FAST NEAREST NEIGHBOR SEARCH IN PSEUDOSEMIMETRIC SPACES

Markus Lessmann and Rolf P. Würtz
Institut für Neuroinformatik, Ruhr-University Bochum, Bochum, Germany

Keywords: Locality-sensitive Hashing, Differing Vector Spaces.

Abstract: Nearest neighbor search in metric spaces is an important task in pattern recognition because it allows a query pattern to be associated with a known pattern from a learned dataset. In low-dimensional spaces a lot of good solutions exist that minimize the number of comparisons between patterns by partitioning the search space using tree structures. In high-dimensional spaces tree methods become useless because they fail to prevent scanning almost the complete dataset. Locality sensitive hashing methods solve the task approximately by grouping patterns that are nearby in search space into buckets. Therefore an appropriate hash function has to be known that is highly likely to assign a query pattern to the same bucket as its nearest neighbor. This works fine as long as all the patterns are of the same dimensionality and exist in the same vector space with a complete metric. Here, we propose a locality-sensitive hashing-scheme that is able to process patterns which are built up of several possibly missing subpatterns causing the patterns to be in vector spaces of different dimensionality. These patterns can only be compared using a pseudosemimetric.

1 INTRODUCTION

Nearest neighbor search (NN search) in high-dimensional spaces is a common problem in pattern recognition and computer vision. Examples of possible applications include content-based image retrieval (CBIR) (Giacinto, 2007), which describes techniques to find the image most similar to a query image in a database, as well as optical character recognition (OCR)(Sankar K. et al., 2010), where letters and words in an image of a text are recognized automatically to digitize it. Another field in which the problem arises is object recognition by feature extraction and matching as in the approach of (Westphal and Würtz, 2009). This system extracts features, which are called *parquet graphs*, from images during learning and stores them together with additional information about object identity and category in a codebook. During recall features of the same kind are drawn from a test image and for each of them the nearest neighbor in the codebook is found. The additional information is then used for voting about possible object identity or category.

All these examples spend a lot of their total computational resources on nearest neighbor search. To make them able to handle large databases (also called *codebooks* from now on) they need methods for fast NN search. In the case of spatially extended features used together with segmentation masks, like in (Würtz, 1997) for face recognition and (Westphal and Würtz, 2009) for general object recognition, a further problem arises.

A multidimensional template or a parquet graph describes a local image patch using Gabor descriptors called *jets* at a small range of positions and can incorporate segmentation information by deactivating jets at positions outside the object. Thus, the parquet graph is enhanced with a segmentation mask with the values active or inactive. The comparison function between two parquet graphs uses only the positions that are active in both graphs. By means of different allocation patterns of active and inactive positions parquet graphs divide into several groups of vectors of different dimensions. This renders known techniques for nearest neighbor search useless because they rely on the triangle inequality which is only valid if all vectors belong to the same metric space. We will present a search technique that is able to handle this problem.

The outline of this article is as follows: in section 2 we will introduce nearest neighbor search techniques for low-dimensional metric spaces. Then we explain the problem of high-dimensional nearest neighbor search (curse of dimensionality) and the locality-sensitive-hashing scheme (LSH) as an approximate solution. In section 3 parquet graphs are

Lessmann M. and P. Würtz R..
FAST NEAREST NEIGHBOR SEARCH IN PSEUDOSEMIMETRIC SPACES.
DOI: 10.5220/0003809006670674
In *Proceedings of the International Conference on Computer Vision Theory and Applications* (VISAPP-2012), pages 667-674
ISBN: 978-989-8565-03-7

presented and it is made clear why the original LSH scheme is not suited for them. Section 4 presents our LSH scheme fitted to the special demands of parquet graphs. Section 5 shows result of experiments that were conducted using our technique, original LSH and exhaustive search for comparison. The conclusion is given in section 6.

2 OVERVIEW NEAREST NEIGHBOR SEARCH

Of course, NN search can always be performed using exhaustive search. That means the query vector is compared to all vectors in the database and minimal distance as well as index of the nearest neighbor are updated after each comparison. For big codebooks this methods becomes prohibitive because of computational cost.

In low-dimensional spaces several efficient methods for NN search exist. The most prominent one is the *kd-tree* (Bentley, 1975). It builds a binary tree structure using the given codebook of vectors and thereby bisects the search space iteratively in each dimension. At first, all vectors are sorted according to the value of their first component. Then the vector whose value is the median is assigned to the root node. This means the root stores the median value and a pointer to the complete vector. All other vectors are now split into two groups depending on whether their first component value was higher or lower than the median. Now both groups are sorted according to their second components and again medians are determined. Their vectors become the root nodes of two new subtrees, which become left and right children of the first root node. This process is repeated until all vectors are assigned to a node. If the depth of the tree gets higher than the number of dimensions the first dimension is taken again for determining median values.

NN search now works in the following way: the query vector is compared to the vector of the root node and the complete and single dimension distance are computed. Since the complete squared Euclidean distance is the sum of the squared Euclidean distances of all single dimensions the complete distance between two vectors can never be smaller than their distance in only one dimension. Therefore, an estimate of the complete distance to the nearest neighbor is updated while traversing the tree. If the distance to a vector in one dimension is already bigger than the current estimate this vector cannot be the nearest neighbor and neither can all its (left or right) children, which are even more distant in the actual dimension. This means that only one of the two subtrees has to be scanned further and thus the number of necessary complete distance calculations and also the search time is $O(\log(n))$ in the optimal case with n being the number of vectors in the codebook. This method does not work in high-dimensional spaces because of the curse of dimensionality. It means that distances between arbitrary points in metric spaces become more and more equal with increasing number of dimensions. This leads to the fact that fewer and fewer subtrees can be discarded in the kd-tree search and almost the complete database needs to be scanned. The search-time thus grows from $O(\log(n))$ to $O(n)$. The first method to provide at least an approximate solution to NN search in high-dimensional spaces is called locality-sensitive-hashing (LSH). In its version for metric spaces (Datar et al., 2004) it uses a special hash function that assigns vectors to buckets while preserving their neighborhood relations. The hash function is based on a (2-stable) Gaussian distribution $\mathcal{G}$. 2-stable means for the dot product of vector $\vec{a}$, whose components are drawn from the Gaussian distribution $\mathcal{G}$ and vector $\vec{v}$:

$$\vec{a} \cdot \vec{v} \propto \|\vec{v}\|_2 \cdot \mathcal{A}\,, \tag{1}$$

where $\mathcal{A}$ is also a Gaussian distributed random number. This dot product projects each vector onto the real line. For two different vectors $\vec{u}$ and $\vec{v}$ their distance on the real line is distributed according to:

$$\vec{u} \cdot \vec{a} - \vec{v} \cdot \vec{a} \propto \|\vec{u} - \vec{v}\|_2 \cdot \mathcal{A}\,. \tag{2}$$

This means that the variance of the distribution increases proportionally to the distance of the vectors in their vector space. Therefore, vectors with a smaller distance in the metric space have a higher probability of being close to each other on the real line. Chopping the real line into bins of equal length leads to *buckets* containing adjacent vectors. The complete hash function is

$$H(\vec{v}) = \lfloor \frac{\vec{a} \cdot \vec{v} + b}{W} \rfloor\,, \tag{3}$$

W being the bin width and b a random number drawn from an equal distribution in the range 0 to W. A query vector is then mapped to its according bucket and compared with all vectors within it. Since only a fraction of all codebook vectors is contained in this bucket search is sped up. It is always possible that a vector is not projected into the bucket of its nearest neighbor but to an adjacent one. Then not the real nearest neighbor is found but only the l nearest neighbor (where l is not known). Thus it is not an exact but an approximate method. To increase the probability of finding the true nearest neighbors (also called hit rate within this paper) the vectors from the dataset

can be mapped to several buckets using several different hash functions. Scanning multiple compilations of adjoining vectors increases the probability to find the true nearest neighbor.

3 PARQUET GRAPHS

Parquet Graphs (Westphal and Würtz, 2009) are features that describe an image patch of a certain size by combining vectors of Gabor descriptors or *jets* (Lades et al., 1993) in a graph structure. The jets are extracted from a 3×3 rectangular grid. Jets contain absolute values of Gabor wavelet responses in 8 directions and 5 scales, which means that they are 40 dimensional vectors. Thus a full parquet graph consists of 9 jets and can be seen as an 360 dimensional vector in a metric space. Two jets can be compared by calculating their normalized dot product resulting in a similarity measure between 0 and 1. The jet is stored in normalized form throughout the process, so a single dot product is enough for calculating similarities. Two parquet graphs can be compared by comparing their jets on corresponding positions and then averaging the similarity values. Segmentation information of a training image can be incorporated into the parquet graph by labeling grid positions inactive that have not been placed on the object. The respective jet then does not contain information about the object and is not used for comparison. For matching of two parquet graphs only grid positions labeled active in both parquet graphs are used for calculating the average similarity. For each parquet graph at least the center position is active by definition, therefore all parquet graphs can be compared to each other.

For NN search in parquet graphs with inactive positions none of the existing methods can be used. Because of the different dimensionality of the features the triangle inequality is invalid, which would be needed to draw conclusions about distances of vectors that have not been compared directly.

4 PROPOSED SYSTEM

4.1 General Application Flow

As LSH is the only applicable technique in high-dimensional spaces we base our method on it. Since the hash function is the essential element that captures neighborhood relations of vectors we have to replace it with a new function that is suited for our requirements. This new function has to somehow capture spatial relationships between parquet graphs. The only way to do this is to use the parquet graphs themselves. This works as follows: a small subset $\mathbb{K}$ of K parquet graphs, which we will call *hash vectors*, is selected from the codebook. Now each vector in the codebook is compared with them, yielding K different similarity values between 0 and 1. These are now transformed into binary values using the Heaviside function with a threshold t_i:

$$b_i = H(s_i - t_i), \tag{4}$$

yielding a bit string of length K. This string, interpreted as a positive integer, is the index of the bucket in which the vector is stored. Storing means that either the bucket contains a list of indices of vectors that belong to it or has a local data array which holds the corresponding vectors. Since we are using specialized libraries for matrix multiplication to compute dot products of query and codebook vectors, local storage is the faster method and used in our approach. By storing the buckets in an array of length 2^K each bucket can be accessed directly by its hash value.

The assumption behind this definition is that parquet graphs behaving similarly in relation to a set of other parquet graphs, are more similar than those that do not. Therefore they are assigned to the same bucket. In a query a hash value is computed for the query vector in the same way. Then the according bucket has to be scanned by exhaustive search. It is possible that this query bucket is empty. Then the non-empty buckets with the most similar bit strings have to be determined. This means a NN search in Hamming space has to be performed by probing the bit strings with Hamming distance of 1, 2, 3 and so forth one after another until an occupied bucket is found for a given distance. If there are several filled buckets in the same Hamming distance they are all scanned. Bit strings with Hamming distance D are easy to compute by applying a bitwise XOR on the query bit string and a string of the same length that contains 1 at D positions and 0 at the remaining ones. These perturbation vectors are precomputed before the search starts. The maximal possible Hamming distance D_{max} of a query can also be computed in advance by comparing all bit strings found in the codebook with all other possible bit strings of length K. The perturbation vectors are then precalculated up to the distance D_{max}.

4.2 Selection of Hash Vectors and Thresholds

The crucial point in this method is the selection of the K parquet graphs used for deriving the hash values and their corresponding thresholds t_i. If vectors and

respective thresholds are chosen poorly, they assign the same hash value to each vector of the database, and the resulting search becomes exhaustive. The ideal case is when the vectors are distributed equally across the possible 2^K bit strings. Then each bucket contains $\frac{n}{2^K}$ vectors and only a small subset of the complete database has to be scanned. Because of the differing metric spaces in which parquet graphs can be compared it is hardly imaginable how such vectors could be calculated from scratch. Instead we will select suitable vectors first and then chose an threshold for each of them one after another such that codebook vectors are distributed most equally onto the buckets.

Hash vectors need to be as dissimilar as possible, such that comparison with them yields a maximum of differing information. Suppose that all hash vectors are chosen. Now all codebook vectors are compared with the first one and a threshold is chosen that gives an optimal distribution (for one half of the codebook vectors b_1 is 0 and for the other 1). Comparison with the second hash vector should now be able to further subdivide these 2 groups into 4 different groups of almost the same size. This will only be possible if the second hash vector is very dissimilar to the first, otherwise it would assign the same bit value to all vectors which are already in the same group, giving 2 buckets of the same size and 2 empty ones. Therefore hash vectors are chosen first in an evolutionary optimization step. In a fixed number of tries a codebook vector is selected randomly. Then it is compared to all other vectors and the one with the lowest similarity is used as second hash vector. Now both vectors are compared with the rest and the codebook vector whose maximum similarity to both of them is minimal is chosen. This is repeated until an expected maximal number of hash vectors that will be needed is reached. The maximal similarity between any two hash vectors is used as a measure of the quality of the selected vectors. After, e.g., 1000 trials the codebook vector with the lowest quality measure is chosen (the remaining hash vectors are exactly determined by that choice). The thresholds are then fixed sequentially: t_i is increased in steps of 0.01 from 0.0 to 1.0 and vectors are divided into groups according to it. To check the created distribution the following fitness function is computed. Let

$$p_i = \frac{n[i]}{n}, \tag{5}$$

with $n[i]$ being the number of vectors in bucket i. This is the percentage of codebook vectors contained in that bucket. The optimal percentage of each bucket is

$$p = \frac{n}{2^K}. \tag{6}$$

Absolute differences of these values are added for each bucket to give the quality measure

$$f(\mathbb{K}) = \sum_{i=1}^{2^K} |p_i - p|. \tag{7}$$

The threshold yielding a minimum $f(\mathbb{K})$ is selected. Another possible measure we tested is the entropy of the distribution, which becomes maximal for an uniform distribution. The problem is that the entropy also becomes high if one bucket contains the majority of vectors and the remaining ones are distributed equally on the remaining buckets, which is a very unfavorable situation.

4.3 Scanning Additional Buckets

As already mentioned the method will not give the real nearest neighbor in some cases. To be useful we expect it to be correct in ca. 90% of all cases. In the original LSH scheme the hit rate was increased by adding further partitionings of the codebook and thus checking several buckets that were constructed using different hash functions. To reduce memory demand(Lv et al., 2007) have devised a way of scanning additional buckets of the same partitioning in search of the true nearest neighbor. This method is called *multi-probe LSH*. The idea behind this approach is, that buckets should be probed first if their hash value would have resulted from smaller displacements of the query vector than the hash values of others would have. If, e.g., in the original LSH we use only one hash function then each buckets corresponds to one of the bins, into which the real line is fragmented. If the current bin does not contain the correct nearest neighbor most likely one of the adjacent bins would. Further, if the query vector is projected on a value closer to the left neighbor bin it is more probable that this bin contains the correct nearest neighbor. Therefore, it should be scanned before the right neighbor bin. If several hash functions are used the number of neighboring bins increases exponentially. In (Lv et al., 2007) Lv et al. present a scheme that assigns each possible bucket (relative to the current) a score that describes its probability to contain the correct nearest neighbor and creates the corresponding changes in the query hash value in the order of decreasing probability (see reference for details). Although we do not have bins of the real line in our approach we can use this scheme as well. In our case everything depends on the similarities to the hash vectors. For an example with $K = 3$, assume the following similarities for a query: 0.49,0.505,0.78 and that all the thresholds are set to 0.5. The first and second value are pretty close to the threshold that decides if their respective

Table 1: 216054 NN searches in a codebook containing 45935 parquet graphs with active and inactive jets.

Method	Time(sec)	av.Time(sec)	% of vectors	Hit rate
Linear search	4798.98	0.0222	100.0	100.0
Fast search, online parameter tuning	773.97	0.0036	14.13	90.61
Fast search, manual parameter tuning	723.99	0.0034	13.42	90.69

Table 2: 216054 NN searches in a codebook containing 45779 parquet graphs with only active jets.

Method	Time(sec)	av.Time(sec)	% of vectors	Hit rate
Linear search	6144.88	0.0284	100.0	100.0
Fast search, online parameter tuning	716.88	0.0033	10.22	90.34
Fast search, manual parameter tuning	654.93	0.0030	9.44	90.66
LSH, online parameter tuning	1321.91	0.0061	7.52	97.25
LSH, offline parameter tuning	997.06	0.0046	13.24	98.60
LSH, manual parameter tuning	579.36	0.0027	5.47	90.03

bit is set to 1 or not. Although the current query bit string is 011, a string of 111 (for the true nearest neighbor) is also possible. A bit string 001 is even more likely because 0.505 is closer to the threshold than 0.49. In this case the proposed scheme decides which bit string should be checked first. The number of additional scanned buckets will be determined during search by devising the percentage P of codebook vectors that will be scanned. The higher this percentage the more additional buckets will be scanned. The percentage will depend on the desired hit rate.

It is also possible to use several partitionings given by different sets $\mathbb{K}$ of hash vectors. If only indices of codebook vectors are stored in the buckets this can help to further reduce search time because in a well chosen additional partitioning the first few scanned buckets have higher probabilities to contain the true nearest neighbors than the last buckets that have to be checked in a single partitioning. Since different partitionings contain the same vectors it can and will happen that a currently probed bucket of the second partitioning contains vectors which were already found in the first partitioning. If only indices of vectors are stored each index can be checked to prevent multiple comparisons of a codebook vector with the query vector. This is not possible for local storage of vectors. In this case unnecessary redundant calculations arise, which keep the search from benefiting from the usage of additional partitionings.

4.4 Parameter Determination

To use the proposed scheme two parameters have to be determined: K and the percentage of codebook vectors P that needs to be scanned. A lot of effort was put on the task of creating test vectors that can be used for determination of these two parameters. In the end none of the devised schemes was satisfying and they were forsaken. The percentage P depends too strongly on the actual query vectors rather than the codebook itself. Methods like the original LSH in the LSHKIT-implementation assume a certain (gamma-) distribution of the codebook vectors and the query vectors and use this assumption for parameter determination. Since parquet graphs are compared in a lot of possibly differing vector spaces we cannot make the same presumption. Additionally, we found during testing of our method in a different setting with clearly not gamma-like distributed data that it is better to exclude any assumptions about the data distribution. Therefore, we do the following: K is determined for each codebook in advance, while the percentage of scanned codebook vectors is adjusted online during search.

The parameter K is set according to the following consideration. In the best case each bucket contains $n[i] = \frac{n}{2^K}$ vectors. For scanning a single bucket one will have then to compute the product of the query vector with K hash vectors and $n[i]$ vectors in the bucket. If K is bigger than $n[i]$ more time is spent for computing the hash value than for probing the bucket. In that case it is better to decrease K such that the hash value is computed faster and instead more vectors in the codebook are scanned, which increases the probability to find the true nearest neighbor there. To guarantee that less time is spent on hash value computation than on scanning of buckets we impose the condition

$$n[i] = \frac{n}{2^K} > 2 \cdot K \tag{8}$$

to get an upper bound for K. On the other hand, K should not be too small, otherwise the codebook is not split up enough and a lot of possible gain in speed search is wasted. Hence we chose just the maximum value for K that fulfills our condition. This scheme provides us with good values for K.

The percentage P of scanned codebook vectors

Table 3: 216054 NN searches in a codebook containing 88951 parquet graphs with only active jets.

Method	Time(sec)	av.Time(sec)	% of vectors	Hit rate
Parallel linear search	2870.75	0.0133	100.0	100.0
Fast search, online parameter tuning	1103.38	0.0051	8.24	90.50
LSH, offline parameter tuning	1453.45	0.0067	11.82	98.79

is adjusted during search. To do this in the first 10 searches all buckets are scanned and the percentage of probed vectors after which the nearest neighbor was found is stored in a histogram. After these linear searches the percentage needed to get the desired hit rate can be read from the histogram.

It may seem more natural to determine instead the number T of visited buckets. If the codebook was distributed uniformly on the buckets this would give exact the same results. But in practice buckets will contain different numbers of vectors and a fixed number of buckets causes the method to scan fewer vectors in some searches than in others, which decreases the probability of finding the true nearest neighbor. Therefore, it turned out that fixing the amount of scanned vectors (via the percentage) works better than fixing T. The percentage P is further changed during search because the statistics of the query data may change. To do this every 1000 searches 10 complete scans are done for determining P.

4.5 Extending the Scheme to k-nearest Neighbor Search

The scheme can also be used for kNN search. The only change that has to be done is that not only the index of the current nearest neighbor estimate is stored but also the indices of the k vectors that have been closest to the query vector.

4.6 Extending the Scheme to Euclidean Pseudosemimetric

Instead of the normalized scalar product the scheme also works with (averaged) squared Euclidean distances. These can also be computed using the dot product because of the relation

$$(\vec{x}-\vec{y})^2 = \|\vec{x}\|^2 - 2\vec{x}\cdot\vec{y} + \|\vec{y}\|^2 . \qquad (9)$$

The squared norms of the codebook vectors and hash vectors are computed once before the search and stored in a container. During search the squared norm of the query vector and its dot products with hash and codebook vectors are computed and the Euclidean distance (averaged over subspaces) is computed. The rest of the scheme stays the same. The hash vectors now have to be as distant as possible (which of course resembles the dissimilarity condition) and the possible distances between a query vector and a hash vector are not restricted to $[0,1]$ but to $[0,\infty]$. Therefore, the minimal and maximal distances of any codebook vector to any hash vector are determined. The thresholds are then checked in steps of $\delta = \frac{maxDis-minDis}{1000}$.

5 EXPERIMENTS

As our scheme is the only one besides exhaustive search that can handle parquet graphs with inactive jets we compare it with linear search as reference. For complete activated parquet graphs we take the LSH implementation of the LSHKIT C++ library by Wei Dong (Dong, 2011) as an additional comparison. This library implements the multi-probe locality sensitive hashing scheme from (Lv et al., 2007). It uses the squared Euclidean distance for comparison of vectors, which gives the same result as normalized dot product, as can be seen from equation (9) with $\|\vec{x}\|^2 = \|\vec{y}\|^2 = 1$. The most similar vector ($\vec{x}\cdot\vec{y} = 1$) is also the closest in Euclidean space.

Linear search was done by storing the complete database in a single matrix and then computing the matrix-matrix-product of query vector and codebook matrix. All components belonging to inactive jets are set to 0. Normalization by the number of coincidentally active jets is done by storing the *allocation pattern* as an unsigned int between 0 and 511. Each bit indicates the status (active or inactive) for a jet. By bitwise AND of two such patterns and subsequent counting the number of bits set to 1 the appropriate normalization factor is found and the dot product is divided by it.

NN search was tested within the frame of an simple object recognition system. This system learns a codebook of parquet graphs from a set of training images by vector quantization. Each extracted parquet graph is added to the codebook if its highest similarity to any codebook parquet graph is less than an threshold set to 0.92 in these tests. Together with these features information about the current object category is stored. During recognition parquet graphs are extracted from test images and their nearest neighbor in the codebook is found. Its corresponding category information is then used in a voting scheme to deter-

mine the category of the current object. Time for NN search was measured separately during recognition. Two different codebooks were used: the first contains inactive jets and can therefore only be handled by linear search and our method, the second comprises only parquet graphs of which every jet is active. The latter codebook is used to compare our method with original LSH.

For being able to evaluate the potential of both systems independent of parameter determination we run the tests for them once with hand-tuned parameters and once with parameters detected by the methods. The LSHKIT also offers the possibility of online adaptation of the number of scanned buckets. This was tested, too. All tests were run 5 times and the average search time was recorded. The biggest standard deviations were 14.5 sec for linear search and 14.34 sec for LSHKIT search with online parameter determination on the second codebook. All others were below 7.0 sec and several also below 3.0 sec. The following tables summarize all test results. The first column gives search time for all searches, the second average time per search, the third the percentage of scanned codebook vectors and the last the hit rate.

Table 1 clearly shows the superiority of our method compared to linear search. Our nearest neighbor search is considerably faster than exhaustive search and needs only 16.13% respectively 15.09% of linear search time, which fits relatively well with the according percentages of scanned codebook vectors.

For the second codebook (table 2) the percentages of search time are (in the order of the table) 11.67%, 10.66%, 21.51%, 16.23% and 9.43%. This is also in relative good accordance to the amount of scanned vectors for our method, for LSHKIT the difference is bigger. It seems that on that codebook LSHKIT spends relatively more time on hash value computation than our approach. The second codebook shows also that our method is potentially not as good as the original LSH in the LSHKIT-implementation, which only needs to look at 5.47% of all vectors for manual tuned parameters. For automatic parameter detection LSHKIT is slower than our method but gives a higher hit rate (at the cost of scanning more vectors than necessary for desired hit rate). In case of online determination the scan amount is lowered to 7.52%, but search time is worse than for offline determination with only slightly smaller recognition rate. Thus online determination seems to scan the codebook more efficiently but does not lead to better approximation to the desired recognition rate and needs additional time for parameter adaptation. This may be due to non-gamma-distributed data in our tests. The superiority of LSHKIT can highly likely be ascribed to a better partitioning of the search space than in our method. In our search scheme this strongly depends on the selected hash vectors. By testing more than 1000 vectors as first hash vector (maybe all codebook vectors) this can be improved, but time for parameter determination increases considerably.

Additionally, we tried to increase search speed by using Intel's MKL (INTEL, 2011) (version 10.2.6.038) for speeding up matrix multiplication. This library is optimized for Intel CPUs to do algebraic calculations as fast as possible. To our astonishment it turned out that using the library actually diminished the performance of our system. Even when using the special functions for matrix-vector and vector-vector products the computation took more time on our test system than when using the boost ublas product functions (BOOST, 2011). MKL needs certain sizes of both matrices for being able to accelerate multiplication. If this condition is fulfilled it makes a substantial difference. We run an additional test with linear search on all parquet graphs of each single image in parallel on the second codebook. This gave a search time of 1009.02 sec (standard deviation 2.97 sec) for parallel linear search (4.7msec per search). This result is clearly slower than our method when aiming at a hit rate of 90 %. But if one compares it to LSHKIT with offline parameter determination and a hit rate close to 100 % it is only slightly slower while being completely exact. Is it therefore recommendable to do parallel linear search if very high hit rates are needed? To test this we created a third codebook of 88951 parquet graph jets, almost twice as big as the others. For this we got the following results (table reftab:tab3) Parallel linear search (standard deviation 2.11 sec) was clearly slower than our method (standard deviation 5.49 sec) and LSHKIT (standard deviation 2.51 sec). This gives a hint, that even with optimized matrix-matrix multiplication it makes sense to use special search methods. It is possible that parallel linear search would have been faster when search would have been done on more search vectors at once, but in most applications one cannot expect to have all search vectors available from the beginning. A real time object recognition system cannot know the search vectors of future images, and the matrix size up to which parallel matrix-matrix-multiplication gives a speedup will highly likely depend on the cache size of the CPU, too.

6 CONCLUSIONS

We have presented the first search scheme that is able to do fast nearest neighbor search on a set of

vectors with different dimensionality that are comparable with each other but do not share a common vector space. The method is (at least in a serial search scheme) considerable faster than exhaustive search and applicable as long as a similarity function can be derived or in (spaces consisting of) Euclidean (sub)spaces. The automatic parameter determination (for hash vectors) takes some time but has to be run only once for a database.

REFERENCES

Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18:509–517.

BOOST (2011). UBLAS basic linear algebra library.

Datar, M., Immorlica, N., Indyk, P., and Mirrokni, V. S. (2004). Locality-sensitive hashing scheme based on p-stable distributions. In *Proc. SCG '04*, pages 253–262, ACM.

Dong, W. (2011). LSHKIT: A C++ locality sensitive hashing library. http://lshkit.sourceforge.net/index.html.

Giacinto, G. (2007). A nearest-neighbor approach to relevance feedback in content based image retrieval. In *Proc. CIVR '07*, pages 456–463, ACM.

INTEL (2011). Intel math kernel library. http://software.intel.com/en-us/intel-mkl/.

Lades, M., Vorbrüggen, J. C., Buhmann, J., Lange, J., von der Malsburg, C., Würtz, R. P., and Konen, W. (1993). Distortion invariant object recognition in the dynamic link architecture. *IEEE Trans. Comp.*, 42(3):300–311.

Lv, Q., Josephson, W., Wang, Z., Charikar, M., and Li, K. (2007). Multi-probe lsh: efficient indexing for high-dimensional similarity search. In *Proc. VLDB '07*, pages 950–961. VLDB Endowment.

Sankar K., P., Jawahar, C. V., and Manmatha, R. (2010). Nearest neighbor based collection OCR. In *Proc. DAS '10*, pages 207–214, New York, NY, USA. ACM.

Westphal, G. and Würtz, R. P. (2009). Combining feature- and correspondence-based methods for visual object recognition. *Neural Computation*, 21(7):1952–1989.

Würtz, R. P. (1997). Object recognition robust under translations, deformations and changes in background. *IEEE Trans. PAMI*, 19(7):769–775.