AH$^+$-Tree: An Efficient Multimedia Indexing Structure for Similarity Queries

Fausto C. Fleites and Shu-Ching Chen
School of Computing and Information Sciences
Florida International University
Miami, FL 33199, USA
{fflei001,chens}@cs.fiu.edu

Kasturi Chatterjee
Hi5 Networks
San Francisco, CA 94105, USA
kasturi.chatterjee@gmail.com

***Abstract*—This paper presents the AH$^+$-tree, a balanced, tree-based index structure that efficiently supports Content-Based Image Retrieval (CBIR) through similarity queries. The proposed index structure addresses the problems of semantic gap and user subjectivity by considering the high-level semantics of multimedia data during the retrieval process. The AH$^+$-tree provides the same functionality as the Affinity-Hybrid Tree (AH-Tree) but utilizes the high-level semantics in a novel way to eliminate the I/O overhead incurred by the AH-Tree due to the process of affinity propagation, which requires a complete traversal of the tree. The novel structure of the tree is explained, and detailed range and nearest neighbor algorithms are implemented and analyzed. Extensive discussions and experiments demonstrate the superior efficiency of the AH$^+$-tree over the AH-Tree and the M-tree. Results show the AH$^+$-tree significantly reduces I/O cost during similarity searches. The I/O efficiency of the AH$^+$-tree and its ability to incorporate high-level semantics from different machine learning mechanisms make the AH$^+$-tree a promising index access method for large multimedia databases.**

***Keywords*-index structures; range queries; nearest neighbor queries**

I. Introduction

Nowadays multimedia data pervade everyone's life and are more and more accessible every day. Online communities such as Facebook [1] handle a large amount of multimedia data that need to be stored, accessed efficiently, and meaningfully-retrieved by users. However, due to the nature of multimedia data, traditional database management systems are unable to provide efficient and meaningful multimedia data retrieval [2] due to the well known problems of (a) semantic gap between low-level features and high-level concepts and (b) the users' subjectivity [3].

Users require a mechanism that effectively accesses and allows the meaningful retrieval of multimedia data. Tree-based index structures offer the best solution since they provide both constant access to the indexed records and data retrieval through similarity queries. Consisting of range and nearest neighbor queries, similarity queries allow the retrieval of objects that are "similar" to a given query object. Given the fact that multimedia data are usually transformed into feature vectors on a multidimensional feature space on which a suitable (dis)similarity function can be defined, similarity queries provide content-based retrieval by allowing the retrieval of objects that are close to the query object. Multidimensional, tree-based indexing mechanisms, therefore, ought to provide similarity queries to access multimedia data.

For multidimensional data, several index structures exist in the literature, which can be categorized into feature-based and distance-based index structures. Please refer to Section VII for a discussion on this taxonomy. Feature-based index structures make difficult the introduction of object-level information that is not represented as a vector because they do not allow any correlation between the feature values in the distance function used to represent the (dis)similarity of data objects [4]. Providing a solution to this issue, distance-based structures index the pairwise distances or similarities of the objects. Representative distance-based index structures are the vp-tree [5], the M-tree [6], and the AH-Tree [4]. To the best of our knowledge, the AH-Tree is the only distance-based, tree-structured index method that tackles the problem of CBIR by incorporating high-level relationships in the index structure.

The AH-Tree combines both vector and metric spaces in a novel way to organize large image databases and supports CBIR [4]. An important characteristic of the AH-Tree is that it utilizes information from a learning mechanism to address the issues of semantic gap and user subjectivity. Specifically, the AH-Tree utilizes the concept of *affinity relationship* from the Markov Model Mediator (MMM) model [7]. Affinity relationships map low-level features to high-level concepts and aim to represent the users' perspective of concept-based relationships between the images. The AH-Tree provides k-nearest-neighbor (kNN) queries that directly support CBIR. Moreover, the AH-Tree was built upon the M-tree, so in the case when no affinity values are available, the AH-Tree has the same efficiency and accuracy of the M-tree.

Notwithstanding, the I/O overhead of the AH-Tree prevents it from being utilized on large multimedia datasets. The limitation arises from the process of *affinity promotion* which is executed every time the AH-Tree is queried. The promotion of affinity values [4] begins at the leaves of the tree and promotes the affinity values of the indexed records with respect to the given query upward the tree up to the root node. The affinity value of each internal node is computed as the maximum affinity value between the node's children. Once the whole tree has been populated with affinity values, the search algorithms are able to visit only nodes whose affinity values with respect to the given query object are higher or equal than a supplied query minimum affinity value, thus computing less distance computations during the query process by avoiding visiting sub-trees that do not comply with the query's minimum affinity requirement. The process of affinity promotion saves distance computations but forces a complete tree traversal for each query, which causes a significant increase in I/O overhead since each node in the tree has to be read from the disk onto the main memory.

To address the I/O overhead problems of the AH-Tree, this paper presents the AH$^+$-tree, which utilizes the affinity values in a novel way to avoid having to perform the process of affinity promotion while still providing the same accuracy of the AH-Tree. Instead of traversing the whole tree to promote affinity values, the AH$^+$-tree efficiently stores and makes use of the affinity information along with object-to-node and node-to-node relationships to identify the sub-trees that need to be trimmed out during the retrieval process. Each query specifies a minimum affinity value, and the AH$^+$-tree only visits nodes whose sub-trees contain indexed objects that comply with the query's affinity, thus providing the same functionality as that of the AH-Tree but without having to traverse the whole tree. Furthermore, the size of the internal data structures used by the AH$^+$-tree to keep track of object-to-node and node-to-node relationships is insignificant when compared to the size of the multimedia data being indexed. The contribution of this paper is in presenting an index structure that utilizes high-level multimedia information in a novel way to significantly improve the I/O efficiency of the AH-Tree.

The remainder of this paper is organized as follows. Section II describes the concept of high-level affinity relationships. Section III details the structure of the AH$^+$-tree. Sections IV and V presents range queries and nearest neighbor queries respectively. Section VI discusses the experiments. Section VII covers related work. Finally, section VIII concludes the work presented in this paper.

II. High-Level Affinity Relationships

The AH$^+$-tree makes use of high-level affinity relationships to tackle the issues of semantic gap and user subjectivity. High-level affinity relationships numerically represent how close objects are from a semantic concept point of view. For the present work, the high-level affinity relationships are obtained from the Markov Model Mediator (MMM) mechanism. Shyu et al. [7] base the affinity concept on the idea that the more frequent two images are accessed together, the more related they are. Given a set of q queries issued over a period of time, the affinity measurement between two images, m and n, is defined as follows:

$$aff_{m,n} = \sum_{k=1}^{q} use_{m,k} \times use_{n,k} \times access_k$$

The terms $use_{m,k}$ and $use_{n,k}$ denote the usage pattern of image m with respect to query k per time period and of image n with respect to query k respectively, and $access_k$ denotes the access frequency of query k per time period. Through usage patterns and frequencies, the affinity measurements capture high-level image relationships and thus provide a model to bridge the gap between low-level image features and semantic concepts.

The MMM model is not the only mechanism that can be used in the AH$^+$-tree as the source for high-level affinity relationships. Other methodologies that make use of mining approaches to identify relationships between multimedia data, such as [8][9][10], can be also utilized in the proposed retrieval routine of the AH-Tree without any loss of generality.

III. The AH-Tree

The AH$^+$-tree is a balanced, distance-based tree structure that improves the performance of the AH-Tree while still providing the same accuracy and functionality. The difference lies in the structure of the tree and how the affinity relationship is utilized during the retrieval process. The following sub-sections describe the representation of multimedia objects, the storage of the affinity relationship, and the structure of the tree.

A. Multimedia Object Representation

The AH$^+$-tree requires multimedia objects to be identified via id's which are integer values that begin at one and increase incrementally by one. This requirement by no means limits the type of multimedia objects that can be indexed by the tree, as these id values function as a primary key on top of any any attributes the multimedia objects may contain. Since the AH$^+$-tree is a distance-based index structure, it only indexes the distances between pairs of objects, and thus multimedia objects can be represented using any suitable mechanism.

B. Storage for Affinity Relationship

The data structure used to store the affinity relationship is critical to the overhead in memory consumption as well as the performance of the tree. The affinity relationship provides an affinity value between two multimedia objects; therefore, the natural method of storing this relationship is via a matrix, which will be called *affinity matrix*. However, in a large data set of multimedia objects, the affinity matrix will most likely be a sparse matrix, since it is not feasible for users to provide enough information to generate affinity values for every possible pair of multimedia objects. Consequently, using the traditional data structure used to store sparse matrices, the AH$^+$-tree makes use of linked lists to store the affinity relationship in a way that enables the tree's retrieval algorithms to efficiently use this information.

The affinity relationship is stored in an array of pointers, where each array index corresponds to an object id, and each pointer points to a sorted linked list ordered decreasingly by affinity. In subsequent sections, for a given query object Q, its corresponding affinity list if denoted $alist(Q)$.

C. Structure

The AH$^+$-tree is composed to two types of nodes: leaf nodes and internal nodes. Leaf nodes store pointers to the indexed multimedia objects (i.e., tuples of the database table) which are represented by a vector of low-level features, and internal nodes store pointers to routing objects. Internal nodes consist of a set of entries, and each entry contains a routing object and a pointer to the root of a sub-tree. All objects accessed through the sub-tree are within a distance r from the routing object; this distance is called the covering radius of the routing object. In addition, each routing object has an associated distance to its parent object. The entries of an internal node are six-tuples consisting of the routing object, the id of the routing object, a pointer to the corresponding sub-tree, the covering radius, the distance to the parent object, and the id of the node referenced by the entry. The entries of leaf nodes are similar to those of internal nodes but without a covering radius and contain the actual object identifier (oid) instead of a pointer to the sub-tree. Figure 1 depicts the structure of the AH-Tree. $L1$, ..., $L3$ represent leaf nodes; O_1, O_{20}, and O_3 are indexed objects; and N is an internal node, whose first entry, $E1$, is composed of a routing object O whose id is id_O, a covering radius r, a distance d to its parent object, a pointer to a sub-tree $ptr(T(O))$, and the id (id_{node}) of the node referenced by E_1.

Moreover, the AH$^+$-tree contains two globally-accessible arrays, namely $ObjectToNode$ and $NodeTrack$, which store integer numbers. The former is of length T and the latter of length M. T is the number of objects, and M is the expected number of nodes necessary to index the T multimedia objects. The value of M can be computed as $\sum_{i=1}^{\log_b T} b^{i-1}$, where b is the minimum node-utilization parameter that is given when the tree is constructed.

Each index i of $ObjectToNode$ corresponds to the multimedia object O_i, and $ObjectToNode[i]$ stores the id of the leave node that holds the reference to O_i. When the object O_i is inserted in some leave node L_k, $ObjectToNode[i]$ is assigned the id of node L_k. Every time a new node is created in the tree, the node is assigned an id equal to the number of nodes in the tree plus one. Consequently, all nodes in the tree are numbered consecutively given their order of insertion. On the same token, each index k of $NodeTrack$ corresponds to the node with id k, denoted as N_k. $NodeTrack[k]$ stores the id of the parent node of N_k. To summarize, $ObjectToNode$ allows, for each multimedia object, the identification of the leave node where the former is located (object-to-node relationship), and $NodeTrack$ allows, for every node in the tree, the identification of its parent node (node-to-node relationship).

It is worth noticing that the sizes in bytes of $ObjectToNode$ and $NodeTrack$ are "negligible" when compared to the size consumed by the T multimedia objects in the database. For example, a database of 1,000,000 images, each represented by an id and a 100-dimensional feature vector, will consume $1,000,000 \times (4 + 8 \times 100)$ = 804,000,000 bytes which is approximately 767 MB; this calculation assumed id's are stored using 4-byte integers and feature values with 8-byte floating-point numbers. For such database of images, $ObjectToNode$ will consume $1,000,000 \times 4$ = 4,000,000 bytes which is approximately 4 MB, and $NodeTrack$, assuming a minimum node utilization of 50, will consume $\sum_{i=1}^{\log_{50} 1,000,000} 50^{i-1}$ which is approximately 0.5 MB.

Constructed in a way similar to that of the R-Tree, the AH-Tree is kept balanced (i.e., all leaves are at the same level) by inserting new keys at the leaves, splitting nodes that overflow, and propagating changes upward in the tree [4]. $ObjectToNode$ is updated appropriately when an object is inserted in a leaf node. When there is an split in the tree due to a node overflow, $NodeTrack$ is updated accordingly to keep track of the parent node of the nodes being modified. Tree construction and node insertion are not covered in this paper in more detail due to space limitations.

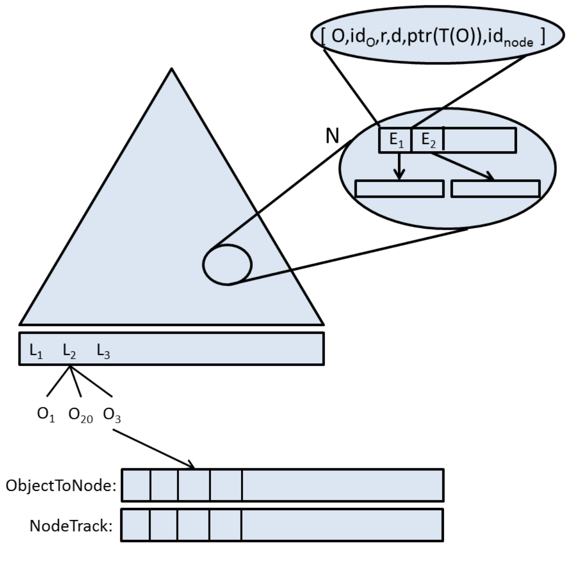

Figure 1. Structure of the AH$^+$-tree

In the rest of the paper, for notational purposes, the entry for object O is denoted as $e(O)$, and the entry's values are denoted as $e(O).r$, $e(O).d$, $e(O).ptr$, $e(O).id$, and $e(O).id_{node}$. In addition, $e(O).id_{node}$ is also denoted as $id(N)$, where N is the node pointed by $ptr(T(O))$.

IV. Range Queries

Range queries usually receive as parameters the query object Q and a radius r and retrieve the objects that lie within a distance less or equal than r from Q. However, in the AH^{+}-tree, the range query algorithm additionally receives the parameter aff which denotes the minimum affinity value the result objects have to comply with.

The range query algorithm works on the basis that $alist(Q)$ dictates the object id's the algorithm has to look for. $ObjectToNode$ tells the algorithm in which leave nodes the object id's are stored, and $NodeTrack$ how to traverse the tree to reach the desired leave nodes. Going down the tree, the algorithm only visits routing objects that both comply with the query Q and whose id's lead to leaf nodes that contain objects listed in $alist(Q)$. As in the AH-Tree [4], a node complies with Q if the distance between the node and Q is less or equal that the sum of the radius of Q and the radius of the node. Upon reaching a leave node, all the entries are checked and those that lie within the query's required minimum radius and have an affinity higher or equal that specified by the query are added to the result set. The algorithm for range queries is shown in Algorithms 1, 2, and 3.

The algorithm for range search starts in Algorithm 1, which begins by setting the current node to the root node (step 1) and locating the affinity list that corresponds to the query parameter Q (step 2). If Q does not have an affinity with any object in the database (step 3), the search is defaulted to the range algorithm of the M-tree [6] (step 4); otherwise, the array NP is obtained by invoking Algorithm 2 (step 8) and the search process is delegated to Algorithm 3 (step 9). The array NP serves to identify the id's of the nodes in the tree that lead to objects that comply with the query's affinity requirement.

Algorithm 1 RangeSearch

```
 1: Set $N = ROOT$ // set the root node as the starting point
 2: Locate $alist(Q)$
 3: if $|alist(Q)| == 0$ then // the list is empty
 4:     Perform search as in the M-tree
 5:     return $Result$
 6: end if
 7: if $|alist(Q)| \geq 0$ then // the list is non-empty
 8:     $NP = ObtainNodePath(alist(Q))$ // mark in NP the nodes that will be visited
 9:     return $InternalSearch(N, Q, r(Q), aff, NP)$ // begin range search on the root node
10: end if
```

Algorithm 2 returns an array NP with size M (refer to Section III-C for the value of M). The array NP is constructed from $alist(Q)$, $ObjectToNode$, and $NodeTrack$ and serves to identify the nodes that lead to objects listed in $alist(Q)$. Holding the highest affinity with respect to Q, O_q (the first object in $alist(Q)$) is obtained in step 2, and steps 3-5 mark in NP the id of the leave node that contains O_q. Steps 7-10 utilize $NodeTrack$ to mark in NP all the nodes that make up the path from the root node to the leave node containing O_q. Then, for all the remaining objects in $alist(Q)$ that comply with the affinity requirement (steps 11-14), the nodes in the tree that lead to such objects are marked in NP. Step 15 obtains the id of each object, and steps 16-19 check that the leave node has not been already marked in NP. Steps 21-28 mark in NP the nodes that lead to the currently selected object in $alist(Q)$, but to avoid re-marking nodes, steps 23-25 stop the marking process for the current node if the latter has already been marked. When all the nodes in the tree that lead to all the objects in $alist(Q)$ that comply with the query's affinity requirement have been marked, NP is returned in step 30.

With regards to number of iterations, the cost of Algorithm 2 is negligible as it only makes use of small arrays that are easily loaded into memory. It marks NP as many times as nodes need to be searched in the tree in order to find the objects listed in $alist(Q)$; only needs to work with $ObjectToNode$ and $NodeTrack$, which only need to be created when the tree is constructed; and returns an array of size M, which is 0.5 MB for 1,000,000 images.

Once all the nodes the tree that lead to objects that comply with the query's affinity requirement are marked in NP, Algorithm 3 carries out the range search starting at the root node. If the current node is an internal node (step 1), the algorithm loops through the node's entries (step 2) and only selects the entries that both comply with Q and point to a node that has been marked in NP (step 3). The algorithm is called recursively on the selected nodes (step 4). If the current node is a leaf node (step 7), the algorithm loops through all the objects contained in the node and only selects those that both comply with the query Q and have an affinity with Q higher or equal than aff (step 9). Selected objects are added to the result set (step 10).

Compared with the AH-Tree's range query without affinity promotion, the extra cost added by the AH^{+}-tree's algorithms for range search consists of Algorithm 2. However, Algorithm 2 still has the same asymptotic cost of the AH-Tree because the former makes as many iterations as the minimum number of nodes that must be visited in the tree in order to retrieve all the objects that comply with the query, and these iterations are carried out every efficiently.

V. Nearest Neighbor Queries

Nearest neighbor queries, also named kNN, usually receive as parameters a query object Q and a value k and retrieve the k objects in the database closest to the query object. Closeness is defined by a given distance function. As with range queries, the kNN queries of the AH^{+}-tree

Algorithm 2 ObtainNodePath

```
1: Create $NP$ with $|NP| = M$ and all values = false
2: Let $O_q$ be the object at the top of $alist(Q)$
3: Let $i$ be the id of $O_q$
4: Set $id = ObjectToNode[i]$ // $id$ is the id of the leaf node that holds $O_q$
5: Set $NP[id] =$ true// mark id in $NP$
6: Set $id = NodeTrack[id]$ // $id$ now is the id of the parent node of the leaf node that holds $O_q$
7: while $id \neq 0$ do // loop through $NodeTrack$ to mark all the nodes that lead to the leaf node holding $O_q$
8:     $NP[id] =$ true
9:     $id = NodeTrack[id]$
10: end while
11: for all $(aff_i, id_i) \in alist(Q)$ do // iterate through the remaining elements
12:     Break the loop if $aff_i < aff$
13:     $id = ObjectToNode[id]$
14:     if $NP[id] ==$ false then
15:         $NP[id] = true$
16:     else // if node was already marked, then continue to next element in the list
17:         continue
18:     end if
19:     $id = NodeTrack[id]$
20:     while $id \neq 0$ do // mark the nodes that lead to the current element in $alist(Q)$, stop if a node has already been marked
21:         Break the loop if $NP[id] ==$ true
22:         $NP[id] =$ true
23:         $id = NP[id]$
24:     end while
25: end for
26: return $NP$
```

also receive the parameter aff. Moreover, $alist(Q)$ has the same purpose as in with range queries. The algorithm for kNN queries is shown in Algorithms 4 and 5. It utilizes a branch-and-bound technique [4] that makes use of a priority queue PR and a k-element array NN. PR holds pointers to sub-trees where qualifying objects can be found, and NN holds nearest neighbors obtained from the visited sub-trees and at the end of execution contains the k nearest neighbors of Q.

In Algorithm 4, initially PR is initialized with the root node of the tree (step 1), and the search radius is ∞ (step 2). The affinity linked list for Q is obtained in step 3, and $ObtainNodePath$ (Algorithm 2) is called on step 4 to generate the array NP. Then, while there are nodes in the tree where nearest neighbors can be found (step 5), the node N in PR such that N has the smallest d_{min} (which is defined as the minimum distance from any object in the sub-

Algorithm 3 InternalSearch

```
1: if $N$ is an internal node then
2:     for all $O_r \in N$ do // loop through $N$'s entries
3:         if $d(O_r, Q) \leq r(Q) + r(O_r)$ and $NP[e(O_r).id_{node}] == true$ then // the entry complies with the query's radius and has been marked in $NP$
4:             $InternalSearch(e(O_r).ptr, Q, r(Q), aff, NP)$ // make recursive call
5:         end if
6:     end for
7: else // the node is a leaf node
8:     for all $O \in N$ do // loop through the objects in $N$
9:         if $d(O, Q) \leq r(Q)$ and $aff(O, Q) \geq aff$ then // the object complies with both the query's radius and affinity requirements
10:            $Result = Result \bigcup oid(O)$ // add object to the result set
11:        end if
12:    end for
13: end if
```

tree to Q) is selected (step 6) and removed from PR (step 7). The chosen N is passed as a parameter to Algorithm 5 (step 8) which performs the kNN search on N.

Algorithm 4 kNNSearch

```
1: Initialize $PR$ with the root node
2: Set $r(Q) = \infty$ // the search radius
3: Locate $alist(Q)$
4: $NP = ObtainNodePath(alist(Q), aff)$ // mark in $NP$ the nodes that will be visited
5: while $PR$ is not empty do // there are nodes where nearest neighbors can be found
6:     $NextNode \leftarrow$ choose node $N$ such that $N$ is the node from $PR$ that has the minimum $d_{min}$ (i.e. the first node in $PR$)
7:     Remove $NextNode$ from $PR$
8:     $NodeSearch(NextNode, Q, k, aff, NP)$ // delegate the search to $NodeSearch$
9: end while
```

Algorithm 5 performs the search on the given node N and updates the queue PR and the array NN. For internal an internal node (step 2), the algorithm loops through the node's entries (step 3) and only selects (step 4) the entries that comply with Q and point nodes that are marked in NP. Step 5 utilizes the triangular inequality property of the metric space to avoid making unnecessary distance computations; if $|d(O_p, Q) - d(O_r, O_p)|$ is not less than $\leq r(Q) + r(O_r)$, then there is not need to compute the distance between O_r and Q. It is important to clarify that $d(O_r, O_p)$ is stored in $e(O_r)$ as $e(O_r).d$. Step 13 also makes use of this property.

For leave nodes (step 11), objects that comply with Q and have an affinity with Q higher or equal than aff (step 13) are orderly inserted in NN (step 15). The current minimum search distance is updated in step 16.

Algorithm 5 NodeSearch

```
1:  Let $O_p$ be the parent key of $N$ // *needed to avoid computing unnecessary distance computations*
2:  **if** $N$ is an internal node **then**
3:      **for all** $O_r$ in $N$ **do** // *loop through $N$'s entries*
4:          **if** $|d(O_p, Q) - d(O_r, O_p)| \leq r(Q) + r(O_r)$ and $NP[e(O_p).id_{node}] ==$ **true then** // *the entry is within the current search radius and has been marked in $NP$*
5:              Compute $d(O_r, Q)$
6:              **if** $d_{min}(e(O_r).ptr) \leq d_k$ **then**
7:                  Add the node $e(O_r).ptr$ to $PR$ // *add node referenced by $e(O_r)$ to $PR$ so that it is later used in the search*
8:              **end if**
9:          **end if**
10:     **end for**
11: **else**
12:     **for all** $O$ in $N$ **do** // *loop through the objects of $N$*
13:         **if** $|d(Op, Q) - d(O, Op)| \leq r(Q)$ and $aff(O, Q) \geq aff$ **then** // *the object complies with both the current search radius and the query's affinity requirement*
14:             Compute $d(O, Q)$
15:             Orderly insert $(O, d(O, Q), oid(O))$ into $NN$
16:             Set $r(Q)$ = k$^{th}$ distance in $NN$ // *update current search radius*
17:         **end if**
18:     **end for**
19: **end if**
```

VI. Experiments

This section provides experiments for the proposed AH$^+$-tree index structure. The work in [4] already presented several experiments on the accuracy of the AH-Tree and proved it is superior to other distance-based index methods. Therefore, this section focuses on experiments that address the improved efficiency of the AH$^+$-tree. The experiments compare the AH$^+$-tree, the AH-Tree, and the M-tree with regards to number of I/O read operations during range and kNN queries. The M-tree is included in the experiments as the AH-Tree stems from the M-tree.

Two data sets were utilized in the experiments: set D_1 and set D_2. Set D_1 consists of 10,000 images from the Corel data set, which is the same set of images utilized in [4]. The elements consist of feature vectors in a 12-dimensional feature space. Extracted from the HSV color space, the features are are "black", "white", "red", "red-yellow", "yellow", "yellow-green", "green", "green-blue", "blue", "blue-purple", "purple", and "purple-red". Set D_2 consists of 1,000,000 elements, but, since an image data set so large was not available, the 12-dimensional vectors of D_2 were generated randomly. The affinity relationship for D_1 was obtained using a CBIR system that captures user perception using the MMM model, and for D_2 the affinity relationships was generated randomly. It is worth pointing out that the chosen number of features is irrelevant in these experiments since a larger or lower number of features would have exactly the same impact on the AH$^+$-tree and the AH-Tree as their difference lies solely in the handling of the affinity relationship.

The environment on which the experiments were carried out was an iMac system running Mac OS X 10.6.8 with a 2.7 GHz Inter Core i5 and 8 GB of main memory. With regards to implementation, the same framework employed in [4] was utilized to develop the AH$^+$-tree.

A. Experimental Results on Dataset D_1

Figure 2 depicts the number of I/O reads performed on set D_1 for a range query using 0.3 as minimum radius and 0.005 minimum affinity. These two values were chosen at random. As can be seen in the figure, the AH$^+$-tree carries out significantly less number of I/O reads. The AH-Tree performs a constant amount of I/O reads more than the AH$^+$-tree as the former visits the same number of nodes as the latter plus the total number of nodes in the tree due to affinity propagation. The M-tree performs better than the AH-Tree as the former does not have to traverse the entire tree for the chosen query parameters.

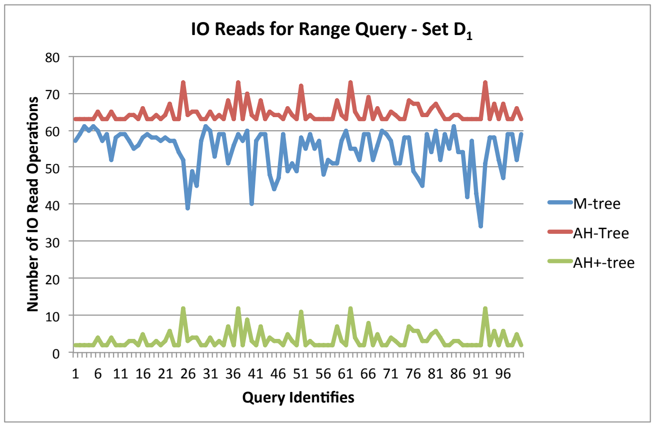

Figure 2. Number of I/O read operations using range queries with radius 0.3 and affinity 0.005

Figure 3 depicts the number of I/O reads for set D_1 for kNN queries with minimum affinity 0.005 and k = 10. The value for the affinity was chosen at random. Overall, the AH$^+$-tree performs better than the M-tree. For some queries, e.g., 64 and 67, both the M-tree and the AH$^+$-tree perform the same number of I/O reads; and for queries 2, 55, 71, and 87, the AH$^+$-tree performed slightly more I/O reads than the M-tree. The reason for these two cases is that these queries have mostly zeroes in their feature values except for two or three features with values higher than zero, and having so few features with values higher than zero causes these queries to have their closest objects in the same node or nearby. These cases are outliers.

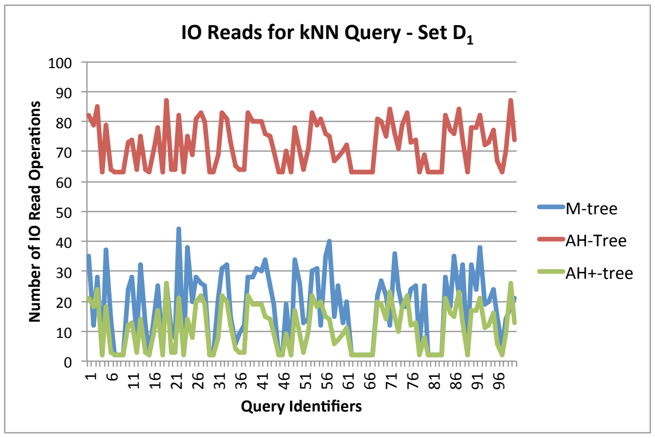

Figure 3. Number of I/O read operations using kNN queries with affinity 0.005 and k = 10

B. Experimental Results on Dataset D_2

On a logarithmic scale, Figure 4 shows the number of I/O reads for range queries on set D_2 using minimum affinity 0.005 and search radius 0.5. Both values were selected at random. The AH$^+$-tree performs significantly better than both the AH-Tree and the M-tree. Over the 100 queries, the average number of I/O reads to the AH-Tree was 19.3, for the AH-Tree 7,719.3, and for the M-tree 3,101.18.

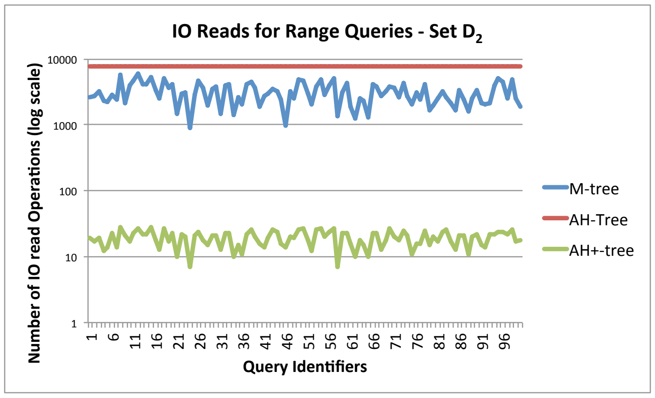

Figure 4. Number of I/O read operations using range queries with radius 0.5 and affinity 0.005

On the same scale, Figure 5 depicts the number of I/O reads performed on set D_2 for kNN queries using 0.005 minimum affinity and k = 10. Echoing the results of the range query, the AH$^+$-tree outperformed both the AH-Tree and the M-tree. Over the 100 queries, the average number of I/O reads for the AH$^+$-tree was 54.83, for the M-tree 2,210.79, and for the AH-Tree 7,754.83.

On both figures, the AH$^+$-tree's huge reduction in I/O reads is driven by the proposed utilization of the affinity relationship during the retrieval process. The values for the AH-Tree appear to be constant only because the number of nodes in the tree is much larger than the tree for set D_1.

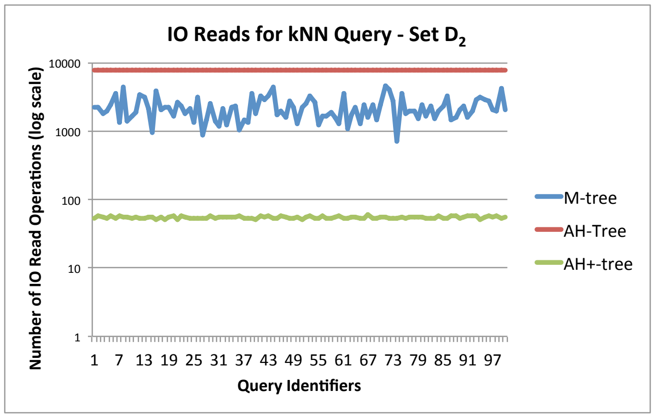

Figure 5. Number of I/O read operations using kNN queries with affinity 0.005 and k = 10

The increased relevance in the results obtained when using the high-level image relationships during the retrieval process has been demonstrated in [4]. The affinity relationship helps tackle the problems of semantic gap and user subjectivity, and thus, helps produce results more relevant to the users.

VII. Related Work

Multimedia data require multidimensional index structures for effective organization and subsequent retrieval. Multidimensional index structures can be categorized into feature-based and index-based. For feature-based index structures, the feature space is partitioned based on the values of the feature vectors along each dimension. Popular feature-based index structures are KDB-tree [11] and R-tree [12]. For distance-based index structures, the feature space is partitioned based on the distances (similarity) between pairs of data points. Thus, only the relative distances between data objects are considered.

Each type of index structure can be further sub-divided as data-partitioned or space-partitioned. For space-partitioned, the multidimensional feature space is recursively partitioned into disjoint sub-spaces which are represented as a hierarchical tree structure. Some common space-partitioned index structures are [13][6]. For data-partitioned index structures, the bounding regions are arranged in a spatial hierarchy in a containment relation. The common data-partitioned index structures are [14][11], etc. As pointed out earlier in this paper, the feature-based index structures have a severe drawback when it comes to handling multimedia data. To introduce data object-level similarity information into the index structure, feature-level index structures require that such similarity be translated into individual feature-level weights. Such approach is quite difficult, given the two important characteristics of multimedia data: namely the semantic gap and the perception subjectivity. Thus, distance-based index structures can be considered as a more natural choice when multimedia data are concerned. They allow the utilization of any representation between data objects during accessing the index structure. Some popular distance-based index structures are the M-Index [15], M-tree [6], and the vp-tree [5]. Nevertheless, none of these distance-based index structures addresses the problems of semantic gap and user subjectivity as they only rely on the distance function which may not map well to high-level concepts.

VIII. Conclusion

This paper has presented the AH$^+$-tree, a distance-based, balanced tree structure that allows CBIR for multimedia objects through similarity searches. The AH$^+$-tree addresses the issues of semantic gap and user subjectivity during the retrieval process by incorporating high-level affinity relationships between multimedia objects. The AH$^+$-tree utilizes the high-level affinity in a novel way to address the AH-Tree's I/O overhead and provide a significantly more efficient indexing mechanism. The structure of the AH$^+$-tree efficiently stores the high-level affinity information and provides a mechanism to keep track of object-to-node and node-to-node relationships, which, along with the affinity information, allow the retrieval algorithms to efficiently prune nodes in the tree that do not lead to relevant objects in the leaves. Supporting the rationale behind the AH$^+$-tree, the improved efficiency of the AH$^+$-tree over the AH-Tree and the M-Tree is demonstrated in the experiments. The ability of the AH$^+$-tree to incorporate high-level affinity information from different models to tackle the problems inherent to multimedia retrieval while maintaining I/O efficiency make the AH$^+$-tree a promising index mechanism for large multimedia databases. For future work, the AH$^+$-tree will be extended to index both images and videos.

Acknowledgment

This research was supported in part by NSF OISE-0730065 and HRD-0833093.

References

[1] Facebook, http://www.facebook.com/, may 2010.

[2] A. Yoshitaka and T. Ichikawa, "A survey on content-based retrieval for multimedia databases," *IEEE Transactions on Knowledge and Data Engineering*, no. 1, pp. 81–93, 1999.

[3] M.-L. Shyu, S.-C. Chen, Q. Sun, and H. Yu, "Overview and future trends of multimedia research for content access and distribution," *International Journal of Semantic Computing (IJSC)*, vol. 1, no. 1, pp. 29–66, Mar. 2007.

[4] K. Chatterjee and S.-C. Chen, "Affinity hybrid tree: An indexing technique for content-based image retrieval in multimedia databases," in *Proceedings of the IEEE International Symposium on Multimedia (ISM2006)*, San Diego, CA, USA, Dec. 2006, pp. 47–54.

[5] P. N. Yianilos, "Data structures and algorithms for nearest neighbor search in general metric spaces," in *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, Austin, Texas, 1993, pp. 311–321.

[6] P. Ciaccia, M. Patella, and P. Zezula, "M-tree: An efficient access method for similarity search in metric spaces," in *Proceedings of the 23rd International Conference on Very Large Data Bases*, 1997, pp. 47–54.

[7] M.-L. Shyu, S.-C. Chen, M. Chen, C. Zhang, , and C.-M. Shu, "MMM: A stochastic mechanism for image database queries," in *Proceedings of the IEEE Fifth International Symposium on Multimedia Software Engineering (MSE2003)*, Taiwan, Dec. 2003, pp. 188–195.

[8] H.-J. Zhang, "Relevance feedback in content-based image search," in *Proceedings of the 1st International Workshop on Pattern Recognition in Information Systems: In conjunction with ICEIS 2001*. ICEIS Press, 2001, pp. 29–31.

[9] D. Wang and X. Ma, "Multimedia data mining for building rule-based image retrieval systems," in *IEEE International Conference on Multimedia and Expo, 2005. ICME 2005*, oct 2005, pp. 197–200.

[10] O. R. Zaïane, J. Han, Z.-N. Li, and J. Hou, "Mining multimedia data," in *CASCON '98: Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 1998, pp. 1–18.

[11] J. T. Robinson, "The k-d-b-tree: a search structure for large multidimensional dynamic indexes," in *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*, Ann Arbor, Michigan, 1981, pp. 10–18.

[12] A. Guttman, "R-trees: a dynamic index structure for spatial searching," in *Proceedings of the 1984 ACM SIGMOD International Conference on Management of data*, Boston, Massachusetts, 1984, pp. 47–57.

[13] S. Berchtold, D. A. Keim, and H. Kriegel, "The x-tree: An index structure for high-dimensional data," in *Proceedings of the 22th International Conference on Very Large Data Bases*, 1996, pp. 28–39.

[14] D. B. Lomet and B. Salzberg, "A survey on content-based retrieval for multimedia databases," *IEEE Transactions on Knowledge and Data Engineering*, no. 4, pp. 625–658, 1990.

[15] M. Novak, D; Batko, "Metrix index: An efficient and scalable solution for similarity searches," in *International Workshop on Similarity Search and Applications (SISAP'09)*, 2009, pp. 65–73.