OPTIMALLY ADAPTED INDEXING TREES FOR MEDICAL IMAGE DATABASES

Xiaoning Qian  Hemant D. Tagare

Dept. of Diagnostic Radiology, Yale University, New Haven CT 06520
Dept. of Electrical Engineering, Yale University, New Haven CT 06520

ABSTRACT

Medical image databases often have high dimensional features and standard indexing trees do not perform well with them. In this paper, we propose an algorithm for eliminating nodes in the indexing tree so that the resulting tree has improved performance. The algorithm (provably) provides the least cost tree among all possible trees that can be generated by eliminating nodes. Experimental data about performance improvement in simulations and real world are provided.

1. INTRODUCTION

Similarity retrieval (also called Content Based Image Retrieval (CBIR)) is often proposed as a search mechanism for medical image databases. The idea is to retrieve all images in a database that have a feature \( y \) which satisfies:

\[
S(x, y) \leq T,
\]

where \( x \) is a feature from an example image, \( S \) is a (dis-)similarity measure and \( T \) is a user-defined threshold. Typically, features are elements of a high-dimensional Euclidean space, and \( S \) is one of the \( l_p \)-norms. Similarity retrieval is possible by a logarithmic search through the database when indexing trees are used.

Feature vectors in image databases have unusual characteristics - they belong to a high dimensional space, but within that space they tend to be distributed along lower-dimensional submanifolds. It is not clear whether standard indexing trees are efficient when used with such data. In practice one observes that nodes near the root of these trees survive most of the queries and are computationally wasteful.

In this paper we propose a technique which adapts the indexing tree so that wasteful nodes are eliminated. We start with a standard tree and increase its efficiency by eliminating nodes from it. We measure the performance of the resulting tree by the average computation required to satisfy similarity queries. We propose a recursive procedure for obtaining a node-eliminated tree with the least average computation. Finally, we present experimental results that measure the performance of the optimal tree in simulated and real-world data sets.

In the context of similarity retrieval other researchers have addressed range queries and k Nearest Neighbor queries. The Nearest Neighbor queries can be easily transformed into range queries [4, 5]. The performance of indexing trees for range queries degrades in high dimensional spaces. This is the infamous "curse of dimensionality". Recently, several indexing trees (SS-tree[6], X-tree[7], M-tree[8], etc.) have proposed for ameliorating the problem, but much remains to be done. One important factor that affects indexing efficiency is the "intrinsic dimension" of the distribution of feature vectors in the database [2, 3]. This has motivated research ([1, 3, 9]) on dimension reduction in indexing structures.

Our research is also motivated by this. But instead of forcing a low-dimensional structure on the feature distribution, we adapt the indexing tree so it fits the distribution. Further, we have an explicit criterion with which to judge the fit - the average computational cost in the tree during similarity retrieval.

2. BACKGROUND

Our results apply to trees that recursively partition the feature space (kd-trees, R-trees etc.) and also to cluster trees. For simplicity we will only consider cluster trees here. Each node of a cluster tree represents a sphere (in the appropriate norm) in the feature space, and the parent of a node represents a larger sphere within which the sphere at the current node is contained. The root node in a cluster tree represents the entire space and the leaf node contains data.

The retrieval procedure traverses the tree from the root to the leaf nodes by applying a node test. The node test checks whether the volume of the feature space represented by the node intersects the volume defined by equation (1). If it does, then the procedure applies the node test to its children. If it does not, then we know that the answer cannot be contained in any node belonging to the subtree rooted at this node, and the entire subtree is rejected. The leaf nodes that survive this procedure are guaranteed to contain all \( y \) that satisfy equation (1).

We can measure the performance of a tree by the average number of node tests carried out in it over a large set of queries. We refer to this as the computational cost of the indexing tree. We can also measure the average number of node tests in any subtree and refer to this as the computational cost of the subtree. The computational cost \( Q(\alpha) \) of the subtree rooted at any node \( \alpha \) of the indexing tree is [10]:

\[
Q(\alpha) = \begin{cases} 
1 & \text{if } \alpha \text{ is not a leaf node,} \\
\sum_{\beta \in C(\alpha)} (1 + p_{\beta/\alpha} Q(\beta)) & \text{otherwise,}
\end{cases}
\]

where, \( C(\alpha) \) is the set of children of \( \alpha \), and \( p_{\beta/\alpha} \) is the conditional probability that the child \( \beta \) passes the node test given that \( \alpha \) passed the node test. If \( p \) is the root node of the tree, then \( Q(p) \) is the computational cost of the whole indexing tree.

Indexing trees get their efficiency from nodes that do not pass the node test. Any node that passes the node test frequently is not useful in an indexing tree and it may well be that we can improve
the performance of the indexing tree by eliminating it. The process of eliminating the node is illustrated in Figure 1 where the node  \( \mu \) is eliminated and its children are attached to its parent node. We call this process node elimination and apply it to nodes other than the root or leaf nodes. By eliminating nodes that are inefficient we hope to lower the computational cost of the indexing tree.

Our goal is to find the tree with the smallest computational cost possible after node elimination. In Section 4, we report the procedure for finding this tree. But before we do that, we need some notation and observations.

3. NOTATION

We begin by developing a notation to be used with node elimination. Let  \( \alpha \) be a node in an original indexing tree  \( T \). We can modify the subtree rooted at  \( \alpha \) by eliminating nodes in it (by definition of node elimination we can eliminate any node except  \( \alpha \) or the leaf nodes). We let  \( T_\alpha \) be the set of all subtrees obtained by eliminating nodes from the subtree rooted at  \( \alpha \). We allow for zero eliminations, so the original subtree is contained in  \( T_\alpha \).

We denote any property of a specific tree  \( T_\alpha \in T_\alpha \) by adding  \( T_\alpha \) as a subscript. Thus, the computational cost of the subtree  \( T_\alpha \in T_\alpha \) is  \( Q(T_\alpha) \) and the children of  \( \alpha \) are  \( C(T_\alpha) \).

For a node  \( \alpha \), the set  \( V_\alpha = \cup T_\alpha \in T_\alpha C(T_\alpha) \) is the collection of all possible children of  \( \alpha \) due to node elimination. If  \( \alpha \) is a leaf node, then  \( V_\alpha \) is empty. Any element  \( C \in V_\alpha \) is a set of nodes  \( C = \{ \beta_1, \ldots, \beta_n \} \). We can also partition  \( C \) into disjoint subsets  \( C_i, i = 1, \ldots, k \) where the elements of each  \( C_i \) come from the subtree in the original tree rooted at the  \( i \)th child of  \( \alpha \). We call this a \textit{canonic} partition of  \( C \).

Any element  \( T_\alpha \in T_\alpha \) is a valid indexing sub-tree rooted at  \( \alpha \). The computational cost of the subtree from any node  \( \mu \in T_\alpha \) is  \( Q(T_\mu) \).

\[
Q(T_\mu) = \begin{cases} 
\sum_{\beta \in C(T_\mu)} (1 + p_{\beta/\mu} Q(T_\beta)) & \text{if } \mu \neq \text{leaf}, \\
1 & \text{otherwise}.
\end{cases}
\]  

(3)

This equation is the same as (2), but with the appropriate subscripts added to  \( Q \). The computational cost has an interesting property -  \( Q(T_\mu) \) depends only on the subtree in  \( T_\mu \) rooted at  \( \mu \) and is independent of all other nodes. We refer to this as the \textit{independence property} of the computational cost. Furthermore, if we alter the subtree rooted at  \( \mu \) keeping the root of the nodes the same, the total computational cost increases or decreases according to increase and decrease in the cost of the altered sub-tree.

Since equation (3) holds for  \( \alpha \), we define

\[
Q^{\text{min}}(\alpha) = \min_{T_\mu \in T_\alpha} Q(T_\mu), \quad \text{and} \quad T^{\text{min}}(\alpha) = \arg\min_{T_\mu \in T_\alpha} Q(T_\mu),
\]

to be respectively the minimum computational cost of subtrees in  \( T_\alpha \) and the tree that achieves the minimum. If  \( \rho \) is the root node then  \( Q^{\text{min}}(\rho) \) and  \( T^{\text{min}}(\rho) \) are the minimum computational cost that can be achieved by trees in  \( T_\alpha \) and the tree that achieves the minimum.

Immediately below, we find the minimizing tree  \( T^{\text{min}}(\alpha) \). Our formulation gives the minimizing tree as a set of subtrees  \( T_1, \ldots, T_k \) which occur immediately below the root node  \( \alpha \). The notation \( \alpha \{ T_1, \ldots, T_k \} \) specifies this tree - it is a tree with  \( \alpha \) as the root node and \( T_1, \ldots, T_k \) as subtrees at depth 1 from  \( \alpha \).

Finally, a comment: if a node of an indexing tree never passes the node test, then the subtree rooted at the node may be completely eliminated without affecting the performance of the tree. We assume that we do not have such nodes in the tree. That is, if  \( \beta \) is any node in a subtree rooted at  \( \gamma \), then  \( p_{\beta/\gamma} \), the probability that  \( \beta \) passes the node test given that  \( \gamma \) did, is greater than 0.

4. FINDING THE OPTIMAL TREE

The recursive form of equation (3) makes it possible to find  \( Q^{\text{min}}(\rho) \) and  \( T^{\text{min}}(\rho) \) by dynamic programming. The following result is key:

**Proposition 1:** If  \( \alpha \) is any node in  \( T^{\text{min}}(\rho) \), then

\[
Q^{\text{min}}(\alpha) = \min_{T_\mu \in T_\alpha} Q(T_\mu) = \min_{C \in V_\alpha} \sum_{\beta \in C} (1 + p_{\beta/\mu} Q^{\text{min}}(\beta)),
\]  

(4)

\[
T^{\text{min}}(\alpha) = \bigcup_{C \in V_\alpha} \{ T^{\text{min}}(C_1, \ldots, C_k) : C = \{ \beta_1, \ldots, \beta_n \} \}
\]  

(5)

where,  \( C_{\text{min}} = \bigcup_{C \in V_\alpha} C_\text{min} \) is the canonical partition of the minimizing  \( C \) in (4), and  \( T^{\text{min}}(C_{\text{min}}) \) are the minimum cost subtrees rooted at the elements of  \( C_{\text{min}} \).

**Proof:** The proof is by contradiction. Let  \( T^* \neq T^{\text{min}}(\rho) \), with  \( T^{\text{min}}(\rho) \) as defined above. Comparing  \( T^* \) and  \( T^{\text{min}}(\rho) \) in a breadth-first fashion, let  \( \gamma \) be the first node at which either of the following two conditions hold: either the set of children of  \( \gamma \) in  \( T^* \) or the subtrees rooted at the children of  \( \gamma \) in  \( T^* \) are not the same as in  \( T^{\text{min}}(\rho) \). Recall that if  \( \beta \) is any node in a subtree at  \( \gamma \), then  \( p_{\beta/\gamma} > 0 \). Thus,

\[
Q(T^*(\gamma)) = \sum_{C \in V^*_\gamma} (1 + p_{\beta/\gamma} Q(T^*(\beta))) \geq \sum_{C \in V_{\text{min}}} (1 + p_{\beta/\mu} Q^{\text{min}}(\beta)) \geq \min_{C \in V_\alpha} \sum_{\beta \in C} (1 + p_{\beta/\mu} Q^{\text{min}}(\beta)).
\]

The first inequality follows from the minimality of  \( Q^{\text{min}}(\beta) \) and, for  \( p_{\beta/\gamma} > 0 \) becomes an equality only when the subtrees rooted at the children of  \( \beta \) in  \( T^* \) minimizing subtrees. The last inequality follows from the fact that  \( Q(T^*(\gamma)) \) is  \( V^*_\gamma \), and for  \( p_{\beta/\gamma} > 0 \) it becomes an equality when  \( C_{\text{min}} \) is the minimizing set of children. By our assumption, the two conditions cannot hold simultaneously, hence at least one of the inequalities is strict. Therefore  \( Q(T^*(\gamma)) \) is not the smallest it can be, and we can lower the cost of  \( T^* \) by replacing the subtree at  \( \gamma \) with the corresponding subtree in  \( T^{\text{min}}(\rho) \) (recall the discussion about the independence property of  \( Q \)). This is a contradiction. End of proof.

The algorithm for finding the minimum cost tree uses Proposition 1 and proceeds from the leaves to the root, at each depth calculating the optimal subtrees using optimal subtrees from previous depths. The optimal subtree at a leaf node is the leaf node itself (since  \( V_\emptyset = \emptyset \) for all leaves):

**Outer loop:**

1. From leaves till root, for each depth level:
2. For each node  \( \alpha \) at current depth:
3. Find the minimum cost tree by
searching for the minimum according to (4) and (5).

The minimization in step 3 of the algorithm can be simplified using the independence property of the computational cost. If \( C = \{ \beta_1, \ldots, \beta_m \} \in V_k \), then the \( \beta_i \)'s are nodes in the subtree rooted at \( \alpha \). We will say that \( C \) has a depth of \( k \) if the minimum depth of all \( \beta_i \) from \( \alpha \) is \( k \). Let \( U_k \) be the set of all \( \beta_i \)'s in \( V_k \) which are at depth \( k \) or more. Then, \( U_k = \prod_{i=1}^{m} \{ \beta_i, V_{\beta_i} \} \), where \( \beta_i = 1, \ldots, m \) are all of the nodes at depth \( k \) in the subtree rooted at \( \alpha \). Also \( U_{k+1} = \prod_{i=1}^{m} V_{\beta_i} \), and \( V_\alpha = U_1 \subset U_2 \subset U_3 \ldots \subset U_n \), where \( n \) is the total depth of the subtree.

We now seek the minimum

\[
\min_{C \in U_k} \sum_{\beta \in C} 1 + p_{\beta_{\alpha}} Q^{\min}(\beta).
\]

The minimization in (4) corresponds to setting \( k = 1 \). Letting \( J(X, \alpha) = \sum_{\beta \in C} 1 + p_{\beta_{\alpha}} Q^{\min}(\beta) \), and noting that the canonical partition of \( C \) must have \( m \) components \( C_i, i = 1, \ldots, m \), because \( C \in U_k = \prod_{i=1}^{m} \{ \beta_i, V_{\beta_i} \} \), we get:

**Proposition 2:**

\[
\min_{C \in U_k} J(C, \alpha) = \sum_{i=1}^{m} \min_{C_i \in \{ \beta_i, V_{\beta_i} \}} J(C_i, \alpha). \tag{6}
\]

**Proof:**

\[
\min_{C \in U_k} J(C, \alpha) = \sum_{i=1}^{m} \min_{C_i \in \{ \beta_i, V_{\beta_i} \}} J(C_i, \alpha) = \sum_{i=1}^{m} \sum_{i=1}^{m} J(C_i, \alpha). \tag{7}
\]

The last step is due to the following: For \( i \neq j \), the variables \( C_i \) and \( C_j \) lie in the two non-intersecting subtrees rooted at \( \beta_i \) and \( \beta_j \) and \( J(C_i, \alpha) \) and \( J(C_j, \alpha) \) are the computational costs of these trees. By the independence property of the cost, \( C_i \) has no effect on \( J(C_j, \alpha) \) and \( C_j \) has no effect on \( J(C_i, \alpha) \). Thus each can be minimized independently. End of Proof.

Further, since \( U_{k+1} = \prod_{i=1}^{m} V_{\beta_i} \), we have

\[
\min_{C \in U_{k+1}} J(C, \alpha) = \sum_{i=1}^{m} \min_{C_i \in \{ \beta_i, V_{\beta_i} \}} J(C_i, \alpha). \tag{8}
\]

By Proposition 2, if \( C_{\min}^{k+1} \) is the minimizing \( C \) in (8), then \( C_{\min}^{k+1} = U_{k+1} \arg \min_{C \in U_{k+1}} J(C, \alpha) \).

Now, if \( C_{\min}^k \) is the minimizing element in (7), then

\[
C_{\min}^k = U_{k} \arg \min_{C_i \in \{ \beta_i, V_{\beta_i} \}} J(C_i, \alpha).
\]

The search space for minimization with respect to \( C_k \) is the union of \( \beta_i \) and \( V_{\beta_i} \). But according to (8), the minimization with respect to \( C_k \) restricted to \( V_{\beta_i} \) is \( C_{\min}^{k+1} \). Thus, we have established:

**Proposition 3:**

\[
C_{\min}^k = U_{k} \arg \min_{C_i \in \{ \beta_i, C_{\min}^{k+1} \}} J(C_i, \alpha),
\]

and we have the algorithm for the minimization in step 3 of the outer loop:

\[
\begin{array}{c|c|c|c|c|c}
\text{Table 1. Performance of the original tree and the optimal tree.} \\
\hline
\text{Average Number of Node Tests} & \text{Dim2} & \text{Orig.} & \text{Opt.} & \text{Original} & \text{Optimal} \\
\hline
0.0 & 0.0 & 75.85 & 67.46 & 1968.72 & 1301.53 \\
0.0 & 0.05 & 272.02 & 5484.99 & 3807.10 \\
0.05 & 0.5 & 70.31 & 1968.72 & 2435.49 \\
0.05 & 0.05 & 241.93 & 3848.99 & 3736.14 \\
\hline
\end{array}
\]

**5. EXPERIMENTAL RESULTS**

We performed three sets of experiments where we measured the performance of the minimal cost tree. The original tree was obtained by agglomerative clustering of feature data. The data sets were (1) Uniformly distributed in the feature space, (2) Uniformly distributed along a lower dimensional manifold in the feature space, and (3) A cardiac ultrasound image database where the features were shapes of the left ventricle. The similarity measure for the first two data sets is the Euclidean distance, and for the last data set it is the Procrustes distance used in shape spaces [11].

The algorithm for finding the optimal tree needs the conditional probability \( p_{\beta_{\alpha}} \) that a node \( \beta \) passes the node test given that its parent \( \alpha \) passed the test. We estimate this by conducting mock queries on the database. The mock queries require us to choose a threshold \( T \) (see eq. (1)) which may or may not be the threshold for any user query. We address this point below.

5.1. Uniformly Distributed Data

We generated 10,000 uniformly distributed feature points in 2 and 10 dimensional Euclidean space. The original tree was constructed by agglomerative clustering. The conditional probability was estimated by the threshold \( T \) set to \( T_{\text{est}} \). The optimal tree was constructed by the algorithm reported here. The performance of the original tree and the optimal tree was measured at \( T \) set to \( T_{\text{est}} \). To fairly compare the performance for 2 and 10 dimensional datasets, we set the thresholds to a value where nearly equal number of data was retrieved in both cases. This corresponded to \( T_{\text{est}} \) equal to 0.05 and 0.5 respectively for 2 and 10 dimensional data. We also evaluated the extreme case \( T_{\text{est}} = 0 \).

Table 1 shows the results. The first column shows the \( T \) at which the conditional densities were estimated and the second column shows the \( T \) at which the performance was evaluated. Each entry in the column is a pair of numbers. The first one applies to the 2 dimensional data set and the second one to the 10 dimensional data set.
Table 2. Performance for fixed intrinsic dimension.

<table>
<thead>
<tr>
<th>Feature Space Dimension</th>
<th>Avg. Query Time (Seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.01383</td>
</tr>
<tr>
<td>10</td>
<td>0.01358</td>
</tr>
<tr>
<td>20</td>
<td>0.01383</td>
</tr>
</tbody>
</table>

Each row in the table gives the result of one experiment. The first and the last line show the results when $T_{est} = T_{test}$. The second line shows the result for $T_{est} < T_{test}$. The third line shows the case when $T_{est} > T_{test}$. In all cases the optimal tree outperforms the original tree.

5.2. Low Intrinsic Dimension Data

In this experiment, we created data that was uniformly distributed in a 2 dimensional subspace of a 2, 10 and 20 dimensional space. The thresholds were $T_{est} = 0.0$ and $T_{test} = 0.025$. Table 2 shows the average number of node tests in the optimized trees as a fraction of the database size. Note that the average number of tests is approximately the same for all extrinsic dimensions. An examination of the optimized trees showed that the optimized tree structures are also roughly comparable. This provides evidence that the optimized tree adapts to the intrinsic dimension of the data.

5.3. Shape Data

Finally we implemented the optimization algorithm for the cardiac ultrasound database. Here too the optimal tree improves the performance. For $T_{est} = 0$ and $T_{test} = 0.1$ the optimal tree improved performance as follows: the average number of node tests in the original indexing tree (expressed as a percentage of the size of the database) was 0.205, after optimization it was 0.171.

Figure 2a shows a typical query to the database. The feature is the shape of the myocardium in left ventricle which is outlined. Three retrieved images with their distances from the query are shown in Figures 2b-d.

6. CONCLUSION

In this paper we proposed an algorithm for eliminating inefficient nodes in an indexing tree. The resulting tree is the one with the minimum number of average node tests from all the trees that can be generated by eliminating nodes from the original tree. Experimental data in support of the performance gain was also reported.

7. REFERENCES