to zero. There are a number of other metric substitution matrices [16, 29].

The challenge we face with respect to scalable disk-based metric-indexes is called the *curse of dimensionality* [6, 7]. Simply stated, in a high dimensional space, as the distance from a point data grows, the number of neighbors in a bounding sphere grows around that point exponentially. Further, most interesting problem statements concerning optimal, single-level, k-clusterings are NP-hard. Consequently, partitioning a high-dimensional space into a hierarchy of disjoint clusters is an unstable, computationally demanding process.

We visit the applicability of the M-tree for the MoBIoS storage manager. To do so we loaded the contents of a local chloroplast sequence database and the yeast proteome into M-trees and developed a test workload by randomly selecting subsequences from the two databases. We were not satisfied with the query performance. We tried all built in alternatives for initializing the index. We found that different initializations impacted the search performance.

We present the results of our initial empirical analysis. We further analyze, algorithmically, M-tree's built-in loading features. We designed a new, bi-directional initialization, taking into consideration the special needs for clustering with respect to secondary storage. Our starting point is a greedy k-center algorithm, farthest-first-traversal [18], which is guaranteed to produce clusterings within a constant factor of two of optimal. Both analysis and experimental results demonstrate our algorithm is scalable. Further, our initialization algorithm leads to better query performance.

The paper is organized as follows. In Section 2, we introduce M-tree and its application for gene sequences indexing. Several algorithms to initialize M-tree are described and analyzed in Section 3, include M-tree bulk loading and our bi-directional initialization, followed by experimental results in Section 4. Section 5 consists of conclusions and future work.

2. Introduction to M-tree and Its Application

All tree-based database index structures may be generalized as follows : Consider a data set S of n objects. Arbitrarily partition S into blocks, such that the content of each block can be qualified by a predicate P. For example, if the data type is rectangles on a plane, then P may be the minimum-bounding rectangle that covers all rectangles in the block. If there are B objects per block, roughly n/B blocks are created. Consider as a new data set the n/B predicates describing these blocks, and repeat the process until a single block, the root, is formed. The result is a balanced tree of height $\log_B n$.

Generally speaking, there are two primary methods used to define the predicates for metric-spaces: vantage points and generalized Hyperplanes. In vantage point methods [2, 7, 8, 30], a point is chosen as

the vantage point, then other points are partitioned according to their distances to the vantage point. Each partition satisfies a predicate $P(V, r_{min}, r_{max})$, i.e., each distance from the vantage point to each point in this partition is within the range (r_{min}, r_{max}). In generalized hyperplanes [4, 7, 10, 30], two points are chosen as centers. Membership of the remaining points to the partitions are determined by computing the distance of each point to the centers, and assigning the point to the closest center. Therefore, each partition's predicate is its center. In either method, the fanout of interior nodes may be increased by choosing many center points, in which case, in Euclidean space the data structure resembles a Voronoi partitioning [7].

M-tree [9, 10] is a balanced General Hyperplane structure with dynamic capabilities. It is page based to accommodate large, disk resident data sets. It is built on the basis of GiST framework[6]. M-tree is open source and the code can be downloaded at the M-tree Project Homepage [23]. The release we use is version 0.911. In this section, we will introduce the node structure, and search strategy of M-tree. Since our contribution concerns initialization, we will not detail the dynamic aspects of the M-tree algorithm. The interested reader may see [10]

2.1 Node structure

M-trees are composed of internal nodes and leaf nodes. Leaf nodes store all the data objects entries. Internal nodes store *routing objects* [9, 10].

O_r: the key of the routing object **(a**
$ptr(T(O_r))$: the pointer to the root node of the sub-trees
$r(O_r)$: the covering radius
$d(O_r, p(O_r))$: the distance from $P(O_r)$, the parent object of O_r

O_j: the key of the data object **(b**
$Oid(O_j)$: the object identifier
$d(O_j, p(O_j))$: the distance from $P (O_j)$, the parent object of O_j

Figure 1 Structures of routing objects (a) and leaf entries (b).

A routing object consists of a pointer to the root node of a sub-tree, the key of the routing object, a covering radius and the distance from the parent routing object to itself. Each leaf entry consists of the key of the data object, the data object identifier and the distance from its parent routing object to itself. The structure of the routing objects and leaf entries are shown in Figure 1[9, 10].

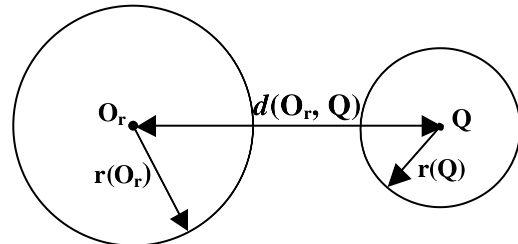


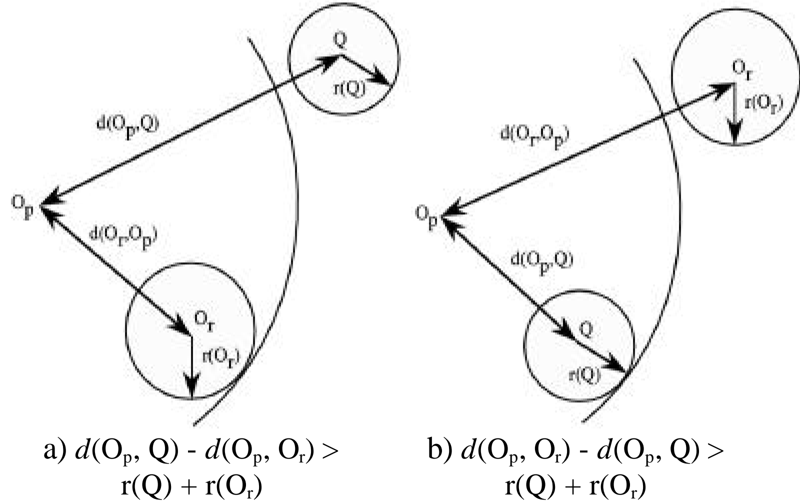
Figure 2. M-tree search rule 1

2.2 Searching in M-tree

M-trees support two types of similarity queries, range query and nearest-neighbor query. Since nearest-neighbor queries can be systematically implemented by range queries [7], to simplify the discussion, we only consider range queries in this paper.

Definition 2: Range query range(Q,r): Given a data point Q and a distance r , a range query returns all objects O in the database such that $d(Q, O) = r$.

The performance of similarity queries for disk based systems must consider both the number of disk I/Os, (i.e., number of nodes accessed), and number of distance computations. To improve performance, the key idea is to store pre-computed distances in the interior nodes, such as covering radii of the children, and exploit the triangle inequality to prune both the number of distance calculations within an interior node and the number of sub-trees recursively searched. Given query (Q, r), routing object O_r with covering radius $r(O_r)$ and parent node O_p, M-tree search exploits two rules [7, 10].



a) $d(O_p, Q) - d(O_p, O_r) > r(Q) + r(O_r)$ b) $d(O_p, O_r) - d(O_p, Q) > r(Q) + r(O_r)$

Figure 3. M-tree search rule 2

Rule 1: If $d(O_r, Q) > r(Q) + r(O_r)$, then, for any data object O_j in the sub-trees of Or, $d(O_j, Q) > r$, i.e., all the sub-trees of O_r can be pruned.

Rule 2: If $| d(O_p, Q) - d(O_p, O_r) | > r(Q) + r(O_r)$, then for any data object O_j in the sub-trees of Or, $d(O_j, Q) > r$, i.e., all the sub-trees of O_r can be pruned..

Rules 1 and 2 are shown in Figures 2 and 3 [10]. Rule 1describes the case that the query Q is so far from a routing object O_r that it is impossible for their covering spheres to intersect. Rule 2 describes the case that the difference between the distance from parent routing

```
rangequery(q: query, r: radius, p:node center)
{ if ( internal node )
  { for (each child c)
      if ( |d(p,q) – d(p,c)| > r(q) + r(c) )
        prune this child;
      for (each remain child c)
      { compute d(c,q);
        if ( d(c,q) > r(q) + r(c) )
          prune this child;
        else
        { rangequery(q, r , c);
          save d(c,q) for future use; }
      } }
  else // leaf node
  { for (each child c)
      if ( |d(p,q) – d(p,c)| > r(q)
        prune this child;
      for (each remaining child c)
      { compute d(c,q);
        if ( d(c,q) <= r(q) )
          add c to the result list; }
  }
```

Figure 4. Algorithm of range query

object O_p to child routing object O_r and the distance from O_p to query Q is so large that it is impossible for their covering spheres to intersect. The interested reader is referred to [7, 10] for details.

The search algorithm can traverse the tree either breadth-first or depth-first. The depth-first search algorithm is detailed in Figure 4.

3. Initializing M-tree

M-tree takes the search requirements of secondary storage into consideration. This implies that the tree should be balanced and within some tolerance have the same utilization. That is each leaf node should contain the same amount of data and be the same depth. Similarly, interior nodes should have the same number of routing objects. We note that this is a sharp contrast with the usual goals of hierarchical clustering; revealing naturally occuring relationships among the data. In particular, the natural clustering of a data set may be skewed, that is some leaves of the tree may be very deep compared to others. Outliers, data points that are far from most other points of the dataset, if each mapped to their own cluster and then each mapped to a disk page, can result in both high I/O overhead and poor disk utilization.

Given a data set D of size n ($D = \{d1, d2, \ldots, dn\}$), we cluster D hierarchically with constraints on minimum and maximum node utilization. We denote the constraints as minimum and maximum children number, m and M, of nodes and k ($m = k = M$) as the average number of objects in index nodes (fanout). A good initialization method is critical to the query performance. M-tree's dynamic operations (insert and delete) make certain local, greedy (clustering) decisions to avoid algorithmically expensive operations. We can expect M-trees will require periodic rebalancing.

As part of its standard distribution, there are two ways to initialize an M-tree, a top-down bulk-load method and repeated insertions. Repeated insertions create a tree bottom-up.

We were not satisfied with the results we first obtained with M-tree and recognized its initialization methods as a contributing factor. In particular, its greedy heuristics intended to minimize extra I/O due to outliers induce large radii and reduce the opportunity for pruning during search. For our implementation and analysis we use the farthest-first traversal k-center algorithm (FFT). FFT is a fast, greedy algorithm that minimizes the maximum cluster radius. It has been shown that FFT is guaranteed to produce a solution within a factor of 2 of optimal [18]. Since there is no consideration of balance, this is not necessarily the best objective function**.** Our thinking is it is better to try heuristics associated with proven algorithmic guarantees, even if the objective function is not optimal, in preference to heuristics that speak only intuitively to desirable objective functions.

In FFT, k points are first selected as cluster centers. Then for each remaining point, the point is added to the cluster whose center is the closest. To select the centers, the first center is chosen as random. The second center is greedily chosen as the point furthest from the first. Each remaining center is determined by greedily choosing the point farthest from the set of already chosen centers, where the furthest point, x, from a set, D, is defined as, $\max_x\{\min\{d(x,j), j \in D\}\}$. The farthest-first traversal algorithm is shown in Figure 5. The time complexity of FFT is $\Theta(sk)$, where s is the number of data objects and k is the number of clusters.

```
Farthest_first_traversal(D: data set, k: integer)
{ randomly select first center;
  //select centers
  for (I= 2,…,k)
  { for (each remaining point)
      calculate distance to the current center set
      select the point with maximum distance as new center;
    }
  //assign remaining points
  for (each remaining point)
  { calculate the distance to each cluster center;
      put it to the cluster with minimum distance; }
}
```

Figure 5. Algorithm of farthest-first traversal

We detail top-down and bottom-up initialization algorithms of M-tree and those derived from FFT. We then introduce a method for initializing M-trees that interleaves top-down and bottom-up clustering steps. The bi-directional method is simple and scalable. Intuitively, it yields better query performance because it considers more global information when merging outliers into other clusters.

3.1 Top-down Initialization Algorithms

Definition 3: Simple Top-down Hierarchical Clustering: Run FFT on the whole data set to establish an initial clustering. Recursively call FFT on each cluster that is too big to be put in a leaf node.

The Simple Top-down Hierarchical Clustering algorithm is detailed in Figure 6. Since the algorithm is applied recursively to sub-clusters,, the maximum children number constraint is satisfied. To satisfy the minimum children number constraint we delete the too small clusters and add each of their data objects to the closest of the remaining clusters respectively.

```
Td-Bulkloading(D: data set, m, M: integer)
{ if ( |D| = M) return a leaf node;

  FFT( D, k);

  For (each sub-cluster Di)
      If ( |Di| < m) delete Di, put each
        point in Di into the closest sub-
        cluster;

  For (each remaining sub-cluster Di)
```

Figure 6. Algorithm of Simple Top-down Hierarchical Clustering

Simple top-down hierarchical clustering does not guarantee the index tree to be balanced. It may be skewed. It does guarantee that each leaf is properly utilized. We use this result for bi-directional clustering.

Lemma 1: *Simple Top-down Hierarchical Clustering's time complexity is* $\Theta(nlgn)$ *in the best case, and* $\Theta(n^2)$ *in the worst case.*

Proof: Best case:

If each recursive call of FFT produces similarly sized clusters, the algorithm degenerates to a simple divide-and-conquer algorithm of k sub-problems, each of which with size n/k. The index tree is balance. The time complexity to build the tree is**T** (nlg^n).

Worst case:

When FFT yields unbalanced clusters, in extreme case, after eliminating the small clusters, only two clusters will be left. One will be a leaf of size k-1. The other node, subject to recursive splitting will contain n-k data objects. To simplify, we assume the leaf size is k.

T(n) = FFT(n,k) + T(n-k,k)

T(n-k, k) = FFT(n-k, k) + T(n-2k, k)

….

? T(n) = FFT(n,k) + FFT(n-k,k) + FFT(n-2k, k) + …

=**?** $_i$ FFT(n - i**?** k, k) , (0 **=** i**=** n/k –1)

= **?** $_i$ k(n - i**?** k) , (**?** FFT(s,t) = O(st))

=**?** $_i$ kn - i**?** k^2 = kn**?** n/k - k^2**?** $_i$ I

= n^2 - k^2**?** 1/2(n/k –1)**?** n/k

=**T** (n^2).

M-tree top-down bulk loading algorithm is also a divide and conquer method. We only introduce the M-tree bulk loading base version. The interested reader may see [9] to see several optimizations.

The M-tree top-down method generates an initial clustering by selecting *k* center objects at random. Each remaining object is then put into the cluster whose center is the closest to this object. To satisfy the minimum node size constraint, the algorithm detects and deletes small clusters and inserts their data objects into the remaining clusters that have the closest center respectively. Similar to simple top-down clustering, the algorithm calls itself recursively for each cluster. It finds the minimum height h_{min} for all the subtrees. Then, it splits the taller subtrees into a forest of subtrees of height h_{min}. Next, it calls itself recursively to build a super tree on the centers of the

```
Mtree-Bulkloading(D: data set, m, M: integer)
{ if ( |D| = M) return a leaf node;
   Randomcenter_clustering(D, k);
   Eliminate small clusters;
   For (each remaining cluster Di)
        Mtree-Bulkloading(Di, m, M);
   Split the resulting index trees into sub-trees of the
     minimum height;
   Construct data set Super of all sub-tree roots;
   Mtree-Bulkloading(Super, m,M);
   Append sub-trees to the super tree.
   r = max{ ri + d(o, oi) };
}
```

Figure 7. Algorithm of M-tree Bulk loading

forest. Finally, it calculates the covering radius of the root routing object as the maximum sum among the sums of the covering radius of its children and distance from the children [9, 10]. Figure 7 presents the algorithm of M-tree bulk loading.

We can see, like the simple top-down clustering, that M-tree bulk loading satisfies the minimum and maximum node size constraints but it also produces a balance index tree. The Algorithm [9] is shown in Figure 7. We analyze the time complexity of M-tree bulk loading by Theorem 1.

Theorem 1: *The time complexity of M-tree bulk loading is $\Theta(nlgn)$ in the best case and $O(n^3)$ in the worst case.*

Proof: Best case:

If the k-center algorithm produces clusters in nearly the same size, the index trees produced by recursively calls on every cluster are of the same height. Therefore, no index tree will be split and no super-tree will be generated. Actually, this case is a Simple Top-down Hierarchical Clustering. From Lemma 1, the time complexity is $\Theta(nlgn)$.

Worst case:

If the k-center algorithm does not produce clusters balanced in size, it takes more time to bulk load. Let's consider an extreme case that only two clusters left after eliminating the small clusters, and one of them is a leaf. Thus, the minimum index tree height is 1, the other higher index tree will be split and number of sub-trees is n/k. So, we have (assume n is big enough)

$T(n) = nk + T(n/k) + T(n-k)$

$T(n-k) = (n-k)k + T(n/k-1) + T(n-2k)$

$T(n-2k) = (n-2k)k + T(n/k-2) + T(n-3k)$

…

$T(n-(n/k-1)k) = (n-(n/k-1)k)k + T(1) + T(k)$

$T(n) = \sum_i (n-ik)k + \sum_i T(n/k-i) + \sum_i T(n-ik), (0 \le i \le n/k-1)$

$= n^2 - k^2 \sum_i i + \sum_i T(n/k-i) + \sum_i T(n-ik)$

$= n^2 - k^2[1/2\ n/k\ (n/k-1)] + \sum_i T(n/k-i) + \sum_i T(n-ik)$

$\approx n^2 - k^2[1/2\ n/k\ n/k] + \sum_i T(n/k-i) + \sum_i T(n-ik)$

$= 1/2\ n^2 + \sum_i T(n/k-i) + \sum_i T(n-ik)$ (1)

$\approx 1/2\ n^2 + \sum_i T(n/k-i)$

$\approx 1/2\ n^2 + \sum_i T(n/k-i)\ , (0 \le i \le n/k - n/2k)$

$\approx 1/2\ n^2 + (n/k - n/2k)T(n/2k)$

$= 1/2\ n^2 + n/2k\ T(n/2k\ )$

$= 1/2\ n^2 + n/2k\ 1/2\ (n/2k)^2$, (from (1), $T(n) = 1/2\ n^2$)

$=1/2\ n^2 + n^3/16k^3$

$= 1/16k^3\ n^3$.

3.2 Bottom-up Initialization Algorithms

Definition 4: Simple Bottom-up Hierarchical Clustering: It clusters the data set level by level in a bottom-up style. For each level, it first computes number of clusters according to index node size, fanout k and size of current level data set. Then it runs FFT to cluster the data set, and finally, it constructs the data set of next level from all the centers of sub-clusters of current level.

```
Bu-Bulkloading(D: data set, m, M: integer)
{ if ( |D| = M) return a leaf node;
  while ( |D| > M)
   {  FFT( D, |D|/k);
      For (each sub-cluster Di)
           If ( |Di| < m) delete Di, put data in Di
             into other sub-cluster;
      D = centers of all Di;
   }
}
```

Figure 8. Algorithm of Simple Bottom-up Hierarchical Clustering

Algorithm of Simple Bottom-up Hierarchical Clustering is shown in Figure 8. The minimum node size constraint can be satisfied by using the same merging technique discussed in last sub-section. However, since in each level FFT only runs once and then continues to cluster the higher level, the maximum node size constraint is not guaranteed to be satisfied. The maximum node size constraint will be satisfied only if FFT does not produce unacceptably large cluster. It will generate a balanced index tree.

Lemma 2: *If maximum node size constraint is satisfied by FFT, Simple Bottom-up Hierarchical clustering generates a balance index tree, and the time complexity is* $T\ (n^2)$.

Proof: If maximum node size constraint is satisfied, given average node size k, cluster number of a level is the size of data set of that level divided by k. So, we have following:

$$T(n) = FFT(n, n/k) + FFT(n/k, n/k^2) + FFT(n/k^2, n/k^3) + \dots$$

$$= ?_i\ FFT(n/k^i, n/k^{i+1})\ ,\ (\ 0 = i = \log_k{}^n - 2\ )$$

$$= ?_i\ FFT(n/k^i, n/k^{i+1})\ ,\ (?\ FFT(s,t) = O(st)\ )$$

$$= ?_i\ n^2\ k^{-2i-1} = n^2 ?_i\ k^{-2i-1}$$

$$= n^2 ?_i\ k^{-2i},\ (\ 0 = i\ ) = n^2\ e = T\ (n^2)\ .$$

M-tree bottom-up initialization is implemented by running insert operations repeatedly for each data object in the given data set. The insert algorithm is an insert-split-promote method similar to that of B-tree [10]. It first descends the tree and selects the target leaf node into which to insert the data object based on

its search algorithm. If the target node exceeds the maximum node utilization after insertion, it splits the node, selects a pair of objects as new centers and promotes them. Several split and promote policies are provided by M-tree, such as minimum volume, minimum overlap, and generalized Hyperplane distribution. The interested reader may see [10].

In the insert-split-promote procedure, no matter what policies are used, the time spent on each level is constant. Thus, given the height h of the tree, the insertion accesses h levels in the best case (no split and promotion) and 2h levels in the worst case (split and promote in every level). If the tree is balanced, the time complexity of insertion is Θ(lgn). Consequently, the time complexity of M-tree bottom-up initialization is Θ(nlgn). Our experimental results confirm the reported results that this method of initialization scales well with the size of the database.

Comparing with M-tree top-down bulk loading, M-tree bottom-up initialization is faster. At the same time, since the top-down uses random k-center algorithm, bottom-up yields out better query performance. The experimental results also show this. On the other hand, since the choice of leaf node is determined by the search procedure and clusters are split pairwise without consideration or reorganization of the remaining clusters, the M-tree bottom-up initialization technique does not produce uniformly tight clustering at the leaves. Moreover, both top-down and bottom up compute the covering radius as the maximum sum of the distance from the center to child center and the covering radius of child routing object among all children, i.e., $r = \max\{ d(O, O_i) + r_i, O_i$ are the children of $O\}$. Radii computed by this way are actually increased unnecessarily. We will detail this in the following empirical analysis.

3.3 FFT Derived Bi-directional Initialization

Summarizing, we can see that both simple top-down and simple bottom-up have their advantages and disadvantages. Specifically, top-down algorithm satisfies the maximum node size constraint but cannot guarantee a balanced index tree. Bottom-up produces a balanced tree, but will satisfy the maximum node size constraint if and only if the data naturally decomposes into approximately equal size clusters, an unlikely scenario given that the number of clusters, (one per leaf) is calculated a priori with any knowledge of the nature of the data.

```
Bidirectional-Bulkloading(D: data set, m, M: integer)
{ if ( |D| = M) return a leaf node;
  while ( |D| > M)
   { Td-Bulkloading(D,m, M);
     get all the leaves Di.
     construct index nodes of current level from Di
     D = {centers of all Di}
     compute the covering radius: r = max{ d(O,di) }
   }
}
```

Figure 9. Algorithm of Bi-directional initialization

How can we achieve the advantages of both approaches while avoid both disadvantages? The answer is to use top-down simple clustering to determine the contents of the data pages, and continue the bottom-up construction of the tree by repeatedly applying the top-down clustering. In other words, the bounding predicates (centers) for each data page (cluster) are treated as a new data set. The top-down method clusters those predicates to form the interior nodes. The process is repeated until the bounding predicates fit into a single node, which forms the root of the tree.

Definition 5: Bi-directional initialization: This is a bottom-up clustering algorithm. In each level of the bottom-up, we first run a simple top-down hierarchical clustering algorithm on current level data set. The leaves of the cluster tree returned by simple top-down are actually clusters of the data set with minimum and maximum node size constraints. Then, we construct the data set of next level from centers of all clusters of current level. The radius is computed as the maximum distance between the center and each data objects in the subtree. The bottom-up procedure terminates when current level data fit one disk page.

The bi-directional initialization algorithm is shown in Figure 9. Now, the top-down algorithm is applied to the predicates that cover the leaves that already satisfy the minimum and maximum node size constraints, building the bottom-layer of interior nodes. Repeated application of the top-down algorithm generates a balanced index tree satisfying both constraints.

Theorem 2: *The time complexity of bi-directional initialization is* $\Theta(nlg^n)$ *in the best case and* $\Theta(n^2)$ *in the worst case.*

Proof: The best and the worst cases of Simple Top-down Hierarchical Clustering are also the best and the worst cases of Bi-directional Bulk-loading respectively.

From Lemma 2:

$$T(n) = FFT(n, n/k) + FFT(n/k, n/k^2) + FFT(n/k^2, n/k^3) + \ldots$$

$$= Top\text{-}down(n) + Top\text{-}down(n/k) + Top\text{-}down(n/k^2) + \ldots$$

Best case:

From Lemma 1: Top-down(n) = O(nlgn)

$$T(n) = nlg^n + n/k\ lg^{n/k} + n/k^2\ lg^{n/k2} + \ldots$$

$$= \Sigma_i\ n/k^i\ lg^{n/ki},\ (0 \le i \le \log_k^n - 1)$$

$$= \Sigma_i\ n/k^i\ (lg^n - i) = nlg^n \Sigma_i\ k^{-i} - \Sigma_i\ i\ k^{-I}$$

$$\le nlg^n \Sigma_i\ k^{-i} = nlg^n\ [\ (n\text{-}1)k / n(k\text{-}1)\ ] = \Theta(nlg^n)$$

Worst case:

From Lemma 1: Top-down(n) = $O(n^2)$

$$T(n) = n^2 + n^2 / k^2 + n^4 / k^4 + \ldots$$
$$= n^2 \sum_i k^{-2i}, (0 \le i \le n/k - 1)$$
$$= n^2 (1- k^{-2n/k}) / (1- k^{-2}) = \Theta(n^2).$$

All of top-down, bottom-up and bi-directional initializations have the same best-case time complexity, $\Theta(nlgn)$, which is the same as for B+-trees. Bi-directional initialization's worst-case time complexity, $\Theta(n^2)$, is far better than that of top-down initialization, $O(n^3)$, while is worse than that of bottom-up initialization, $\Theta(nlgn)$. In practice, for gene sequence datasets, the load time of both the bi-directional initialization and M-tree bottom-up scale linearly with database size (Figure 14).

Top-down uses random k-center algorithm to cluster a single level of data. Bottom-up only considers local context. In bi-directional initialization, farthest-first traversal algorithm, which gives a solution guaranteed to be within a factor of 2 of optimal, is used to cluster a single level or data with global consideration.

Moreover, in both M-tree initializations, cover radii are computed as the maximum of the sum of child-parent distance and child node radii among all children. Although this does not affect the correctness, it results in larger stored radii and therefore does affect the performance. In our structure, we compute the radius as the maximum distance between the node center and data object in the sub-tree. This takes more time in initialization but compensated by better query performance.

4. Experimental Results

Our experimental results comprise, the initialization running time, index structure information pertinent to index quality and query performance. We've made measurements of both M-tree initialization methods and bi-directional initialization. We use the M-tree open source release v0.91, which is written in C++ [23]. Our own implementation of M-tree with bi-directional initialization is written in JAVA. Therefore, we report on implementation independent measurements such as counting the number of distance computations and nodes visited. Nodes visited is proportional to the number of disk I/Os. We use the same source data and parameters, such as node utilization, dataset size, for both the original M-tree release and our Java implementation. The dataset we use is Yeast gene sequence data downloaded from [31], and local chloroplast genome sequence data. We split the sequences into equal size segments of 10 amino acids.

We first compare the query performance of M-tree's top-down and bottom-up initializations. For bottom-up initialization, we tried several combinations of split and promotion policies. The best combination is to use balanced-GH split policy and minimum overlap promotion policy together. We initialize M-trees with

chloroplast gnome sequence data and ran queries on them. For both initializations, the relationship between the number of distance computations and the database size, and relationship between the number of disk I/Os and the database size are shown in Figures 10 and 11. The results suggest that bottom-up initialization has better performance than top-down. Therefore, for the remainder of the paper we present M-tree results only with respect to bottom-up initialization.

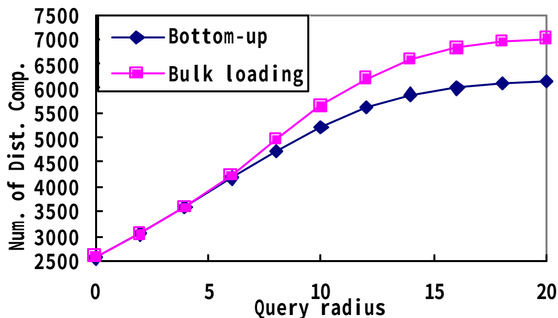


Figure 10. Number of distance computations vs. database size of M-tree initializations.

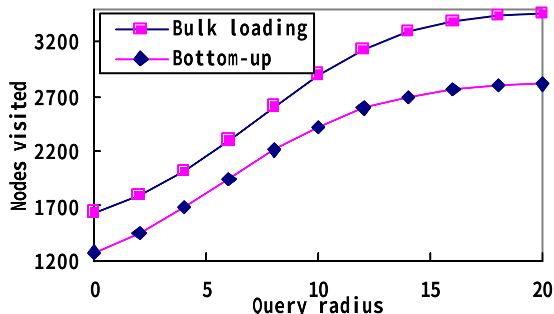


Figure 11. Number of nodes visited vs. database size of M-tree initializations.

In the rest of this section, we present the results of the bi-directional initialization with respect to the yeast proteome sequence dataset. To get deep insight of the data and held determine the scope of realistic range query radiiwe first compute the distance distribution for all pairs of points in the dataset (Figure 12). According to the figure, the data objects distribute normally in the space. The maximum distance between two object is around 60. Moreover, after distance 20, the curve climbs up rapidly, which indicates that due to vast amount of results, range query radius greater than 20 is meaningless. For 75% conserved peptide sequences of length 10, the average mPAM score is around 7.

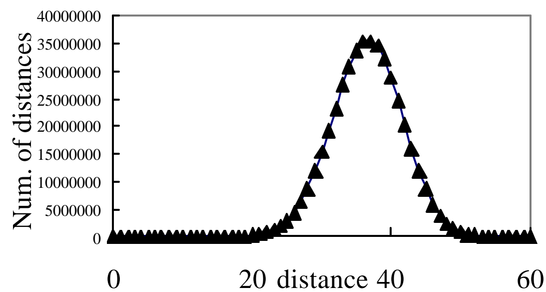


Figure 12. Yeast distance distribution

To lend insight into the quality of the clusterig of the two methods, bi-directional initialization and bottom-up initialization, we compute the covering radii of each tree level. See Figure 13.This figure shows that the radius of bottom-up initialization decreases significantly while descending the tree, which suggests that the dataset is clustered significantly.

However, we can also see that for higher levels, the radii of bottom-up initialization are much higher than those of bi-directional initialization. From Figure 12 we know that the maximum distance between two objects is around 60. However, in the seven higher levels of bottom-up initialized M-tree, the average radii are all greater than 60. According to the pruning rules, due to the large covering radii, almost none of

the children will be pruned in the higher levels, i.e., nearly all the internal nodes in the seven highest levels will be visited. This will appear as both I/O cost and distance computations. For bi-directional initialization, no radius exceeds 60, which means the search will prune tree branches immediately. Finally, in any level, the radius of bi-directional initialization is smaller than that of bottom-up initialization, indicating better clustering and implying better query performance than bottom-up initialization.

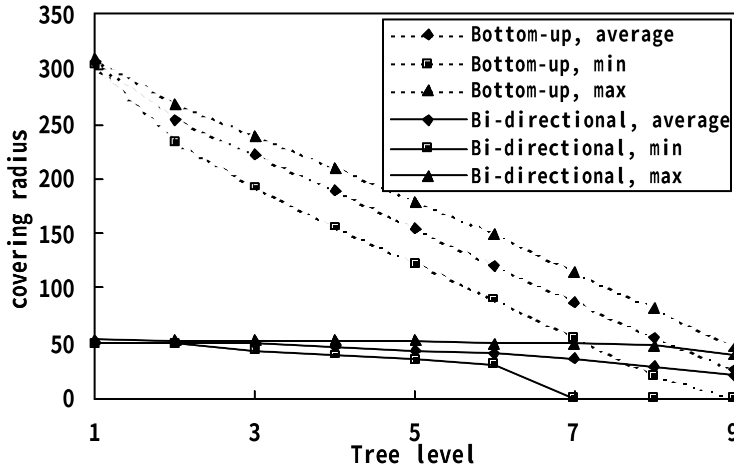


Figure 13. Covering radii of both initializations

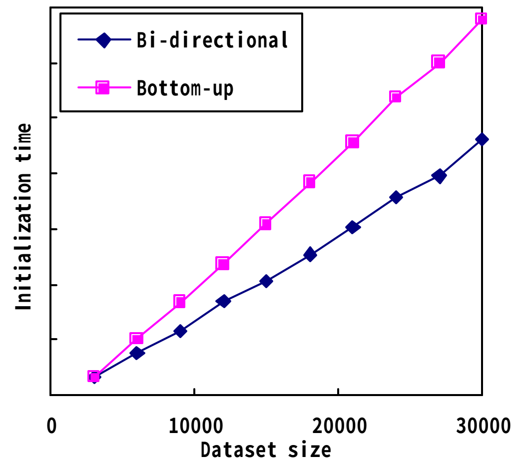


Figure 14. Running time of both initializations

We also show the initialization running time of the two initializations to examine their scalability (Figure 14). Because the bottom-up initialization is implemented in C++ and bi-directional initialization is implemented in JAVA, we scale the date when put them into the figure. Figure 14 show that both initializations have good scalability in building the database.

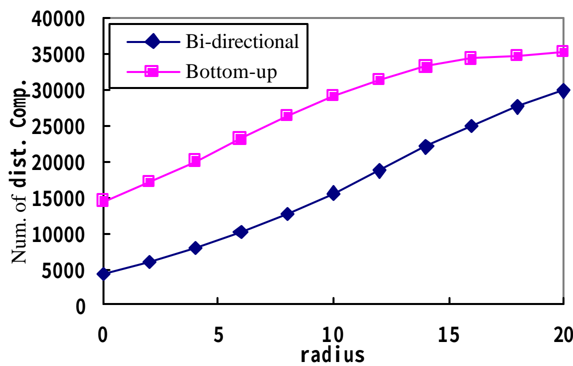


Figure 15. Comparison of number of distance computations vs. radius

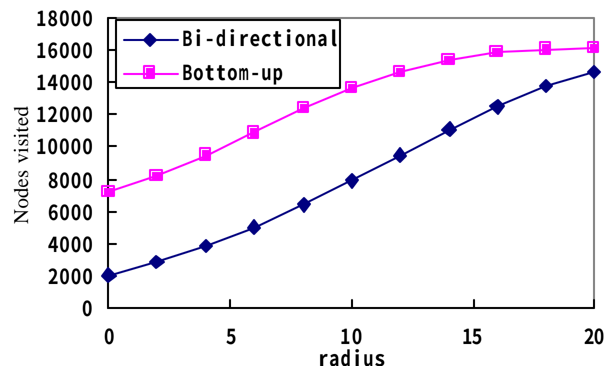


Figure 16. Comparison of number of nodes visited vs. radius

Next, we present the results of queries. Two main measurements of query performance are the number of distance calculations and number of disk I/Os. Figure 15 shows the comparison of number of distance computations of bi-directional initialization and bottom-up initialization. First, we can see that the numbers of distance computations of bottom-up initialization are much greater than those of bi-directional

initialization. Especially, for radius 0, (exact matches) bottom-up's number is three times of that of bi-directional initialization's, and for radius 10, bottom-up's number is two times that of bi-directional initialization. We can also see that as the radius increases, the two numbers become closer. Given the curse of dimensionality we would expect that increasing the radius would quickly induce the entire tree to be searched, independent of the initialization.

Comparison of I/O number vs. radius is shown in Figure 16. Similar conclusion can be drawn as from Figure 15.

From all the experimental results shown above, we can conclude that the bi-directional initialization owns better performance than M-tree bulk loading.

5. Conclusions and Future Work

Due to the curse of dimensionality the performance of a metric space indexing structure is very sensitive to the initialization algorithm. Moreover, periodic bulk loading and re-balancing is also critical to maintain an index structure for dynamic disk-based operation.

Comparing with M-tree top-down and bottom-up initializations, bi-directional initialization is simpler and produces better results. Specifically, top-down and bi-directional initialization have the same best case time complexity, T (nlgn), while top-down's worst case time complexity, O (n^3), is far slower than that of bi-directional initialization, T (n^2). Currently, we use bi-directional initialization on the same data structure and search strategy as M-tree. The differences are the single level clustering algorithm, the hierarchical clustering algorithm, and the way they calculate the covering radii of internal nodes.

Experimental results show that both bottom-up and bi directional initializations are scalable. However, bi-directional initialization produces better index structure. From the experiments, we can also conclude that bi-directional initialization owns better query performance than M-tree.

Our results for sequence data contradict those reported by the M-tree project on their synthetic test data. It is quite normal in the development of search-tree that heuristics are individualized to particular workloads [17]. In ongoing work we are exploring other clustering algorithms in hopes of finding a single generally applicable solution. Toward that end, we are exploring approximation algorithms for the capacitated k-median problem in hopes that a simple top-down clustering can be developed that will guarantee a balanced tree and potentially useful approximation guarantees for the index as a whole.

6. Acknowledgement

Our thanks to the M-tree project team for making M-tree open source.

7. References

[1] D. K. Agrafiotis, . A new method for analyzing protein sequence relationships based on Sammon maps. Protein Sci. 1997, Vol. 6, 287-293 .

[2] D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, B. A. Rapp, D. L. Wheeler, GenBank. Nucleic Acids Research, 2002, Vol 20, No 1. pp 17-20

[3] T. Bozkaya and M. Ozsoyoglu. Distance-based indexing for high dimensional metric spaces. In Proceedings of the 1997 ACM SIGMOD, pages 357--368. ACM, 1997.

[4] S. Brin. Near neighbor search in large metric spaces. In Proc. VLDB'95, pages 574--584, 1995.

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

[6] B. Chazelle. Computational geometry: a retrospective. In Proc. ACM STOC'94, pages 75--94, 1994.

[7] E. Chavez, G. Navarro, R. Baeza-Yates, and J.L. Marroquin, Searching in Metric Spaces (Survey). ACM Computing Surveys, 2001.

[8] T. Chiueh. Content-based image indexing. In Proc. of the 20th Conference on Very Large Databases (VLDB'94), pages 582--593, 1994.

[9] P. Ciaccia and M. Patella. Bulk loading the M-tree. In Proceedings of the 9th Australasian Database Conference (ADC'98), pages 15--26, Perth, Australia, February 1998.

[10] P. Ciaccia, M. Patella, and P. Zezula, M-Tree: An Efficient Access Method for Similarity Search in Metric Spaces. Proc. VLDB, 1997.

[11] M.O. Dayhoff, R. SChwartz and B.C. Orcutt. Atlas of protein sequence and structure. Vol. 5, Suppl. 3, Ed. M. O. Dayhoff, 1978.

[12] V. Gaede and O. Gunther. Multidimensional Access Methods. ACM Computing Surveys, 1997.

[13] D. Gusfield, Algorithms on strings, Trees, and Sequences: computer Science and Computational biology. Cambridge University Press, 1997.

[14] A. Guttman. R-trees: A Dynamic Index Struc- ture for Spatial Searching. Proc. of SIGMOD, 1984.

[15] E. Giladi, M. G. Walker, J. Z. Wang, and W. Volkmuth SST: an algorithm for finding near-exact sequence matches in time proportiona1 to the logarithm of the database size Bioinformatics 2002 18: 873-877.

[16] D.Haig and L.Hurst. A quantitative measure of error minimization in the genetic code. J. Mol. Evol., 33:412-417, 1991

[17] J. M. Hellerstein, J. F. Naughton and A. Pfeffer. Generalized Search Trees for Database Systems. Proc. 21st Int'l Conf. on Very Large Data Bases, Zürich, September 1995, 562-573.

[18] D.S. Hochbaum and D.B. Shmoys. A best possible heuristic for the k-center problem. Mathematics of Operational Research, 10(2):180-184, 1985.

[19] I. Kalantari and G. McDonald. A data structure and an algorithm for the nearest point problem. IEEE Transactions on Software Engineering, 9(5), 1983.

[20] D. Karger and M. Ruhl, Finding Nearest Neighbors in Growth-restricted Metrics, Proc. Of the 34th ACM Symposium on Theory of Computing, 2002

[21] W. J. Kent. BLAT: The BLAST-like Alignment Tool Genome Research 2002 Apr; 12(4):656-664.

[22] M. Linial, N. Linial, N. Tishby and G. Yona, Global self organization of all known protein sequences reveals inherent biological signatures, J. Molecular Biology 268 (1997) 539 - 556.

[23] M-tree project home page. http://www-db.deis.unibo.it/Mtree/index.html

[24] E.Myers, A sublinear algorithm for approximate keyword searching. Algorithmica, 1994, Vol. 12(4/5):345-374.

[25] R Mott, J Schultz, P. Bork , CP Ponting . Predicting Protein Cellular Localisation Using a Domain Projection Method. Genome Res (2002), 12, 1168-74

[26] S.B.Needleman and C.D.Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. Journal of Molecular Biology, 48:443-453,1970.

[27] R. Mao, D. P. Miranker, J. N. Sarvela and W. Xu. Clustering Sequences in a Metric Space. Poster on ISMB 02, Edmonton, Canada, August 2002.

[28] O.Sasson, N.Linial and M. Linial. The metric space of proteins -- comparative study of clustering algorithms, Proceedings The 10th International Conference on Intelligent Systems for Molecular Biology 2002

[29] W. R. Taylor and D. T. Jones. Deriving an amino acid distance matrix. Journal of Theoretical Biology, 164:65-83, 1993.

[30] J. K. Uhlmann. Satisfying General Proximity/Similarity Queries with Metric Trees. Information Processing Letter, 40(4):175-179, November 25, 1991.

[31] Yeast data file web page.