Speeding up Queries in a Leaf Image Database

Daozheng Chen

May 10, 2007

Abstract

We have an Electronic Field Guide which contains an image database with thousands of leaf images. We have a system which can take a photo of a leaf and use this image to query the database and bring out a set of leaf images which have the most similar shape as the query leaves from different species. The existing query algorithm is based on a brute force approach, and it does not have a very attractive query processing time. After we have tried some approaches to speed up the system, we discovered that two methods using a distance matrix show some usable improvement.

Key Terms: Nearest Neighbor Query, Metric Space, Data Structures, Image Database

1. Introduction

In the project, "An Electronic Field Guide: Plant Exploration and Discovery in the 21st Century", a prototype electronic field guide for the flora of Plummers Island in the District of Columbia is constructed. The field guide integrated a novel computer vision algorithm, Inner Distance Shape Comparison (IDSC), for computing leaf similarity into an image browser that "assists a user in navigating a large collection of images to identify the species of a new specimen" [2]. One of these functions is to bring out a list of species whose leaf shape is most similar to the query leaf's shape. Given a positive number n, the system can compute the similarity value from the query to every existing leaf, and then choose the k nearest leaves which are from n different species. However, because the computation of similarity between two leaves is time consuming, the query processing time is not very attractive if the number of leaves in the database is very large.

To reduce the searching time, we notice that the similarity value of two leaves approximately satisfies the property of a metric in a metric space, so we look at similarity search methods in a metric space. Currently, there are "two alternatives when the only information that we have is the distance function." The first one is the embedded method, in which it "derived 'features' purely based on the distance function." This method can replace the original expensive distance computation function to be a much less expensive distance computation function, and it can also index the elements in the database "using multidimensional indexes thereby speeding up the search" [3]. However, we have not yet considered this method.

The other method uses "the given distance function to index the data with respect to their distance from a few selected objects," which is called a distance-based indexing method [3]. This method can reduce the number of distance computations compared with sequentially scanning the entire database, and it is the primary approach we have used so far.

In this paper, I will first describe the problem, define a metric space, and propose a general search algorithm in section 2. In section 3, I will describe the search methods in

detail. In section 4, I will show some results of the experiment using the methods described in section 4.

2. The Problem, Metric Space, and General Search Algorithm

In this section, to begin, I will give the description of the problem, then the definition of a metric space, and then a general search algorithm which is used in three of the five algorithms proposed in the following section.

The problem is that in a universal set UL of leaves from a set US of plant species with distance function D defined for any two elements in UL, given a nonempty subset L of UL from a set S of plant species, a query q from UL, and a positive integer k ($0 < k <=$ the size of S), what is the k nearest species in S with the smallest distance from q. The distance for a species **A** in S from q is defined to be the shortest distance among all distances between the query and leaves in S which belong to species A. Because the distance function d for any two leaves approximately satisfies the properties of the metric in a metric space, we decided to use methods for nearest neighbor search in metric space to approach this problem.

A metric space is a 2-tuple (S, D) where S is a set of elements, and D is function such that $D: S \times S \rightarrow R$ (the set of real numbers), where for $a, b, c \in S$,

(1) $D(a, b) \geq 0$
(2) $D(a, b) = 0$ if $a = b$
(3) $D(a, b) = D(b, a)$
(4) $D(a, b) + D(b, c) \geq D(a, c)$

The function D can be called metric, distance function, or simply distance. In our project, the current distance function between two leaves obey properties (1), (2), (3), and almost always obey (4).

Before looking at those methods in the next section, I will introduce the general idea of how to search the database based on the nearest neighbor algorithm discussed in Hjaltason and Samet's papers [1, 3]. Three of the five algorithms introduced below use this general idea. It treats the whole database as an object, we first pick a pivot out from this database and compute the distance d between the pivot and the query, the rest of the elements in the object can be viewed as one or more nonempty sets, each of which can be viewed as an object. Using the triangular inequality and the existing distance relation between the pivot and the rest of the elements, we can obtain lower bound of distances between the query and the rest of sets (objects). We then put these objects and the pivot into a priority queue, and then retrieve an element from the queue whose distance or lower bound to the query is the shortest. If it belongs to a single element in the database, then we know that the distance between the query and the rest of the elements in the database must be greater than or equal to the distance between the query and this element. So we can remove it from the queue and report it to be a nearest neighbor. If it is a set, then we conduct a similar procedure as we have done for the whole set. We continue this procedure, until we finally get a list of nearest neighbors which contains k plant species. To make this idea clear, I have provided the following pseudocode of the algorithm below:

Precondition of the algorithm:

1. k: number of top species needed to be returned. (0 < k <= the number of species in the group passed)
2. q: the query leaf
3. group: an object which refers to the whole set of leaves in the dataset. Its initial distance to query is 0.

Pseudocode

```
0 NearestSpeciesSearch(k, q, group)

1 queue ← an empty priority queue storing objects referring to a set of leaves or referring
a single leaf.
2 nearestLeafList ← an empty list storing the nearest leaves found so far
3 nearestSpeciesList ← an empty list storing the nearest species found so far
4 Enqueue(group) /* please see the precondition (3) above for meaning of group */

5 While (true)
6       E ← Dequeue(queue)
7       /* this operation returns an object whose distance or distance lower bound to
          query is the shortest in the queue */
8       If E is a single leaf
9               Put E at the end of the nearestLeafList
10              If the species type of E does not exist in nearestSpeciesList
11                      Put the species type of E at the end of the nearestSpeciesList
12              End
13              If (size(nearestSpeciesList) = k)
14                      End the algorithm and return the nearestSpeciesList
15              End
16      End
17      If E is referring to a set ES of leaves
18              pivot ← pick a leaf out from the set ES
19              dist ← compute distance between q and pivot
20              enqueue the pivot into the queue
21              partition ES/{pivot} into one or more sets of leaves. For each of these sets,
                compute its distance lower bound to the query, and put it into the queue
22      End
23 End
```

The behavior of this algorithm is determined by the following three factors:

1. It can use different schemes to choose a pivot from the set. This corresponds to step 18 in the pseudocode.
2. It can use different schemes to view the rest of the elements in the set as one or more nonempty sets. This corresponds to step 21 in the pseudocode.
3. It can use different schemes to compute distance or lower bound between the query and the set. This corresponds to step 21 in the pseudocode.

3. Search Methods

Below, I discuss five algorithms to do the nearest neighbor queries. The first one is a tree structure which organizes the database according to the distance function before performing a search. The next three are variants from the general scheme which use a distance matrix when performing the search. The last one is a raw algorithm which treats each species as an object and uses a distance matrix to do the search.

3.1. Using the Vantage Point Tree (vp-tree) – VP method

The vp-tree [4] is a data structure which organizes the dataset into a binary tree structure according to the distance function. I refer to this method as VP. When building the tree, it first selects a pivot (Yianilos terms this pivot to be vantage point) p from the whole set S of elements in the metric space (S, d), and computes its distance to all remaining elements in S. Then it partitions the set S/{p} into subsets S_1 and S_2 with roughly equal size according to the median r of those distances, that is

S_1 = {o belongs to S/{p} | d(o, q) <= r} and
S_2 = {o belongs to S/{p} | d(o, q) > r}

So for a root node in a vp-tree, it represents the whole set of elements in the metric space. Specifically, it contains the selected pivot p for partitioning the set, and the left and right children representing S_1 and S_2. After building up the root node, it recursively applies this partition scheme to S_1 and S_2 to produce the whole tree. The leaf nodes of the tree will store "one or more elements, depending on the desired capacity" [3].

3.1.1. Lower and Upper Bounds

For each internal node in the tree, it will also store the boundary values of its two children for the purpose of performing a search, that is,

for any $o \in S_1$, we have $d(q, o) \in [r_{low,1}, r_{high,1}]$
for any $o \in S_2$, $d(q, o) \in [r_{low,2}, r_{high,2}]$

The simplest way to store the boundary value is to let $r_{low,1} = 0$, $r_{high,1} = r_{low,2} = r$, and $r_{high,2}$ be ∞ (in this case, the closed bracket ']' following $r_{high,2}$ should be ')'). This allows us to store just the median r in an internal node. But to improve the search performance, we can store the tightest boundary values, "perhaps at the price of excessive storage cost" [3].

3.1.2. Pivot Selection

The simplest way to select a pivot is to randomly choose an element in the metric space which is currently being dealt with. But better schemes of choosing the pivot can yield results in "nontrivial search-time savings." One way is to choose a sample from the current metric space, evaluate each element in the symbol, and then choose an element within this sample as the vantage point based on the evaluation. The evaluation of an element is done by choosing another sample in the current metric space, "from which the median of IIp(S), and a corresponding moment are estimated" [4]. Finally, the vantage is chosen as the one with the highest moment.

3.1.3. Algorithm

To perform the nearest species search, we use the general search algorithm in section 2. Initially, we put the root node representing the whole set of leaves into the queue. When we pull out a node N from the queue, if it is a leaf node, we simply compute the distance between the query and each leaf stored in this leaf node, and put all of these leaves into the queue. If it is an internal node, we will choose its vantage point as the pivot to compute its distance to query and put it into the queue. Then we partition N into two nodes representing its two children. To compute their lower bound distance to query, we use max{d(query, pivot) – r_{hi}, r_{lo} – d(query, pivot), 0}, where d(a, b) be the distance between two leaves. This formula is correct because for an element a in N which is not the pivot, we have |d(query, pivot) – d(pivot, a)| ≤ d(query, a).
Let F = d(query, pivot) – d(pivot, a), then if F < 0, then

d(query, a) ≥ d(pivot, a) –d(query, pivot) ≥ rlo – d(query, pivot).

If F ≥ 0, then

d(query, a) ≥ d(query, pivot) – d(pivot, a) ≥ d(query, pivot) – rhi.

Also, d(query, a) ≥ 0. So d(query, a) ≥ max{d(query, pivot) – rhi, rlo – d(query, pivot), 0}

3.2. Using INN1, INN2, and INN3 with Distance Matrix

In this method, before performing the search, we first compute the distances between any two distinct elements in the database and store them in a matrix. This matrix is called a distance matrix, and it serves to build the distance relation between the pivot and other sets of elements in the database. To do the search, we address three methods which are described in Hjaltason and Samet’s 2000 technical report [1], which I refer to as INN1, INN2, and INN3.

3.2.1. INN1 and INN3

In INN1, initially an object referring to the whole set of leaves with distance to query being 0 is put into the queue, and the lower bound of distances of all leaves to the query is set to be zeros. When pulling out an element from the queue, if it is a single leaf, we report it to the nearest neighbor list. If it is the object referring to the whole set of leaves, we randomly pick a leaf as pivot, compute its distance to query, and put it into the queue. Then we use this distance to update distance lower bounds to query for the rest of the leaves in the set. To update the lower bound of a leaf L, we compute D = |d(query, pivot) – d(pivot, L)|, where d(query, pivot) is the distance between query and pivot and d(pivot, L) is the distance between pivot and L, which is stored in the distance matrix. If D is greater than the existing lower bound distance between L and the query, then the lower bound is set to be D. Otherwise, there is no update needed. Then we put an object containing the set of leaves without the pivot into the queue. Its distance lower bound to query is the smallest lower bounds of all leaves in this set. If the newly pulled out element is an object referring to a proper subset S of the whole set of leaves, we pick the

leaves with the lowest lower bound distance as pivot. We then use its distance to query to update the lower bounds of the rest of leaves in the S as what we do for the object referring to the whole set of leaves. We keep doing this until we have the expected number of leaves species we want.

In INN3, we do a similar operation as we do for INN1. However, when the newly pulled out element is an object referring to a set of leaves, and after we finish updating the distance lower bounds for the leaves in the set, we partition the set into two or more subsets and put them into the queue. For each of these subsets, its lower bound distance to query is the smallest lower bound of leaves within that set. In the case that we partition the set into two subsets, the search algorithm behaves similarly to when the vp-tree is used. However, it has a different way to choose the lower bound of distance for the set and the size of the partitioned two sets may not be the same. Now, according to these two partition schemes, a natural idea is to partition it into N subsets, and each subset contains only one leaf. This is exactly the idea INN2 uses.

3.2.2. INN2

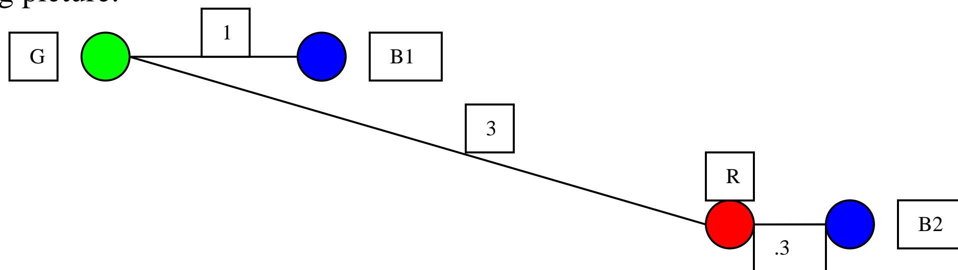
This algorithm first randomly picks a pivot p from the database, computes its distance to query (denote it d(q, p)), and then uses d(q, p) and the distances between p and the rest of the leaves to obtain the lower bounds of distance from query for the rest of the leaves. Then it puts all the leaves, including p, into a queue and the one with shortest distance or lower bound is retrieved. If the retrieved element E has its actual distance from query computed, we know that all the rest of the elements in the queue have distances greater than E's. Therefore, we remove it from the queue and report it to be a nearest neighbor. If the retrieved element E only has its lower bound of distance, then we use all the computed actual distances to query to refine a new lower bound of the element. That is, let d(q, r) be the actual distance between query and the element r, let d(r, E) be the distance between r and E. If $| d(q, r) - d(r, E) | >$ the lower bound of E, then make the lower bound of E be $| d(q, r) - d(r, E) |$. After finishing the refinement, if the lower bound of E is still greater than or equal to any other actual distance and lower bounds in the queue, then we remove E from the queue, compute the distance between E and the query, and then put it back into the queue. We keep doing this until we have a list of nearest neighbors which includes the expected number of species.

3.2.3. Incremental Nearest Species (INS) with Distance Matrix

In order to bring out a list of nearest species, we also consider treating all leaves from the same species as an object. I refer to this method as Incremental Nearest Species (INS). First, we need to define a proper distance function D which is possible to use triangular inequality to refine the distance. As Professor David Jacobs points out, we can use the distance function proposed at the beginning of section 2. That is, the distance for a species **A** from a query q is defined to be the shortest distance among all distances between the query and leaves in the database which belongs to species A. This idea corresponds to the idea above to locate the nearest species based on nearest leaves. Now we have a raw searching algorithm which uses this idea. Let the expected number of species to be brought out be k. In the algorithm, it maintains the upper bound and lower bound of distance from query for each species, and it sorts the list of species according to their lower bounds. The algorithm keeps refining the lower bound and upper bound of

each species until the $(k+1)^{th}$ smallest lower bound is greater than or equal to all upper bounds of the previous k species which have smaller lower bounds than the $(k+1)^{th}$ one. By doing this, we can ensure that no species outside these k species will have a distance smaller than those for any of these k species (because if it is, then let the species outside these k species be P, let the query be q, and let d(q, X) be the distance between q and a species X. We know that lowerBound(d(q, P)) ≥ upperBound(d(q, L)) for any L within that k species, but by our assumption, we will have lowerBound(d(q, P)) ≤ d(q, P) < upperBound(d(q, W)) for some W within that k species -- A contradiction).

Now I will describe how the algorithm refines the bounds. The algorithm keeps picking new pivots out of the database, computes its distance to query, and then uses this distance to refine the bound. Perhaps the simplest way to pick a pivot is to randomly select one whose distance to query has not yet been computed. But according the scheme of refinement of bounds below, we can have another one. To refine the bound, consider that if the pivot is refining the bounds of its own species, then by the definition of the distance between query and species, we can let the upper bound be the d(query, pivot). If all the leaves within a species have been picked out as pivots, then we can directly let the upper bound and lower bound be the smallest distance for those pivots. However, in general, if a pivot is refining the bounds for a species X, we are choosing the smallest of d(query, pivot) + d(pivot, J), where J is a leaf within that species in the database, for d(query, X) ≤ d(query, J) ≤ d(query, pivot) + d(pivot, J). For the lower bound, notice that we may have d(query, X) ≤ abs(d(query, pivot) - d(pivot, J)), which is illustrated in the following picture:



The green ball is the query, the two blue balls is from the species, the red ball is the pivot, and d(G, B1) = 1, d(G, R) = 3, d(R, B2) = 0.3

To still obtain some lower bound, I choose the

$$\max\{0, (\text{abs}(d(\text{query}, \text{pivot}) - d(\text{pivot}, J)) - M)\}$$

where M is the maximum distance from J to the rest of the leaves within the species. This is true because d(query, X) = d(query, y) ≥ d(query, pivot) – d(pivot, y) ≥ abs(d(query, pivot) - d(pivot, J)) – M, for some leaf y within the species, and the d(query, X) = d(query, y) ≥ 0.

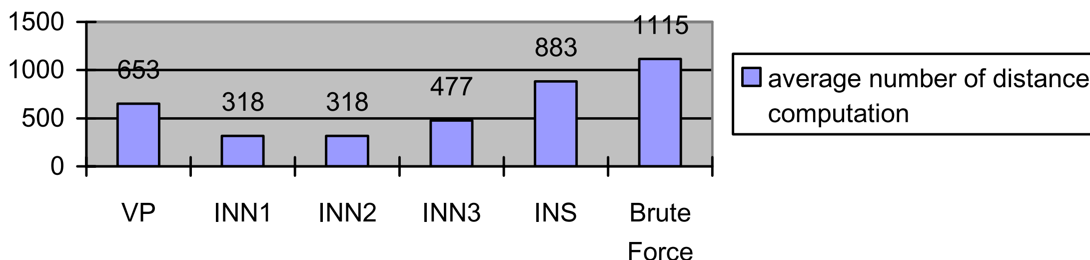
However, the scheme does not refine the lower bound quickly and the result in my experiment shows that it does not provide much speed up. However, notice that if all the leaves from a species have been picked up as pivots, we can obtain the exact distance between query and the species because all its leaves' distances to query are computed.

Also, many species do not have many leaves in it, so we will be able to obtain the exact distance quickly by only picking out the leaves from the same species as pivots.
Based on this insight, I make the first (0.05 * the size of the dataset) pivot to be randomly chosen. Then it switches to choose pivot within only one species until all leaves within this species have been chose and then switch to pick leaves from another species. Call this approach C, and it produces better results in the experiment. Notice that these lower and upper bound update schemes require us to know the distance between the pivot and the leaves in the database, and a simple approach is to use a distance matrix.

4. Experiments

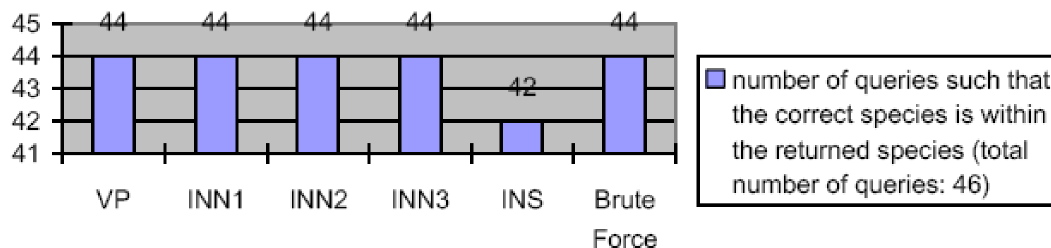
I performed some experiment for method VP, INN1, INN2, INN3, INS. In each of these experiments, I have 1,115 leaves in the database and 46 different leaves outside the database as input in queries. They are all 10 nearest species queries. Among these 1161 leaves, 64,008 triples out of 260,149,140 possible triples of leaves do not obey the triangular inequality using the current distance function. That is, 0.0246% of possible triples of leaves do not obey the inequality. The numbers of distance computation needed using the above 5 methods and the brute force method are showed in the following chart:

Figure 6.1



Since the distance function does not exactly obey triangular inequality, we also need to consider the correctness of the above search method, except for the brute force search, which is not using the triangular inequality to find nearest species.

Figure 6.2



Notice that for the VP algorithm, I choose the simplest method to pick pivots and define bounds for a child of an internal node which is addressed in 3.1.2 and 3.1.1 respectively. For the INN3 method, I partition a newly pulled out subset into two subsets with random sizes and put them back into the queue. For the INS method, it is using approach C and the distance matrix discussed at the end of section 3.2.3. In this

experiment, we can see that INN1 and INN2 have the best improvement compared with brute force, and they also maintain the same correctness rate. INS does not gain much improvement and also has a lower correctness rate compared with the brute force approach.

In a further experiment for INN1, INN2, and INN3, I expand the number of queries to be 1,161. In every query, I let one of those 1,161 leaves to be query, and the remaining leaves are put into the database. The result is showed in figure 6.3 and 6.4.

Figure 6.3

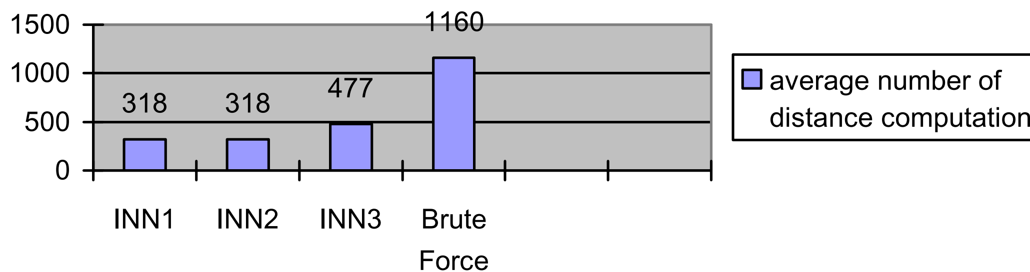
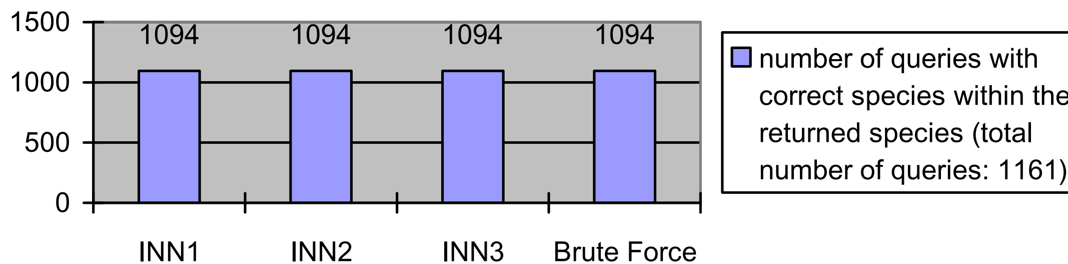
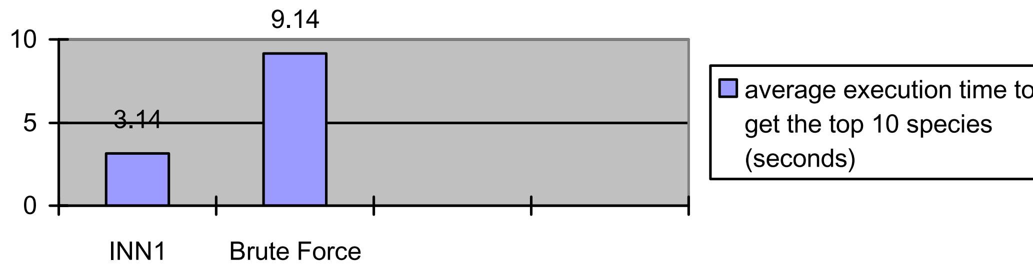


Figure 6.4



In this experiment, we can see that INN1 and INN2 still show the best improvement compared with brute force and maintain the same correctness rate. In a later experiment, I used 2,004 leaves from 32 different species in the database, 128 queries from 32 species, and INN1 as the search method. We have the search result compared with brute force approach in the following chart:

Figure 6.5



The program for this experiment is written in Matlab, and it is running under Microsoft Windows XP operating system with Intel Xeon(TM) CPU (1.50 GHz clock rate) and 1 GB RAM. From this experiment, we can see the INN1 method indeed produces some usable speed-up for the current system. Due to the time constraint, I have not yet checked the correctness of the INN1 method in this experiment and performed an experiment for INN2.

However, using a distance matrix, the size of the matrix will grow to be very big. Consider a database with 10,000 leaf images and a distance between any two leaves is stored as a 4 bytes integer, then the matrix will need about 381 megabytes. However, because of the symmetric property of the distance function, the value in the ith row and the jth column of the matrix is equal to the value in the jth row and the ith column. So we really just need to store half of the matrix, which reduces the memory needed for the matrix by two. In the coming experiment, we will have 6,000 leaf images in the database, which needs about 69 megabytes to store the distance matrix. However, although we can reduce the storage size by half, the distance matrix is still impractical to store if the size of the database is very large, such as 10^8 images.

6. Conclusions

Distance-based indexing methods in a metric space, such as VP, INN1, INN2, and INN3, can be used to speed up the query processing time of the image database in the current electronic field guide. Among them, INN1 and INN2 show the best improvement on reducing the number of distance computations needed and maintain the same correctness rate as the brute force approach does. INN1 has improved the searching time of the current system by a factor of 3. However, these two methods require potentially expensive storage of the distance matrix. The current INS method shows some speed-up, but the improvement is small and does not maintain the same correctness rate.

Since the INN1 and INN2 methods need the distance matrix and the storage of the matrix may be expensive when the size of the database is very large, we may want to consider other methods which do not use a distance matrix. We might consider some other distance-based indexing method such as using a multi-vantage point (mvp) tree and M-tree [3]. We might also consider trying some embedded methods other than distance-based indexing methods. Also, the current INS is just a raw algorithm, so we might also consider improving it. Furthermore, Professor David Jacobs suggests using a fast but less accurate algorithm, Fourier Descriptors, to help choose pivots when doing the search, and it will be a main idea we consider to improve the search in the next step.

7 Acknowledgements.

The author would like to thank Professor David Jacobs for his discussion and direction on this project. The author would also like to thank graduate student Sameer Shirdhonkar for teaching him how to write and debug programs in Matlab.

Bibliography

[1] Hjaltason, G. R. and Samet, H. 2000. Incremental similarity search in multimedia databases. Computer Science Department TR-4199, Univ. Maryland, College Park, Md., Nov.

[2] G. Agarwal, H. Ling, D. Jacobs, S. Shirdhonkar, W. J. Kress, R. Russell, P. Belhumeur, N. Dixit, S. Feiner, D. Mahajan, K. Sunkavalli, R. Ramamoorthi, and S. White, First StepsToward an Electronic Field Guide for Plants," Taxon, in press.

[3] G. R. Hjaltason and H. Samet. Index-driven similarity search in metric spaces. ACM Trans. Database Syst., 28(4):517–580, 2003.

[4] Yianilos, P. N. 1993. Data structures and algorithms for nearest neighbor search in general metric spaces. In Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms (Austin, Tex.). ACM, New York, 311–321.