Optimizing Hashing Functions for Similarity Indexing in Arbitrary Metric and Nonmetric Spaces

Pat Jangyodsuk [1], Panagiotis Papapetrou [2], and Vassilis Athitsos [1]

[1] *Department of Computer Science and Engineering, University of Texas at Arlington, Arlington, Texas, USA*

[2] *Department of Computer and Systems Sciences, Stockholm University, Stockholm, Sweden*

Abstract

A large number of methods have been proposed for similarity indexing in Euclidean spaces, and several such methods can also be used in arbitrary metric spaces. Such methods exploit specific properties of Euclidean spaces or general metric spaces. Designing general-purpose similarity indexing methods for arbitrary metric and non-metric distance measures is a more difficult problem, due to the vast heterogeneity of such spaces and the lack of common properties that can be exploited. In this paper, we propose a generally applicable method for similarity-based indexing in arbitrary metric and nonmetric spaces, based on hashing. We build upon the technique of Distance-Based Hashing (DBH), which organizes database objects in multiple hash tables, so that two similar objects tend to fall in the same bucket in at least one of those hash tables. The main contribution is in showing how to optimize the hashing functions for accuracy and efficiency, using training data. The proposed optimizations significantly improve performance in experiments on three public datasets.

1 Introduction

Suppose that we have a database of objects that are elements of a space with some arbitrary (metric or nonmetric) distance measure. Given such a database, and given a new query object that does not appear in the database, the similarity retrieval problem is the problem of identifying the nearest database objects for that query. This problem arises in a multitude of important real-world domains, including bioinformatics, searching web and multimedia content, and pattern recognition.

The Euclidean distance is a frequent choice for measuring similarity, and a large number of similarity indexing methods have been proposed for this case. These methods exploit the specific structure and properties of the Euclidean distance, e.g., [27, 30, 32]. Non-Euclidean metrics, such as the Levenshtein distance (edit distance) [24] and the Earth Mover's Distance [26] have also been used in several applications. General-purpose indexing methods [18, 19] have been proposed for such spaces, that exploit metric properties, and more specifically the triangle inequality.

There are also important cases where the distance measure of choice is nonmetric. Examples include dynamic time warping for time series [21], the chamfer distance [4] and shape context matching [5] for edge images, and the Kullback-Leibler (KL) distance for probability distributions [8]. In spaces with such measures, indexing methods designed for general metric spaces can still be applied, albeit only heuristically. Methods that are exact, or that exhibit specific mathematical properties when applied to metric spaces, lose such properties in nonmetric spaces, due to the vast heterogeneity of nonmetric spaces, and the lack of even a single common property (e.g., the triangle inequality for metric spaces) that might be leveraged for mathematical analysis.

Distance-Based Hashing (DBH) [3] is a similarity indexing method that has been proposed for arbitrary spaces (metric and nonmetric). DBH works by constructing multiple multibit hashing functions. Each such function maps each database and query object into a bucket of a hash table. We say that two objects "collide" if they are mapped into the same bucket in at least one hash table. The goal in DBH is to obtain hashing functions such that:

- If two objects are close to each other, the objects have a good chance of colliding.
- If two objects are dissimilar (far from each other), their chance of colliding is much smaller than it is for similar objects.

DBH can be used for the filter step of a filter-and-refine retrieval framework: at the filter step, we use the hash tables to quickly find, given a query object Q, all database objects that collide with Q. At the refine step we measure the distances between Q and each database object retrieved at the filter step, to select the most

similar objects. The primary measures of performance are accuracy (what fraction of "similar" objects are successfully retrieved) and time cost, which is primarily determined by the number of dissimilar objects that were retrieved at the filter step (i.e., irrelevant results that the filter step failed to filter out).

The main **contribution** in this paper is a method for optimizing the hashing functions of DBH using training data. DBH, as described in [3], constructs the hashing functions in an entirely random manner. We show that using sample objects from the existing database we can obtain useful information about the performance of different hashing functions. Using this information we can select better hashing functions, so as to optimize the obtained accuracy vs. efficiency tradeoffs. Experiments with three public datasets demonstrate that the proposed method significantly improves performance compared to the original DBH method.

2 Related Work

A plethora of methods exist in the literature for speeding up nearest neighbor retrieval, while many comprehensive reviews have been presented on this subject [6, 18, 19]. Efficient nearest neighbor retrieval in multidimensional vector spaces using an L_p metric has been the focus of many research papers, e.g., [27,30,32]. Nonetheless, many commonly used distance measures are not L_p metrics, and thus cannot be indexed with such methods.

A number of nearest neighbor methods can be applied for indexing arbitrary, non-L_p metrics, such as the Levenshtein distance (edit distance) [24] and the Earth Mover's Distance [26]. The reader can refer to [7, 19] for thorough surveys on the topic. Typical examples include VP-trees [33] and metric trees [31], which hierarchically partition the database into a tree structure by splitting, at each node, the set of objects based on their distances to pivot objects.

However, in many important application, the distance measure of choice is nonmetric. Examples include dynamic time warping for time series [21], the chamfer distance [4] and shape context matching [5] for edge images, and the Kullback-Leibler (KL) distance for probability distributions [8]. In spaces with such measures, indexing methods designed for general metric spaces can still be applied. However, methods that are exact for metric spaces become inexact in non-metric spaces, and no theoretical guarantees of performance can be made.

In domains and spaces involving computationally expensive distance measures, significant speed-ups can be obtained by embedding objects into another space with a more efficient distance measure. Several methods have been proposed for embedding arbitrary spaces into a Euclidean or pseudo-Euclidean space [11, 18, 20, 25]. Nonetheless, such methods simply substitute a fast approximate distance for the original distance, and still use brute force to compare the query to all database objects, albeit using the fast approximate distance instead of the original one.

A method explicitly designed for indexing non-metric spaces is DynDex [15], which is tailored for a specific non-metric distance measure, and is not applicable to arbitrary spaces. An alternative method is proposed by Skopal in [28]. In that method, distances are directly modified in a nonlinear way, to become more metric or less metric, i.e., conform more or less with the triangle inequality. That method can be combined with any distance-based indexing scheme and is orthogonal to such schemes, including the method proposed in this paper. A tree-based reverse kNN was proposed [10] for indexing non-metric spaces; however, this problem is orthogonal to ours, while the asymmetric relationship between kNN and reverse kNN makes it hard to adapt techniques for kNN to reverse kNN.

Distance-Based Hashing (DBH) [3] is a method for indexing spaces with arbitrary distance measures for approximate nearest neighbor retrieval. The method employs a domain-independent distance-based technique for constructing a family of binary hash functions. The method most closely related to DBH is locality-sensitive hashing (LSH) [1, 14]. LSH, and its various improvements and extensions (e.g., [12, 13, 17, 29]), provide mathematical guarantees of sublinear retrieval time, for any desired accuracy under 100%.

A key limitation of LSH is that it can only be applied to specific spaces (e.g., Euclidean spaces), for which a family of locality sensitive hashing functions can be constructed. Hence, LSH is not a method that can be applied to arbitrary spaces and distance measures, such as, for example, the spaces and distance measures used in our experiments. DBH, on the other hand, can be applied to arbitrary spaces and distance measures, with the price of losing the mathematical properties and guarantees of LSH. As mentioned earlier, the heterogeneity and lack of common properties of general nonmetric spaces makes it impossible for any indexing method to obtain mathematical guarantees of retrieval accuracy and/or efficiency in such spaces. In this paper, the focus is on hash-based indexing of arbitrary spaces, and on how to optimize the hash functions of DBH.

3 Review: Distance-Based Hashing

In this section we briefly review Distance-Based Hashing (DBH), the method that we build on top of. DBH constructs multiple multibit hashing functions. Each

function maps database and query objects into buckets of a hash table. In order for DBH to be applicable to arbitrary spaces, the definition of the hash functions only depends on distances between objects.

Let $\mathbb{X}$ be the space to which database and query objects belong, with distance measure D. The building block for defining hash functions in DBH is the pseudo line projections proposed in [11]. Given two arbitrary objects $X_1, X_2 \in \mathbb{X}$, we define a "line projection" function $F^{X_1,X_2} : \mathbb{X} \to \mathbb{R}$ as follows:

$$F^{X_1,X_2}(X) = D(X, X_1)^2 - D(X, X_2)^2 \ . \tag{3.1}$$

If $(\mathbb{X}, D)$ is a Euclidean space, then $F^{X_1,X_2}(X)$ projects points X on a unique line defined by points X_1 and X_2. If $\mathbb{X}$ uses a non-Euclidean distance measure D, then $F^{X_1,X_2}(X)$ does not have a geometric interpretation, but is still well-defined mathematically, and provides a simple way to project $\mathbb{X}$ into $\mathbb{R}$.

We obtain discrete-valued hash functions from F^{X_1,X_2} using thresholds $t_1, t_2 \in \mathbb{R}$ as follows:

$$F^{X_1,X_2}_{t_1,t_2}(X) = \begin{cases} 0 & \text{if } F^{X_1,X_2}(X) \in [t_1, t_2] \ . \\ 1 & \text{otherwise} \ . \end{cases} \tag{3.2}$$

In practice, t_1 and t_2 are chosen so that $F^{X_1,X_2}_{t_1,t_2}(X)$ maps approximately half the objects in $\mathbb{X}$ to 0 and half to 1, so that we can build balanced hash tables. We can formalize this notion by defining, for each pair $X_1, X_2 \in \mathbb{X}$, the set $\mathbb{V}(X_1, X_2)$ of intervals $[t_1, t_2]$ such that $F^{X_1,X_2}_{t_1,t_2}(X)$ splits the space in half:

$$\mathbb{V}(X_1, X_2) = \{[t_1, t_2] | \Pr_{X \in \mathbb{X}}(F^{X_1,X_2}_{t_1,t_2}(X) = 0) = 0.5\} \ . \tag{3.3}$$

A family $\mathbb{H}$ of hash functions for an arbitrary space $(\mathbb{X}, D)$ can be defined simply by randomly choosing many quadruples X_1, X_2, t_1, t_2:

$$\mathbb{H} = \{F^{X_1,X_2}_{t_1,t_2} | X_1, X_2 \in \mathbb{X}, [t_1, t_2] \in \mathbb{V}(X_1, X_2)\} \ . \tag{3.4}$$

By concatenating k binary hash functions h from $\mathbb{H}$ we can define a k-bit hash function. DBH actually constructs l k-bit hash functions. Thus, pointers to each database object are stored at l buckets, one per hash table. Given a query object Q, the system collects the database objects found in the l buckets that the query is mapped to, and only measures distances between the query and those objects.

4 Measuring DBH Performance

The two performance criteria that we care about optimizing are retrieval accuracy and retrieval time. Retrieval accuracy is the percentage of test queries for which the true nearest neighbor is retrieved. In spaces with computationally expensive distance measures, retrieval time is primarily determined by the fraction of database objects that collide with the query in at least one of the l buckets that the query falls in. The refine step must evaluate the exact distance between the query and each such database object.

This section discusses how to estimate retrieval accuracy and cost (time) on a specific data set. As before, let $(\mathbb{X}, D)$ be the underlying space and distance measure. Let $\mathbb{U} \subset \mathbb{X}$ be a database of objects from $\mathbb{X}$. Let F be a line projection function defined using Equation 3.1, and let Q and X be objects of space $\mathbb{X}$.

We define $C_F(Q, X)$ to be the collision probability of Q and X under F. Function F can be converted to multiple binary functions, depending on the choice of thresholds t_1 and t_2 in Equation 3.2. We say that Q and F collide in a binary function if that binary function maps Q and X to the same binary value. Function $C_F(Q, X)$ is the percentage of binary functions derived using F that maps Q and X to the same binary value.

In practice, the only thresholds t_1 and t_2 that our system uses in Equation 3.2 are actual values $F(U)$, where U is a database object. Furthermore, since we want any binary function to split the space $\mathbb{X}$ in half, t_2 is fully constrainted given t_1. Therefore, we can compute $C_F(Q, X)$ by simply counting the number of values $F(U)$ that are between $F(Q)$ and $F(X)$. If we define $I_{\mathbb{U},F}(X)$ to be the number of values $F(U)$ that are smaller than $F(X)$, then:

$$C_F(Q, X) = \frac{|\ |\mathbb{U}| - 2|I_{\mathbb{U},F}(Q) - I_{\mathbb{U},F}(X)|\ |}{|\mathbb{U}|} \tag{4.5}$$

The collision probability of two objects Q and X over a set of line projections $\mathbb{F}$, $C_{\mathbb{F}}(Q, X)$ is the percentage of binary functions (derived from line projections in $\mathbb{L}$) that map Q and X to the same binary value. This can be computed as follows:

$$C_{\mathbb{F}}(Q, X) = \text{mean}\{C_F(Q, X) | F \in \mathbb{F}\} \tag{4.6}$$

Suppose that, given k and l, we construct l k-bit hash tables by choosing randomly, uniformly, and with replacement, kl line projections from a set $\mathbb{F}$ of line projections, and by choosing threshold pais t_1, t_2 for each of those kl line projections. The probability $C^k_{\mathbb{F}}(Q, X)$ of collision between two objects on a k-bit hash table is:

$$C^k_{\mathbb{F}}(Q, X) = C_{\mathbb{F}}(Q, X)^k \ . \tag{4.7}$$

The probability $C^{k,l}_{\mathbb{F}}(Q, X)$ that two objects collide in at least one of the l hash tables is:

$$C^{k,l}_{\mathbb{F}}(Q, X) = 1 - (1 - C_{\mathbb{F}}(Q, X)^k)^l \ . \tag{4.8}$$

Suppose that we have a database $\mathbb{U} \subset \mathbb{X}$ of finite size $n = |\mathbb{U}|$, and let $Q \in \mathbb{X}$ be a query object. We denote by $N(Q)$ the nearest neighbor of Q in $\mathbb{U}$. The probability that $N(Q)$ will collide with Q in at least one of the l hash tables is simply $C_{\mathbb{F}}^{k,l}(Q, N(Q))$. The accuracy of DBH, i.e., the probability over all queries Q that we will retrieve the nearest neighbor $N(Q)$, is:

$$\text{Accuracy}_{\mathbb{F}}^{k,l} = \int_{Q \in \mathbb{X}} C_{\mathbb{F}}^{k,l}(Q, N(Q)) \Pr(Q) dQ \ , \tag{4.9}$$

where $\Pr(Q)$ is the (uniform) probability density of Q being chosen as a query.

Quantity $\text{Accuracy}_{\mathbb{F}}^{k,l}$ can be easily estimated by:

1. Sampling queries $Q \in \mathbb{X}$, and finding the nearest neighbors $N(Q)$ of those queries in the database $\mathbb{U}$.
2. Estimating $C_{\mathbb{F}}(Q, N(Q))$ for each sample Q by sampling from $\mathbb{F}$.
3. Applying Equation 4.8 to compute $C_{\mathbb{F}}^{k,l}(Q, N(Q))$.
4. Computing the average value of $C_{\mathbb{F}}^{k,l}(Q, N(Q))$ over all sample queries Q.

Besides accuracy, the other important performance measure for DBH is retrieval time. In practice, retrieval time primarily depends on the fraction of database objects that collide with each query in at least one of the l hash tables. The expected number of database objects that collide with a query Q is denoted as $\text{LookupCost}_{\mathbb{F}}^{k,l}(Q)$, and can be computed as:

$$\text{LookupCost}_{\mathbb{F}}^{k,l}(Q) = \sum_{X \in \mathbb{U}} C_{\mathbb{F}}^{k,l}(Q, X) \ . \tag{4.10}$$

For efficiency, $\text{LookupCost}_{\mathbb{F}}^{k,l}(Q)$ can be estimated using only a sample of database objects.

An additional part of the retrieval cost is the cost of computing the outputs of the l k-bit hash functions. Overall, we must compute kl binary hash functions of the form specified in Equation 3.2, and for each such binary function we must compute the distances $D(Q, X_1)$ and $D(Q, X_2)$. We denote by $\text{HashCost}(\mathbb{F})$ the number of such distances we need to compute for all binary hash functions, where $\mathbb{F}$ is the set of all line projections used to define the kl binary hash functions. $\text{HashCost}(\mathbb{F})$ is simply the number of pivot objects, i.e., the number of unique objects used as X_1 and X_2 in the definitions of the kl binary hash functions. In the worst case, $\text{HashCost}(\mathbb{F}) = 2kl$. However, in practice $\text{HashCost}(\mathbb{F})$ is much smaller, close to $2\sqrt{kl}$. We achieve that by first choose a relatively small number B of pivot objects, and then we can define up to $\frac{B(B-1)}{2}$ unique line projections using all pairs of those B pivots.

The total cost $\text{Cost}_{\mathbb{F}}^{k,l}(Q)$ of processing a query is therefore the sum of the two separate costs:

$$\text{Cost}_{\mathbb{F}}^{k,l}(Q) = \text{LookupCost}_{\mathbb{F}}^{k,l}(Q) + \text{HashCost}(\mathbb{F}) \ . \tag{4.11}$$

Finally, the average query cost can be computed using sample queries, as was done for computing indexing accuracy. In particular:

$$\text{Cost}_{\mathbb{F}}^{k,l} = \int_{Q \in \mathbb{X}} \text{Cost}_{\mathbb{F}}^{k,l}(Q) \Pr(Q) dQ \ . \tag{4.12}$$

In conclusion, the accuracy and efficiency of DBH, given parameters k and l, can be measured by sampling from the space of queries, sampling from the set of database objects, and sampling from the set $\mathbb{F}$ of line projection functions.

5 Optimizing DBH Performance

In this section we discuss how to optimize DBH so that we get better accuracy versus efficiency trade-offs. The original DBH method of [3] did not perform any optimizations, and defined binary hash functions by randomly choosing line projections and thresholds.

As discussed in Section 4, to keep $\text{HashCost}(\mathbb{F})$ low, we first choose a relatively small number of pivot objects, and then we construct line projection functions by applying Equation 3.1 to multiple pairs of those pivot objects. This approach was also followed at the original DBH paper [3]. However, in that paper the pivot objects were chosen entirely randomly. In this section we propose a method for choosing pivot objects so as to optimize performance.

Once the pivot objects are chosen, we need to select line projections and thresholds for the hash functions. Again, the original DBH method [3] chooses line projections randomly. We propose a method for optimizing the set of line projections.

While pivot selection is done first, and line projection selection (given the selected pivots) is done second, we describe them in reverse order, because line projection selection is more simple, and the pivot selection method uses as building blocks some concepts from the line projection selection method.

5.1 Optimizing the Set of Line Projections

Given a set of pivot objects, we want to select a set of line projections. We use a greedy algorithm, that executes in multiple rounds. Each round adds one new line projection to the set of selections. Thus, at round t, $t-1$ line projections have already been selected, and a number T of line projections are candidates for selection at this round. To choose a line projection at this round, we simply evaluate each of the T candidates to esti-

mate the performance we would obtain by adding that candidate to the $t-1$ already selected line projections. This greedy algorithm provides no guarantees of finding the globally optimal solution, but nonetheless leads to significant improvements in experimental performance, compared to selecting line projections randomly.

The benefit of adding each line projection is measured based on accuracy and cost, using Equation 4.9 and 4.12. In those equations, parameters k and l must be provided. In practice we start by specifying the desired accuracy (e.g., 90%), and a parameter k (typically between 10 and 30). Then, starting with $l = 1$, consecutive values of l are evaluated, to find the smallest l that provides the desired accuracy (any accuracy less than 100% can be attained with a sufficiently high value of l). Since k and accuracy are provided as input to the training algorithm, we run this training algorithm multiple times, for different choices of k and different levels of accuracy. For each level of accuracy, we record the choice of k (and corresponding value l, returned by the algorithm) that gives the best efficiency, and that gives us the k, l parameters that we use at testing time. We choose multiple k and l pairs, to obtain different tradeoffs of accuracy vs. efficiency.

Therefore, at round t, when we evaluate each of the T candidate line projections, we have a specific k and desired accuracy. First, we find the smallest l that provides the desired accuracy, using Equation 4.9. Then, we apply Equation 4.12 to compute the retrieval cost for that candidate line projection. In both Equation 4.9 and Equation 4.12, we use as $\mathbb{F}$ the result of inserting the candidate line projection to the set of the $t-1$ already selected projections. The candidate line projection chosen at round t is the one with the lowest associated retrieval cost.

Equation 4.12 is a double summation formula. It sums costs over queries, and for each query it sums collision probabilities between that query and different database objects. Equation 4.9 is a single summation formula, as it sums over all queries the collision probability between each query and its nearest neighbor. To evaluate these equations, we use a set of sample queries that are actually database objects (and that are disjoint from queries used at test time), and as "database" we use either the entire database, or (to speed up calculation of Equation 4.12) a random subset of the database. Details about the number of samples we have used, and all other parameters we have used, are provided in the Experiments section.

5.2 Optimizing the Set of Pivot Objects The set of pivot objects is also chosen greedily. We start with a set of candidate pivot objects, randomly sampled from the database. Each round of the greedy algorithm evaluates each candidate pivot, to estimate the retrieval cost obtained by adding that candidate to the $t-1$ already selected pivots.

Evaluating a set of pivots could be done, in theory, by selecting and evaluating an optimal subset out of all line projections formed from those pivots, as described in Section 5.1. However, that would require running the algorithm of Section 5.1 an infeasible number of times: for each of the T candidate pivots, for each round of pivot selection. Instead, we evaluate a set of pivot points by computing accuracy and efficiency using all line projections that can be defined using those pivots. Evaluation of this set of line projections is done as in Section 5.1. The algorithm is given k and the desired accuracy, and finds the smallest l that provides the desired accuracy. Then, we apply Equation 4.12 to compute the corresponding retrieval cost for that k and l. The candidate pivot that is selected at round t is simply the one such that, adding that candidate to the $t-1$ already selected pivots, yields a set of line projections with the lowest cost.

Once pivot selection has completed, we run the line projection selection of Section 5.1, to further improve performance compared to simply using all line projections that can be defined those pivots. Since k and accuracy are inputs to the pivot selection process, this process is run multiple times, for different accuracy values, and we record for each accuracy value the (k, l) pair (and associated set of pivots) giving the lowest cost.

6 Experiments

In the experiments we use three public datasets: the isolated digits benchmark (category 1a) of the UNIPEN Train-R01/V07 online handwriting database [16] with dynamic time warping [22] as the distance measure, the MNIST database of handwritten digits [23] with shape context matching [5] as the distance measure, and a database of hand images [2] with the chamfer distance [4] as the distance measure.

6.1 Datasets Here we describe each of the three datasets. We should emphasize that, in all datasets and experiments, the set of queries used to measure performance (retrieval accuracy and retrieval cost) was completely disjoint from the database and from the set of sample queries used for the optimizations of Section 5.

UNIPEN. We use the isolated digits benchmark (category 1a) of the UNIPEN Train-R01/V07 online handwriting database [16], which consists of 15,953 digit examples (see Fig. 1). The digits have been randomly and disjointly divided into training and test sets with a 2:1

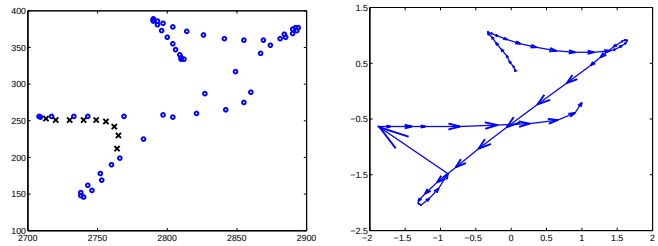

Figure 1: Left: Example of a "seven" in the UNIPEN data set. Circles denote "pen-down" locations, x's denote "pen-up" locations. Right: The same example, after preprocessing.

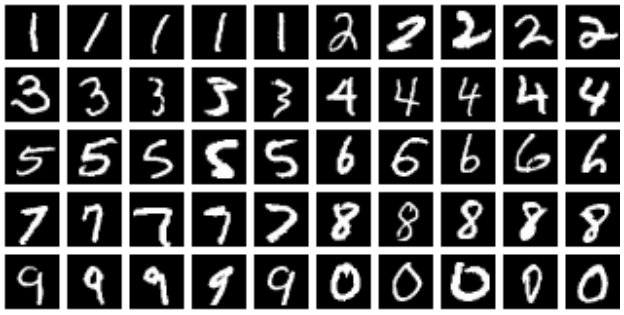

Figure 2: Example images from the MNIST dataset of handwritten digits.

ratio (or 10,630:5,323 examples). We use the training set as our database, and the test set as our set of test queries. The target application for this dataset is automatic real-time recognition of the digit corresponding to each query. The distance measure D used is dynamic time warping [22]. On an AMD Athlon 2.0GHz processor, we can compute on average 890 DTW distances per second. Therefore, nearest neighbor classification using brute-force search takes about 12 seconds per query. The nearest neighbor error obtained using brute-force search is 2.05%.

MNIST. The MNIST dataset of handwritten digits [23] contains 60,000 training images, which we use as the database, and 10,000 test images, which we use as our test query set. Each image is a 28x28 image displaying an isolated digit between 0 and 9 (see Fig. 2). The distance measure used in this dataset is shape context matching [5], which involves using the Hungarian algorithm to find optimal one-to-one correspondences between features in the two images. The time complexity of the Hungarian algorithm is cubic to the number of image features. As reported in [2], NN classification using shape context matching yields an error rate of 0.54%. Based on the MNIST website (`http://yann.lecun.com/exdb/mnist/`), shape context matching outperforms in accuracy a large number of other methods that have been applied to MNIST.

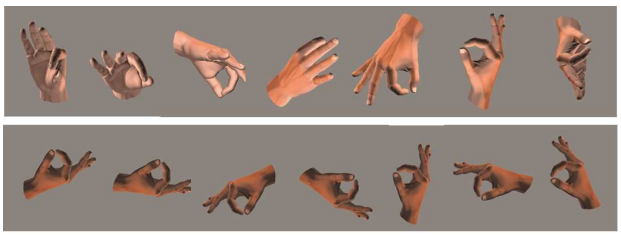

Figure 3: Examples of different images in our hand image dataset.

Using an optimized C++ implementation, and running on an AMD Opteron 2.2GHz processor, we can compute on average 15 shape context distances per second. As a result, using brute force search to find the nearest neighbors of a query takes on average approximately 66 minutes.

The hand image data set. This dataset, described in [2], consists of a database of 80,640 synthetic images of hands, generated using the Poser 5 software [9], and a test set of 710 real images of hands, used as queries. Fig. 3 displays example images from this dataset. The distance measure used here is the chamfer distance [4]. On an AMD Athlon processor running at 2.0GHz, we can compute on average 715 chamfer distances per second. Consequently, finding the nearest neighbors of each query using brute force search takes about 112 seconds.

6.2 Implementation Details We implemented 5 versions of DBH: (i) "Regular DBH", which is our reimplementation of the method of [3]. As we note below, this gives better results than the ones reported at [3], due to a slight change; (ii) Hierarchical DBH, also as described in [3], reimplemented by us; (iii) Optimizing pivot objects; (iv) Optimizing line projections, (v) Optimizing both pivot objects and line projections.

For each dataset, to construct a DBH index, we follow this process: First, a set $\mathbb{X}_{\text{small}}$ of 100 pivot objects are selected. Depending on the version of DBH, this selection is either random or optimized following Section 5.2. Then, we select line projections, again either randomly or according to Section 5.1. Finally, we choose thresholds $[t_1, t_2]$ separately for each line projection F, by choosing t_1 randomly among the smallest half of all values $F(U)$ of database objects, and choosing t_2 so that half of all values $F(U)$ fall inside interval $[t_1, t_2]$.

For the pivot set optimization of Section 5.2, we selected 2,000 candidate pivots randomly among database objects. The optimization algorithm selects 100 of those

pivots, using which we can define 4,950 line projections. The line projection selection of Section 5.1 selects 1,000 line projections out of the 4,950 candidates.

To estimate retrieval accuracy using Equation 4.9, we use 10,000 database objects as sample queries. To estimate the lookup cost using Equation 4.10 we use the same 10,000 database objects as both sample queries (Q in Equation 4.10) and sample database objects (X in Equation 4.10). We should emphasize that Equations 4.9, 4.10 and 4.12 were only used in the offline stage of constructing the DBH index. The accuracy and efficiency values shown in Figures 4, 5 were measured experimentally using previously unseen queries, that were completely disjoint from the samples used during optimization.

For the hierarchical version of DBH, described in [3], we used $s = 5$ for all data sets, i.e., the hierarchical DBH index structure consisted of five separate DBH indexes, constructed using different choices for k and l.

6.3 Results Fig. 4 shows the results obtained on the three data sets for our reimplementaitons of regular DBH and Hierarchical DBH, compared to the original results reported at [3], and also compared to VP-trees [33], used as a baseline method. For each dataset we plot retrieval time versus retrieval accuracy. Retrieval time is completely dominated by the number of distances we need to measure between the query object and database objects. The number of distances includes both the hashing cost and the lookup cost for each query. To convert the number of distances to actual retrieval time, one simply has to divide the number of distances by 890 distances/sec for UNIPEN, 15 distances/sec for MNIST, and 715 distances/sec for the hands data set. Retrieval accuracy is simply the fraction of query objects for which the true nearest neighbor was returned by the retrieval process.

In Fig. 4, it can be seen that our reimplementations performed much better than the original implementations of [3]. This was caused by a small change: in the original work [3], the threshold t_1 for each line projection is always the median of all values $F(U)$ of database objects U, whereas in our implementation it is chosen randomly. We have verified that choosing t_1 as in [3] produces indeed the results reported at [3]. We also note that hierarchical DBH generally performs better than regular DBH, except for the MNIST dataset where performance is comparable to regular DBH. Finally, all DBH versions outperformed VP-trees for all datasets.

Next, Fig. 5 compares the performance of our reimplementation of regular DBH with the optimized versions proposed here. We evaluate a version doing the line projection optimization of Section 5.1 (preceded by random pivot selection), a version doing only the pivot object optimization of Section 5.2 (followed by random line projection selection), and a version doing

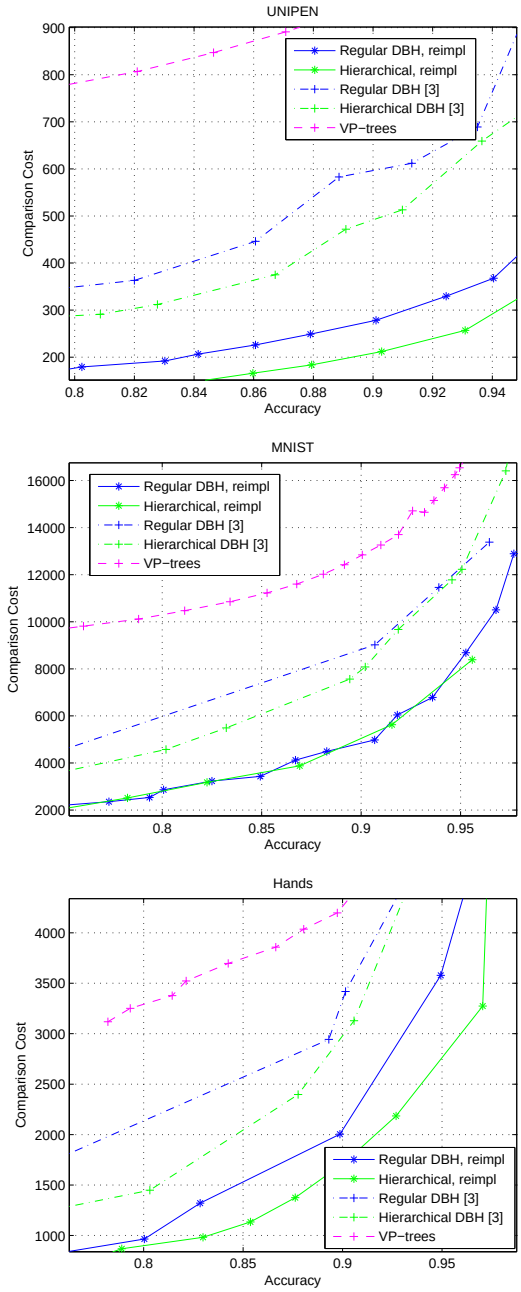

Figure 4: Comparisons of our re-implementations of the regular DBH and Hierarchical DBH methods of [3], the original results reported at [3], and VP-trees [33]. The x-axis is retrieval accuracy, i.e., the fraction of query objects for which the true nearest neighbor is retrieved. The y-axis is the average number of distances that need to be measured per query object.

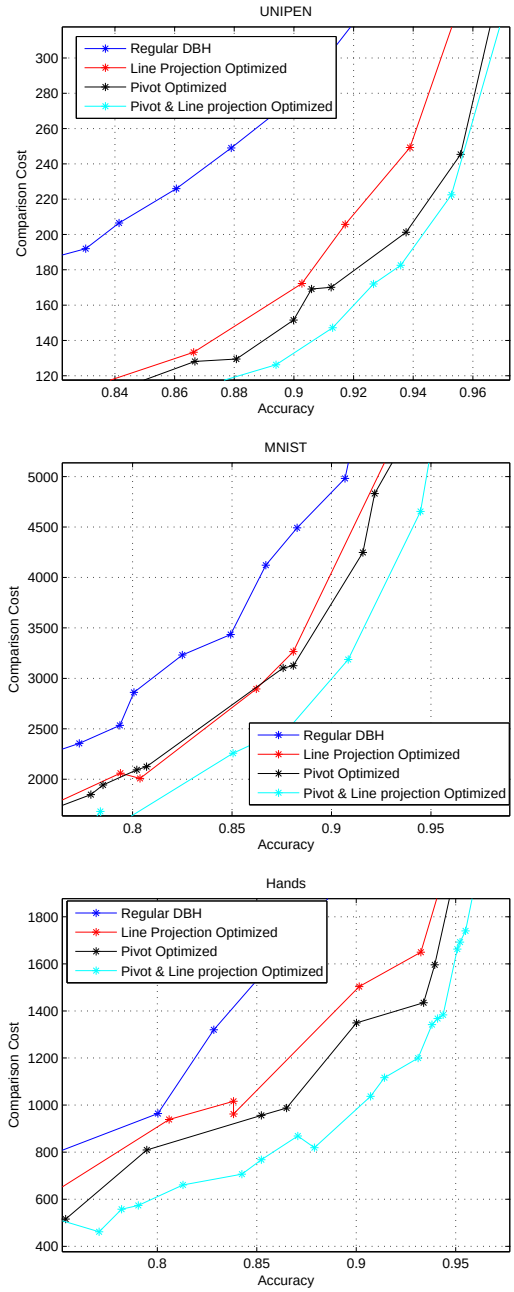

Figure 5: Comparisons of the original DBH method of [3] with the optimized versions proposed here. We evaluate a version doing only the line projection optimization of Section 5.1 (preceded by random pivot selection), a version doing only the pivot object optimization of Section 5.2 (followed by random line projection selection), and a version doing both pivot optimization and line projection optimization. The x-axis is retrieval accuracy, i.e., the fraction of query objects for which the true nearest neighbor is retrieved. The y-axis is the average number of distances that need to be measured per query object.

both pivot optimization and line projection optimization. Optimizing only pivot objects, or optimizing only line projections, always improves performance over regular DBH. Optimizing both pivots and line projections always improves performance over doing only one of the two optimizations, and outperforms both regular DBH and hierarchical DBH.

In conclusion, the results show that each of the two optimizations proposed in this paper, for line projections for pivot objects, leads to better performance in all our experiments. We also see that combining these two optimizations leads to even better results.

7 Conclusions

We have described two methods for optimizing the performance of the Distance-Based Hashing (DBH) method of [3]. In the original paper, the DBH index was constructed using several random choices. In this paper we have described methods for constructing a better index, by making non-random choices based on specific optimization criteria, for selecting pivot objects and line projections. In our experiments, the two proposed optimizations have improved performance, both when applied separately, and even more when combined.

The problem of improving retrieval efficiency in non-Euclidean spaces remains challenging. While the proposed method improves performance over the original DBH, further work is needed to establish whether, and to what extent, better performance is possible, using either improved versions of DBH, or possibly using other, novel indexing approaches.

Acknowledgements

This work was partially supported by National Science Foundation grants IIS-1055062, CNS-1059235, CNS-1035913, and CNS-1338118. The work was also supported by project High-Performance Data Mining for Drug Effect Detection at Stockholm University, funded by Swedish Foundation for Strategic Research under grant IIS11-0053.

References

[1] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 459–468, 2006.

[2] V. Athitsos. *Learning Embeddings for Indexing, Retrieval, and Classification, with Applications to Object and Shape Recognition in Image Databases.* PhD thesis, Boston University, 2006.

[3] V. Athitsos, M. Potamias, P. Papapetrou, and G. Kollios. Nearest neighbor retrieval using distance-based

hashing. In *Proceedings of the IEEE International Conference on Data Engineering*, pages 327–336, Washington, DC, USA, 2008. IEEE Computer Society.

[4] H. Barrow, J. Tenenbaum, R. Bolles, and H. Wolf. Parametric correspondence and chamfer matching: Two new techniques for image matching. In *International Joint Conference on Artificial Intelligence*, pages 659–663, 1977.

[5] S. Belongie, J. Malik, and J. Puzicha. Shape matching and object recognition using shape contexts. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(4):509–522, 2002.

[6] C. Böhm, S. Berchtold, and D. A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 33(3):322–373, 2001.

[7] E. Chávez and G. Navarro. Metric databases. In L. C. Rivero, J. H. Doorn, and V. E. Ferraggine, editors, *Encyclopedia of Database Technologies and Applications*, pages 366–371. Idea Group, 2005.

[8] T. M. Cover and J. A. Thomas. *Elements of information theory*. Wiley-Interscience, New York, NY, USA, 1991.

[9] Curious Labs, Santa Cruz, CA. *Poser 5 Reference Manual*, August 2002.

[10] P. Deepak and P. M. Deshpande. Efficient rknn retrieval with arbitrary non-metric similarity measures. *Proc. VLDB Endow.*, 3(1-2):1243–1254, Sept. 2010.

[11] C. Faloutsos and K. I. Lin. FastMap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In *ACM International Conference on Management of Data (SIGMOD)*, pages 163–174, 1995.

[12] J. Gao, H. V. Jagadish, W. Lu, and B. C. Ooi. Dsh: Data sensitive hashing for high-dimensional k-nnsearch. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 1127–1138, 2014.

[13] K. Georgoulas and Y. Kotidis. Distributed similarity estimation using derived dimensions. *The VLDB Journal*, 21(1):25–50, Feb. 2012.

[14] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *International Conference on Very Large Databases (VLDB)*, pages 518–529, 1999.

[15] K.-S. Goh, B. Li, and E. Chang. DynDex: a dynamic and non-metric space indexer. In *ACM International Conference on Multimedia*, pages 466–475, 2002.

[16] I. Guyon, L. Schomaker, and R. Plamondon. Unipen project of on-line data exchange and recognizer benchmarks. In *12th International Conference on Pattern Recognition*, pages 29–33, 1994.

[17] E. Hassan, S. Chaudhury, and M. Gopal. Word shape descriptor-based document image indexing: a new dbh-based approach. *International Journal on Document Analysis and Recognition (IJDAR)*, 16(3):227–246, 2013.

[18] G. Hjaltason and H. Samet. Properties of embedding methods for similarity searching in metric spaces. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25(5):530–549, 2003.

[19] G. R. Hjaltason and H. Samet. Index-driven similarity search in metric spaces. *ACM Transactions on Database Systems*, 28(4):517–580, 2003.

[20] G. Hristescu and M. Farach-Colton. Cluster-preserving embedding of proteins. Technical Report 99-50, CS Department, Rutgers University, 1999.

[21] E. Keogh. Exact indexing of dynamic time warping. In *International Conference on Very Large Data Bases*, pages 406–417, 2002.

[22] J. B. Kruskall and M. Liberman. The symmetric time warping algorithm: From continuous to discrete. In *Time Warps*. Addison-Wesley, 1983.

[23] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[24] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics*, 10(8):707–710, 1966.

[25] P. Papapetrou, V. Athitsos, M. Potamias, G. Kollios, and D. Gunopulos. Embedding-based subsequence matching in time-series databases. *ACM Trans. Database Syst.*, 36(3):17:1–17:39, Aug. 2011.

[26] Y. Rubner, C. Tomasi, and L. J. Guibas. A metric for distributions with applications to image databases. In *IEEE International Conference on Computer Vision*, pages 59–66, 1998.

[27] Y. Sakurai, M. Yoshikawa, S. Uemura, and H. Kojima. The A-tree: An index structure for high-dimensional spaces using relative approximation. In *International Conference on Very Large Data Bases*, pages 516–526, 2000.

[28] T. Skopal. On fast non-metric similarity search by metric access methods. In *International Conference on Extending Database Technology (EDBT)*, pages 718–736, 2006.

[29] Y. Tao, K. Yi, C. Sheng, and P. Kalnis. Efficient and accurate nearest neighbor and closest pair search in high-dimensional space. *ACM Trans. Database Syst.*, 35(3):20:1–20:46, July 2010.

[30] E. Tuncel, H. Ferhatosmanoglu, and K. Rose. VQ-index: An index structure for similarity searching in multimedia databases. In *Proc. of ACM Multimedia*, pages 543–552, 2002.

[31] J. Uhlman. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40(4):175–179, 1991.

[32] R. Weber and K. Böhm. Trading quality for time with nearest-neighbor search. In *International Conference on Extending Database Technology: Advances in Database Technology*, pages 21–35, 2000.

[33] P. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 311–321, 1993.