Clustering Billions of Images with Large Scale Nearest Neighbor Search

Ting Liu
tingliu@google.com

Charles Rosenberg
chuck@google.com

Henry A. Rowley
har@google.com

Google Inc., Mountain View, CA, USA

Abstract

The proliferation of the web and digital photography have made large scale image collections containing billions of images a reality. Image collections on this scale make performing even the most common and simple computer vision, image processing, and machine learning tasks non-trivial. An example is nearest neighbor search, which not only serves as a fundamental subproblem in many more sophisticated algorithms, but also has direct applications, such as image retrieval and image clustering. In this paper, we address the nearest neighbor problem as the first step towards scalable image processing. We describe a scalable version of an approximate nearest neighbor search algorithm and discuss how it can be used to find near duplicates among over a billion images.

1. Introduction

One of the most challenging areas in the field of computer vision and image processing is scalability. As an example, a modern image search engine may contain billions of images, which makes some of the most common tasks non-trivial. One such task is nearest neighbor search, which is often seen as the first step for a variety image processing problems, such as image clustering, object recognition and classification. In this paper, we illustrate the usefulness of large scale nearest neighbor search to tackle a real-world image processing problem.

Very large scale image collections are difficult to organize and navigate. One operation which can facilitate this task is the identification of near duplicate images in the collection. Near duplicate images of popular items, such as book covers, CD covers, and movie posters, appear frequently on the web. This is because they are often scanned or photographed multiple times with varying resolutions and color balances. To tackle this problem at the scale of the whole web is a very challenging task, one which needs efficient, scalable, and parallelizable algorithms for locating and clustering nearest neighbors in the image feature space.

In this work, we tackle the problem of finding approximate nearest neighbors for a repository of over one billion (10^9) images, and perform clustering based on these results. To accomplish this, we introduce a parallel version of a state of art approximate nearest neighbor search algorithm, known as spill trees [11]. Existing spill tree algorithms scale very well with both feature space dimensionality and data set size, but break down when all of the data does not fit into a single machine's memory. We also present a new parallel search method for the parallel spill trees. In addition to the scalable algorithms, we also report on interesting patterns and statistics observed from the experiment. We believe this work can be the basis of more active research in large scale image processing.

2. Background

Nearest Neighbor Search Nearest neighbor search is a subproblem in many machine learning and clustering algorithms. Each object is described by a feature vector, often with many dimensions. Given a new object's features, the goal is to find the existing object which has the closest feature vector according to some distance measure, such as Euclidean distance. This has direct applications in numerous areas, such as information retrieval [5], pattern recognition, databases and data mining, image and multimedia search [6].

Over the years, techniques for solving the exact and approximate k nearest neighbor (k-NN) problem have evolved from doing a linear search of all objects, to k-D trees [7] which do axis parallel partitions of the data, to metric trees (or ball trees) [13] which split the data with arbitrary hyperplanes, to spill trees [11] and LSH [8]. Unfortunately, these methods are all designed to run on a single machine. For a large scale image clustering problem like ours, which cannot fit on a single machine, the traditional algorithms simply cannot be applied. One way to solve this problem is to store the data on disk, and load part of the data into main memory as needed. Although there exist sophisticated paging algorithms, these types of algorithms are far slower than memory based methods. An alternate solution is to use multiple

IEEE Workshop on Applications of Computer Vision (WACV'07)
0-7695-2794-9/07 $20.00 © 2007 IEEE

IEEE COMPUTER SOCIETY

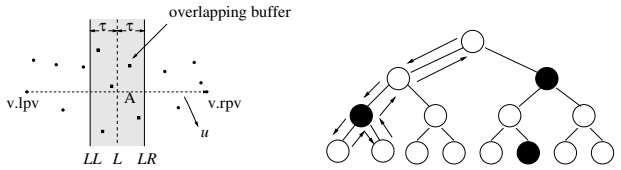

Figure 1. (a) Overlap buffer. (b) Hybrid spill tree. Non-overlap nodes are black, and overlap nodes are white.

machines simultaneously, and perform the k-NN search in parallel. In this work, we use spill trees as a starting point, and adapt the spill tree algorithms to work in parallel. We next review spill trees, then introduce our parallel version of spill trees.

Spill Trees Spill trees are a variant of metric trees which support efficient approximate k-NN searches. Unlike metric trees, the children of a spill tree node can share objects. Formally, we use v to denote a node in the spill tree, and use v.lc and v.rc to denote its left and right children. We first choose two pivots v.lpv and v.rpv, and find the decision boundary L that goes through the midpoint A between the pivots. The partition procedure of a metric tree implies that point sets of v.lc and v.rc are disjoint, separated by the decision boundary L, as shown in Figure 1(a). In a spill tree, the splitting criteria is relaxed to allow overlaps between the two children. We define two new separating planes, LL and LR, both of which are parallel to and at distance τ from L. Then, all the objects to the *right* of plane LL belong to the child v.rc, and all the objects to the *left* of plane LR belong to the child v.lc. Since all objects that fall in the region between LL and LR are shared by v.lc and v.rc, this region is called *overlap buffer*, and we call 2τ the *overlap buffer width*.

A spill tree based k-NN search uses *defeatist search*, which descends the tree quickly using the decision boundaries at each level without backtracking [11]. In practice, hybrid spill trees are used which are a combination of spill trees and metric trees, where a decision is made at each node whether to use an *overlap node* or *non-overlap node*. We only use defeatist search on overlap nodes, and for non-overlap nodes, we still do backtracking as in conventional metric trees, as illustrated in Figure 1(b). Notice, for hybrid spill trees, τ is the critical parameter for both tree generation and search. In general, the greater τ is, the more accurate and the slower the search algorithm becomes.

The above algorithms are serial algorithms, running on a single machine and requiring random access to the entire set of objects to be placed in the tree. This work focuses on three extensions to the above work: making the tree building algorithms work in parallel to handle large data sets which cannot fit on a single machine, doing a large number of queries efficiently in parallel, and automatically setting the overlap buffer size.

Although we have focused on extending hybrid spill trees, there exist other algorithms for fast approximate nearest neighbor search. Among the most popular is locality sensitive hashing (LSH) [8]. One of the deficiencies of this algorithm is the number of parameters which need to be tuned for optimal performance in different domains. Some of the parameters which need to be adjusted and are critical to the accuracy and speed are: the number of LSH tables to use, the number of bins in each table, and the stop value after finding too many neighbors. The difficulty of setting these LSH parameters is one reason that its efficiency often does not match that of hybrid spill trees [11]. Another issue with LSH is that if you cannot find enough neighbors in the bins examined, there is no way to expand the set.

Image Features Before building a search tree of images, we need to define how the images will be represented as feature vectors. We first normalize an image by scaling the maximum value of each color channel to cover the full range of intensities, and then scale the image to a fixed size of 64×64 pixels. From here, one obvious representation might be the image pixels themselves, however this would likely be quite sensitive to noise and other small image variations. Instead we used an adaptation of the technique presented by [9], in which the image is converted to the Haar wavelet domain, all but the largest 60 magnitude coefficients are set to 0, and the remaining coefficients are quantized to ± 1. The feature vector as described is quite large, $64 \times 64 \times 3$, so random projection [1] using random unit-length vectors is used to reduce the dimensionality of the feature vector to 100 dimensions. The average of each color channel (the ranges were $0-1$, $-0.596-0.596$, and $-0.523-0.523$ for Y, I and Q, respectively) and the aspect ratio $w/(w+h)$ (range $0-1$) are appended to this feature vector for a total of 104 dimensions. No effort was made to tune the relative scalings of the features. The nearest neighbor algorithm described in this paper is designed to handle generic feature vectors, and is not restricted to this particular representation.

Parallel Computing Framework All of the parallel algorithms described will be expressed in terms of MapReduce operations [4], which provide a convenient framework hiding many of the details necessary to coordinate processing on a large number of machines. An operation in the MapReduce framework takes as input a collection of items in the form of key-value pairs, and produces a collection of output in the same format. It has three basic phases, which are described in Figure 2. In essence, an operation in the MapReduce framework is completely described by the *map operation*, the *shuffle operation*, and the *reduce operation*. The algorithms below will be described in terms of these

IEEE COMPUTER SOCIETY

Map A user-defined *Map Operation* is performed on each input key-value pair, optionally generating one or more key-value pairs. This phase works in parallel, with the input pairs being arbitrarily distributed across machines.

Shuffle Each key-value pair generated by the Map phase is distributed to a collection of machines, based on a user-defined *Shuffle Operation* of their keys. In addition, within each machine the key-value pairs are grouped by their keys.

Reduce A user-defined *Reduce Operation* is applied to the collection of all key-value pairs having the same key, optionally producing one or more output key-value pairs.

Figure 2. The three phases which make up the MapReduce framework. All steps run in parallel on many machines.

operations, however this should not be taken as the only way to implement these algorithms.

3. Algorithms

Building Hybrid Spill Trees in Parallel The main challenge in scaling up the hybrid spill tree generation algorithm is that it requires all the objects' feature vectors to be in memory, and random access to this data. When the number of objects becomes large enough, it is no longer possible to store everything in memory. For our domain, with 104 floating point numbers to represent each object, or around 416 bytes, this means we could typically fit eight million objects comfortably on a machine with 4GB of memory. In a collection of over a billion images, there are nearly a thousand times as many image as can fit into one machine's machine.

The first question is how to partition the data. One possibility is to randomly partition the data, building a separate hybrid spill tree for each partition. However, at query time, this would require each query be run through all the trees. While this could be done in parallel, the overall query throughput would be limited.

Another alternative is to make a more intelligent partition of the data. We propose to do this through the use of a metric tree structure. We first create a random sample of the data small enough to fit on a single machine, say $1/M$ of the data, and build a metric tree for this data. Each of the leaf nodes in this *top tree* then defines a partition, for which a hybrid spill tree can be built on a separate machine. The overall tree consisting of the top tree along with all the *leaf subtrees* can be viewed conceptually as a single hybrid spill tree, spanning a large number of machines.

At first glance it might appear that the top tree should also be a spill tree, because of the benefits of not needing to backtrack during search. However, a negative aspect of spill trees is that objects appear in multiple leaf subtrees. In practice however we found that this data duplication lead to an unacceptable increase in the total storage required by the

Sample Data Input is all the objects, output is a sampled subset for building the top tree.

- **Map** For each input object, output it with probability $1/M$.
- **Shuffle** All objects map to a single machine.
- **Reduce** Copy all objects to the output.

Build Top Tree On a single machine, build the top tree using the standard metric tree building algorithm as described in [11], with an upper bound U and lower bound L on the number of objects in each leaf node.

Partition Data and Create Leaf Subtrees Input is all the objects, output is the set of leaf subtrees.

- **Map** For each object, find which leaf subtree number it falls into, and output this number as the key along with the object.
- **Shuffle** Each distinct key is mapped to a different machine, to collect the data for each leaf subtree.
- **Reduce** For all the objects in the leaf subtree, use the serial hybrid spill tree algorithm [11] to create the leaf subtree.

Figure 3. Building a parallel hybrid spill tree.

system. The resolution was to force the top tree to be a metric tree, and to make modifications to the search procedure which will be described in the next subsection.

The metric tree building procedure needs a stopping condition for its leaves. Typically the condition is an upper bound on the leaf size. In order for each partition to fit on a single machine, we set the upper bound U such that the expected number of objects $U \cdot M$ which will fall into a single leaf subtree can fit on a single machine. We typically set $U \cdot M$ a factor of two or more smaller than the actual limit, to allow for variability in the actual number of objects ending up in each leaf. In addition we set a lower bound on the number of nodes. The lower bound L is set empirically to prevent individual partitions from being too small, typically we use a value of five.

The algorithm as described so far is implemented in a sequence of two MapReduce operations and one sequential operation, as shown in Figure 3.

Efficient Queries of Parallel Hybrid Spill Trees After the trees have been built, they can be queried. As mentioned earlier, the top tree together with the leaf subtrees can be viewed as one large hybrid spill tree. The normal way to query such a tree allows for backtracking through non-overlap nodes, such as those which appear in the top tree. However such an approach would be expensive to implement since the entire tree is not stored on a single machine. Instead, we speculatively send each query object to multiple leaf subtrees when the query appears to be too close to the boundary. This is effectively a run-time version of

IEEE COMPUTER SOCIETY

Find Neighbors in Each Leaf Subtree Input is the set of query objects, output is the k-NN lists for each query object for each subtree the query was routed to.

- **Map** For each input query, compute which leaf subtree numbers it falls into. At each node, the query may be sent to both children if it falls within the overlap buffer width of the decision plane. Generate one key-value pair for each leaf subtree to be searched.
- **Shuffle** Each distinct key is mapped to a different machine, grouping the data for each leaf subtree so they can be searched in parallel.
- **Reduce** The standard hybrid spill tree search is used for the objects routed to each leaf subtree, and the k-NN lists for each query object are generated.

Combine k-NN Lists Inputs are k-NN lists for each object in each leaf subtree, outputs merged k-NN list for each query.

- **Map** Copy each query, k-NN list pair to the output.
- **Shuffle** The queries (object numbers) are partitioned randomly by their numerical value.
- **Reduce** The k-NN lists for each query are merged, keeping only the k objects closest to the query.

Figure 4. Batch k-NN search.

the overlap buffer which was previously only applied at tree building time. The benefit of this is that fewer machines are required to hold the leaf subtrees (because there is no duplication of objects across the subtrees), but with the expense that each query may be sent to several leaf trees during search. In practice we can adjust the overlap buffer size to control the amount of computation done at query time.

For our application of image clustering, we need the k-NN lists for every object, so we organize the searches as a batch process, which takes as input a list of queries (which will be every image), and produces their k-NN lists. The process is described in Figure 4, using two MapReduces.

Parameter Estimation As mentioned earlier, one of the critical parameters for spill tree generation and search is the overlap buffer width. Ideally, the overlap buffer width should be large enough to always include the k nearest neighbors of a point, because this will guarantee that they will always be found. However, a smaller value may be acceptable if we are willing to tolerate a certain number of errors. Below we describe an algorithm to estimate the ideal overlap buffer size, which is then relaxed in practice (by making the buffer smaller) to improve speed.

To estimate the overlap buffer size, we need to estimate R_S, the average distance (averaged over the objects in set S) to their nearest neighbors. Following the heuristic described in [2] (after Equation 8), if we make the approximation that points are uniformly distributed, we expect that the number of objects falling within a certain radius of a given object is proportional to the density of the objects (which is in turn proportional to the number of samples N_S) raised to the power of the dimensionality of the manifold d on which the objects are distributed. In particular, if we fix the expected number of points to k, then the radius of interest is R_S, giving the following equation:

$$k \propto N_S \cdot {R_S}^d \quad \rightarrow \quad R_S = \frac{c}{{N_S}^{1/d}} \tag{1}$$

where c is a proportionality constant. To compute $R_{S_{all}}$ for the whole set of objects S_{all}, we first need to estimate the constant c and the effective dimensionality d. These can be estimated by generating a number of different sized subsets of the data, typically between 20 and 500 samples. For each of these sets, we can find the nearest neighbor of each point by computing all ${N_S}^2$ distances, and recording the average distance to the nearest neighbor of each point. By taking the log of both sides of the expression for R_S in Equation 1, we can then estimate c and d with linear regression. Plugging these values along with the full sample set size into Equation 1, we arrive at an estimate of the average nearest neighbor distance over the whole set.

At first it might appear that we should set the overlap buffer width to $R_{S_{all}}$. However, we need to take into account that the partition hyperplanes are unlikely to be perpendicular to the vector between objects which are $R_{S_{all}}$ apart. According to the Johnson-Lindenstrauss lemma [10], after randomly projecting a point from the effective dimensionality d of the samples down to the one dimensional space normal to the partitioning hyperplane, the expected distance will be as follows:

$$2\tau = \frac{R_{S_{all}}}{\sqrt{d}} \tag{2}$$

This yields an estimate of the overlapping buffer size. Because our original assumption of a uniform distribution is the worst case, we usually use a smaller value for the overlap buffer than what is computed above for greater efficiency. This procedure provides an efficient method to get close to the right value.

4. Experiments

Data Sets There were two main data sets of images used for these experiments, one in which the clusters were hand labeled for setting various algorithm parameters, and the second larger set which is our target for clustering.

The labeled set was generated by performing text-based image search queries on several large search engines and collecting the first 60 results for each query. The queries were chosen to provide a large number of near duplicate images. Queries for movie posters, CDs, and popular novels worked well for this. The duplicate sets within the results

IEEE COMPUTER SOCIETY

Map Input is the k-NN list for each image, along with the distances to each of those images. We first apply a threshold to the distances, shortening the neighbor list. The list is then treated as a prototype cluster, and reordered so that the lowest image number is first. The generated output consists of this lowest number as the key, and value is the full set. Any images with no neighbors within the distance threshold are dropped.

Shuffle The keys (image numbers) are partitioned randomly by their numerical value.

Reduce Within a single set of results, the standard union-find algorithm [3] is used to combine the prototype clusters.

Figure 5. Algorithm for initial clustering.

of each query were manually labeled. In addition, 1000 images were chosen at random to represent non-duplicate images. The full collection consisted of 3385 images, in which each pair of images is labeled as either a duplicate or not.

The second much larger set of images consisted of nearly 1.5 billion images from the web (hereafter the 1.5B image set). This was our target for clustering. We have no way of knowing in advance how many of these images are duplicates of one another.

Clustering Procedure Most of the work described so far was concerned with efficiently finding the k nearest neighbors of points, either for single points or in a batch mode. In order to adapt this for clustering, we compute the k nearest neighbors for all images in the set and apply a threshold to drop images which are considered too far apart. This can be done as a MapReduce operation as shown in Figure 5.

The result of this algorithm is a set of prototype clusters, which further need to be combined. Once singleton images are dropped in the 1.5B image set, we are left with fewer than 200 million images, which is a small enough set to run the final union-find algorithm on a single machine.

Clustering Results To evaluate the image features, we first performed clustering on the smaller labeled data set. For each pair of images, we compute the distance between their feature vectors (since this is a small enough data set this is practical). As the distance threshold is varied, we compute clusters by joining all pairs of images which are within the distance threshold of one another. Each image pair within these clusters is then checked against the manual labelling. The results are shown in Figure 6. From this graph, along with manual examination of the clusters, we determined that a distance threshold of 0.45 works well. The graph also shows the result of using at most 10 nearest neighbors (instead of all within the distance threshold), and the approximate 10 nearest neighbor lists generated by the spill tree algorithm and hybrid spill tree algorithms. All of these results are quite close in accuracy, although the spill tree-based algorithms are almost 20 times faster for

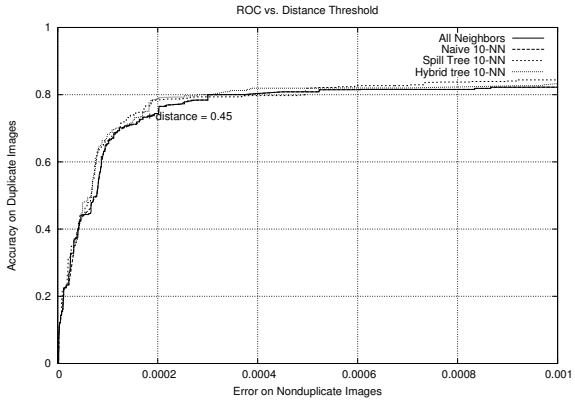

Figure 6. Plot of the error rate on duplicate image pairs vs. the error rate on non-duplicate image pairs on the small labeled test set.

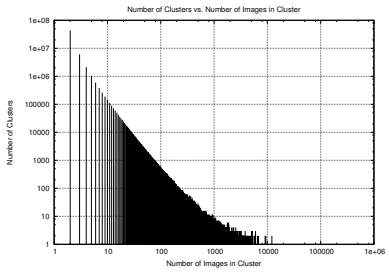

Figure 7. Histogram of cluster sizes for the 1.5B image set. Note the logarithmic scale on both axes.

this smaller set. This difference in speed will grow as the size of the set grows.

We then applied the parallel nearest neighbor finder and clustering procedure to the 1.5B image set. The entire processing time from start to finish was less than 10 hours on the equivalent of 2000 CPUs. Much of that time was spent with just a few machines running, as the sizes of the leaf subtrees was not controlled directly (this will be a direction for future work). Although not discussed here, the computation of the features themselves was also done using the MapReduce framework, and took roughly the same amount of time as the clustering (but with fewer machines). The resulting distribution of cluster sizes is shown in Figure 7. Around 50 million clusters are found, containing nearly 200 million images. The most common cluster size is two, which is perhaps not surprising given the number of thumbnail-fullsize image pairs which exist on the web.

As there is no ground truth labeling for clusters in this larger set, we could not to objectively evaluate the accuracy of the clustering. For a subjective evaluation, we show subsets of some of the actual clusters in Figure 8. As can be seen the images tend to be quite similar to one another, al-

IEEE COMPUTER SOCIETY

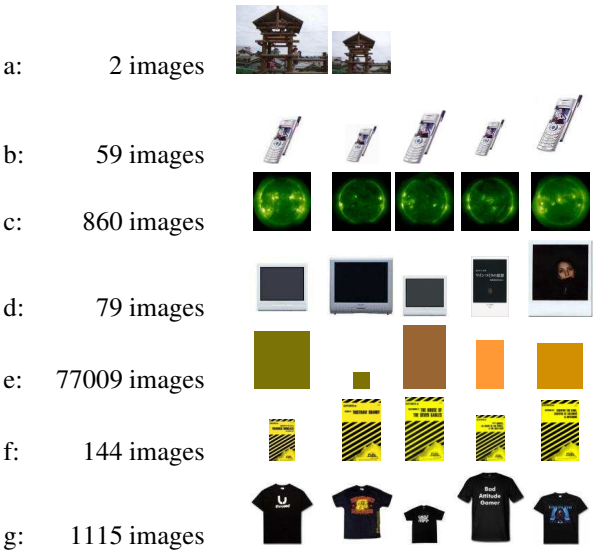

Figure 8. Selection of clusters found by the algorithm. Note the many different sizes of the object in B, and the different words on the same pattern in F and G, as well as additional visually similar false matches in D.

though in some cases images which are quite far apart are grouped together. It is expected that by combining these results with the results of a text query, it will be possible to get more precise clusters when displaying results to users. Another alternative will be to apply a postprocessing step to cut clusters which are "long and thin" into smaller clusters.

5. Summary and Future Work

We have described an algorithm for building parallel distributed hybrid spill trees which can be used for efficient online or batch searches for nearest neighbors of points in high dimensional spaces. Although at first glance a parallel extension of the original spill tree method seems straightforward, there were many non-trivial issues that needed to be addressed. These included how to create a roughly balanced tree, how to automatically find the intrinsic dimensionality and other parameters, how to adapt the hybrid tree to avoid constant communication between machines. This algorithm has enabled us to perform clustering on a set of over a billion images with the goal of finding near duplicates. To our knowledge, this is the largest image set that has been processed in this way.

We choose to apply the algorithm to the image near-duplicate domain because it is relatively straightforward and well understood, allowing us to focus on scaling to larger data sets. However the algorithm does not depend on the types of objects or the application; all it requires is that the objects be described by a feature vector in a metric space. Because of this, we look forward to seeing its application in a wide variety of domains, for instance face recognition, OCR, matching SIFT descriptors [12], and machine learning and classification problems. All of these applications have online settings in which a query object is presented and we want to find the nearest neighbor, and offline or batch settings in which we want to find the nearest neighbors of every point in our collection.

References

[1] E. Bingham and H. Mannila. Random projection in dimensionality reduction: applications to image and text data. In *Knowledge Discovery and Data Mining*, pages 245–250, 2001.

[2] K. L. Clarkson. Nearest-neighbor searching and metric space dimensions. In T. Darrell, P. Indyk, G. Shakhnarovich, and P. Viola, editors, *Nearest-Neighbor Methods for Learning and Vision: Theory and Practice*. MIT Press, 2006.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (Second Edition)*. McGraw-Hill, 2002.

[4] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Symposium on Operating System Design and Implementation*, San Francisco, CA, Dec. 2004.

[5] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990.

[6] M. Flickner, H. Sawhney, W. Niblack, J. Ashley, Q. Huang, B. Dom, M. Gorkani, J. Hafner, D. Lee, D. Petkovic, D. Steele, and P. Yanker. Query by image and video content: the qbic system. *IEEE Computer*, 28:23–32, 1995.

[7] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, September 1977.

[8] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the 25th VLDB Conference*, Edinburg, Scotland, 1999.

[9] C. E. Jacobs, A. Finkelstein, and D. H. Salesin. Fast multiresolution image querying. In *Proceedings of SIGGRAPH*, pages 227–286, 1995.

[10] W. B. Johnson and J. Lindenstrauss. Extensions of lipshitz mapping into hilbert space. In *Conference in Modern Analysis and Probability*, volume 26, pages 189–206. American Mathematical Society, 1984.

[11] T. Liu, A. W. Moore, A. Gray, and K. Yang. An investigation of practical approximate nearest neighbor algorithms. In *Advances in Neural Information Processing Systems*, Vancouver, BC, Canada, 2004.

[12] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, Nov. 2004.

[13] A. W. Moore. The Anchors Hierarchy: Using the Triangle Inequality to Survive High-Dimensional Data. In *Twelfth Conference on Uncertainty in Artificial Intelligence*. AAAI Press, 2000.

IEEE COMPUTER SOCIETY