Parallel Multi-Tree Indexing for Evaluating Large Descriptor Sets

Levente Kovács
Distributed Events Analysis Research Laboratory
MTA SZTAKI, Kende u. 13-17, Budapest, Hungary
http://web.eee.sztaki.hu/~kla
Email: levente.kovacs@sztaki.mta.hu

***Abstract*—This paper presents a method towards easier evaluation of a large number of different image/video content descriptors, by using a multiple descriptor-tree based parallel indexing scheme instead of classical index structures with high dimensional multi-feature vectors. We will show that the proposed scheme is flexible and easily extensible, and it is not just faster to build, but provides good retrieval precision as well. The primary goal is to provide a flexible and modular indexing scheme for descriptor evaluation and feature selection purposes, but it can be used for generic content-based retrieval tasks as well.**

I. Introduction

The field of content based image/video retrieval contains a lot of different indexing structures, schemes and methods aiding the related retrieval tasks. The goal of the indexing structures is to take the available datasets and the extracted descriptors/features, and produce a concise and easier to handle index which can be used to search for similar content. From the practical point of view, those index structures are the most beneficial and usable, which can include a very large number of dataset elements, and at the same time be able to produce search results quickly. In this paper we will present an indexing and retrieval scheme which has similar features, but with a slightly different goal: we wanted to create an indexing method with the goal of easy descriptor evaluation capabilities, robustness against the addition and removal of features without the need of complete index rebuilding, high modularity, and at the same time capable of quick servicing of nearest neighbour-types of queries with high precision.

Some of the traditional and well-known indexing structures include KD-trees [1] which provide $\log N$ time nearest neighbour searches among k dimensional feature vectors by creating a tree structure based on space partitioning along the k dimensions. Here we need to have a dissimilarity metric between two feature vectors, and we need to be able to calculate the distance between two vectors from accumulating the partial differences along the dimensions. R-trees [2] are similar in the sense that they are also space partitioning trees, with the main difference that here the nodes represent not space partitioning planes, but k-dimensional bounding boxes which can be overlapping. DBM-trees [3] are density-based structures where the height of the tree is larger in denser regions to achieve a trade-off between breadth-searching and depth-searching, with the goal of minimizing disk reads and improving on the performance of M-trees (which is a string matching metric tree). Vantage point trees [4] address the indexing of such datasets whose elements can not be represented as classical feature vectors and can only work with available pairwise distance information. NV-trees (nearest vector trees) [5] are disk-based indexing structures for very large datasets, providing good approximate nearest neighbour retrievals based on a single random disk read using a combination of projections of data points to lines and partitioning of the projected space. BK-trees [6] have been used for string matching algorithms, essentially being representations of point distributions in discrete metric spaces. The tree is built so as to have each sub-tree contain sets of strings that are at the same distance from the sub-tree's root. Our presented indexing scheme is based on such trees, and this last property is an important one for our purposes (to be detailed later).

Traditional indexing schemes work by picking a set of descriptors/features, creating a high dimensional feature vector based on them, and use it to create an index of a dataset, which in turn can be used to produce responses to queries. However, in such schemes, when we want to add or remove features, the whole index needs to be recreated, which can take a lot of time for large datasets. Our goal was to create a scheme where features (and indices built from them) can be added and removed quickly in a modular architecture, and a query service which can take whichever indices are available and be able to produce results using all of the indices. This reproduces the classical behaviour of multi-feature querying, but at the same time is more flexible and easier to manipulate.

In the presented scheme we build index trees for each descriptor, each having its own metric and distance functions. This enables us to combine a lot of different feature descriptors, without the need for common normalization (i.e. joint equal contribution or some logistic regression). In this paper we will present such an indexing and retrieval scheme, along with evaluation data. The main idea is that we will produce separate index structures in parallel for all currently available descriptors on the dataset, while the query service picks up these indices, performs parallel searches over them, and produces a ranked combined result list. When a new descriptor becomes available, the indexer just builds its separate index, the query service picks it up, and includes it in the later retrievals.

The main contributions of this paper are: i). providing a flexible and modular indexing scheme for evaluating large descriptor sets; ii). presenting the difference interval based BK*-tree structure for fast nearest neighbour types of searches; iii). presenting an associated result ranking retrieval step for servicing high-precision multi-feature content-based queries.

We will compare our approach with a classical multi-feature indexing scheme, using two datasets, and will show that the proposed approach is highly modular, fast, produces good precision rates in *k*NN-type (*k* nearest neighbours) of retrievals, and is a viable scheme serving as the basis of higher level descriptor evaluation and feature selection methods.

II. Descriptors and datasets

We use two datasets for the purposes of this paper. The first one is the MIRFLICKR25000 dataset [7], which contains 25000 images gathered from Flickr, along with tags, and roughly partitioned into 24 categories with lots of overlaps (detailed description can also be found at http://press.liacs.nl/mirflickr/). Another dataset we used is our own video and image dataset (which we will call CDB7000 here) which contains 7000 video segments collected from television captures in 13 categories, e.g. sports, nature, cartoons, music, cooking, news, street surveillance, outdoor, indoor. The videos were automatically cut into shots and for each shot a representative frame was automatically extracted - these representative frames will be used for the purposes of this paper.

For both datasets, we selected a set of 10 descriptors, using which we extracted the features for all images, and stored them in a MYSQL database. These descriptors were the following: average colour (custom descriptor gathering average representative colour over local image areas), curvelets [8], colour layout, colour structure, dominant colour, edge histogram, homogeneous texture [9], LBP [10], PHOG [11], focus regions [12]. However, it is important to note that there is no limitation regarding the number of used descriptors. As for the size of the datasets used, since the current implementation uses in-memory indices, there could be physical memory limitation in case of very large sets (which can be circumvented with a disk-based implementation).

III. Index structure

As an indexing structure we use a variant of in-memory BK-trees [6] which we first introduced in [13], with nodes containing an arbitrary number (descriptor dependent) of children, each representing an interval of difference between a node and its parent. We call these trees as BK*-trees.

This structure can be used to build quickly searchable index trees for any descriptor which has a metric. We build these trees for every descriptor we use, separately. The query service will then pick up these index trees and perform queries upon them, when requested by the retrieval interface. Naturally, these trees are only able to generate results for a single descriptor at a time, but all of them can be used when performing multi-dimensional (i.e. multi-feature) queries.

Fig. 1 shows a simplified diagram of the indexing-retrieval scheme. As described above, the indexer creates indices for each descriptor. When a new descriptor is added, its index will be built, and added to the existing ones, without the need to rebuild the others. The query service reads and uses all available indices.

The original BK-trees have been used in string matching algorithms. Essentially they are representations of point distributions in discrete metric spaces. For classical string matching purposes, the tree is built so as to have each sub-tree contain sets of strings that are at the same distance from the sub-tree's root, i.e. for all l leaves below sub-root sr the difference $d(l, sr) = \varepsilon$ is constant.

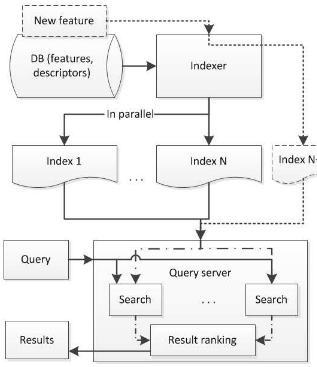


Fig. 1. Simplified architecture of the framework. The indexer builds indices in parallel from the selected features; the query server serves results based on the available indices. For a new feature, the other indices are not rebuilt.

In our case, the used structure contains tree nodes that can have an arbitrary number of children (M), where the leaves below each child contain elements for which the distance d falls in a difference interval: $d(l, sr) \in [\varepsilon_i; \varepsilon_{i+1})$, where $i \in [0, M-1] \cap \mathbb{N}$. The distance intervals in the child nodes (denoted by $\varepsilon_i, \varepsilon_{i+1}$ above) depend on the maximum error E_{max} that the feature-dependent distance metric can have, more specifically, $\|\varepsilon_{i+1} - \varepsilon_i\| = E_{max}/M$, thus the difference intervals are linearly divided buckets. This is a very important aspect of this indexing scheme, since when performing nearest neighbour types of searches we can decide very early in the process that which sub-trees of the index will contain all of the nearest neighbours (i.e. elements that fall in a specified distance interval).

This property was one of the main reasons for starting to use the BK* structure in the first place, since it can enable to discard large irrelevant parts of the index tree during the query process (it can be decided early on that all the result candidates which are in the vicinity of the query are located in one or more sub-trees). Also, in case of eventual larger datasets, the query process of a single tree can also be easily parallelised for higher efficiency, forking the process along the selected candidate sub-trees and combining the results in the final ranking step.

If we have feature points with an associated distance metric, then we can populate a BK*-tree with these points in the following way:

1) Pick one of the points as the root node, n_R.
2) In a tree (i.e. for a descriptor) each node will have a constant number of M child nodes.
3) A point P_j will be placed into the child node N_i ($i = 0...M-1$), if

$$i \cdot \frac{E_{max}}{M} < d(P_i, P_j) < (i+1) \cdot \frac{E_{max}}{M} ,$$

where P_j is P_i's parent node. Thus, a node will contain a point if its distance from the parent falls into the interval specified above; each node representing a difference interval $\varepsilon_i, \varepsilon_{i+1}$=$[i \cdot d/M; (i+1) \cdot d/M)$.

4) Continue recursively until there are no more points left to insert.

IV. Querying

In this section we will present a querying scheme based on the above index structure which enables the testing of retrievals using any number of available descriptors/features, providing efficient parallel content-based searches.

Given a content-based query (Q), the index trees are searched for similar entries (all trees are searched in parallel, based on the different descriptors and their distance metrics):

1) If $d_0 = d(Q, n_R) < t$ (t is an adjustable threshold), then root n_R element is a result. Let $t_1 = d_0 - t$ and $t_2 = d_0 + t$.
2) For each node P_i (where $P_0 = n_R$ and $d_0 = d(Q, n_R)$) which has M intervals (children) c_j, $j = 1...M$, if

$$\left\{ \begin{array}{ll} j\frac{E_{max}}{M} \in [t_1, t_2] & \text{or} \\ (j+1)\frac{E_{max}}{M} \in [t_1, t_2] & \text{or} \\ j\frac{E_{max}}{M} \leq t_1 \text{ and } (j+1)\frac{E_{max}}{M} \geq t_2 & , \end{array} \right. \quad (1)$$

then:

   a) let $d_j = d(Q, c_j)$,
   b) if $d_j < t$ then c_j is a result,
   c) update $t_1 = d_j - t$ and $t_2 = d_j + t$ and iterate step 2.

After getting all result candidates from all used index trees, the results are retrieved by ordering them by placing those in front, which are closer to the query according to multiple descriptors. Basically, if we imagine that all descriptor graphs are viewed from the location of the query node, then results are retrieved from a high dimensional spherical neighbourhood around query Q.

Thus, we perform multi-feature queries not by creating very high dimensional feature vectors in a single index structure, but by building separate index trees for all descriptors, and performing parallel searches for the query in all indices, then combining the results. This enables us to easily extend the framework with descriptors without the need to rebuild the entire dataset index when adding or removing a descriptor.

An important issue to note is that in this scheme *k*NN-type searches get translated into:

- finding the closest matches to a query in each index, then
- combining the result lists, and
- returning the first *k* best matches.

For creating the combined retrieval lists from multiple descriptors, we define a weighting scheme which gives a result a higher overall rank if it also has high individual ranks according to multiple descriptors. This weighting scheme improves the retrieval process in itself (e.g. compared to producing an interleaved merging of the results), but we will show that it can be further improved by exploiting the descriptor ranking information we obtain from the graph analysis steps.

While this result list combination step is a kind of rank aggregation [14] of partial lists, the purpose of this paper is not the creation of a Kemeny-optimal aggregation method. However, our ranking scheme can be thought of as a special form of Borda scoring of partial/top-*k* lists, without assigning scores to elements not present in a specific list, but producing a final ranked list which is based on the aggregation of weights obtained from individual result lists. The evaluation of this ranking procedure itself could be the topic of a separate paper.

Let us denote by $R = \cup_{j=1}^{n_d} R_j$ the combined retrieval result list from all indices (n_d is the number of descriptors). If R_j is an ordered result list obtained by using descriptor d_j , ($j = 1...n_d$), and ordered by increasing distance δ from the query node q, then let $r_{j,i}$ be the identifier of the i^{th} result in R_j:

$$R_j = \{r_{j,i} | 0 \leqslant i < n_j, j = 1...n_d, \delta(r_{j,i}) \leqslant \delta(r_{j,i+1})\}, \quad (2)$$

where n_j is the number of results in R_j. Each $r_{j,i}$ will be assigned a weight

$$\hat{w}(r_{j,i}) = \left\{ \begin{array}{ll} 1 - i/n , & \text{if } i < n/2 , \\ 0, & \text{otherwise.} \end{array} \right. \quad (3)$$

where $n = |V|$ the number of all dataset elements. The main goal of the retrieval process is to have relevant results at the beginning of the result list. In our retrievals we retrieve a fixed k number of best matches, and we would like to have a high precision among these k responses. However, when searching in the separate index trees, we do not enforce this k limit (only when creating the final combined result list), thus the search from an index can produce different numbers of individual results. It is because of this property that we use the $i < n/2$ limit in Eq. 3, assuming no single category has more elements than half of the dataset: should an extreme case occur when the number of results from an index return too many results (in this case more than half the cardinality of the whole dataset), then we do not assign special weight for results above this limit, since practically most of them would be irrelevant anyway: it is important to keep in mind, that the individual result list of an index is in itself an ordered list (according to the specific descriptor used of the index), thus realistically - and if the descriptor's metric is valid - results beyond the $n/2$ limit should not be relevant.

In the final result list E, if a result e occurs in multiple result lists R_j, then the total weight of e will be the sum of all its weights from all R_j:

$$W(e) = \sum_{j=1}^{n_d} \sigma(j) , \quad (4)$$

where

$$\sigma(j) = \left\{ \begin{array}{ll} \hat{w}(r_{j,i}), & \text{iff } e = r_{j,i} \in R_j , \\ 0, & \text{otherwise.} \end{array} \right. \quad (5)$$

and results will be retrieved ordered by their decreasing weights W:

$$E = \{e_k | k > 1, W(e_k) \geqslant W(e_{k+1})\}. \quad (6)$$

Thus, the final position of a result will be better if it has a higher position by multiple descriptors.

V. Evaluation

As described above, our primary goal here is not to create a best performing retrieval method, but a framework suitable for the evaluation of large descriptor sets. However, it is also important to see how the proposed framework behaves inside a retrieval scheme, to prove its overall usability and viability.

Thus, as a basis of comparison for evaluating the usability and performance of the proposed scheme, we used the hierarchical K-means tree based indexing from the FLANN library (Fast Library for Approximate Nearest Neighbors) [15] (also available at http://www.cs.ubc.ca/~mariusm/index.php/FLANN/FLANN). We built the index by inserting feature identifiers into the tree and creating a custom distance functor class which pulls the feature vectors from the database and performs the combined difference calculations. The above BK* indices also pulls the features from the database, so this step does not introduce additional complexity of running time differences. The combined feature vectors have variable lengths, containing all the features of the selected 10 descriptors (mentioned above). The length of a full feature vector is approx. 11000 bytes. The difference of two vectors is the sum of all differences, i.e. ($N = 10$)

$$d(F_i, F_j) = \sum_{k=1}^{N} d_k(F_i(k), F_j(k)), \tag{7}$$

where F_i,F_j are the full feature vectors and d_k is the distance function associated to feature k.

The first n best matches (50 in the tests) were obtained by using the *knnSearch* function to find the 50 nearest neighbours around a query. Here, the matches are already a result of a combined multi-feature querying process, thus in this case there is no further need for additional re-arrangement or re-ranking of the result list since it will not affect the precision of the results.

We performed comparisons of the proposed scheme with this approach, from the points of view of indexing time, retrieval time, retrieval performance (precision and average precision), in order to show that the proposed method is fast in building indices, retrieving results, and has good retrieval performance when combined with the above described result ranking step. All these properties combined with the design points of high modularity and the easy descriptor plug-in/out capabilities prove the presented scheme's usability for descriptor evaluation purposes, and also for regular content based retrieval tasks.

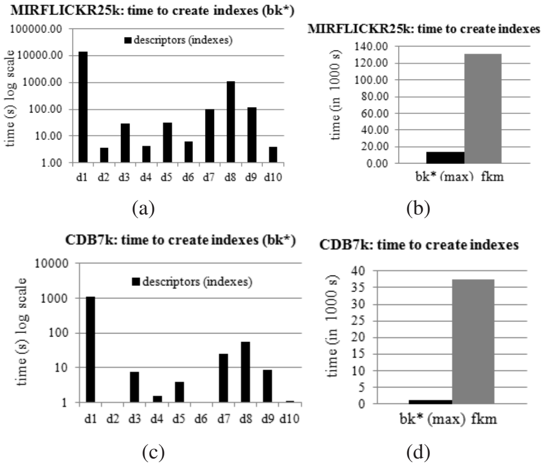


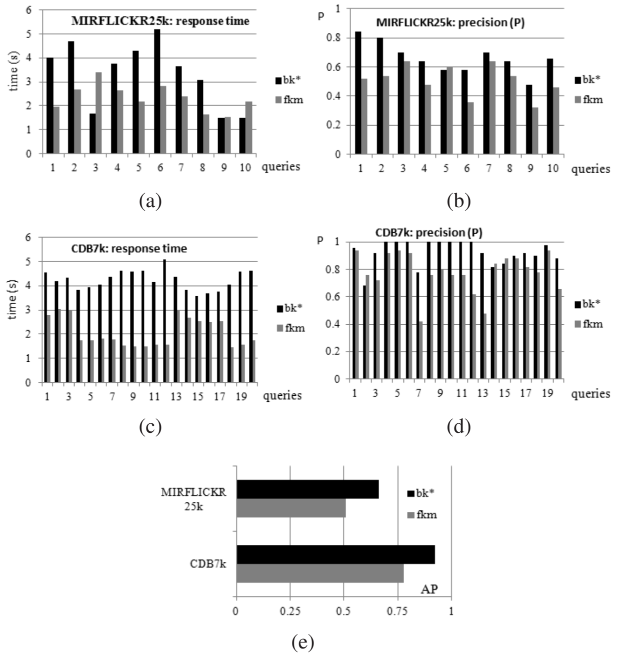
Fig. 2. Time to create the indices for the MIRFLICKR25000 dataset by the proposed (*bk**) and the FLANN/Kmeans (*fkm*) indexing scheme. (a) Indexing time with the proposed scheme for each descriptor. (b) Comparing the total indexing times of the proposed and the FLANN/Kmeans scheme. (c-d) The same time values in the case of the CDB7000 dataset.

Fig. 2 shows times to create the indices for the MIRFLICKR25000 and the CDB7000 datasets, by using the FLANN/Kmeans index structure (denoted by *fkm*) and the proposed BK*-tree indices (denoted by *bk**). In this figure (a) and (c) show - in logarithmic scale - the time to create the separate indices for each descriptor by *bk**. The indices are built in parallel (in a multi-threaded implementation using OpenMP http://openmp.org), and the overall total time to build all indices can be thought of as the maximum build time from all the indices (since all other indices will be finished earlier). Thus in Fig. 2 (b)-(d) we compare the slowest building time from the *bk** indices with the indexing time of the FLANN/Kmeans scheme. Overall *bk** indexing takes less time by more than an order of magnitude (in this case 4 hours vs. 36 hours and 19 minutes vs. 626 minutes respectively). This would be important in the case when this structure is used inside a complete retrieval framework, providing the capability for fast re-indexing of the dataset.

Then, in Fig. 3 we present information regarding the retrieval time (time to give a response to a query) and performance (in precision). While in average *bk** takes 1.5-2 times more to produce the results as *fkm*, here we have to note that in *bk** the query service searches the index trees in parallel, but we have to wait for all searches to finish before producing the combined ranked results. Thus the overall response time of *bk** will be the slowest response of all indices. However, Fig. 3 (b)-(d) show that *bk** with the associated ranking retrieval has better retrieval performance regarding precision, which is an important issue when searching for first k best matches (nearest neighbours).

Here the precision is measured as the number of relevant results from the first best 50 matches in all cases. For both datasets, a result is deemed relevant, if it has at least one category/class which is common with the query's. Fig. 3 (e) shows the average precision values - averaged over all performed queries - for both datasets and indexing schemes, showing the better overall performance of the proposed scheme.

Regarding the similarity threshold used in Sec. IV, its important property is that it determines which dataset elements will be treated as "close enough" by an individual index, when gathering the results. If it is set too high, then a large number of elements will be treated as similar to the query which also means a very large percentage of the index tree will be walked when gathering result candidates. Thus, as a general rule, in the proposed scheme we always set this threshold so as the percentage of the visited index tree nodes will stay below 50%. We used such settings to produce all the above presented measurements. Fig. 4 shows the percentages of visited nodes.



(a) (b) (c) (d) (e)

Fig. 3. Response times (time to answer a query) and precision (P) values for several queries, on the MIRFLICKR25000 (a-b) and the CDB7000 (c-d) datasets. (e) Average precision values for the two datasets over all queries.

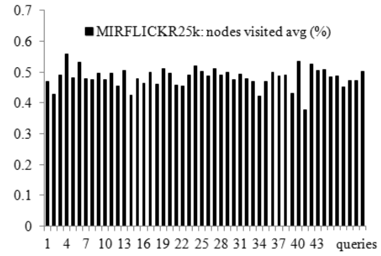


Fig. 4. Percentage of nodes visited during a *bk** retrieval for the MIRFLICKR25000 dataset averaged over descriptors for each query.

VI. Conclusions

We have presented an indexing scheme usable for the evaluation of large descriptor sets and at the same time providing quick multi-feature retrieval capabilities. The goal was to create a scheme which is modular, independent on the number of used features/descriptors, and does not need to rebuild indices when adding or removing descriptors. The used BK-trees based structure provides a fast kNN-type search capability by structuring dataset elements in an index tree based on distance intervals. The resulting indexing scheme combined with the presented result ranking retrieval provides a robust framework for descriptor evaluation and testing content based similarity searches.

Also, while the index structure of the proposed scheme is currently an in-memory one (all the indices need to be kept in memory by the query service to perform retrievals), it can be fairly easily translated into a disk-based storage design (in order to overcome available memory limitations in case of very large datasets), where nodes would be folders/directories, subnodes would be sub-directories, and feature data of the nodes would be files in a directory. For such a scheme to work in a usable manner, very high speed disk solutions (e.g. high performance SSDs) would be needed.

Acknowledgment

This work has been supported by Hungarian Scientific Research Fund (OTKA) grant nr. 83438.

References

[1] J. H. Friedman, J. L. Bentley, and R. A. Finkel, “An algorithm for finding best matches in logarithmic expected time,” *ACM Trans. on Mathematical Software*, vol. 3, no. 3, pp. 209–226, 1977.

[2] A. Guttman, “R-Trees: A dynamic index structure for spatial searching,” in *Proc. of the ACM SIGMOD Intl. Conference on Management of Data*, vol. 14, no. 2, 1984, pp. 47–57.

[3] M. R. Vieira, C. Traina, F. J. T. Chino, and A. J. M. Traina, “DBM-Tree: A dynamic metric access method sensitive to local density data,” *Journal of Information and Data Management*, vol. 1, no. 1, pp. 111–127, 2010.

[4] A. W.-C. Fu, P. M.-S. Chan, Y.-L. Cheung, and Y. S. Moon, “Dynamic vp-tree indexing for n-nearest neighbor search given pair-wise distances,” *Intl. Journal on Very Large Data Bases*, vol. 9, no. 2, pp. 154–173, 2000.

[5] H. Lejsek, F. H. Asmundsson, B. T. Jonsson, and L. Amsaleg, “NV-tree: An efficient disk-based index for approximate search in very large high-dimensional collections,” *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 31, no. 5, pp. 869–883, 2009.

[6] W. Burkhard and R. Keller, “Some approaches to best-match file searching,” in *Proc. of CACM*, 1973.

[7] M. J. Huiskes and M. S. Lew, “The MIR Flickr retrieval evaluation,” in *Proc. of ACM International Conference on Multimedia Information Retrieval (MIR)*. ACM, 2008, pp. 39–43.

[8] E. Candès, L. Demanet, D. Donoho, and L. Ying, “Fast discrete curvelet transforms,” *Multiscale Modeling & Simulation*, vol. 5, no. 3, pp. 861–899, 2006.

[9] B. S. Manjunath, J. R. Ohm, V. V. Vasudevan, and A. Yamada, “Color and texture descriptors,” *IEEE Trans. on Circuits and Systems for Video Technology*, vol. 2, no. 6, pp. 703–715, 2001.

[10] T. Ojala and M. Pietikainen, “Multiresolution gray-scale and rotation invariant texture classification with local binary patterns,” *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 24, no. 7, 2002.

[11] A. Bosch, A. Zisserman, and X. Munoz, “Representing shape with a spatial pyramid kernel,” in *Proc. of the 6th ACM international conference on Image and video retrieval (CIVR)*, 2007, pp. 401–408.

[12] L. Kovács and T. Szirányi, “Focus area extraction by blind deconvolution for defining regions of interest,” *IEEE Tr. on Pattern Analysis and Machine Intelligence*, vol. 29, no. 6, pp. 1080–1085, 2007.

[13] L. Kovács, Á. Utasi, and T. Szirányi, “Visret - A content based annotation, retrieval and visualization toolchain,” in *Proc. of ACIVS (Advanced Concepts for Intelligent Vision Systems) Lecture Notes in Computer Science*, vol. 5807, 2009, pp. 265–276.

[14] C. Dwork, R. Kumar, M. Naor, and D. Sivakumar, “Rank aggregation methods for the Web,” in *Proc. of the 10th international conference on World Wide Web*, 2001, pp. 613–622.

[15] M. Muja and D. G. Lowe, “Fast approximate nearest neighbors with automatic algorithm configuration,” in *Proc. of International Conference on Computer Vision Theory and Applications (VISAPP)*, 2009, pp. 331–340.