How to Use SIFT Vectors to Analyze an Image with Database Templates

Adrien Auclair[1], Laurent Cohen[2], and Nicole Vincent[1]

[1] CRIP5-SIP, University Paris-Descartes,
45 rue des Saint-Pères, 75006 Paris, France
{adrien.auclair,nicole.vincent}@math-info.univ-paris5.fr
[2] CEREMADE, University Paris-Dauphine,
Place du Maréchal De Lattre De Tassigny 75775 PARIS, France
cohen@ceremade.dauphine.fr

Abstract. During last years, local image descriptors have received much attention because of their efficiency for several computer vision tasks such as image retrieval, image comparison, features matching for 3d reconstruction... Recent surveys have shown that Scale Invariant Features Transform (SIFT) vectors are the most efficient for several criteria. In this article, we use these descriptors to analyze how a large input image is formed by small template images contained in a database. Affine transformations from database images onto the input image are found as described in [16]. We introduce a filtering step to ensure that found images do not overlap themselves when warped on the input image. A typical new application is to retrieve which products are proposed on a supermarket shelf. This is achieved using only a large picture of the shelf and a database of all products available in the supermarket. Because the database can be large and the analyze should ideally be done in a few seconds, we compare the performances of two state of the art algorithms to search SIFT correspondences : Best-Bin-First algorithm on Kd-Tree and Locally Sensitive Hashing. We also introduce a modification in the LSH algorithm to adapt it to SIFT vectors.

1 Introduction

In this article, we propose a method to analyze how a large input image is formed by small template images contained in a database. Examples of images to analyze are shown on figures 2.(a) and 2.(b). A typical application is to analyze a picture of a supermarket shelf. The input is a large image of the shelf taken in the supermarket. The database contains images of all the available products. The output is a list of products contained in the image with their corresponding positions.

Products in database and products on the supermarket shelf can be slightly different. For example, a price sticker or a discount sticker can be added. Or the product can be partly hidden (e.g., by the shelf itself, by a discount or any decorative item). For these reasons, local descriptors matching is a natural algorithm for this problem.

During last years, Scale Invariant Features Transform (SIFT) features [16] have received much attention. It has been shown in a recent survey ([17]) that it leads to the best results compared to other local descriptors. Several minor modifications of the initial SIFT features have also been presented (PCA-SIFT [12], or Gloh-SIFT [17]), but the gain is not obvious in all experiments. Thus, we will only focus on the original SIFT descriptors in this article.

These SIFT features will be used to compute several affine transformations between products in database and the input image. An important remark is that some products in database can be very similar. For example, a brand icon will appear on a large number of images in database. Thus, if the input image contains a single product with this brand icon, all the database images having this icon will be found by affine matching. We will thus need a filtering step that will keep only a small subset among all found affine transformations. This filter will use the fact that found templates images cannot overlap themselves in the input image.

Because our goal is to propose an interactive solution, we compare two state of the art optimization methods to search for SIFT correspondences. These algorithms are related to the problem of finding approximate nearest neighbors in high dimensional space. First one is based on Kd-Tree : Best-Bin-First algorithm of [3]. Second one is using hash table : Locally Sensitive Hashing of [9]. We also introduce a modification in the LSH algorithm to adapt it to SIFT vectors.

In the next sections, we first recall the SIFT construction algorithm and the classical method we used to compute affine transformations from SIFT correspondences. Then we introduce a filtering step to keep only valid matchings. Eventually, we compare two optimization methods and introduce a modification in LSH algorithm.

Used Databases Our goal in this article is to apply the described method to an actual private database. It contains 440 images of supermarket products. These images are compressed in JPEG. The image size approximately varies from 100x100 pixels to 500x500 pixels. Images of database lead to more than 270.000 descriptors, each one being in $\Re^{128}$. We call this database DB_{440}.

We also tested our algorithm on the publicly available Amsterdam Library of Object Images [8]. We picked the dataset where illumination direction changes, using the gray-value images of size 384x288 pixels. Within this dataset, we used the 1000 pictures that were taken from light position number 4 and camera number 3. This database generates 170.000 local descriptors. Figure 1 shows some images of this database, noted ALOI.

2 Using SIFT for Affine Matching

In this part, we introduce the first building blocks of our method : SIFT descriptors and robust affine matching. More details about these two steps can be found in [16].

Fig. 1. Four images from the ALOI database

2.1 Descriptors

Like any local descriptor algorithm, it can be split in two distinct steps. First one is to detect points of interest where to compute the local descriptors. Second one is to actually compute these local descriptors. The first stage is achieved by finding scale-space extrema in the difference of Gaussian pyramid. A point is said extremum if it is below or above its 8 neighbors at same scale and the 9 and 9 neighbors at up and down scale. Thus, a point of interest is found at a given scale. Its major orientation is computed as the major direction of a patch of pixels around its position. Then, the descriptor vector is computed at the feature scale. It is a vector of 16 histograms of gradient. Each histogram contains 8 bins, leading to a 128 dimensional descriptor.

Due to their construction, SIFT vectors are invariant by scale change and rotation. And experiences show that they are also robust to small viewpoint changes or illumination variations. This is particularly adapted to our problem as our images (both input and database images) are taken from a frontal point of view and templates from database can be rotated and scaled in the image to analyze.

3 Finding Affine Matchings

As a pre-process step, SIFT vectors are computed from the input image, and noted $SIFT_{IN}$. All the descriptors from the database images have also been extracted offline and are noted $SIFT_{DB}$. Features of the i^{th} database image are noted $SIFT^{i}_{DB}$. The first step is to identify correspondences between $SIFT_{IN}$ and $SIFT_{DB}$.

3.1 Linear Search

Each feature from $SIFT_{IN}$ is compared to each feature from $SIFT_{DB}$ and only correspondences whose L_2 distance is lower than a threshold ϵ are kept. For a query feature q of $SIFT_{IN}$, this can be seen as finding the $\epsilon - neighborhood$ of q in $SIFT_{DB}$. Once we have obtained for each descriptor of $SIFT_{IN}$ a list of its neighbors in $SIFT_{DB}$, these correspondences are fitted to affine transformations. The goal of this fitting step is twofold. First, it is needed to remove outliers from the correspondences. Then, it gives the mapping of database images onto the input image.

3.2 Affine Fitting

In our experiments, database images and input images are taken from a frontal point of view. The affine model is thus well adapted. An affine matrix transforms a point $p1 = (x_1, y_1)$ in the first image, to a point p_2 in the second image :

$$p_2 = \begin{bmatrix} a & b & t_x \\ c & d & t_y \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix}$$

Fitting is achieved independently for each database image. For the i^{th} image in database, we consider correspondences between $SIFT_{IN}$ and $SIFT^i_{DB}$. These correspondences are fitted by several affine matrices corresponding to the multiple occurrences of this database image within the input image.

As introduced in [16], all the correspondences are clusterized. Each correspondence c between feature $f1$ in $SIFT_{IN}$ and feature $f2$ in $SIFT^i_{DB}$ can be seen as a four dimensional point : $c(\theta, \eta, x, y)$ where θ is the rotation between the orientations of $f1$ and $f2$, η is the scale ratio between $f1$ and $f2$, and (x, y) is the coordinates of $f1$ in the input image. Each correspondence is projected in a 4d grid. For being less sensitive to the grid tile sizes, each point is projected on its two closest tiles on each dimension. Thus, a correspondence is projected in 16 tiles. Eventually, every cluster with at least 3 correspondences can be fitted by an affine transformation and can be seen as a potential product match.

Estimating an affine matrix from a cluster of correspondences requires robust method as outliers are common. We used a RANSAC [6] for this task. The affine matrix needs only 3 samples (i.e., correspondences) to be estimated. Once the matrix with the major consensus is obtained, it is optimized by least square.

4 Filtering Potential Images Occurences

Eventually, we obtain a list of potential template image occurrences. Each one can be seen as a triplet :

$$\langle i, A, n \rangle$$

where i is the index of the database image, A is the found affine matrix and n is the number of SIFT correspondences that agree with this matrix.

Because several products of the database are very close (e.g., same brand icons), some potential occurrences are incorrect. Some of them are also overlapping and a few ones are completely wrong. These wrong product matchings are mostly due to affine matrices which were fitted with only 3 or 4 correspondences. Our solution to filter these results is to set up a spatial checking. Product matches are sorted according to the number of points supporting their affine transformation, in decreasing order. Then, they are iteratively pasted in this order onto the final result only if their underlying pixels have not already been reached by another products. This is achieved with several threshold on the number of pixels that must be free to paste a product match. Using this method,

correct product matchings with many SIFT correspondences are pasted first and are accepted while wrong ones with few correspondences cannot be pasted and are discarded. The final result is a list of the product matchings that does not overlap themselves when warped on the input image.

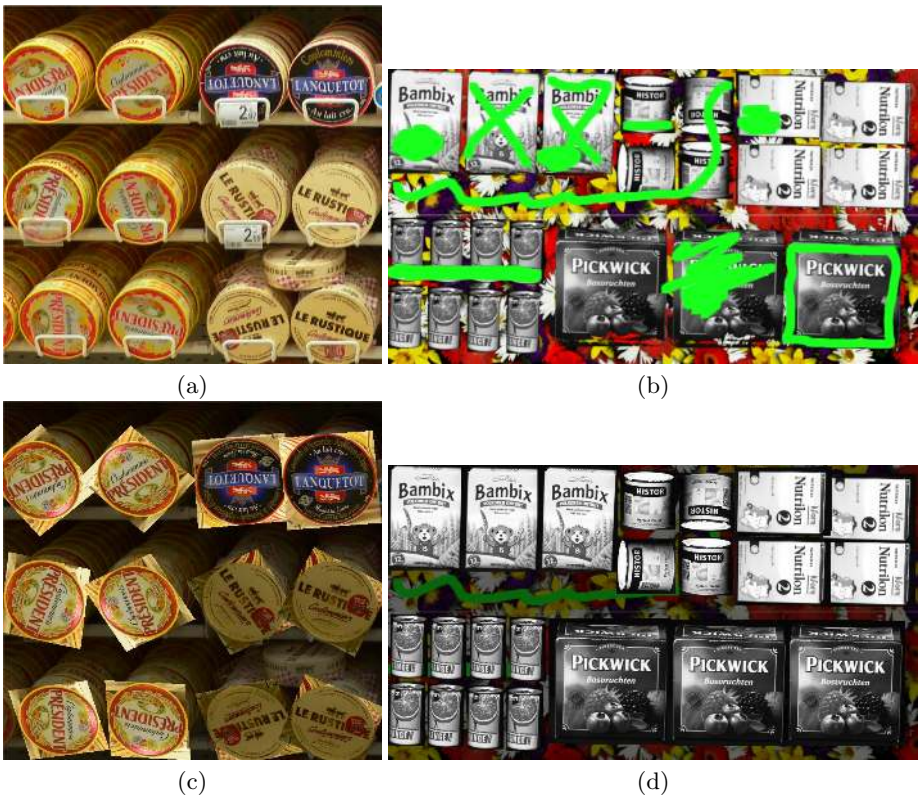


Fig. 2. (a) : A supermarket shelf image to analyze, using DB_{440}. (b) : A test image made up manually from images of the ALOI database. Green painting is for being more challenging. (c) and (d) : Found database images are warped on their found location onto input image. Backgrounds of these result images are input images with lower intensity.

The figure 2 shows the result of the previous algorithm on two input images. Image 2.(a) contains twelve templates of 3 distinct products. Some of them are partially hidden by their corresponding price stickers. The database contains 440 products. Image 2.(b) is made up manually from images of the ALOI database. Some templates were pasted on a flower background, and partially occluded by painting on it. Then a gaussian blur was applied. In both examples, database templates images are correctly found.

The problem of this approach is it slowness. In the example of the figure 2.a, the input image contains 6475 descriptors and the DB_{440} database contains 274.587 descriptors thus there are more than one milliard of euclidean distances to compute. The table 1 shows the running time on our machine (Pentium 1.7GHz) for the linear search. The time is of course linear in the number of SIFT vectors in the database. For convenience, table 1 uses the number of products in database (using an average value of 600 descriptors per database image). It clearly shows that running times are far from acceptable for any interactive application as soon as the database contains more than 10 products. The two other steps of robustly computing affine transformations and checking the spatial coherence are insignificant in time compared to the search of SIFT neighbors. This is why in the next section, we present and compare two optimization methods for searching nearest neighbors in high dimensional space.

Table 1. Computation time for linear search of SIFT correspondences

number of products in database	time to compute correspondences
10	30 seconds
100	5 minutes
450	22 minutes

5 Optimization Methods

The presented algorithm is very slow because of the time needed to search for SIFT correspondences. This is due to the large amount of 128 dimensional euclidean distances to be computed. An idea explored in [12] was to reduce the dimension of local descriptors using PCA. Changing from a 128 dimensional vector to a 36 dimensional vector is an interesting gain but still not enough to get to interactive applications. Moreover, PCA-SIFT is a little less efficient in term of quality (see [17]), thus we will not use this method. In [7], the authors prune a large amount of the descriptors for each database image. But they say this approach is only efficient for near-duplicate image detection. In our case, input image and database images can have different lighting for example because of a photographer using a flash on reflective surface. Images can be blurred because of a bad focus. Thus, pruning many descriptors would lead to miss database images with low quality. This is confirmed by our results as the number of SIFT descriptors is sometime less than 10 for a correct affine matrix. Thus, pruning a large amount of SIFT vectors would lead to miss some products.

Instead, we concentrate more on optimization algorithm for searching neighbors in high dimensional spaces. Methods have been proposed in the literature for exact nearest neighbors computation (one can find a review of this problem in [1]). Tree based methods include Kd-Trees [4], Metric-Trees [18] or SR-Trees [11].

Kd-Tree hierarchically divide the space along one dimension at a time, choosing the median value along this dimension as the pivot. Metric Trees use the same concept but hyperplanes are not aligned with axis. SR-Trees merge the concepts of rectangles trees (R*-Trees [2]) and similarity search trees that uses englobing spheres (SS-Tree [19]).

In [15], the authors compare these tree-based exact nearest neighbors algorithms. The metric-Tree method gives the best results. But even this algorithm has a little gain compared to an exhaustive search. The results obtained in [15] are not better that one order of magnitude in dimension 64. In fact, if the problem is to look for exact neighbors, there is no method that optimize much the linear one presented in 3.1. This is especially true in high dimensional space (i.e., with more than 20 dimensions). This particularity is known as the 'curse of dimensionality'. Thus, we will focus on another class of algorithms which are looking for approximate nearest neighbors (so called ANN problem).

5.1 Approximate Nearest Neighbors Problem

We restrict ourselves to two classes of algorithm : tree-based methods (hierarchical split of the space) and hashing methods. In the next sections, we will test two state of the art algorithms, one of each class. The first one uses a Kd-Tree coupled with the Best-Bin-First algorithm presented in [3]. The second one is the Locally Sensitive Hashing method of [9]. Another method not presented here uses Hilbert Curve to map the high dimensional space to a one-dimensional data space. The search is then achieved in this space ([14]). One can then restrict the search on some portions of the curve to find neighbors as done in [10] in a video retrieval system.

Before presenting the tested methods, we need to define how accuracy will be measured. The goal of these algorithms is to find for each query feature q its $\epsilon - neighborhood(q)$ (i.e., all database features whose distance from q is below a threshold ϵ). The ground truth is found by a linear search. Then, for a given *epsilon* and for a query feature q, the optimized algorithm will give its $\epsilon - neighborhood_{approx}(q)$. We have of course :

$$\epsilon - neighborhood_{approx} \subseteq \epsilon - neighborhood(q).$$

This is achieved by eliminating point further than ϵ from the found neighborhood. The quality of an algorithm will be function of its time of execution and of its recall value. Recall is averaged for a large amount of query points q :

$$mean\_value_{\{q\}} \left(\frac{\#\left\{\epsilon - neighborhood_{approx}(q)\right\}}{\#\left\{\epsilon - neighborhood(q)\right\}} \right).$$

In our application, some products are detected with only a few correspondences. This is why it is important to keep a high recall on this step of the algorithm.

5.2 Using Best-Bin-First with Kd-Trees

Kd-Trees have been introduced in [4]. They are successful for searching exact nearest neighbors when the dimensionality is small. In higher dimensional spaces, this is not anymore true. In our case (i.e., dimension 128), it can only be used for approximate nearest neighbors using the Best-Bin-First algorithm ([3]).

A Kd-Tree is constructed with all the features from the whole database. Each node splits its point cloud into two parts according to a split plane. Each split plane is perpendicular to a single axis and positioned at the median value along this axis. Eventually, each leaf node contains one point. For exact search, a depth first search is used to initialize the closest neighbor. Backtracking is then achieved on a limited number of sub-branches that can have a point closer than the current closest one.

The Best-Bin-First algorithm does not achieve a complete backtracking. To limit its search, it keeps a list of node where search has already been done. This list is sorted according to the distance between the query point and the split plane of the given node. Then, instead of full backtracking from the initial leaf found by the depth first search, backtracking is done on a limited number of branches. The next branch to visit is the one at the head of the sorted list of nodes. The user can then decide the number of branches to visit. We will call this parameter E_{max}. When reducing its value, the user increases speed but more neighbors are missed by the algorithm. Results are presented in section 5.5.

5.3 Using Locally Sensitive Hashing

This hashing has been introduced in [9]. It has been successfully used on image retrieval in very large database [13]. The idea is that if two points are close, they will be hashed with high probability in the same bucket of an hash table. And if they are far, they will be hashed with low probability in the same bucket. Because of the uncertainty of this method, points are hashed in several hash tables using several hash functions.

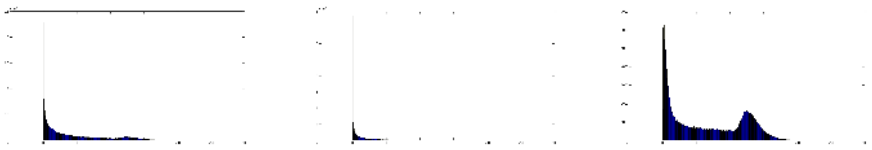
More formally, points are hashed by l different hash functions g_i, leading to store points in l different hash tables. The hash functions are parametrized by the number of hashed dimensions k. Each g_i function is parametrized by two vectors : $D_i = \langle D_0^i, D_1^i \dots D_{k-1}^i \rangle$ and $T_i = \langle t_0^i, t_1^i \dots t_{k-1}^i \rangle$. Values of D_i are randomly drawn with replacement in [0...127]. Thresholds of T_i are drawn in $[0...C]$, where C is the maximum value of the vectors along one dimension. Each g_i maps $\Re^{128}$ to $[0...2^k - 1]$. $g_i(p)$ is computed as a $k - bits$ string $b_0^i, b_1^i \dots b_{k-1}^i$ such that :

$$b_j^i = 0 \;\; if \;\; \left(p(D_j^i) < t_j^i\right) \;\; else \; 1.$$

This $k - bits$ string is the hash index for the point p in the i^{th} hash table. To search for neighbors of a query point q, it is hashed by the l functions. Then, the corresponding l buckets are linearly tested for points closer than the given threshold ϵ. Modifying both l and k allows to tune the algorithm for speed or accuracy. Because k can be chosen high (e.g., above 32), the destination space of

the hash functions can be too large. This is why a second hash function is added to project the result of g_i functions to an actual bucket index whose domain is smaller. Because this will add collisions in the table, a third hash function computes a checksum from the bit string. When going through the linked list of a bucket, only points with the same checksum than the query point are tested. After tuning, we choose to use $l = 20$ hash tables and to adjust k to choose performance or efficiency. The algorithm is benchmarked in section 5.5.

5.4 Adapting LSH to the SIFT Vectors



(a) (b) (c)

Fig. 3. Histograms of SIFT vectors values along several dimensions : (a) Dimension 0 (b) Dimension 10 (c) Dimension 48

In the literature, some authors adapted nearest neighbor algorithms for non uniform data distribution. BOND algorithm of [5] is a natural method for such data. But the exact search method proposed in this article leads to a gain below one order of magnitude that are not enough for our application. In [20], the authors claim that LSH is not adapted for non uniform distribution and thus create a hierarchical version of LSH.

Figure 3 shows the distribution of the coordinates of SIFT vectors on three chosen dimensions. These figures were obtained from the 170.000 SIFT descriptors from the ALOI database. But we obtained similar histograms using the DB_{440}. Histograms of figures 3.a and 3.b are almost representative of all the 128 histograms. Just a few ones are different (e.g., 3.c). These different distribution are a consequence of the SIFT vectors construction. As explained in 2.1, each dimension is a bin where local gradients of a given direction are accumulated. This direction is measured relatively to the major direction of the SIFT descriptor. Thus, local bins which represent gradient of the same direction as the major one are naturally the largest. Histogram of 3.c corresponds to dimension 48 which accumulated gradient in a direction equal to the major one. But excepted a few dimensions with this type of distribution, most of the histograms looks like 3.a or 3.b.

The consequence is that coordinates of the 128 dimensional descriptors which are very low are much more common than those with high values. Thus, when the LSH threshold on one dimension is low, a large amount of points will be

projected in different buckets even if they are close. Accepting low thresholds will thus lead to bad hash functions. For this reason, we tried to choose the thresholds vectors T_i of the LSH hash functions in the range $[min, max]$ where min is much higher than zero. Experimentally, we tuned this range and found that $[60, 120]$ gives the best results.

We can analyze this modification by measuring the probability of collisions for close points. We say two points are close if their distance is below the threshold $\epsilon = 260$. The probability of two close points to be projected by a hash function in the same bucket is noted $P_{close \to same}$. The probability of two points which are not close to arrive in the same bucket is noted $P_{far \to same}$. A family of hash function is efficient if $P_{close \to same}$ is relatively high and $P_{far \to same}$ is relatively low. These probabilities are experimentally measured, averaging the obtained values over the 20 used hash functions. Results are shown in table 2.

Table 2. LSH probabilities of collisions

range to draw LSH thresholds	$P_{close \to same}$	$P_{far \to same}$
$[0...C]$	0.001	0.0002
[60...120]	0.008	0.0005

Using the range [60...120] multiplies $P_{close \to same}$ by 8 while only multiplying $P_{far \to same}$ by 2.5. This confirms the fact that restricting the interval of hashing thresholds leads to better hash functions. In the next paragraph, we will call this method *adaptedLSH* and compare its performance to standard LSH.

5.5 Results

We benchmarked the three presented algorithms : BBF on Kd-Tree, LSH and *adaptedLSH*. Tests were achieved on the two presented databases to ensure being independent of the images. These databases contain respectively 270.000 and 170.000 SIFT vectors.

Figure 4 shows the speed gain of each optimized method, according to the obtained accuracy. BBF algorithm on Kd-Tree is outperformed by both methods based upon LSH. Obtaining a ratio of 70% with this algorithm on the DB_{440} database is almost no faster than using a linear search. If the requirement is to be 100 times faster than linear search on the ALOI database, the LSH algorithm still finds 70% of the neighborhoods, while the *adaptedLSH* method finds 90% of the points. If the need is to find 80% of the points, *adaptedLSH* is twice faster than LSH.

With the first input image, linear search time is around 21 minutes on our machine. A gain of two order of magnitude means the same computation is achieved in 13 seconds, while finding around 80% of the points. This quality is good enough so that all the products found by the exact linear search are also found by this approximate method. For an application that would require only

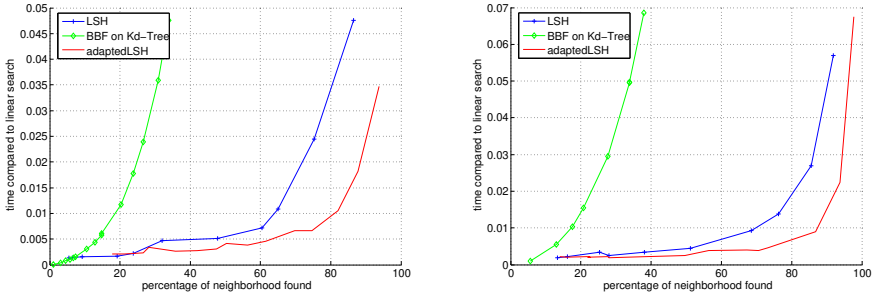


Fig. 4. Results for Kd-Tree, LSH and *adaptedLSH*, using $\epsilon = 260$ on (a) : $DB440$ and (b) : ALOI.

20% of the correspondences, the required time for a query could be divided by a factor 500 relatively to the linear search.

6 Conclusion

The contributions of this article are double. First we propose a complete algorithm to analyze an input image using database template images. This work uses the initial SIFT matching algorithm of [16]. It adds a filtering step to ensure that found images do not overlap themselves when warped on the input images. Our second contribution concerns speed limitations. We compared two optimization algorithm for the approximate nearest neighbors problem. In these tests, LSH outperforms the BBF-KdTree algorithm. We also introduce a modification in the LSH algorithm to adapt it to the SIFT distributions. If the quality requirement is to find 80% of the correspondences, this modified LSH is at least twice faster than standard LSH. Comparatively to a linear search, the gain is of two order of magnitude. Being able to keep a high percentage of the correspondences is a major advantage for our application as it can be sensitive to missing points because some templates matchings are based only on a few points. In the tested images, result are encouraging as we exactly find all the database images at their correct location. For these experiments, we tuned the parameters of the optimization algorithm to find 80% of the SIFT correspondences. We plan to investigate performances of the overall algorithm in terms of recall-precision when modifying the parameters of the LSH algorithm. Moreover, we believe that the criteria we used to filter the matchings (i.e., the number of points validating the found affine transformation) is not optimal.

References

1. Christian Böhm, Stefan Berchtold, and Daniel A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Comput. Surv.*, 33(3):322–373, 2001.
2. Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: an efficient and robust access method for points and rectangles. In *SIGMOD '90: Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pages 322–331, New York, NY, USA, 1990. ACM Press.
3. Jeffrey S. Beis and David G. Lowe. Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In *CVPR '97: Proceedings of the 1997 Conference on Computer Vision and Pattern Recognition (CVPR '97)*, page 1000, Washington, DC, USA, 1997. IEEE Computer Society.
4. Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.
5. Arjen P. de Vries, Nikos Mamoulis, Niels Nes, and Martin Kersten. Efficient k-nn search on vertically decomposed data. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 322–333, New York, NY, USA, 2002. ACM Press.
6. Martin A. Fischler and Robert C. Bolles. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, 1981.
7. Jun Jie Foo and Ranjan Sinha. Pruning sift for scalable near-duplicate image matching. In James Bailey and Alan Fekete, editors, *Eighteenth Australasian Database Conference (ADC 2007)*, volume 63 of *CRPIT*, pages 63–71, Ballarat, Australia, 2007. ACS.
8. Jan-Mark Geusebroek, Gertjan J. Burghouts, and Arnold W. M. Smeulders. The Amsterdam library of object images. *Int. J. Comput. Vision*, 61(1):103–112, 2005.
9. Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *The VLDB Journal*, pages 518–529, 1999.
10. Alexis Joly, Carl Frélicot, and Olivier Buisson. Feature statistical retrieval applied to content-based copy identification. In *ICIP*, pages 681–684, 2004.
11. Norio Katayama and Shin'ichi Satoh. The sr-tree: an index structure for high-dimensional nearest neighbor queries. In *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 369–380, New York, NY, USA, 1997. ACM Press.
12. Yan Ke and Rahul Sukthankar. Pca-sift: A more distinctive representation for local image descriptors. In *CVPR (2)*, pages 506–513, 2004.
13. Yan Ke, Rahul Sukthankar, and Larry Huston. An efficient parts-based near-duplicate and sub-image retrieval system. In *MULTIMEDIA '04: Proceedings of the 12th annual ACM international conference on Multimedia*, pages 869–876, New York, NY, USA, 2004. ACM Press.
14. Jonathan K. Lawder and Peter J. H. King. Querying multi-dimensional data indexed using the hilbert space-filling curve. *SIGMOD Record*, 30(1):19–24, 2001.
15. Ting Liu, Andrew W. Moore, Alexander G. Gray, and Ke Yang. An investigation of practical approximate nearest neighbor algorithms. In *NIPS*, 2004.
16. David G. Lowe. Distinctive image features from scale-invariant keypoints. In *International Journal of Computer Vision*, volume 20, pages 91–110, 2003.
17. Krystian Mikolajczyk and Cordelia Schmid. A performance evaluation of local descriptors. *IEEE Trans. Pattern Anal. Mach. Intell.*, 27(10):1615–1630, 2005.

18. Jeffrey K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Inf. Process. Lett.*, 40(4):175–179, 1991.
19. David A. White and Ramesh Jain. Similarity indexing with the ss-tree. In *ICDE '96: Proceedings of the Twelfth International Conference on Data Engineering*, pages 516–523, Washington, DC, USA, 1996. IEEE Computer Society.
20. Zixiang Yang, Wei Tsang Ooi, and Qibin Sun. Hierarchical, non-uniform locality sensitive hashing and its application to video identification. In *ICME*, pages 743–746, 2004.