



UNIVERSIDAD DE BUENOS AIRES
FACULTAD DE CIENCIAS EXACTAS Y NATURALES
DEPARTAMENTO DE COMPUTACIÓN

Efficient Large-scale image search with a vocabulary tree

Tesis presentada para optar al título de
Licenciado en Ciencias de la Computación

Esteban Uriza

Director: Francisco Gomez Fernandez

Codirector: Martin Rais

Buenos Aires, 2016

BÚSQUEDA EFICIENTE DE OBJETOS EN IMÁGENES MEDIANTE UN ÁRBOL DE VOCABULARIO

La tarea de buscar y reconocer objetos en imágenes se ha convertido en un importante tema de investigación en el área de procesamiento de imágenes y visión por computadora.

Una solución general al problema que no puede aprovechar características particulares de un dominio específico en las imágenes de entrada, requiere procesar mucha información en la imagen. Si el reconocimiento se plantea sobre un número importante de imágenes, el volumen de datos crece muy rápido.

Redes sociales en la web, aplicaciones en smart-phones, etc. plantean la necesidad de resolver este problema a gran escala ya que cada vez trabajan con volúmenes de datos más grandes que superan el orden de los millones de imágenes. Poder desarrollar métodos eficientes en términos de búsqueda e indexación, que devuelvan resultados correctos en tiempo real representa un desafío importante.

En este trabajo se estudia el enfoque de Bag of Features, en particular la variante denominada vocabulary tree. En esta variante, métodos de clustering jerárquico son aplicados sobre descriptores locales de imágenes, para formar un vocabulario visual jerárquico. Para indexar las imágenes y crear el dataset de índices invertidos, los descriptores son cuantizados en términos del vocabulario para formar vectores esparsos, que permiten computar un ranking de similaridad entre imágenes de manera muy eficiente.

Se incluyen también explicaciones detalladas del método con ejemplos que permiten un mejor entendimiento y reproducibilidad de los resultados.

Se realiza además un análisis del impacto de la performance del método variando distintos factores tales como: los parámetros sobre la construcción del vocabulario y distintas técnicas de extracción de descriptores locales. Se observa que la performance de recuperación (retrieval) aumenta con un vocabulario más rico y decae muy lentamente a medida que el tamaño del dataset crece. Se muestra que los descriptores open source KAZE y AKAZE probaron tener resultados iguales o superiores a los métodos clásicos SIFT, SURF y ORB.

Los experimentos realizados muestran que aplicar una reducción de dimensionalidad a los descriptores, usando técnicas como PCA, mejoran o igualan la calidad de la recuperación de imágenes, permitiendo reducir los requerimientos de memoria necesarios y brindando al método de una mayor escalabilidad aún.

Finalmente se muestran diversas aplicaciones concretas para el método como búsqueda de objetos o escenas en videos, reconocimiento de billetes de banco o de etiquetas de vino, contando con una demo funcional online.

Palabras claves: árbol de vocabulario, búsqueda escalable, procesamiento de imágenes, visión por computadora, reconocimiento de objetos en imágenes, bag of features.

EFFICIENT LARGE-SCALE IMAGE SEARCH WITH A VOCABULARY TREE

The task of searching for and recognizing objects in images has become an important research topic in the area of image processing and computer vision.

A general solution to the problem that can not take advantage of particular characteristics of a specific domain in the input images, requires processing a considerable amount of information from the image. If the recognition is done on a large number of images, the volume of data to be processed grows very fast.

Social networks on the web, applications on smart phones, etc. settle the need to solve this problem on a large scale as they have to work increasingly with larger volumes of data, exceeding the order of millions of images. Thus, being able to develop efficient methods, that return correct results in real time represents a major challenge.

In this paper the Bag of Features approach is studied, in particular the variant called vocabulary tree. In this variant, hierarchical clustering methods are applied to local image descriptors to form a hierarchical visual vocabulary. In order to index the images and create the inverted indexes, the descriptors are quantized in terms of the vocabulary forming sparse vectors, which allows computing a ranking of similarity for images very efficiently.

Detailed explanations of the method with examples that allow a better understanding and reproducibility of the results are included.

The impact on method performance is analyzed varying different factors such as: the parameters on the vocabulary construction and different techniques of local descriptors extraction. It can be observed that the retrieval performance increases with a richer vocabulary and decays very slowly as the size of the dataset grows. New open-source descriptors KAZE and AKAZE proved to have results equal to or greater than the classic ones SIFT, SURF and ORB methods.

The experiments also show that applying a reduction of dimensionality to the descriptors, using techniques such as PCA, improves or equals the quality of the image recovery, allowing to reduce the necessary memory requirements and giving to the method even greater scalability.

Finally, several specific applications for the method are shown, such as searching for scenes in videos, recognition of bank notes or wine labels with a functional demo on-line.

Keywords: vocabulary tree, scalable search, image processing, computer vision, image object recognition, bag of features.

Índice general

1.. Introduction	1
1.1. Image Similarity	1
1.2. Large Scale Object Recognition	1
1.3. Local invariant features	1
1.3.1. Features Detection	2
1.3.2. Features Description	3
1.4. Local Features Memory Consumption	3
2.. Related Work	4
2.1. Holistic Methods	4
2.2. BoF based methods	5
2.3. Hashing Techniques	6
3.. Bag of Features representation	7
3.1. Review of text retrieval	7
3.2. Building a visual vocabulary	7
3.3. Visual indexing and weighting (TF-IDF)	7
3.4. Object Retrieval - Inverted Index Files	8
3.5. Spatial consistency	8
3.6. Bag of Features method summary	9
4.. Vocabulary tree approach	11
4.1. Training Phase	11
4.1.1. Keypoint Detection and Descriptor Extraction	11
4.1.2. Building the visual vocabulary	11
4.1.3. K-Clustering feature vectors to build vocabulary	13
4.1.4. Vocabulary Memory Consumption	13
4.1.5. Building inverted indexes	14
4.1.6. Building Bag of Features	14
4.1.7. Normalizing BoFs	15
4.2. Runtime Phase	17
4.2.1. Scoring	17
4.2.2. Query	17
4.3. Example	17
4.3.1. Building the Tree	17
4.3.2. Inverted Indexes Creation	19
4.3.3. Computing BoFs	21
4.3.4. Normalizing BoFs	24
4.3.5. Performing Query	25

5.. Results	28
5.1. Data Sets	28
5.2. Evaluation Metrics	28
5.3. K-H best values	30
5.4. Performance vs training size	30
5.5. Performance vs database size	30
5.6. Performance vs dimension reduction	33
6.. IPOL Demo	36
7.. Applications	37
8.. Conclusions	41
8.1. Future work	41
Appendices	43
.1. Scoring Derivation	44
.2. K-Means Clustering	45
.2.1. External K-means clustering	47
.3. Implementation Details	47
.3.1. Tree representation using an array	47

1. INTRODUCTION

1.1. Image Similarity

In this work we are going to address the problem of searching for similar images in a large scale set of images. Similar images are defined as images of the same object (or scene) viewed under different imaging conditions (rotations and scale changes coming from pictures taken from different points of views, changes on illumination conditions, partial occlusions, noise). See figure 1.1.



Fig. 1.1: Similar Images with Different Imaging conditions - scale, rotation, light changes, occlusion, etc.

1.2. Large Scale Object Recognition

A large scale object recognition system, takes an image query Q as input, performs a search process over a large scale image data set (more than one million images), and retrieves a ranked short list of images in the dataset as output. See figure 1.2.

As the size of the data set is large, is not possible to perform a linear scan, taking image by image and computing its similarity against the query at run-time. Thus, this kind of systems use data preprocessed in a training stage. Query response time is constrained and should be in the order of the state-of-the-art text document retrieval web engines (milliseconds). It is necessary to find efficient data structures to index precomputed retrieval data, that is used in the search. Indexing structures must fit entirely in RAM, accessing to hard disk drives will be simply too slow for this requirements. Such kind of restrictions makes the problem challenging.

1.3. Local invariant features

A local feature is an image pattern which differs from its immediate neighborhood. It is usually associated with a change of an image property or several properties simultaneously, although it is not necessarily localized exactly on this change. The image properties commonly considered are intensity, color, and texture.

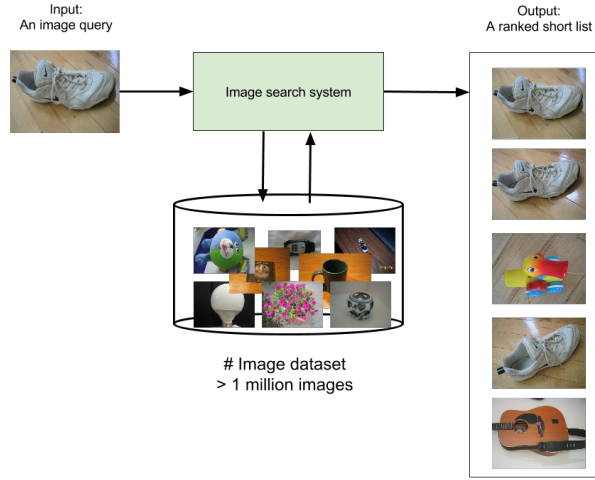


Fig. 1.2: large scale image search system



Fig. 1.3: Local invariant feature detection, extraction and matching

Given an image as input, this technique typically extracts features and then some measurements are taken from a region centered on each features generating a set of descriptors as output. Similarity estimation can be achieved by matching features by the descriptors similarity. See figure 1.3.

1.3.1. Features Detection

Most descriptive features are located in special positions in the images, such as mountain peaks, building corners, or patches of snow.

Given two or more images of the same object, a good features detection algorithm should find a high percentage of the same features in all the given images despite different imaging conditions. Typically corners are used as features (but in some cases blobs are also considered good features).

Here we mention some examples of popular feature detection algorithms: HARRIS [13], GFTT [39], MSER [28], SIFT [25], SURF [6], FAST [34], STAR (CenSurE) [1], ORB [35], BRISK [24], KAZE [4], AKAZE [3].

Feature detection can be expressed as a function $Detect_{md}$ that takes an image as

input and returns a set of features.

$$Detect_{md}(I) = \{ft_1, \dots, ft_n\} \quad (1.1)$$

where md is a feature detection method technique.

1.3.2. Features Description

Once features are detected, it is possible apply a descriptor extraction algorithm which gives as result the descriptors of the image. A descriptor is often expressed as a D-dimensional vector that describes the appearance of the patches of pixels surrounding the feature locations.

Examples of popular descriptor extraction algorithms are

SIFT [25], SURF [6], BRIEF [7], BRISK [24], ORB [35], FREAK [2], KAZE [4], AKAZE [3].

Generally, descriptor extraction can be expressed as a function $Extract_{mx}$ that takes an image I and a set of detected features Ft , and returns a set of descriptor vectors

$$Extract_{mx}(I, Ft) = \{\alpha_1, \dots, \alpha_n\}, \quad (1.2)$$

where mx is a feature description method. $\alpha_i \in \mathbb{R}^D$ are the are the resulting the descriptor vectors of the image I .

Many local features techniques, specify both a detection and an extraction method (SIFT, SURF), although there are other techniques that only specify a detection (HARRIS, MSER), or an extraction method (BRIEF, BRISK). Thus we could employ different combinations from feature detection and feature extraction, but in practice not all combinations are compatible, and only a few combinations will give us good results.

1.4. Local Features Memory Consumption

In this work we are using local invariant features technique. The amount of memory required to store the features of an image is simply the memory consumed by a single extracted feature descriptor, multiplied by the amount of all extracted descriptors for that image. Memory consumption of a single descriptor depends on the description technique used. For example, each descriptor SIFT has 128 dimensions, which if they are represented by chars, it will consume 128 bytes per descriptor.

The number of features also varies depending on the chosen method, the intrinsic parameters of each method employed, and also varies depending on elements such as the size of the images (larger images will give us more features), as well as nature images (images with trees with many leaves, or rough surfaces, will give many more features than images of flat surfaces).

For example, if we have a set of one million images, and we use SIFT as extracting process. Let's assume that the images produce 1,000 descriptors SIFT in average (optimistic number with real images), we would need $128bytes \times 1,000descriptors \times 1,000,000images \sim 120Gigabytes$ to store all descriptors. **It does not fit in RAM** for current standard hardware machines, and it does not scale well if we add more images.

2. RELATED WORK

There are currently two main approaches to face this problem. The first approach known as “holistic” uses global image descriptors. The second one is based on processing local invariant descriptors, and is called “bag of features” (BoF). Both approaches have its strengths and weaknesses. Recently some authors are combining both approaches for achieving better results. BoF technique has proven to be good enough working with image sets in the order of one million images, and it has had better accuracy than holistic methods. However in the context of indexing hundreds of thousands of images, as occurs in most popular web sites, is not clear how BoF techniques can be applied. The main problem is that the indexing structure needed wouldn’t fit in RAM, and accessing to HDD would severely damage the efficiency. In this context is where begins to be suitable the use of the holistic approach although its limitations.

2.1. Holistic Methods

We can mention GIST [31] as one of the first holistic method. The idea of GIST is to generate a global representation of the scene using a low dimension vector. This vector is built according to a set of perceptual dimensions (naturalness, openness, roughness, expansion, ruggedness) that represent the dominant spatial structure of the scene. These dimensions can be estimated using spectral information. Each image is divided in a 4 by 4 grid, where orientation histograms are extracted. In [41] y [42] different strategies are proposed to compress GIST descriptor.

Recently with the rise of the machine learning techniques, global image descriptors are represented with an unique high dimension vector (about the oder of dozens of thousands dimensions). Some examples include VLADs [18], Fischer Vectors [37], descriptors based on convolutional neural networks [21], [38]. Compression techniques are applied over these global descriptors to generate compact binary codes. Ideally if global vectors could be compressed to 32 or 64 bits without losing precision, then they could fit into RAM. Furthermore if the descriptor is binary it is possible to use the Hamming distance which is very fast to compute. Consequently, several hashing methods have been developed attempting to shrink those codes as compact as possible.

In [32], Fischer Vectors are compressed using Locality Sensitive Hashing (LSH) [14] and Spectral Hashing [42]. Spectral Hashing shows poor performance, and LSH needs thousands of bits in order to have good performance.

In [18] principal component analysis (PCA) is used and random rotations to get a very compact code, although it is not binary.

In [8] a neural networks approach is used to propose a compression scheme based in *Stacked Restriction Boltzmann Machines* (SRBM) making codes between 64 y 1024 bits.

In [27], a graph fusion method is used to combine multiple local descriptors into Fisher Vectors, producing global descriptors. Performance is analyzed in different time and memory constrained scenarios.

2.2. BoF based methods

BoF is covered in detail in section 3. BoF representation is built from a local descriptors set. This methods first extract local descriptors for all the images generating a set of descriptors. Most of authors use SIFT [25] as description technique. BoF main idea comes from text search engines, that in particular exploit inverted files indexing data structures [44]. A document is represented as a sparse vector. Inverted files is a convenient data structure to compute distances between those sparse vectors using *Term Frequency e Inverse Document Frequency* (TF-IDF) weighting mechanisms. Typically k-means clustering is applied over the descriptors set to create a set of “visual words” (this set is called visual vocabulary). Each descriptor is then quantized in its nearest visual word (or nearest cluster centroid) forming the visual vocabulary.

Josef Sivic and Andrew Zisserman first time proposed this technique in [40]. Since then different authors have contributed with improvements.

David Nister and Henrik Stewenius [30] proposed using hierarchical k-means, to arrange visual words in a tree. Their approach named “vocabulary tree” allows to work with a dataset bigger and a potential runtime object addition once the vocabulary is already created.

Works as [23] and [33] used randomized kd-trees, allowing to build the vocabulary much faster, without losing performance.

In [33] a variant of RANSAC [9] is used to re-rank obtained results in function of a local spatial consistency verification of the keypoints distribution, improving search results.

In [29] both methods are compared hierarchical k-means (HKM) and Randomized kd-trees (RKD). The performance depends on the considered data set and the wished system requirements (precision, vocabulary build time, memory constraints). Kd-trees is faster to train, and requires less memory. Is shown that HKM is better than RKD in performance for some data sets, and vice versa.

In [19] two contributions are proposed. The first one called *contextual dissimilarity measure* (CDM) improves precision. It is iteratively computed as correction factor on the distance between visual words based on the nearest neighbors favoring isolated vectors, and penalizing vectors in dense areas. The second contribution allows to speed up query time, using an inverted files structure of two levels. First level partitions the set of images in clusters, then only a subset of images is used in the second level when query is done.

In [15], and [16] additional information is embedded to the descriptors to improve results. Two contributions are proposed. First *Hamming Embedding* (HE) where SIFT descriptor location within the Voronoi’s space cell is binary encoded. Second, weak geometric consistency constraints (WGC) that filters matching descriptors that are not consistent in terms of angle and scale. Both require the use of extra memory.

In [43] greater descriptiveness of the features is explored in order to construct the vocabulary, and the concept of Bag of Features is extended with visual phrases, which have a context and a spatial disposition, in the same way words have in a text document.

The use of SIFT descriptor in de BoF has proven to be successful, but the computation is expensive compared to other description techniques. In [12] the use of ORB [35] within the BoF approach. Computing ORB is up to two orders of magnitude faster than SIFT. One problem is faced here is that k-means algorithm needs to use euclidean distance, but ORB is a binary descriptor, that works with Hamming Distance. Therefore it is necessary to modify the clustering algorithm. In this paper they proposed a similar k-means

algorithm, called *k-majority* based on a scheme of votes.

2.3. Hashing Techniques

More recent work has focused on getting binary codes for visual descriptors or words as compact as possible. The motivation is simple, fitting indexing structures of web scale image data sets in a few Gigabytes of memory, leaves us a budget of only a few bytes per image.

Compact binary codes allow very fast queries on conventional hardware using hashing techniques. The pioneer method for compact binary descriptors is *Local Sensitive Hashing* (LSH) [14]. This method uses random projections, to solve approximated nearest neighbor problem in constant time with elements belonging to a high-dimensional euclidean space.

In [5] is shown that using LSH near elements will have high probability to fall in the same hashing buckets, and it is possible to exploit this fact to solve A-NN in a very efficient way.

This technique requires storing a considerable amount of bytes per descriptor, then is not scalable if it is to be used with local descriptors. Furthermore, the problem becomes intractable when scale in the order of one million images, but it becomes interesting when considering global descriptors.

Recently many hashing algorithms have been developed motivated by generating compact codes [17], [20], [22], [36]. These algorithms “learn” hash functions so that the similarities in data space is preserved in Hamming space.

3. BAG OF FEATURES REPRESENTATION

Bag of Features representation was proposed by Sivic and Zisserman in [40]. The idea of Bag of Features was taken from text retrieval systems (such as Google page rank).

3.1. Review of text retrieval

Text retrieval systems generally employ a number of standard steps. The documents are first parsed into words. Second the words are represented by their stems, for example “walk”, “walking” and “walks” would be represented by the stem “walk”. Third a stop list is used to reject very common words, such as “the” and “an”, which occur in most documents and are therefore not discriminating for a particular document. The remaining words are then assigned a unique identifier, and each document is represented by a vector with components given by the frequency of occurrence of the words the document contains. In addition the components are weighted in various ways, and in the case of Google the weighting of a web page depends on the number of web pages linking to that particular page. All of the above steps are carried out in advance of actual retrieval, and the set of vectors representing all the documents in a corpus are organized as an inverted file to facilitate efficient retrieval. An inverted file is structured like an ideal book index. It has an entry for each word in the corpus followed by a list of all the documents (and position in that document) in which the word occurs. A text is retrieved by computing its vector of word frequencies and returning the documents with the closest (measured by angles) vectors. In addition the match on the ordering and separation of the words may be used to rank the returned documents.

3.2. Building a visual vocabulary

Instead of using text words, the idea is to use “visual words”. In Sivic and Zisserman’s work, descriptors are clustered using k-means algorithm (though other clustering methods are possible). Once clustering is done, each cluster represents a visual word, and the union of all the visual words represents the visual vocabulary. When a new image is processed each descriptor of this image is assigned to the nearest cluster, and this immediately generates matches for all the images in the image set.

3.3. Visual indexing and weighting (TF-IDF)

In text retrieval each document is represented by a vector of word frequencies. However, it is usual to apply a weighting to the components of this vector, rather than use the frequency vector directly for indexing.

Each visual word is assigned with a weight w_i value based on entropy. In text retrieval, this weight is called (inverse document frequency or idf)

$$idf_i = \log \frac{N}{N_i} \quad (3.1)$$

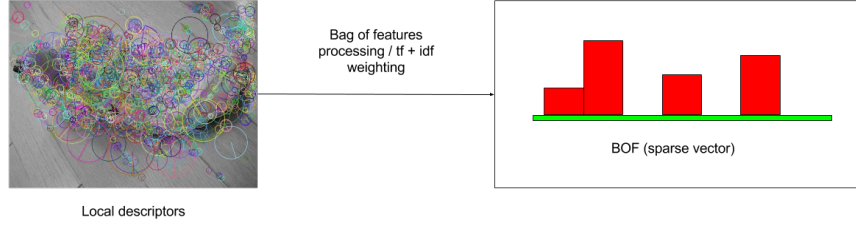


Fig. 3.1: Each BoF d can be thought as a sparse vector histogram of words frequencies

where N is the number of total training images and N_i is the number of occurrences of the word i in the whole database. Note this weighting mechanism penalizes most frequent words (least descriptive) with lower weights.

Each descriptor for each processed image I has descriptors that are quantized into visual words. Term frequency (tf) is defined in order to measure the relevance of a particular word in the image I as follows

$$tf_{I,i} = \frac{nI_i}{nI} \quad (3.2)$$

where nI_i is the number of times that word i appears in the image I , and nI is the total count of words in image I .

Each image produces a Bag of Features, defined as follows

$$d_{I,i} = tf_{I,i} idf_i \quad (3.3)$$

3.4. Object Retrieval - Inverted Index Files

Object retrieval exploits the use of inverted files. In a classical file structure all words are stored in the document they appear in. An inverted file structure has an entry for each word where all occurrences of the word in all documents are stored. Sivic and Zisserman used an inverted file with an entry for each visual word, which stores all the occurrences of the same word in all the images in the set, see figure 3.2. Since the BoF vector is very sparse, the use of an inverted file makes the retrieval very fast.

3.5. Spatial consistency

Text retrieval algorithms increases the ranking for documents where the searched words appear close together in the retrieved texts (measured by word order). Analogously visual words spatial disposition within the images can be used to improve retrieval performance, discarding (or re-ranking) images where visual words disposition is not spatially consistent with the words in the query. Spatial consistency can be measured quite loosely simply by requiring that neighbouring matches in the query region lie in a surrounding area in the retrieved image. It can also be measured very strictly by requiring that neighbouring matches have the same spatial layout in the query region and retrieved image. This kind of spatial consistency considerations could be time consuming, but since it is only applied

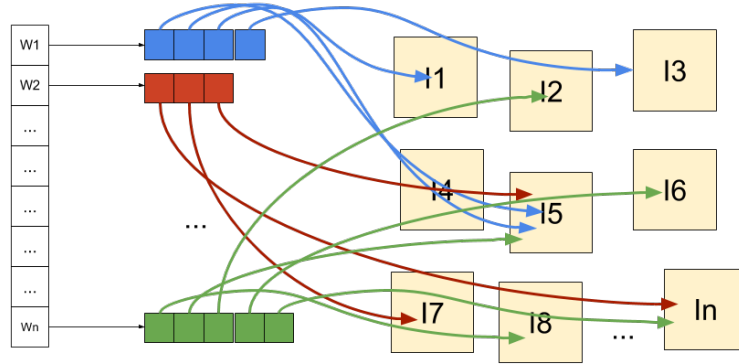


Fig. 3.2: Each cell in the vertical vector represents a visual word, forming the visual vocabulary. Each visual word has a list of pointers to the images where this visual word occurs.

over the short ranked list of retrieved images, it can be implemented in such a way that does not degrade the algorithm retrieval time.

3.6. Bag of Features method summary

In order to solve response time constraints, BoF technique has two phases (off-line phase and on-line phase).

Off-line phase (or training phase). It will be run once, in order to create the pre-computed information. This phase can take a considerable amount of time, depending in the number of input images, and the local feature technique employed (for example SIFT is much more time consuming than ORB). In this phase first features are extracted from all the images in the data set. Then a clustering technique is applied to build the visual vocabulary. All descriptors are quantized to its nearest visual words, producing sparse vectors named Bag of Features. BoFs quantization is arranged in a convenient way using inverted index files.

On-line phase (or run-time phase) occurs when training phase is done. Multiples queries can be run fast, the figure 3.3 summarizes the main steps to solve the query. When the system receives a query image, local descriptors are extracted for that query image. The resulting descriptors are quantized in its nearest visual words, generating the Bag of Features for this query. The query BoF is compared with the precomputed BoFs computed in the training phase. This comparison process is performed efficiently using the inverted indexes files, producing a short list of best ranked images. A further process will improve the output by checking the geometric consistency on the short list to re-rank or filter the results.

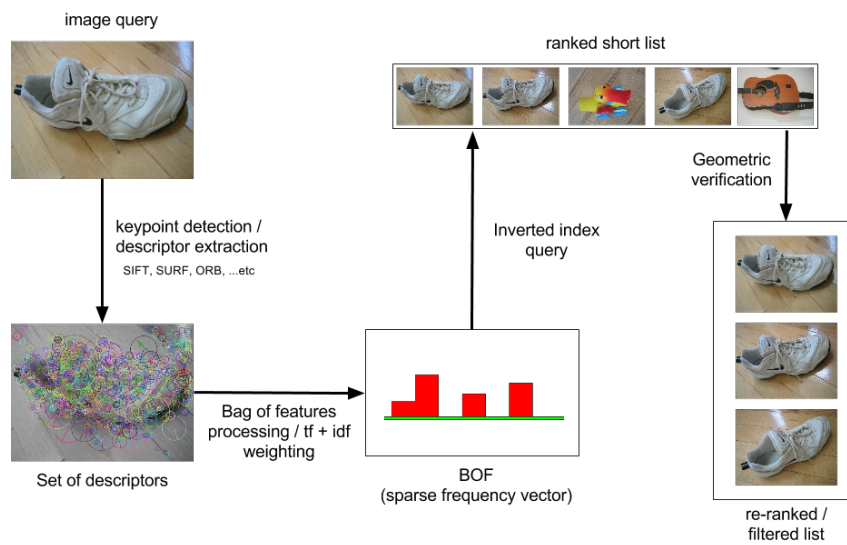


Fig. 3.3: BoF query process summary

4. VOCABULARY TREE APPROACH

Nister et al. [30] approach generalizes Sivic’s idea, adding hierarchical clustering. Thus a visual vocabulary tree is generated.

This approach has several advantages over the previous work. It allows us to use a larger vocabulary, traverse smaller portions of the database to perform the query, and do potential on-the-fly insertion of new objects into the database in run-time phase.

4.1. Training Phase

Algorithm 1 shows a high level overview of training phase process. First it extracts descriptors for all the images in the dataset, then it builds the visual vocabulary, builds the inverted indexes and finally computes the BoFs.

Algorithm 1: Training Phase

TrainingPhase

input : Γ a set of N images, \mathbf{K} number of children per node, \mathbf{H} maximum height for the tree

output: \mathbf{T} the vocabulary tree, **BoFs** the bag of features vectors

begin

```

     $D \leftarrow \text{ExtractDescriptors}(\Gamma)$ 
     $T \leftarrow \text{BuildVocabularyTree}(D, 0, K, H)$ 
     $T \leftarrow \text{BuildInvertedIndexes}(D, T)$ 
     $BoFs \leftarrow \text{ComputeBoFs}(T)$ 

```

4.1.1. Keypoint Detection and Descriptor Extraction

Descriptor extraction is applied to all the images of Γ storing the resulting descriptors into a set. Algorithm 2 shows this process. Note that descriptors are mixed all together into the result set, but we will still need to know which image each descriptor belongs to, thus in a real implementation will be necessary to mark in any way the descriptors before adding them to the set to avoid lose the image index where it comes from. Hereafter we will adopt the super index notation to indicate that an object corresponds to an specific image. For example, α^j means that this descriptor belongs to the j th image.

4.1.2. Building the visual vocabulary

K-clustering is applied on the descriptors set, generating \mathbf{K} cluster centers. This \mathbf{K} centers are used as the first level of nodes in the vocabulary tree. The descriptors set is then split into \mathbf{K} subsets, assigning each descriptor to it’s closest center.

K-clustering is applied in the same way, recursively for each subset Process is repeated on every branch until it reaches a maximum height \mathbf{H} (or it runs out of descriptors).

Each node stores the cluster center vector (or visual word).

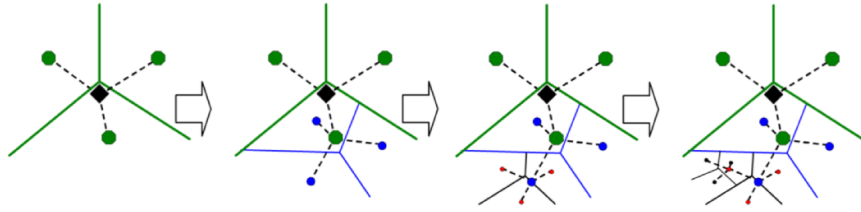
Algorithm 2: Descriptors Extraction**ExtractDescriptors****input** : Γ a set of N images**output**: \mathbf{D} the set of extracted descriptors**begin** $D \leftarrow \{\}$ **for** each image I_i in Γ **do** $Features_i \leftarrow \mathbf{Detect}_{md}(I_i)$ $Descriptors_i \leftarrow \mathbf{Extract}_{mx}(I_i, Features_i)$ $D \leftarrow D \cup Descriptors_i$ 

Fig. 4.1: Nister's hierarchical clustering

Algorithm 3 shows a pseudo code of tree building. $|D|$ is the number of elements in the set D , **ComputeClusters** function performs descriptors clustering, it can be used K-means or any other clustering technique. **GetCentroid** function returns the centroid for the given cluster, if K-means is used, centroid will be the mean. To see an example of how this algorithm works, see “Building the Tree” on example section 4.3.

Algorithm 3: Building the vocabulary tree**BuildVocabularyTree****input** : \mathbf{D} a set of descriptors, \mathbf{L} current level of the tree, \mathbf{K} number of children per node, \mathbf{H} maximum height for the tree**output**: \mathbf{T} the vocabulary tree**begin** **if** ($|D| < K$ or $L \geq H$) **then** $T \leftarrow \text{build a new leaf node}$ **else** $Clusters \leftarrow \mathbf{ComputeClusters}(D, K)$ $T \leftarrow \text{build a new branch node}$ $T.children \leftarrow \{\}$ **for** each cluster C in $Clusters$ **do** $Child \leftarrow \mathbf{BuildVocabularyTree}(C, L + 1, K, H)$ $Child.\mu \leftarrow \mathbf{GetCentroid}(C)$ $T.children \leftarrow T.children \cup \{Child\}$

4.1.3. K-Clustering feature vectors to build vocabulary

Let's call K-clustering to the family of clustering algorithms that receive a set of data, and return K clusters. The classic approach is to employ K-means clustering using euclidean distance to compare feature vectors. K-means strongly uses the fact that euclidean distances can be averaged to compute the mean vector. For detailed explanation of K-means see appendix .2. When comparing binary feature vectors (ORB, AKAZE, ...), usually Hamming distance is employed, and the mean vector is undefined. There are many other techniques that allow clustering without computing means, but most of them require computing the distance matrix, making the problem intractable considering the training input size. In [12] a workaround to this problem is introduced allowing to use the vocabulary tree technique with binary descriptors. This clustering algorithm is called "K-majority". K-majority uses a voting scheme to decide which components of the centroid would have 0 or 1. K-majority can be also thought as a forced way to apply K-means over binary vectors, by reinterpreting them as real-valued vectors, each component would be 1,0 or 0,0. In this way it is feasible to calculate the mean. Mean components are rounded to 0 or 1 to reinterpret it as a binary vector.

4.1.4. Vocabulary Memory Consumption

Once K and H are fixed, we can estimate some aspects of memory consumption. For example each level of the tree has as much K children then the maximum number of nodes is limited by the geometric sum

$$\sum_{i=0}^H K^i = \frac{K^{H+1} - 1}{K - 1} \quad (4.1)$$

Since each node stores a D-dimensional cluster center (or visual word), the amount of memory needed to store the vocabulary will be the number of used nodes multiplied by the consumption of a single visual word. For example with $D = 128$, $K = 10$ and $H = 6$, it will produce at most 1,111,111 cluster centers (or nodes), and it would consume less than 136MB of memory. This amount is promising in terms of scale, is fixed for any database size, and can perfectly fit in RAM. It is important to note that the data type used to represent the components of feature vectors also affects the memory consumption required, and this is a non-minor decision. For example, SIFT descriptors components have integer values between 0 and 127, thus it is possible to represent it using 1 byte for each dimension (which in C++ would be *unsigned char* or simply *uchar* data type). Other descriptors such as SURF and KAZE have real-valued components, and their representation may use C++ *float* data type, but a floating point consumes 4 bytes. Even though SIFT vector components can be represented with bytes, combining floating point operations with byte values in the same algorithm would require convert data type to float type every time, giving poor execution time. Consequently, the float data type is a trade-off efficiency and amount of memory required for the vocabulary, and the above example would require actually $136MB * 4 = 544MB$, and still perfectly fits in RAM.

Here is being considered only the amount of memory needed to store the vocabulary, but other data structures are needed to compute the score.

4.1.5. Building inverted indexes

After vocabulary tree is created, it is time to build inverted indexes. Algorithm 4 shows this process.

Inverted indexes are physically stored in leaf nodes. At the beginning each leaf starts with an empty inverted index. An inverted indexes are bags (can be implemented with a set with repetitions or multi-set), which stores the ids of images as elements, and the number of descriptors of this image as repetitions.

Each descriptor vector of the image j is descended along the tree from the root to a leaf following in each level the nearest center (for example according to L_2 norm). When the descriptor reaches a leaf, it adds one repetition for j into the multi-set in the leaf.

Later, inverted indexes for branch nodes (virtual inverted indexes), will be computed combining inverted indexes of children nodes.

To see an example of how this algorithm works, see “Inverted Indexes Creation” on the example section 4.3.

Algorithm 4: Building inverted index

BuildInvertedIndexes

input : \mathbf{D} a set of descriptors, \mathbf{T} vocabulary tree

output: \mathbf{T} tree with inverted indexes on its leaves

begin

```

  for each descriptor  $\alpha^j$  in  $D$  do
     $A \leftarrow \mathbf{FindLeaf}(T, \alpha^j)$ 
     $A.InvertedIndex \leftarrow A.InvertedIndex \cup_{bag} \{j\}$ 

```

FindLeaf

input : \mathbf{T} the vocabulary tree, α^j : a descriptor of the image j

output: \mathbf{A} the leaf which α^j has felt

begin

```

   $A \leftarrow T$ 
  while  $A$  is not a Leaf do
     $A \leftarrow \mathit{argmin}_{C \in A.children} d_C : d_C = \|\alpha^j - C.\mu\|_L$ 

```

4.1.6. Building Bag of Features

In [30] different ways to adapt the weighting mechanisms are evaluated. IDF and TF are proposed in the following way:

IDF

Each node i is assigned with a weight $w_i \in \mathbb{R}$ value based on entropy

$$w_i = \log \frac{N}{N_i} \quad (4.2)$$

where N is the number of images in the database, and N_i is the number of images in the database with at least one descriptor vector path through node i . Note N_i is the number

of elements of the inverted index.

TF

Let $n_i^j \in \mathbb{N}$ be the number of descriptor vectors of the image j with a path through the node i . Similarly, let $m_i \in \mathbb{N}$ be the number of descriptor vectors of the query image with a path through the node i .

d-vectors and q-vector

Let $q \in \mathbb{R}^n$, $q = (q_1, \dots, q_n)$ be the query BoF-vector and $d^j \in \mathbb{R}^n$, $d^j = (d_1^j, \dots, d_n^j)$ be the database BoF-vectors, defined as follows:

$$\begin{aligned} d_i^j &= n_i^j w_i \\ q_i &= m_i w_i \end{aligned} \tag{4.3}$$

where j is the database image index, $1 \leq j \leq N$, and n is the number of tree nodes.

TF and BoF for the query are actually computed at run-time phase, but it is mentioned here because it is computed in an analogous way than the images in the database.

Note n_i^j is the number of repetitions of element j in the inverted index of node i . Note also that this equation establishes a mapping between the nodes and the components of d-vectors (and q-vector). Each node is mapped to a different d component. These vectors can also be thought as a big sparse matrix with N rows mapped to images and n columns mapped to tree nodes. Sub-index A for d_A^j is used to indicate the component of d^j that is mapped to node A .

Algorithm 5 shows a pseudo-code of how d-vectors computation could be done. It iterates over all the nodes in the tree, and for each node computes its inverted index. Then it computes the node weight, and for each image index in the inverted index computes the corresponding d-vector component value. This d-vector component is stored in the inverted index of the node as a pair (j, d_i^j) . The sub-function **ComputeInvertedIndexes** returns the inverted index for the specified node. Note it uses the union of bags function \cup_{bag} that sums repetitions if an element is on both sets.

4.1.7. Normalizing BoFs

To achieve fairness between database images with few and many descriptor vectors, a L_p -norm normalization is made to each vector: $\bar{d}^j = \frac{d^j}{\|d^j\|}$, $\bar{q} = \frac{q}{\|q\|}$.

As suggested, L_1 -norm normalization is done for this implementation. Then L_1 normalization is simply done performing the sum of all components of each database vector, and then dividing each component by each sum. For an example see “d-vectors normalization” on Example section 4.3.

Algorithm 5: Computing Bag of Features

ComputeBoFs**input** : **T** a vocabulary tree with inverted indexes**output**: **T** tree nodes have its weight computed**output**: **BoFs** the bag of features**begin** $BoFs \leftarrow$ new sparse matrix ($N \times \#T$) **for** each node index $i < \#T$ **do** $A \leftarrow T[i]$ $ii \leftarrow$ **getVirtualInvertedIndexes**(A) $A.weight \leftarrow \log(\frac{N}{|ii|})$ **for** each j in ii **do** $n_i^j \leftarrow ii[j]$ $BoFs[j, i] \leftarrow A.weight * n_i^j$

// normalizes BoFs

for each $j < BoFs.rows$ **do** $d \leftarrow BoFs[j]$ $\bar{d} \leftarrow \frac{d}{|d|_p^p}$ $BoFs[j] \leftarrow \bar{d}$ **getVirtualInvertedIndexes****input** : **A** a tree node**output**: **ii** a bag containing the inverted indexes of A **begin** **if** (A is a Leaf) **then**

// (it is a real inverted index)

 $ii \leftarrow A.invertedIndex$ **else**

// computing virtual inverted index

 $ii \leftarrow \{\}_{bag}$ **for** each node C in $A.children$ **do** $cii \leftarrow$ **getVirtualInvertedIndexes**(C) $ii \leftarrow ii \cup_{bag} cii$

4.2. Runtime Phase

4.2.1. Scoring

Score between query and image j is defined as the distance between its normalized BoF vectors \bar{q} and \bar{d}^j , distance is computed as the norm of the difference:

$$s^j(\bar{q}, \bar{d}^j) = \|\bar{q} - \bar{d}^j\| \quad (4.4)$$

This metric is also known as Minkowsky distance, and can be considered as a generalization of both the Euclidean distance and the Manhattan distance. In [30] is shown that if vectors are normalized, any Minkowsky distance can be computed inspecting only components where both vector components are non zero (see appendix .1 for full derivation). Thus, equation 4.5 is going to be used to compute scoring. Because it will be applied to BoFs in a real world case most of the components will have zero value. This will reduce considerably the number of operations to do at scoring time.

$$\|\bar{q} - \bar{d}^j\|_p^p = 2 + \sum_{i|\bar{d}_i^j \neq 0, \bar{q}_i \neq 0}^n (|\bar{q}_i - \bar{d}_i^j|^p - |\bar{q}_i|^p - |\bar{d}_i^j|^p) \quad (4.5)$$

4.2.2. Query

Algorithm 6 shows the process of query. Descriptor extraction is performed over Q with the same function we used for training images. Function **ComputeQBoF** calculates q -vector, it uses the function $\#(T)$ that returns the number of nodes on the tree T . It acumulates weights on q componentes while traverses each descriptor of Q through the tree, from root to leaves. The function **indexOf**(A) returns the index of the q -vector component mapped to the node A , and it can be implemented in $O(1)$. First index of q is not considered as it is mapped to root node and it has weight 0. q -vector is then normalized using L1-norm. An empty scoring set S is initialized, and for each image index in the database, we will check only components where q is different from zero. Equation 4.5 is then applied on \bar{d} and \bar{q} vectors to compute the scoring.

4.3. Example

Suppose we have a set of 3 images $\Gamma = \{I_1, I_2, I_3\}$ (in a real case we will have much more images). Suppose now we apply SIFT descriptor extraction over all the images in the set, and we obtain the following image descriptors: $Extract_{SIFT}(I_1) = \{\alpha_1, \alpha_2, \alpha_3, \alpha_4\}$, $Extract_{SIFT}(I_2) = \{\beta_1, \beta_2, \beta_3, \beta_4, \beta_5\}$ $Extract_{SIFT}(I_3) = \{\gamma_1, \gamma_2, \gamma_3\}$ (in a real case we will have much more descriptors). Since we used SIFT, each descriptor belongs to \mathbb{R}^{128} .

4.3.1. Building the Tree

Figures 4.2 and 4.3 show the tree building process using $K = 3$ parameter for K-means, and $H = 4$ as maximum height for the tree. (a) The extracted descriptors are put together in a set and clusterized with K-means. In this example $K = 3$ is used. The result is 3 different clusters, each one with its corresponding cluster center μ_1 , μ_2 and μ_3 . (b) K-means is applied recursively over the first cluster. The data of the first child cluster is divided again onto 3 different new clusters. The root node A is created, and then the

Algorithm 6: Performing Query

Query

input : Q the image query
input : T the vocabulary tree
input : BoFs the set of precomputed bag of features
input : t size of the resulting short-list
output: R the short-list with the indexes of top ranked results
begin
 $E \leftarrow \text{ExtractDescriptors}(Q)$
 $q \leftarrow \text{ComputeQBoF}(E, T)$
 $\bar{q} \leftarrow \frac{q}{\|q\|_p^p}$
 $nzi \leftarrow \{i | 1 \leq i \leq \#T \text{ and } \bar{q}_i \neq 0\}$
 $S \leftarrow \{\}$
 for $1 \leq j \leq N$ **do**
 $d^j \leftarrow \text{BoFs}[j]$
 $score \leftarrow 2$
 for *each* i *in* nzi **do**
 $score \leftarrow score + |\bar{q}_i - \bar{d}_i^j|^p - |\bar{q}_i|^p - |\bar{d}_i^j|^p$
 $S \leftarrow S \cup \{(j, score)\}$
 $Sorted \leftarrow \text{sort}(S, (s1, s2) \rightarrow \{s1.score < s2.score\})$
 $Indexes \leftarrow \text{collect}(Sorted, (s) \rightarrow \{s.j\})$
 $R \leftarrow \text{take}(Indexes, t)$

ComputeQBoF

input : E a set of descriptors
input : T the vocabulary tree
output: q the BoF for the query
begin
 $q \leftarrow \text{new sparse vector } (\#T)$
 for *each* α *in* E **do**
 $A \leftarrow T$
 while (A *is not a Leaf*) **do**
 $A \leftarrow \text{argmin}_{C \in A.children} \delta_C : \delta_C = \|\alpha - C.\mu\|_L$
 $i \leftarrow \text{indexOf}(A)$
 $q_i \leftarrow q_i + A.weight$

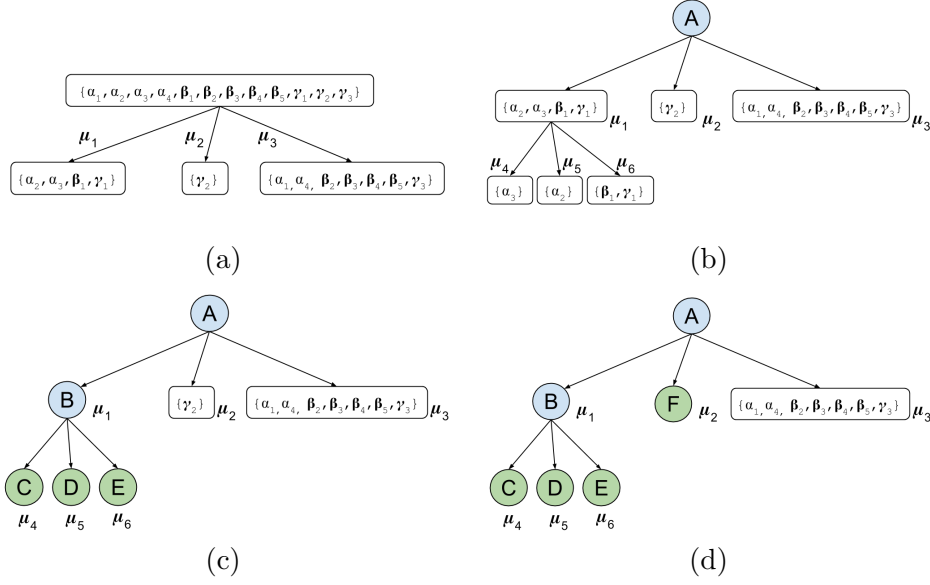


Fig. 4.2: tree building example

original set on the root is released in order to gain some memory. (c) Node B is created, the cluster center μ_1 is stored on the node B and the set on node B is released to gain some memory. All the 3 children clusters of node B have less than $K = 3$ elements, thus these sets can not be split any more and then leaf nodes C , D and E are created, and cluster centers μ_4 , μ_5 and μ_6 are stored in each node respectively. (d) Leaf node F is created as there are not enough elements to split the second child set onto $K = 3$ subsets. Cluster center μ_2 is stored on node F . (e) K-means is applied recursively over the third child cluster. This cluster is split further, creating 3 new subclusters. (f) Node G is created, and memory set is released. Cluster center μ_3 is stored on node G . As first subcluster of G has more than K elements, this cluster is split further creating the third level on the tree. (g) Node H is created, cluster center μ_7 is stored on node H and memory is released. First and third children clusters have less than 3 elements and can not be split further. Note that second cluster has 3 elements, but since tree has reached the maximum height $h = 4$ it is not allowed to continue splitting, then leaf nodes I , J and K are created (h) Nodes L and M are created, cluster centers are stored on the nodes. The tree is now complete.

4.3.2. Inverted Indexes Creation

Figures 4.4 and 4.5 show the inverted index creation of the example described above. (a) the first descriptor α_1 of the first image I_1 is taken and the algorithm positions at root node A . Using $L_2 - norm$, it computes the distance from α_1 to each of the centers μ_1 , μ_2 and μ_3 located at level 1 of the tree. (b) The descriptor vector is descended to the node which has the nearest center to it. In this case the nearest center is the one stored into the node G . The process is repeated until the descriptor reaches a leaf. (c) Once the descriptor α_1 reaches the leaf, the inverted index is updated. A new link to the image I_1 is created. The inverted index is represented as the image id, and the number of descriptor vectors that fell into this leaf. (d) At this moment on the node K we have one descriptor that points to image 1 then the pair $(1, 1)$ is added to the inverted index of node. (the

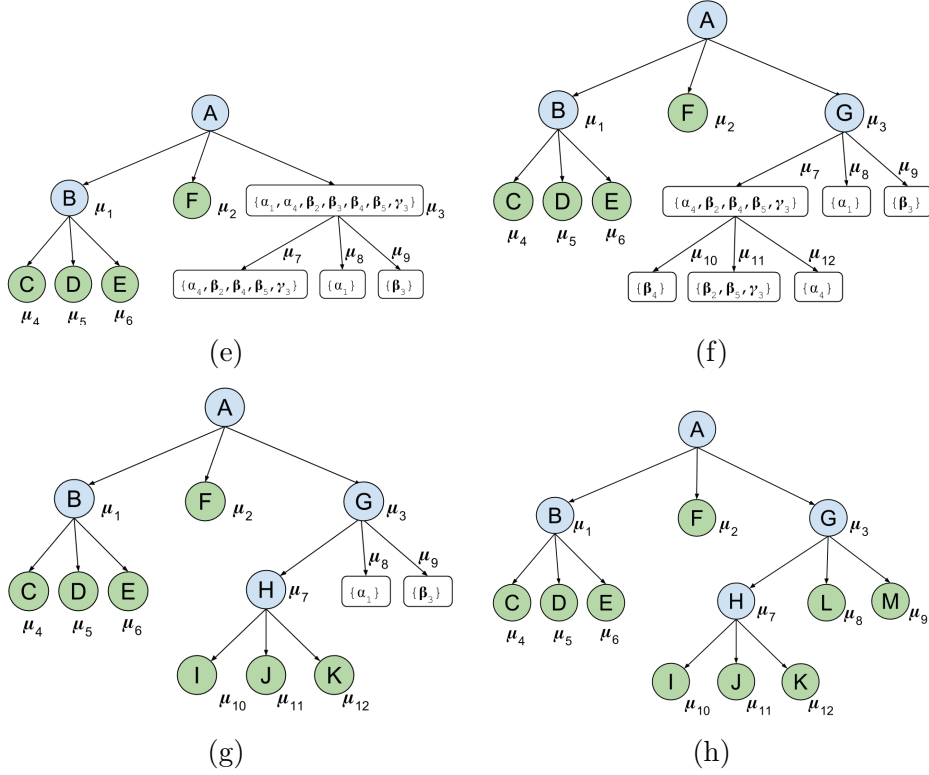


Fig. 4.3: tree building example (continued)

first component is the image id, and the second is the descriptor quantization). The same procedure is repeated with the next descriptor α_2 . (e) In this case α_2 falls through node F , as it is the nearest to μ_2 . Note that when the tree was generated, α_2 fell in a different branch, but due to K-means is an heuristic it is not guaranteed that it will fall through the same path it did when the tree was forming. (f) α_2 reaches the leaf F and inverted index on node F is updated. (g) All remaining descriptors of image 1, and the descriptors of the rest of the images are processed in the same way.

In order to reduce memory consumption actually inverted indexes are stored only in the leaves of the tree. For the internal nodes virtual inverted indexes are recursively computed combining child inverted indexes.

One efficient way to implement virtual inverted index is to store inverted indexes entries ordered by image id, and use a method similar to merge part of merge sort algorithm. Figure 4.6 shows how leaf inverted indexes are combined to generate virtual inverted indexes for the internal nodes. (a) Merge pointers are set at the start of each inverted index. Virtual inverted index for node H is now empty. (b) Pointer on node K is advanced as it has the lowest image index. A new entry $(1, 1)$ on the node H is added to the virtual inverted index. (c) Pointers on nodes I and J are advanced and values combined onto a new entry $(2, 1 + 2)$ added to the virtual inverted index on node H . (d) Pointer on node J is advanced as is the only one remaining, and a new entry is added to the virtual inverted index on H $(3, 1)$.

Figure 4.7 shows the merged inverted indexes complete example:

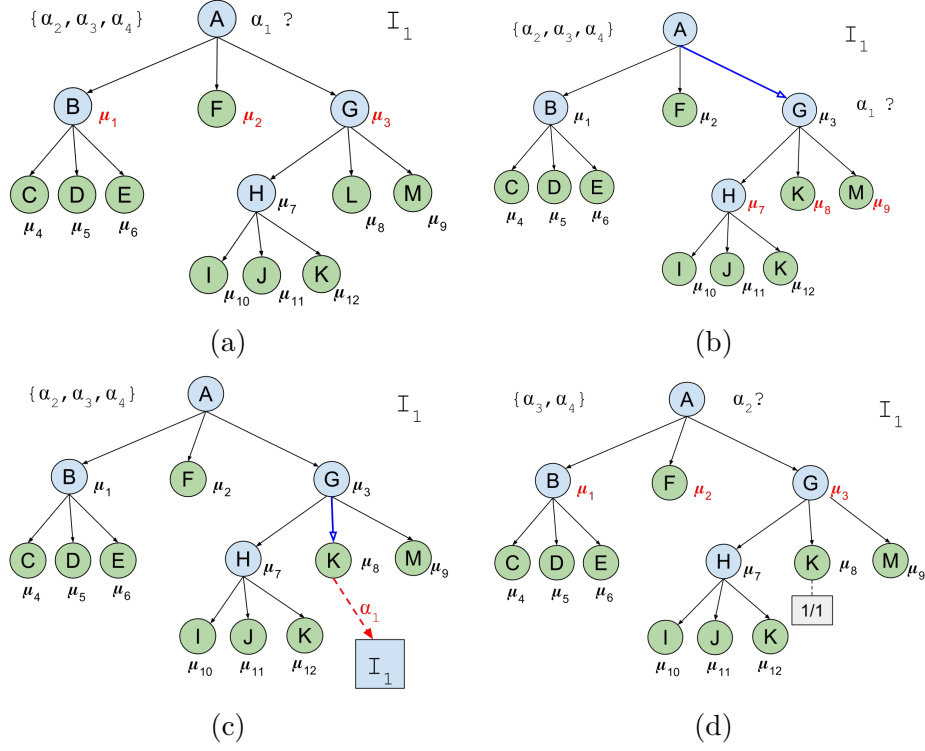


Fig. 4.4: inverted indexes building

4.3.3. Computing BoFs

Let's compute some bag of features. The database example above has ($N = 3$) images. Figure 4.8 shows inverted indexes for three example nodes I , J and H . As there is only one image with descriptors that passes through node I (the image 2) then $N_I = 1$. On the same way, there are two different images with descriptors that passes through node J (images 2 and 3) then $N_J = 2$, and all the images of the database pass through node H , then $N_H = 3$.

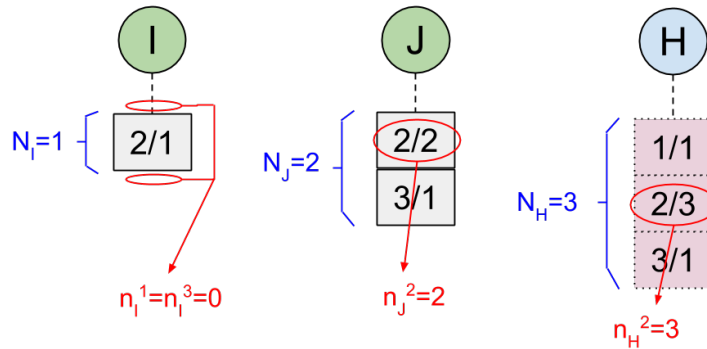


Fig. 4.8: computing d-vectors

Then the weights for these nodes will be:

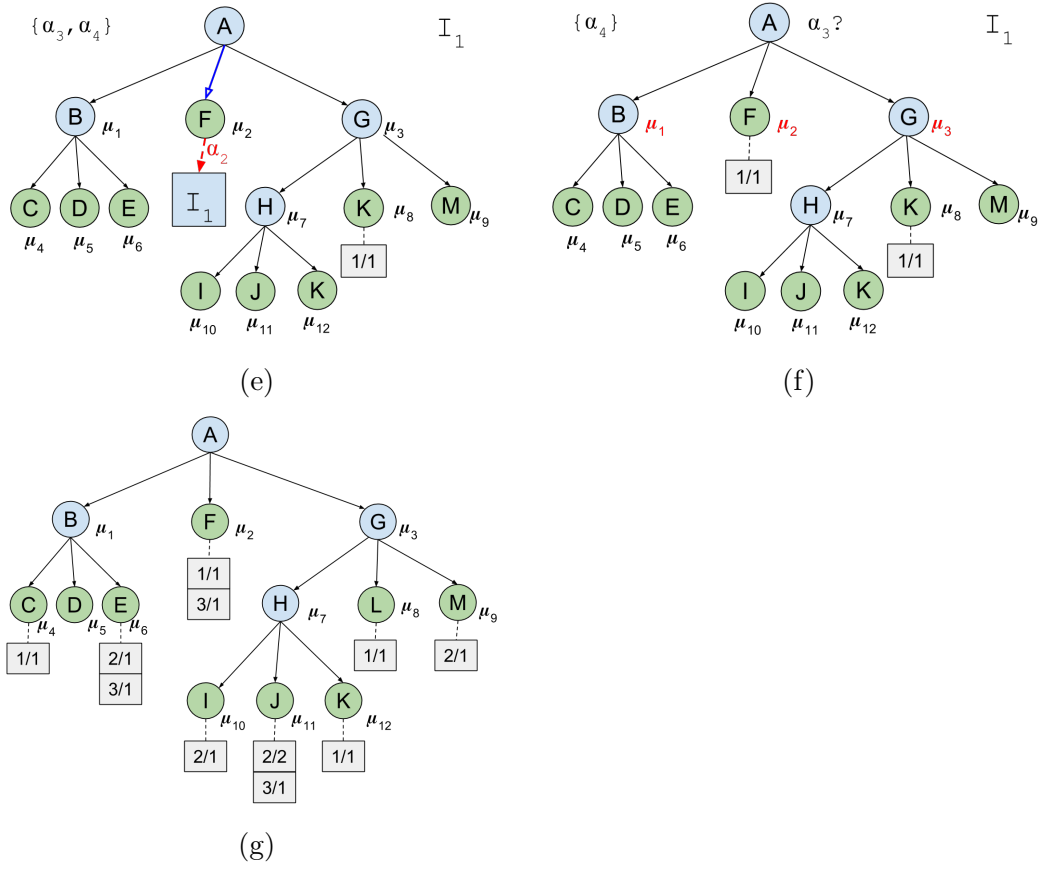


Fig. 4.5: inverted indexes building (continued)

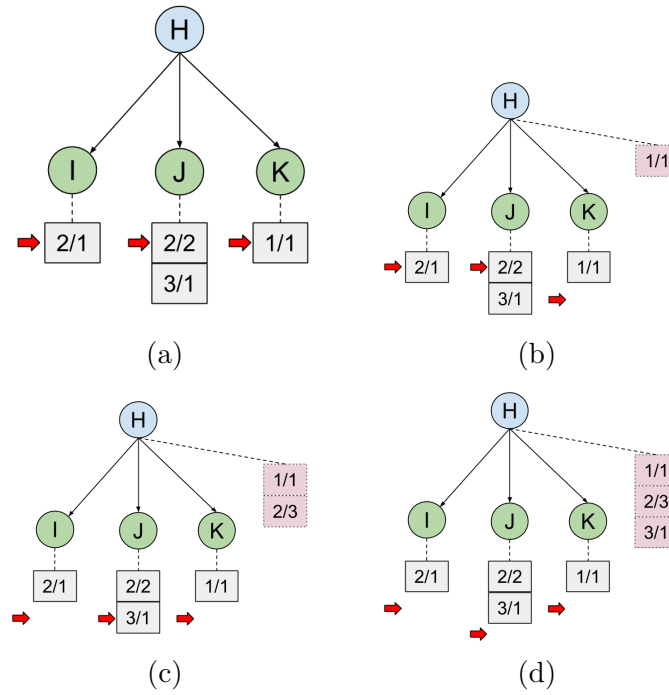


Fig. 4.6: merging inverted indexes

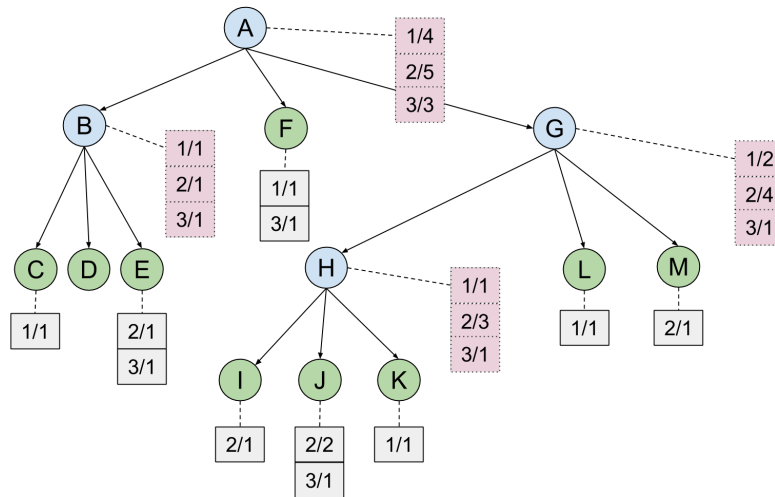


Fig. 4.7: complete inverted indexes

$$\begin{aligned}
w_I &= \log N/N_I = \log 3/1 \approx 0,47712 \\
w_J &= \log N/N_J = \log 3/2 \approx 0,17609 \\
w_H &= \log N/N_H = \log 3/3 = 0
\end{aligned}$$

Note that weight is smaller when the node has more different images with descriptors that passes through it. For instance, as all the images passes through node H it does not provide any information to reduce the search space of the images in the database (because it points to all of the images), then it weights 0.

Table 4.1 shows the complete weight vector for the complete example:

	A	B	C	D	E	F	G	H	I	J	K	L	M
$w =$	0	0	$\log 3$	0	$\log 3$	$\log 3/2$	0	0	$\log 3$	$\log 3$	$\log 3$	$\log 3$	$\log 3$

Tab. 4.1: weight vector example

Figure 4.8 shows also how n_i^j is the number of repetitions in the inverted index entry. For example on node I we have no inverted index entries for images 1 and 3, then $n_I^1 = n_I^3 = 0$, and $n_I^2 = 1$. For node J we have $n_J^1 = 0$, $n_J^2 = 2$ and $n_J^3 = 1$. And for node H we have $n_H^1 = 1$, $n_H^2 = 3$ and $n_H^3 = 1$.

Table 4.2 shows n-vectors for the complete example:

	A	B	C	D	E	F	G	H	I	J	K	L	M
$n^1 =$	4	1	1	0	0	1	2	1	0	0	1	1	0
$n^2 =$	5	1	0	0	1	0	4	3	1	2	0	0	1
$n^3 =$	3	1	0	0	1	1	1	1	0	1	0	0	0

Tab. 4.2: n-vectors example

Using equation 4.3 we compute d-vectors and we obtain the results shown in the table 4.3:

d-vectors can be represented as a vector of pointer to vectors to pairs. These pairs will have the image id as first component and d value as second component. This data can be computed at the same time inverted indexes are built.

4.3.4. Normalizing BoFs

Normalization is simply done by iterating twice over BoF structure. First iteration will compute the norm for each BoF. Second pass will update in place each component of the BoF.

Let's compute normalized d-vectors \bar{d}^j for the example using L1-norm:

$$\begin{aligned}
\sum_{i=1}^n |d_i^1| &= \log\left(\frac{3}{2}\right) + 3 \log(3) = \log\left(\frac{3^4}{2}\right). \\
\sum_{i=1}^n |d_i^2| &= 2 \log\left(\frac{3}{2}\right) + 3 \log(3) = \log\left(\frac{3^5}{2^2}\right). \\
\sum_{i=1}^n |d_i^3| &= 2 \log\left(\frac{3}{2}\right) + \log(3) = \log\left(\frac{3^3}{2^2}\right). \\
\bar{d}^1 &= \frac{d^1}{\sum_{i=1}^n |d_i^1|}, \quad \bar{d}^2 = \frac{d^2}{\sum_{i=1}^n |d_i^2|}, \quad \bar{d}^3 = \frac{d^3}{\sum_{i=1}^n |d_i^3|}.
\end{aligned}$$

	A	B	C	D	E	F	G	H	I	J	K	L	M
$d^1 =$	0	0	$\log 3$	0	0	$\log 3/2$	0	0	0	0	$\log 3$	$\log 3$	0
$d^2 =$	0	0	0	0	$\log 3$	0	0	0	$\log 3$	$2 \log 3/2$	0	0	$\log 3$
$d^3 =$	0	0	0	0	$\log 3$	$\log 3/2$	0	0	0	$\log 3/2$	0	0	0

Tab. 4.3: d-vectors example

	A	B	C	D	E	F	G	H	I	J	K	L	M
$\bar{d}^1 =$	0	0	$\frac{\log 3}{\log(\frac{3^4}{2})}$	0	0	$\frac{\log(\frac{3}{2})}{\log(\frac{3^4}{2})}$	0	0	0	0	$\frac{\log 3}{\log(\frac{3^4}{2})}$	$\frac{\log 3}{\log(\frac{3^4}{2})}$	0
$\bar{d}^2 =$	0	0	0	0	$\frac{\log 3}{\log(\frac{3^5}{2^2})}$	0	0	0	$\frac{\log 3}{\log(\frac{3^5}{2^2})}$	$\frac{2 \log(\frac{3}{2})}{\log(\frac{3^5}{2^2})}$	0	0	$\frac{\log 3}{\log(\frac{3^5}{2^2})}$
$\bar{d}^3 =$	0	0	0	0	$\frac{\log 3}{\log(\frac{3^3}{2^2})}$	$\frac{\log(\frac{3}{2})}{\log(\frac{3^3}{2^2})}$	0	0	0	$\frac{\log(\frac{3}{2})}{\log(\frac{3^3}{2^2})}$	0	0	0

Tab. 4.4: \bar{d} -vectors example

4.3.5. Performing Query

Suppose now we have a query image Q . We apply SIFT descriptor extraction in the same way we did with the images in the entire image set, and we obtain the following image descriptors: $Extract_{SIFT}(Q) = \{\delta_1, \delta_2, \delta_3, \delta_4\}$.

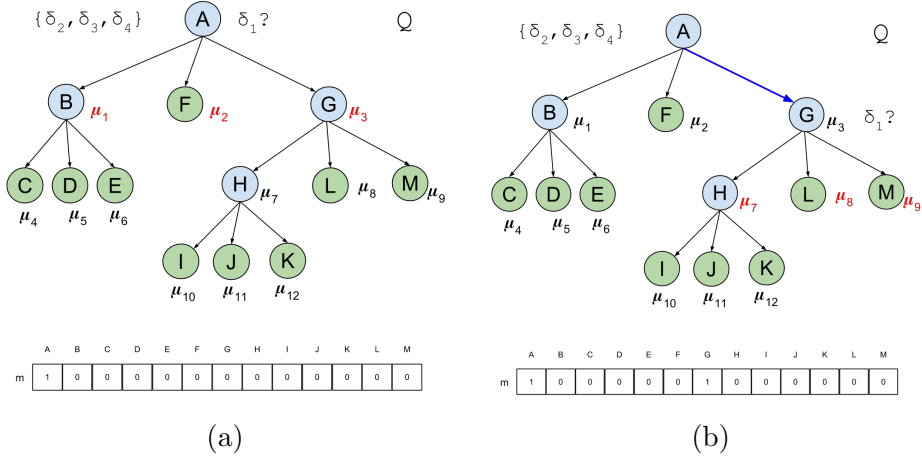
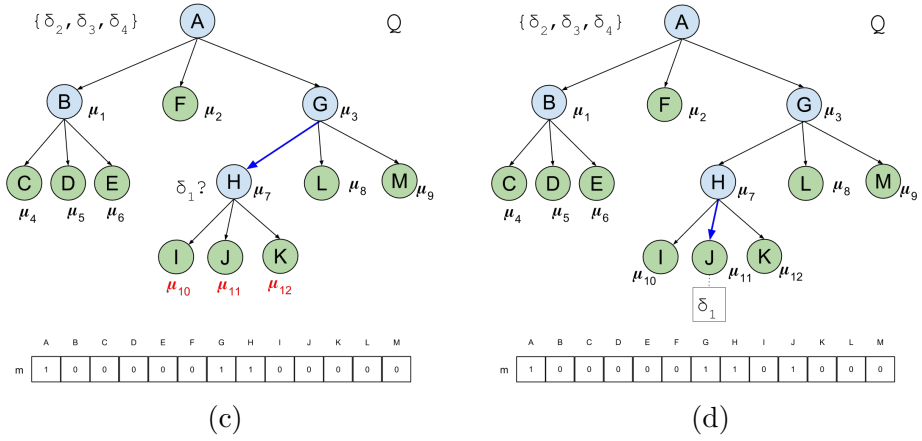
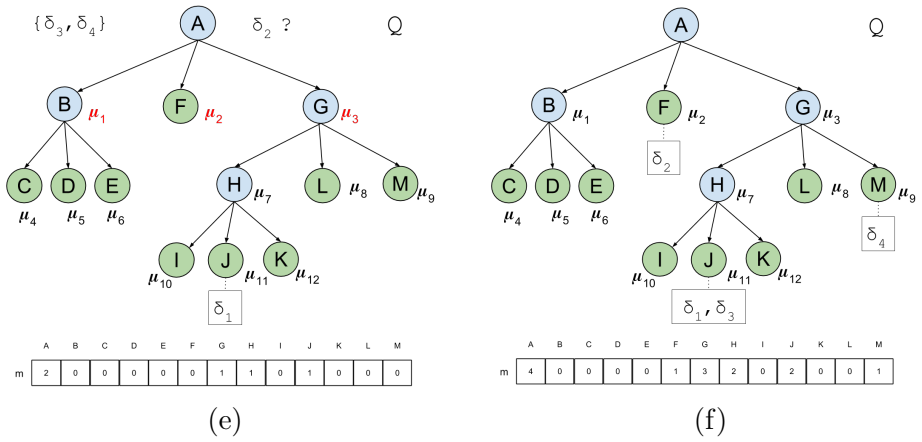
Figures 4.9, 4.10 and 4.11 show how the q vector is computed. At start m vector is initialized with zeroes. (a) The first descriptor of the query image δ_1 is taken and the algorithm steps at root node A . m_A vector component is updated (+1). Using the norm L_2 , it computes the distance from δ_1 to each of the centers μ_1, μ_2 and μ_3 located at level 1 of the tree. (b) The descriptor vector is descended to the node which has the nearest center to it. In this case the nearest center is the one stored into the node G . m_G vector component is updated (+1). The process will be repeated until the descriptor reaches a leaf. δ_1 is now compared with child nodes centers μ_7, μ_8 and μ_9 . In this case the nearest one is μ_7 . (c) δ_1 is descended to node H , and m_H vector component is updated (+1). (d) The descriptor δ_1 reaches the leaf J. m vector is updated. (e) The algorithm performs the same process with the next descriptor δ_2 . And when δ_2 reaches a leaf, it starts over with next descriptor δ_3 until all the descriptors are processed. (e) All descriptors of query image have been processed and now we have completely computed m vector.

Table 4.5 shows m-vector, w-vector and q-vector for the example above. q-vector is computed as the component by component product of the other two vectors (see equation 4.3).

	A	B	C	D	E	F	G	H	I	J	K	L	M
$m =$	4	0	0	0	0	1	3	2	0	2	0	0	1
$w =$	0	0	$\log 3$	0	$\log 3$	$\log 3/2$	0	0	$\log 3$	$\log 3$	$\log 3$	$\log 3$	$\log 3$
$q =$	0	0	0	0	0	$\log 3/2$	0	0	0	$2 \log 3$	0	0	$\log 3$

Tab. 4.5: q-vector example

Now we have computed q vector, we need to normalize it using L1-norm

Fig. 4.9: calculating vector q for exampleFig. 4.10: calculating vector q for example (continued)Fig. 4.11: calculating vector q for example (continued)

$$\sum_{i=1}^n |q_i| = \log\left(\frac{3}{2}\right) + 3 \log(3) = \log\left(\frac{3^4}{2}\right).$$

$$\bar{q} = \frac{q}{\sum_{i=1}^n |q_i|}.$$

	A	B	C	D	E	F	G	H	I	J	K	L	M
$\bar{q} =$	0	0	0	0	0	$\frac{\log 3/2}{\log(\frac{3^4}{2})}$	0	0	0	$\frac{2 \log 3}{\log(\frac{3^4}{2})}$	0	0	$\frac{\log 3}{\log(\frac{3^4}{2})}$

Tab. 4.6: q-vector example

Now that we have computed \bar{q} is time to compute the scoring for each image in the database. Using equation 4.5 we only need perform operations over components where both vectors have values different from zero. First we can project only those components where \bar{q} is not zero (in this case will be F , J and M). Table 4.7 shows these projected vectors.

	F	J	M
$\bar{q}_{\langle F, J, M \rangle}$	$\frac{\log \frac{3}{2}}{\log(\frac{3^4}{2})}$	$\frac{2 \log 3}{\log(\frac{3^4}{2})}$	$\frac{\log 3}{\log(\frac{3^4}{2})}$
$\bar{d}_{\langle F, J, M \rangle}^1$	$\frac{\log \frac{3}{2}}{\log \frac{3^4}{2}}$	0	0
$\bar{d}_{\langle F, J, M \rangle}^2$	0	$\frac{\log 3}{\log \frac{3^5}{4}}$	$\frac{2 \log \frac{3}{2}}{\log \frac{3^5}{4}}$
$\bar{d}_{\langle F, J, M \rangle}^3$	$\frac{\log 3}{\log \frac{3^3}{4}}$	$\frac{\log \frac{3}{2}}{\log \frac{3^3}{4}}$	0

Tab. 4.7: projected components for scoring

To compute s^1 we now look in d^1 projected values. As only d_F^1 is different from zero the score s^1 will be:

$$s^1 = 2 + (|\bar{q}_F - \bar{d}_F^1| - \bar{q}_F - \bar{d}_F^1) \approx 1,78091$$

To compute s^2 we now look in d^2 projected values. As d_J^2 and d_M^2 are different from zero the score s^2 will be:

$$s^2 = 2 + (|\bar{q}_J - \bar{d}_J^2| - \bar{q}_J - \bar{d}_J^2) + (|\bar{q}_M - \bar{d}_M^2| - \bar{q}_M - \bar{d}_M^2) \approx 1,07005$$

To compute s^3 we now look in d^3 projected values. As d_F^3 and d_J^3 are different from zero the score s^3 will be:

$$s^3 = 2 + (|\bar{q}_F - \bar{d}_F^3| - \bar{q}_F - \bar{d}_F^3) + (|\bar{q}_J - \bar{d}_J^3| - \bar{q}_J - \bar{d}_J^3) \approx 1,35623$$

This means that for this example, within the database the most similar to Q image is I_2 , followed by I_3 and then by I_1 .

5. RESULTS

In this chapter we show the results of the following experiments:

- Different combinations of K and H were experimented, varying feature extraction techniques (SIFT, SURF, KAZE, ORB, AKAZE).
- How it impacts on performance to vary the size of the vocabulary.
- Performance with respect to increasing database size, up to 1 million images.
- Performance with respect to reducing descriptors dimensionality using Principal Component Analysis.

Shown experiments were done using $L1$ norm for scoring, as it gives good results and fast implementation. Experiments using $L2$ norm for scoring gave worse performance, and their results are not shown here.

5.1. Data Sets

There are many canonical image data sets, used for experimentation. In our case to experiment we used mainly the following two datasets:

UKBench Ukbench is the set of images used in [30] to perform their experiments. This data set consists of pictures of different objects (toys, clocks, cd covers, mostly taken indoors). The pictures were taken with different illumination, angle and distance conditions. There are pictures of 2250 objects, each object has 4 pictures, totaling 10,200 images. All images are 640x480 pixels. It can be downloaded from <http://vis.uky.edu/~stewe/ukbench/>.

MirFlickr1M MirFlickr1M has 1 million Flickr images under the Creative Commons license. This set is commonly used for image retrieval evaluation. At Aug 1st, 2015, it has over 600 Google Scholar citations and 32 thousand downloads from universities (MIT, Cambridge, Stanford, Oxford, Columbia, UIUC, NUS, Tsinghua, Univ. Tokyo, KAIST, etc.) and companies (IBM, Microsoft, Google, Yahoo!, Facebook, Philips, Sony, Nokia, etc.) worldwide. More information can be found here <http://press.liacs.nl/mirflickr/>.

5.2. Evaluation Metrics

In order to automatize algorithm effectiveness measurement, a fixed image data set is considered, a fixed set of queries is defined, and data within the corpus has to be classified as relevant or non-relevant for each query in the query set, this classification will provide the ground truth. If an image from the indexed corpus is queried, Query function could be overloaded to receive the index of that image in the set. That is $Query(q) \rightarrow [r_1, \dots, r_t]$ where $1 \leq q \leq n$ is the index of the image to be queried, t is the maximum size for the short ranked list, and r_i are the indexes of the top ranked resulting images. The ground truth can be expressed as a function $\mathbf{isRelevant}(q, k) \rightarrow \{0, 1\}$, that values 1 if the k -th

image is a relevant as result for the image query q . Parameters q and k are the indexes of the images in the dataset. Manually assigning each image to an object class, will provide the result value for the *isRelevant* function (1 when both images are images of an object of the same class, and 0 else), and it will give us a way for automatize the evaluation. We present two main metrics used to evaluate retrieval performance **Metric₄** and **mAP** used in many other works [33], [16].

Metric₄ In [30] based on UKBench data set, this performance metric is defined, basically a real number between 0 and 4, defined as follows:

$$Metric_4 = \sum_{q=1}^Q \frac{CountRelevantTop_4(q)}{Q} \quad (5.1)$$

where $CountRelevantTop_4(q) = \sum_{k=1}^4 IsRelevant(q, Query(q)[k])$

It counts how many of the images of the same object are ranked within the top 4 better results. For example, getting everything right gives a score of 4, getting nothing right gives a score of 0. Getting only identical image right gives a score of 1. A score of 3 means that we find the identical image plus 2 of the 3 other images of the set. Each object individual performance is averaged for all the objects in the database, resulting in a global performance value.

Metric₄ assumes 4 images of each class, and is not suitable to use with other datasets. Image datasets usually doesn't have a fixed number of object classes. Even more, if the performance is measured with a number between 0 and 4, it is difficult to compare it with results from another data sets.

mean Average Precision (mAP) Mean average precision (mAP) is a common metric used to evaluate retrieval effectiveness. This metric is similar to the previous defined. The main difference is that this metric will favor algorithms where good results are put on top of the ranking. *mAP* can be defined as follows:

$$mAP = \sum_{q=1}^Q \frac{AveragePrecision(q)}{Q} \quad (5.2)$$

where:

- $AveragePrecision(q) = \frac{\sum_{k=1}^n Precision(q,k)}{TotalRelevant(q)}$
- $Precision(q,k) = \frac{\sum_{i=1}^{min(k,t)} IsRelevant(q,Query(q)[i])}{k}$
- $TotalRelevant(q) = \sum_{k=1}^n IsRelevant(q,k)$

Figure 5.1 shows an example of mAP calculation. mAP is commonly used because it combines precision as well as recall in a single value. A filter or re-rank stage will improve considerably this metric. In [30] these improvements are not applied.

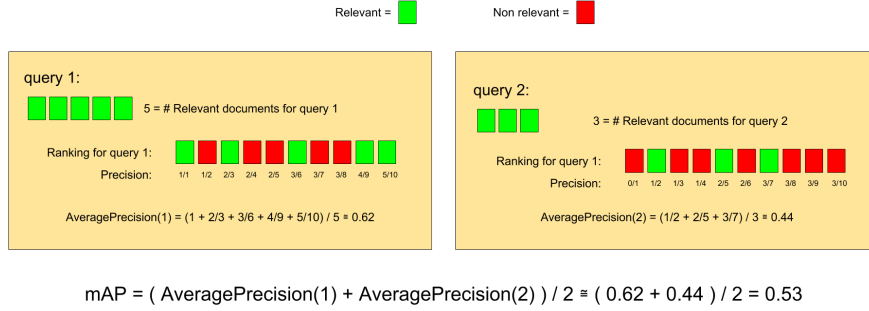


Fig. 5.1: Example. Computing Mean Average Precision

5.3. K-H best values

In [30] values $K=10$, $H=6$ (see Chapter 4) were proposed as good parameters to build the vocabulary tree. We computed mAP varying different combinations of K and H , and feature extraction algorithm, with the first 1000 images UKBench dataset, to find which combinations performs better. K and H were chosen, in such a way that vocabulary construction could fit in fixed amount of RAM. Thus, for $H = 6$ only a few values of K were tested. Figures 5.2 and 5.3 show mAP behavior varying K between 8 and 32 for $H = 2, 3, 4, 5, 6$ values, using L_2 norm, and Hamming distance, respectively.

Results are consistent with [30].

Note that methods that use L_2 norm resulted superior to the ones using Hamming distance. Methods using L_2 norm have mAP peaks of 0,92 while the ones that use Hamming distance obtain only 0,88. The new methods KAZE y AKAZE (available from de OpenCV 3) in some cases resulted slightly better than the classic ones. One important advantage of using KAZE and AKAZE is that these are open source, while SIFT y SURF are patented and non-free [26], [10].

5.4. Performance vs training size

Figure 5.4 shows the result of varying the number of images used to train the vocabulary. For each feature description technique K and H parameters were chosen according to the best results of K - H exploration experiment.

UKBench dataset was used both for training and for runtime. Training was performed varying incrementally the number of images from 1000 to 10000. Metrics in runtime were computed using always the entire dataset (10200 images).

Results show that a bigger vocabulary gives a better performance. Results are consistent with [30].

5.5. Performance vs database size

Figure 5.5 shows the result of varying N given a fixed vocabulary. UKBench dataset was used both for vocabulary training and as part of the input dataset. MirFlickr1M was used as distracting images for large scale image search. Features were extracted using SIFT (with 500 per image), and dimensions were reduced to 64. Experiment shows how

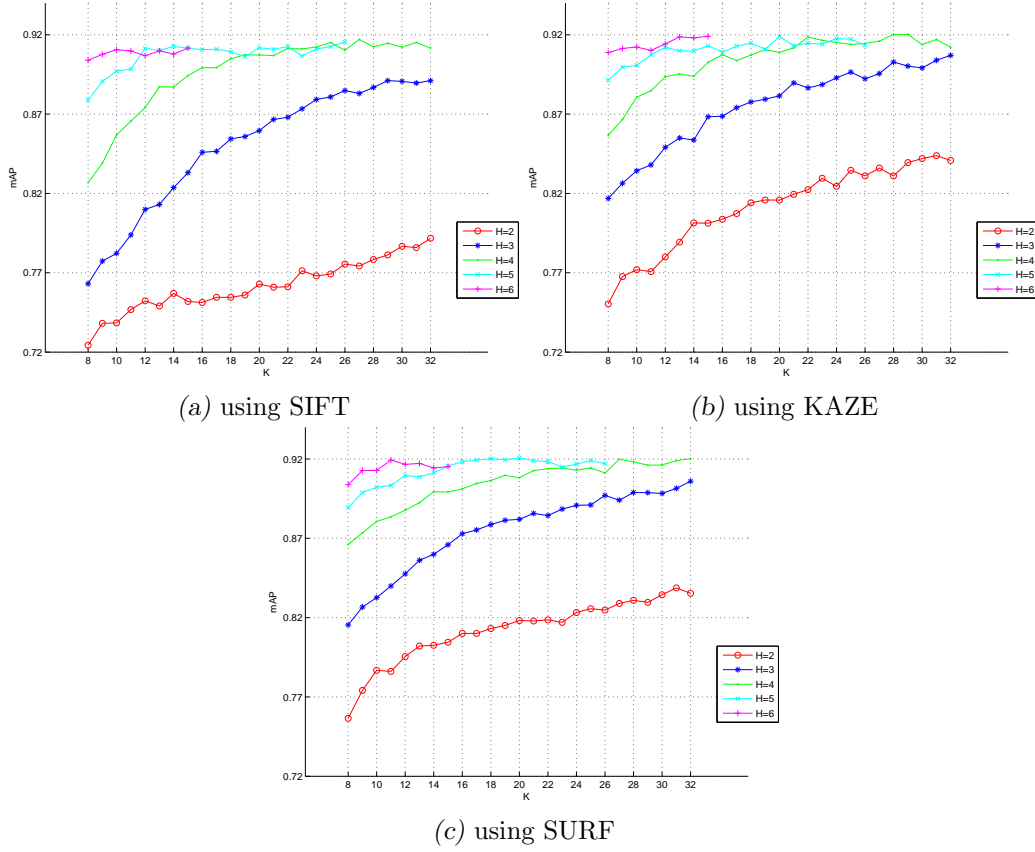


Fig. 5.2: mAP behavior varying K between 8 and 32 for $H = 2, 3, 4, 5, 6$ values, using L_2 norm, using a) SIFT, b) KAZE, c) SURF. L1 norm was used for scoring. For KAZE, parameters were *extended = true*, *threshold = 0,0001*

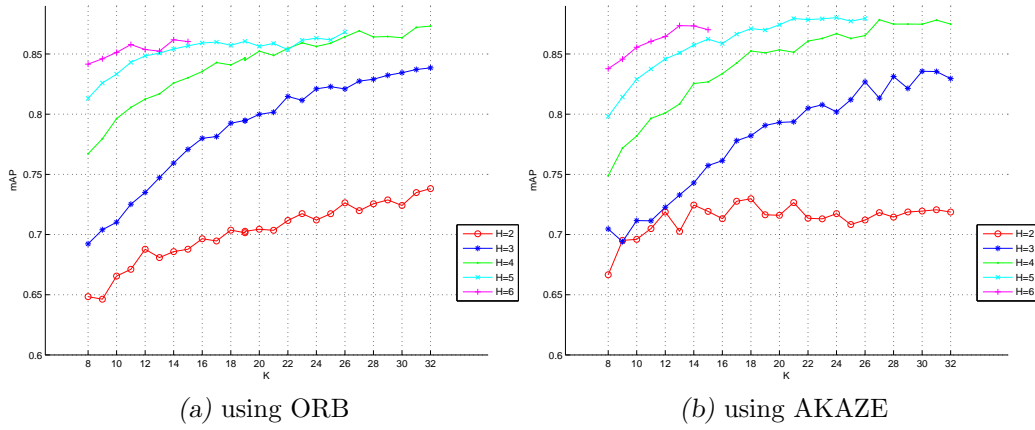


Fig. 5.3: mAP behavior varying K between 8 and 32 for $H = 2, 3, 4, 5, 6$ values, using Hamming distance, and a) ORB, b) AKAZE. L1 norm was used for scoring. For AKAZE parameter *threshold = 0,0001* was used.

this method scales well with the dataset size. The size of the dataset was varied from 10200 to 1 million images.

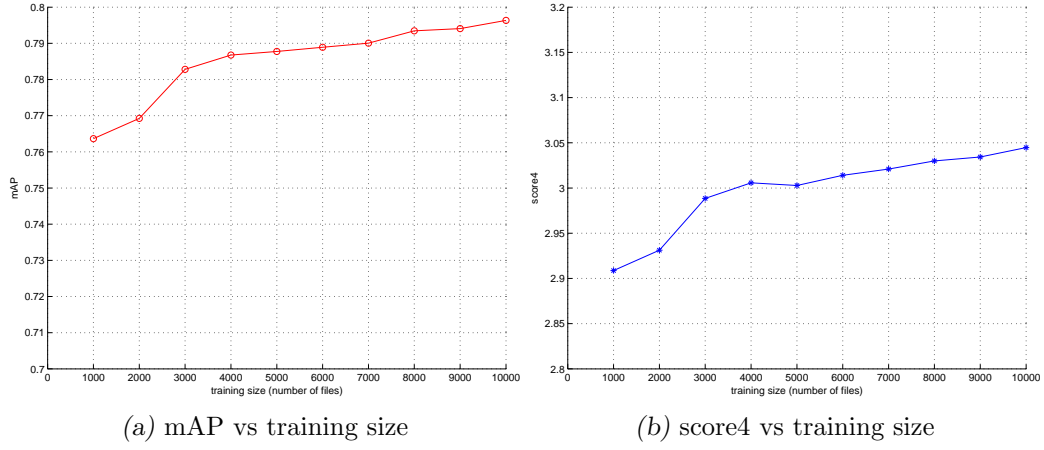


Fig. 5.4: Performance vs training size (number of images). Vocabulary was built varying the number of input training files from 1000 to 10000. UKBench dataset. Method used SIFT, with $K = 27$, $H = 4$, a) mAP, b) score4.

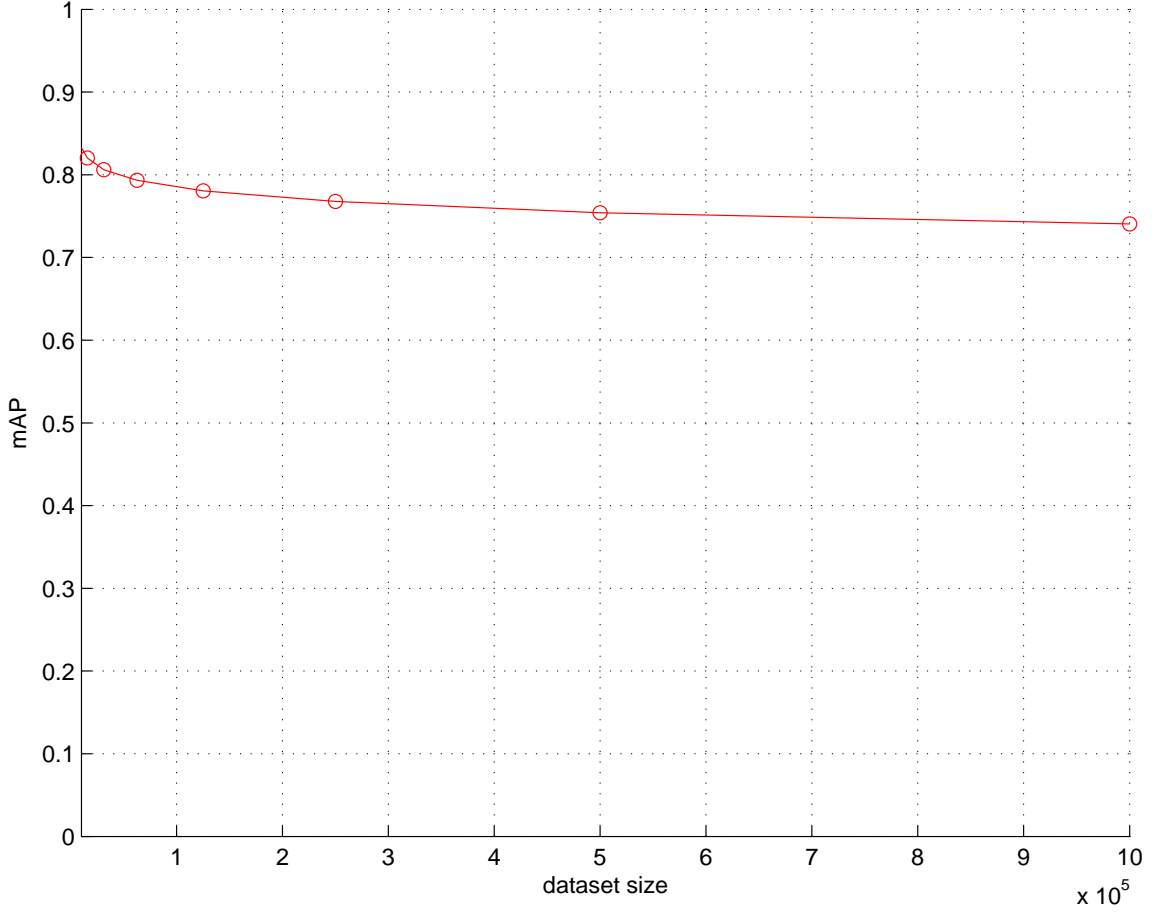
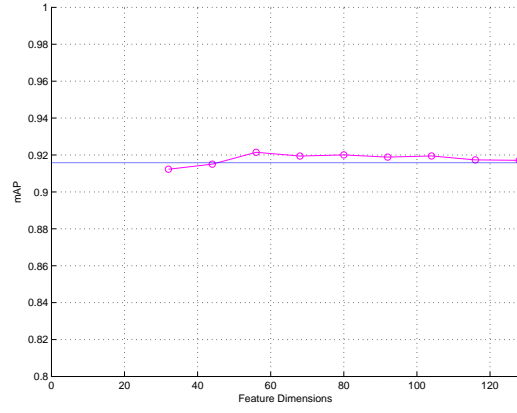


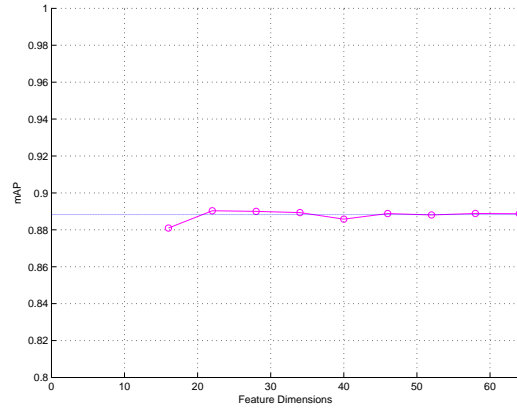
Fig. 5.5: mAP vs dataset size (up to 1 million of image files). UKBench dataset. SIFT + PCA, 64 dims.

5.6. Performance vs dimension reduction

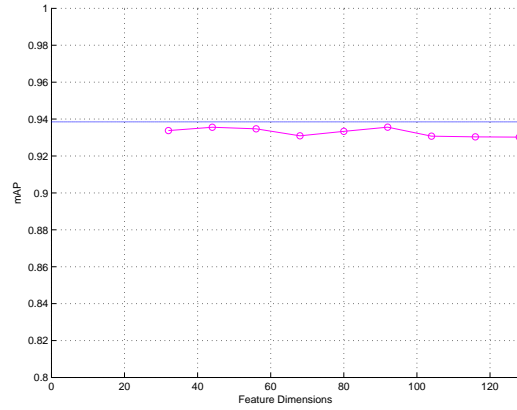
A good improvement can be achieved by using Principal Component Analysis (PCA) to reduce the dimensionality of the features. Dimensionality reduction is done before generating the vocabulary, as well as at runtime phase. Figure 5.6 shows the results of applying PCA on the training features of the first 1000 images of the UKBench dataset. Figure 5.7 shows the results of the same experiment, but applying PCA to reduce dimensions of SIFT descriptors from 128 to 2 dimensions. The results show that if the dimensionality of the features is reduced by PCA, the performance is not deteriorated, and can even grow. The reduction of dimensionality has no impact on the query execution time or could even slightly improve it. However, a greater advantage can be achieved by reducing the required storage space and execution time in the training phase. For example SIFT experiments show that half of the descriptor dimensions could be discarded without making a worse performance, and requiring half space to store features. One of the possible causes of this small performance increase could be related to the fact that less dimensions avoid the effects of the curse of dimensionality, and K-clustering algorithms can deal better with less dimensions.



(a) mAP vs feature dimensions. SIFT.



(b) mAP vs feature dimensions. KAZE.



(c) mAP vs feature dimensions. KAZE extended.

Fig. 5.6: mAP vs feature dimensions. UKBench dataset. Blue line represents mAP without applying PCA for the same input. $K = 27$, $H = 4$. a) SIFT descriptors, b) KAZE descriptors, c) KAZE descriptors with parameters *extended* = true, *threshold* = 0,0001

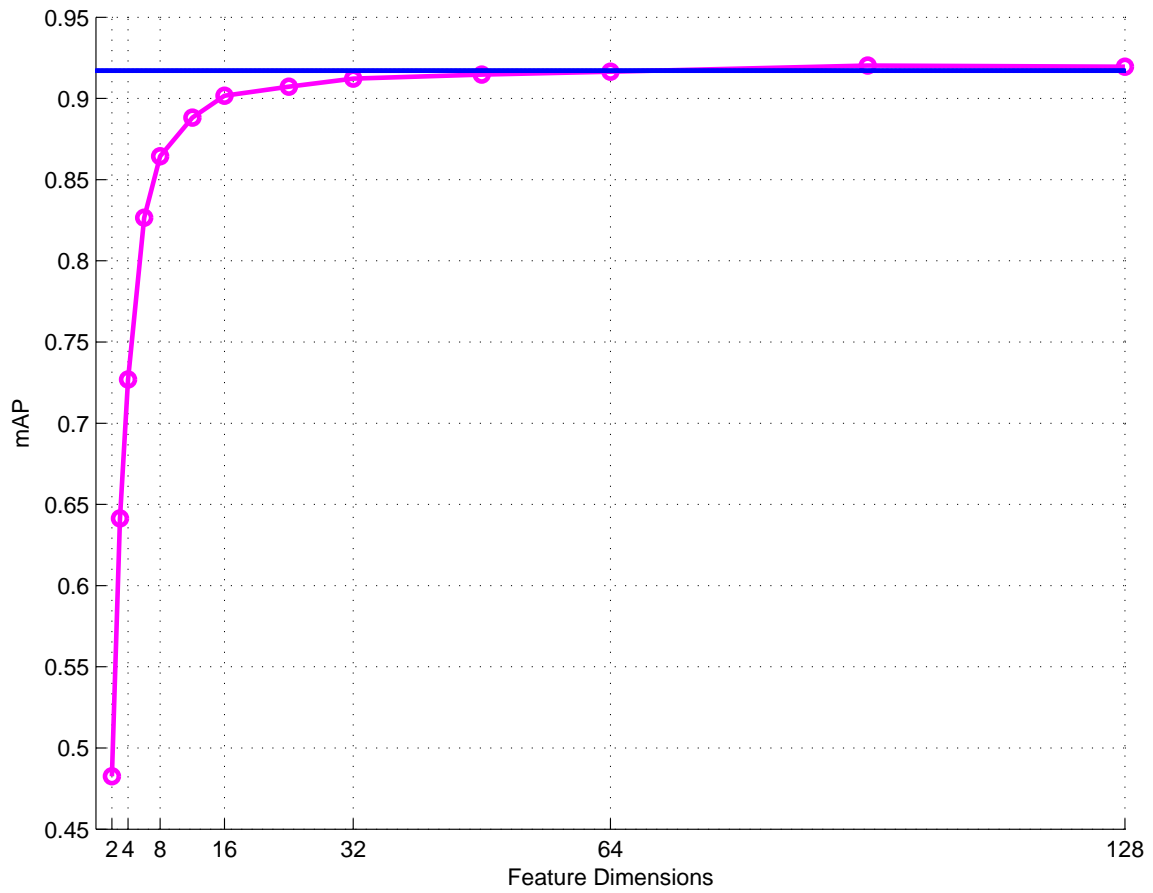


Fig. 5.7: mAP vs feature dimensions. SIFT descriptors were reduced from 128 to 2 dimensions using PCA. mAP was computed using the first 1000 images of the UKBench dataset.

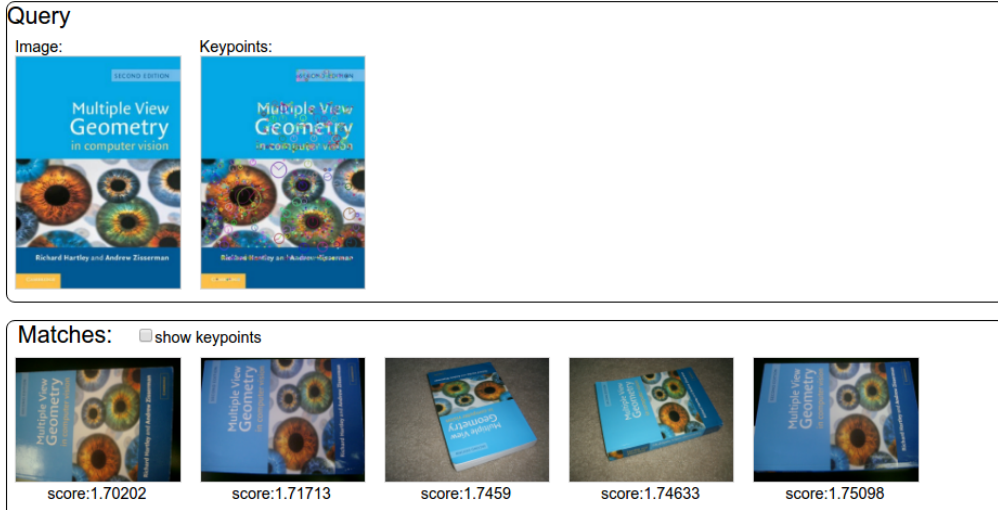
6. IPOL DEMO

Online demo can be found in http://dev.ipol.im/~estebanuri/ipol_demo/voctree/. Figure 6.1 show some example results.

In the demo, a process is listening for queries. When first query arises, precomputed data is loaded into memory, so the first query takes some time until data is loaded. Next queries are run fast, it works like a database engine. When no query is received for a while, the memory is released.



(a) example 1



(b) example 2

Fig. 6.1: Examples of queries results on Ipol interactive demo. UKBench dataset. Query was done using a) a query file from within the dataset, b) uploading an image downloaded from Amazon

7. APPLICATIONS

There are several common problems in many applications where the Vocabulary Tree approach could be applied. Among others we could mention:

Object/Scene search in videos Vocabulary Tree technique can be used in any set of pictures. A video can be thought as sequence of images. Hence, one possible application could be to recognize objects within the frames of movies, tv shows, or commercials. For example if we make a rough estimation, a full length movie in mean could last about 1.5 hours, if it would have 24 frames per second we could reach to 1 million of images with about 7 or 8 full length movies, and scalability provided by this method could be exploited to address the problem in an efficient way. Figure 7.1 shows an example of this application.

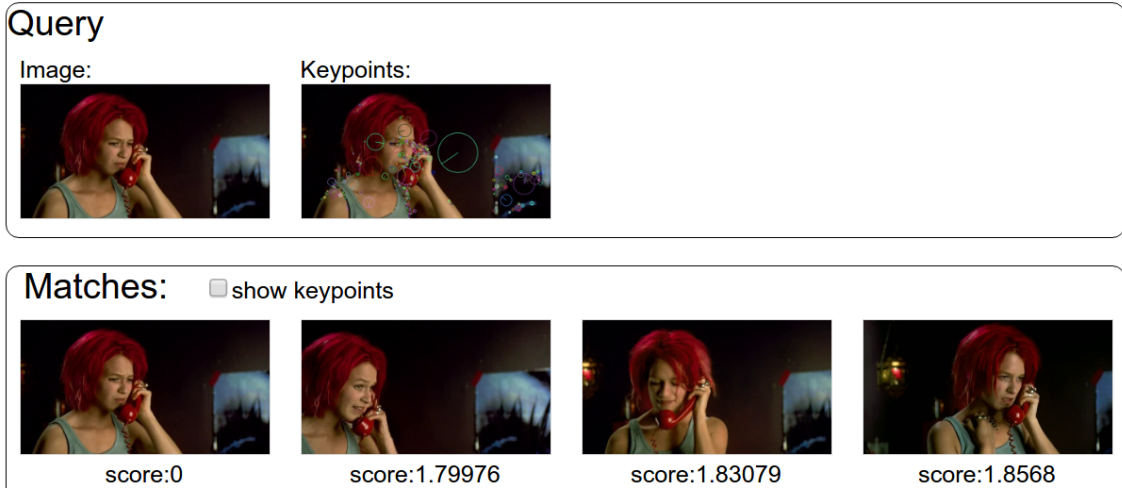


Fig. 7.1: Scenes retrieval from the movie Run Lola Run

Visual Mobile Location Vocabulary tree has been adopted in works like [11] because it provides the ability to insert new samples on-the-fly into the system at run time. Robotic mobile applications usually require the construction of a representation or map of the environment and the reliable location. Similarity of images taken from the environment can be evaluated, in order to improve pose estimation. Even though if the map information is given, since GPS does not work indoors, smart phones often use a coarse location information based on the surrounding telephony cell towers. Providing an accurate indoor position could be a challenging task. Many current indoor positioning systems are based on wireless context sensing (fingerprinting retrieved from wifi access points, beacons) but location accuracy is not good enough for a wide range of uses that would require accuracy. A location scheme based on visual words would provide an interesting and much more accurate alternative to improve the estimation of both position and pose, when GPS is not available.

Bank notes / wine labels recognition An image recovery system can be used as a multi class classifier. The query is performed, and the system returns the t nearest neighbor images. If the images retrieved are labeled with classes, a voting scheme can be employed to determine which class the query corresponds to. This would be a t -nearest neighbors classifier.

An application for bank notes recognition on a smart phones could be very useful for visually impaired people. Most countries do not have regulations that require adding tactile marks to bank notes, in order to would allow blind people to recognize the bank note denomination. A smart phone could apply the image recognition and tell the denomination with sound.

Figures 7.2 and 7.3 illustrates the idea.

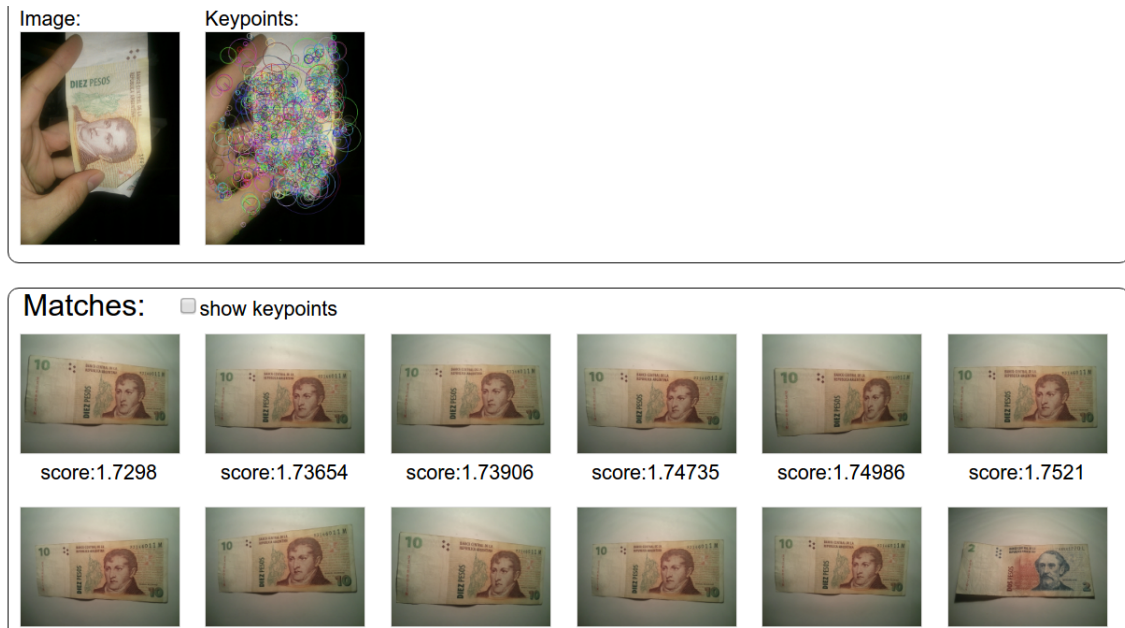

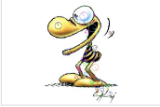


Fig. 7.2: Bank note Recognition. Recognition of 10 Argentinean pesos bank note. 11 samples of each denomination were used to train. There are enough elements on top that votes the same class. There is a majority consensus and then the response is “10 pesos”.

A similar application could be done to recognize wine labels. Figures 7.4 and 7.5 shows examples of Argentinean wine labels recognition using vocabulary tree. Once a wine label is detected, a post-processing algorithm can be applied in order to detect the vintage and the variety of grape.

Query

Image:  Keypoints: 

Matches: ☐ show keypoints














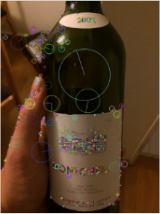
					
score:1.8297	score:1.84867	score:1.85267	score:1.85464	score:1.86501	score:1.86552
					
score:1.86585	score:1.86661	score:1.86718	score:1.86725	score:1.86744	score:1.86863

Fig. 7.3: Bank note Recognition. Rejection example. A picture of “Clemente” is given as query. There is no consensus and the response is “not a bank note”.

Query

Image:  Keypoints: 

Matches: ☐ show keypoints







					
---	---	---	---	--	---

Fig. 7.4: Wine label recognition example 1.

Query

Image: 

Keypoints: 

Matches: ☐ show keypoints



Fig. 7.5: Wine label recognition example 2.

8. CONCLUSIONS

An implementation of object recognition using a vocabulary tree was presented here, as well as a detailed explanation of the involved algorithms. Source code in `c++` is provided and its functionality can be executed with the on-line IPOL interactive demo. This implementation was evaluated using different feature extraction methods, and varying vocabulary construction parameters, in order to measure how these elements affect retrieval performance.

Performed experiments shown that:

- Larger values of branch factor K and tree maximum height H increase the number of tree nodes, allowing to work with more visual words that generate a richer vocabulary, and providing better performance.
- More training samples increases the retrieval performance, both increasing the number of images, and the number of features per image.
- The method scales well with the size of the input.

Even though vocabulary creation depends strongly on the amount of memory available, an external clustering implementation is provided, that allows using training sizes bigger than RAM availability.

Several applications for vocabulary tree were proposed. Standard hardware gives a fixed amount of memory constraint, and hierarchical clustering is a suitable approach to address this problem. This constrained memory requirements could be exploited in several applications on weaker hardware devices such as smart phones, mobile robots, among others.

As contributions to this method can be mentioned that:

- First, feature dimensionality reduction, using techniques such as PCA at least does not worsen retrieval performance, but can allow to work with less memory storage.
- Second, the use of the new feature extraction methods like *KAZE* and *AKAZE* performs as well as the classic ones and provide a open source alternative to the non-free ones.

8.1. Future work

Many parts of this implementation could be improved, or further developed. As future extensions can be mentioned:

- Adding re-ranking stage for results. This could be achieved using geometric consistency as proposed in [33].
- Improving descriptiveness to features used for building the vocabulary, and the use of visual phrases as proposed in [43]

-
- Embedding extra information to the features to improve results as proposed in [15], and [16]
 - Exploring different alternatives for clustering method. Actually *K-means* and *K-majority* were experimented. While there are many other clustering techniques, there are not many that scale well with the input size. It would be interesting to explore different clustering variants. *K-clustering* forces a fixed branch factor, but it would be interesting to experiment with an adaptive branch factor, that perhaps would give better description of the data.

Appendices

.1. Scoring Derivation

If vectors are normalized, any Minkowsky distance can be computed inspecting only components where both vector components are non zero

To compute normalized difference in L_p -norm can be used that:

$$\|\bar{q} - \bar{d}^j\|_p^p = \sum_{i=1}^n |\bar{q}_i - \bar{d}_i^j|^p \quad (.1)$$

This sum can be split in three terms. One where \bar{d}^j components are zero, other where \bar{q} components are zero, and another one where both vector components are different from zero:

$$\sum_{i=1}^n |\bar{q}_i - \bar{d}_i^j|^p = \left(\sum_{i|\bar{d}_i^j=0}^n |\bar{q}_i|^p \right) + \left(\sum_{i|\bar{q}_i=0}^n |\bar{d}_i^j|^p \right) + \sum_{i|\bar{d}_i^j \neq 0, \bar{q}_i \neq 0}^n |\bar{q}_i - \bar{d}_i^j|^p \quad (.2)$$

On the other hand we can write the norm of \bar{q} and the norm of \bar{d}^j split in the same way we did for equation .2 (one term where the components of counterpart vector are zero and other term where not:

$$\|\bar{q}\|_p^p = \sum_{i=1}^n |\bar{q}_i|^p = \sum_{i|\bar{d}_i^j=0}^n |\bar{q}_i|^p + \sum_{i|\bar{d}_i^j \neq 0}^n |\bar{q}_i|^p \quad (.3)$$

$$\|\bar{d}^j\|_p^p = \sum_{i=1}^n |\bar{d}_i^j|^p = \sum_{i|\bar{q}_i=0}^n |\bar{d}_i^j|^p + \sum_{i|\bar{q}_i \neq 0}^n |\bar{d}_i^j|^p \quad (.4)$$

Let's isolate on equations .3 and .4 the left sum.

$$\Rightarrow \sum_{i|\bar{d}_i^j=0}^n |\bar{q}_i|^p = \|\bar{q}\|_p^p - \sum_{i|\bar{d}_i^j \neq 0}^n |\bar{q}_i|^p \quad (.5)$$

$$\Rightarrow \sum_{i|\bar{q}_i=0}^n |\bar{d}_i^j|^p = \|\bar{d}^j\|_p^p - \sum_{i|\bar{q}_i \neq 0}^n |\bar{d}_i^j|^p \quad (.6)$$

Now replacing equations .5 and .6 into equation .2, we have:

$$\|\bar{q} - \bar{d}^j\|_p^p = \left(\|\bar{q}\|_p^p - \sum_{i|\bar{d}_i^j \neq 0}^n |\bar{q}_i|^p \right) + \left(\|\bar{d}^j\|_p^p - \sum_{i|\bar{q}_i \neq 0}^n |\bar{d}_i^j|^p \right) + \sum_{i|\bar{d}_i^j \neq 0, \bar{q}_i \neq 0}^n |\bar{q}_i - \bar{d}_i^j|^p \quad (.7)$$

Note that on equation .7, the conditions of the first two sums can be replaced with the same condition of the third sum (that is stronger). For example the first sum can be split in two sums, one where $\bar{q}_i = 0$ and other where $\bar{q}_i \neq 0$:

$$\sum_{i|\bar{d}_i^j \neq 0}^n |\bar{q}_i|^p = \sum_{i|\bar{d}_i^j \neq 0, \bar{q}_i=0}^n |\bar{q}_i|^p + \sum_{i|\bar{d}_i^j \neq 0, \bar{q}_i \neq 0}^n |\bar{q}_i|^p = 0 + \sum_{i|\bar{d}_i^j \neq 0, \bar{q}_i \neq 0}^n |\bar{q}_i|^p \quad (.8)$$

Then we can combine the three sums of equation .7 onto equation .9:

$$\|\bar{q} - \bar{d}^j\|_p^p = \|\bar{q}\|_p^p + \|\bar{d}^j\|_p^p + \sum_{i|\bar{d}_i^j \neq 0, \bar{q}_i \neq 0}^n (|\bar{q}_i - \bar{d}_i^j|^p - |\bar{q}_i|^p - |\bar{d}_i^j|^p) \quad (.9)$$

Since \bar{q} and \bar{d}^j are normalized, its norm is equal to 1.

$$\|\bar{q} - \bar{d}^j\|_p^p = 2 + \sum_{i|\bar{d}_i^j \neq 0, \bar{q}_i \neq 0}^n (|\bar{q}_i - \bar{d}_i^j|^p - |\bar{q}_i|^p - |\bar{d}_i^j|^p) \quad (.10)$$

For the case of L_2 -norm, using Newton's Binomial, equation .10 simplifies further to

$$\|\bar{q} - \bar{d}^j\|_2^2 = 2 - 2 \sum_{i|\bar{d}_i^j \neq 0, \bar{q}_i \neq 0}^n \bar{q}_i \bar{d}_i^j \quad (.11)$$

.2. K-Means Clustering

Cluster analysis or clustering is the task of grouping a set of objects in such a way that objects in the same group (called a cluster) are more similar to each other than to those in other groups (clusters).

K-means clustering aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean. Given a set of observations $X = (x_1, x_2, \dots, x_n)$, where each observation is a d -dimensional real vector, k-means clustering aims to partition the n observations into $k (\leq n)$ sets $S = \{S_1, S_2, \dots, S_k\}$ so as to minimize the within-cluster sum of squares (WCSS). In other words, the objective is to find:

$$\underset{s}{\operatorname{argmin}} \sum_{i=1}^k \sum_{x_j \in S_i} \|x_j - \mu_i\|^2 \quad (.12)$$

where μ_i is the mean of S_i .

As it is a heuristic algorithm, there is no guarantee that it will converge to the global optimum, and the result may depend on the initial clusters. As the algorithm is usually very fast, it is common to run it multiple times with different starting conditions. However, in the worst case, k-means can be very slow to converge: in particular it has been shown that there exist certain point sets, even in 2 dimensions, on which k-means takes exponential time, that is $2^{\Omega(n)}$, to converge, but these point sets do not seem to arise in practice: this is corroborated by the fact that the smoothed running time of k-means is polynomial.

stop conditions

- maximum number of iterations
- most of the observations does not change from cluster, or
- means have moved less than a threshold

fixing empty clusters There is an improbable but possible case, at the end of the assignment step of the algorithm. K-Means guarantees K clusters, but the assign_clusters step might end with one or more empty sets S_i . One way to handle this case is as follows:

Let p be a cluster with no elements.

Let h be the cluster to which belong more elements, and Let x_f be the farthest element in the cluster h then: remove x_f from h and assign x_f to cluster p , to create a new cluster with 1 element.

Algorithm 7: standard K-means algorithm

K-Means

input : X set of D-dimensional observations

output: S set of partitions

begin

```

    C ← initialize_means( X )
    while ( "the stop condition is not fulfilled" ) do
        S ← assign_clusters ( X, C )
        "fix empty clusters if any"
        C ← update_means ( S )

```

initialize_means

input : X set of D-dimensional observations

output: C the initial K means

begin

```

    C ← "choose distinct K random elements from X"

```

assign_clusters

input : X observations, C means

output: S set of partitioned observations

begin

```

    Let  $S_i$  be K empty sets
    // assign each  $x_j$  to its nearest mean
    foreach  $x_j$  in  $X$  do
         $i \leftarrow \operatorname{argmin}_i \|C_i - x_j\|^2$ 
         $S_i \leftarrow S_i \cup \{x_j\}$ 
     $S \leftarrow \cup_{i=1}^K S_i$ 

```

update_means

input : S set of partitioned observations

output: C updated cluster means

begin

```

    for  $1 \leq i \leq K$  do
         $C_i \leftarrow \frac{1}{|S_i|} \sum_{x_j \in S_i} x_j$ 

```

.2.1. External K-means clustering

In the results section it can be seen as a larger vocabulary gives better recovery performance. It can also be observed that for a fixed size of vocabulary, using more training images improves the results. However, since clustering algorithms need to have all the information stored in RAM, the number of images that can be used to train the vocabulary is limited.

The solution found is to implement an external clustering algorithm that clusters the information while it reads from the hard disk. The disadvantage of reading from the hard disk is that is much slower than accessing to memory, then we should optimize the algorithm to access as few times as possible to disk (a widely studied topic in databases area). According to the tests we did with a standard hard disk drive we get a reading speed of 100MB/sec. while with a solid-state drive (or SSD) was around 500MB/sec. it would take around 20 and 4 minutes respectively to do only one pass to read 120 Gigabytes of data (and k-means requires to do several passes).

divide and conquer approach We tried to address the problem with a “divide and conquer” strategy, similarly as is done for external sorting. The idea was basically, to read a chunk of data in main memory, perform clustering with standard k-means algorithm and write the clustered data to disk. This process is repeated until every n chunk has been clustered. Merge process is done joining the K means of the n chunks, and apply standard k-means to this resulting set, producing a new set of final K means. Finally the n chunks are read again, and each element in chunk is assigned to one of the final cluster center.

According to our experiments, this approach seems to work well with a limited number of chunks, but apparently the difference between the result obtained by the standard k-means and this method increases when the number of partitions grows. Then we abandoned the approach.

k-means external implementation Our trade-off implementation does several passes over the disk performing the k-means standard algorithm. It takes a file as input containing the data to be clustered and it returns K cluster output files. Tree building requires to cluster again and again each one of the K files, producing other K output files for each one. At each step, each of the K files are smaller than the original file, so that at one point the data fits entirely in memory, and then is applied conventional clustering.

For initializing centers, we are taking a number p of random samples from the entire data set, and then we apply standard k-means to produce starting means.

.3. Implementation Details

.3.1. Tree representation using an array

Every node in the tree is either a leaf or a branch. If it is a leaf it has 0 children nodes, and if it is a branch it has exactly K children nodes. Taking advantage of this fact, we can place level by level in an array, first level 0, then level 1, then level 2 ... and so on, starting with the node 0 (the root node) at position 0.

Figure .1 depicts this representation. The vertical lines show the boundaries of the levels. Sign $-$ represents an empty node on the array.

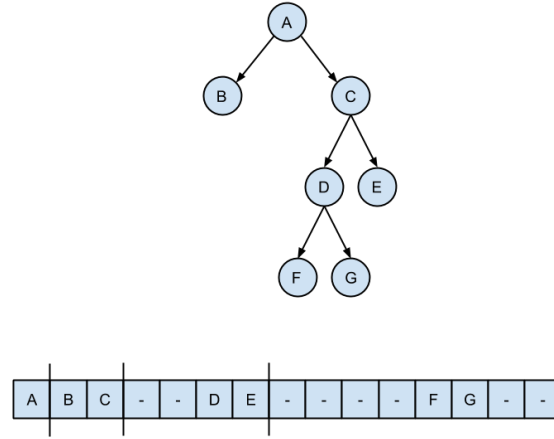


Fig. .1: Example of tree representation by levels using an array (K=2, H=3).

Level size A tree is complete when all its levels are complete. If level 0 is complete then it has 1 node, if level 1 is complete then it has K nodes, if level 2 is complete it has K^2 nodes, and so on. In general if a level L is complete then it has K^L nodes.

If a level is not complete, we leave empty array cells as if it was complete (as it is shown on figure .1). Doing so, given a node index i it is possible to know with a scalar computation where the children of node i are placed. Consequently, cells will be reserved as if the level was complete (.13).

$$Size(L) = K^L \quad (.13)$$

Level start index Each level L starts where the preceding levels finish. Level 0 has 0 nodes preceding, then it starts at index 0, Level 1 has 1 node preceding (the root node), then it starts at index 1, Level 2 has $K + 1$ nodes preceding (the root node + level 1), then it starts at index $K + 1$, Level 3 has $K^2 + K + 1$ nodes preceding (the level 2 + the level 1 + the root), then it starts at index $K^2 + K^1 + 1$, and so on. In general a level L has as start index the sum of the sizes of the preceding levels (.14).

$$Start(L) = K^0 + K^1 + \dots + K^{L-1} = \sum_{i=0}^{L-1} K^i \quad (.14)$$

Using the geometric sum we have the start index of L is (.15) :

$$Start(L) = \frac{K^L - 1}{K - 1} \quad (.15)$$

Children nodes indexes As we need to traverse the tree from root to leaves, given a node i , we need to know the indexes of its children nodes in the array. For the example in figure .1, the children of node D (at position 5) are F and G (at positions 11 and 12).

Let i be a node index, in general if node i is on the level L , then the children of node i are on level $L + 1$. Let Pn be the number of preceding-to- i nodes on level L , then

$$Pn = i - Start(L) \quad (.16)$$

For each preceding-to- i cell on the level L , in the level $(L + 1)$ will be exactly K cells before the cells of the children of the node i .

Then on level $L + 1$ the offset where the children of i start is

$$\begin{aligned} offset &= KPn \\ &= K(i - Start(L)) \\ &= Ki - KStart(L) \\ &= Ki - K\left(\frac{K^L - 1}{K - 1}\right) \\ &= Ki + \frac{-K^{L+1} + K}{K - 1} \end{aligned} \quad (.17)$$

Let c ($0 \leq c < K$) be the c th child node of i . The index of child c will be shifted c positions on the next level from where the children of i start, that is:

$$Child(i, c) = Start(L + 1) + offset + c \quad (.18)$$

Then if we use equations .13 and .17 we have:

$$\begin{aligned} Child(i, c) &= \left(\frac{K^{L+1} - 1}{K - 1}\right) + \left(Ki + \frac{-K^{L+1} + K}{K - 1}\right) + c \\ &= Ki + c + \frac{K^{L+1} - 1 - K^{L+1} + K}{K - 1} \\ &= Ki + c + \frac{K - 1}{K - 1} \\ &= Ki + c + 1 \end{aligned} \quad (.19)$$

That means that we can access directly to the children of i performing a simple arithmetic operation and jumping to the array child index.

This array representation would waste memory if the tree is not complete. Another drawback is the fact nodes need to store different type of information depending on whether the node is a branch or the leaf, and then the size of the node differs depending of the node type.

The solution for both problems could be to store only the nodes indexes (integer numbers) into the array.

Given K and H , first we instantiate the array of integers having length equal to the maximum number of nodes that K and H would allow to have. For example with $K = 10$ and $H = 6$, the tree (if complete) would have 1111111 nodes, in terms of memory 1111111 integers consume about 4MB. And this is the maximum amount of memory we could waste (when the tree is empty). This is not too much waste compared with the amount of memory required for the other data structures needed such as the clusters centers, the dvectors, or the inverted indexes.

At the beginning this array is initialized with blanks (example with -1), and a counter u of used nodes is initialized in 0. Each time a new node i is created the memory for that node is reserved in another data structure, the counter u is incremented by 1, and its value is copied to the array as the node index of i .

Bibliografía

- [1] Motilal Agrawal, Kurt Konolige, and Morten Rufus Blas. Censure: Center surround extremas for realtime feature detection and matching. In *European Conference on Computer Vision*, pages 102–115. Springer, 2008.
- [2] Alexandre Alahi, Raphael Ortiz, and Pierre Vanderghenst. Freak: Fast retina key-point. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 510–517. Ieee, 2012.
- [3] Pablo F Alcantarilla and TrueVision Solutions. Fast explicit diffusion for accelerated features in nonlinear scale spaces. *IEEE Trans. Patt. Anal. Mach. Intell*, 34(7):1281–1298, 2011.
- [4] Pablo Fernández Alcantarilla, Adrien Bartoli, and Andrew J Davison. Kaze features. In *European Conference on Computer Vision*, pages 214–227. Springer, 2012.
- [5] Alexandr Andoni, Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab Mirrokni. Nearest-neighbor methods in learning and vision: theory and practice, 2006.
- [6] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. Surf: Speeded up robust features. In *Computer vision—ECCV 2006*, pages 404–417. Springer, 2006.
- [7] Michael Calonder, Vincent Lepetit, Christoph Strecha, and Pascal Fua. Brief: Binary robust independent elementary features. In *European conference on computer vision*, pages 778–792. Springer, 2010.
- [8] Vijay Chandrasekhar, Jie Lin, Olivier Morere, Antoine Veillard, and Hanlin Goh. Compact global descriptors for visual search. In *2015 Data Compression Conference*, pages 333–342. IEEE, 2015.
- [9] 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.
- [10] R. Funayama, H. Yanagihara, L. Van Gool, T. Tuytelaars, and H. Bay. Robust interest point detector and descriptor, September 24 2009. US Patent App. 12/298,879.
- [11] Dorian Gálvez-López and Juan D Tardos. Bags of binary words for fast place recognition in image sequences. *IEEE Transactions on Robotics*, 28(5):1188–1197, 2012.
- [12] Costantino Grana, Daniele Borghesani, Marco Manfredi, and Rita Cucchiara. A fast approach for integrating orb descriptors in the bag of words model. In *IS&T/SPIE Electronic Imaging*, pages 866709–866709. International Society for Optics and Photonics, 2013.
- [13] Chris Harris and Mike Stephens. A combined corner and edge detector. pages 147–151, United Kingdom, 1998. Plessey Research Roke Manor.

-
- [14] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613. ACM, 1998.
 - [15] Herve Jegou, Matthijs Douze, and Cordelia Schmid. Hamming embedding and weak geometric consistency for large scale image search. In *Computer Vision–ECCV 2008*, pages 304–317. Springer, 2008.
 - [16] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. Improving bag-of-features for large scale image search. *International Journal of Computer Vision*, 87(3):316–336, 2010.
 - [17] Herve Jegou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128, 2011.
 - [18] Hervé Jégou, Matthijs Douze, Cordelia Schmid, and Patrick Pérez. Aggregating local descriptors into a compact image representation. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 3304–3311. IEEE, 2010.
 - [19] Herve Jegou, Hedi Harzallah, and Cordelia Schmid. A contextual dissimilarity measure for accurate and efficient image search. In *2007 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8. IEEE, 2007.
 - [20] Jianqiu Ji, Jianmin Li, Shuicheng Yan, Bo Zhang, and Qi Tian. Super-bit locality-sensitive hashing. In *Advances in Neural Information Processing Systems*, pages 108–116.
 - [21] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.
 - [22] Brian Kulis and Trevor Darrell. Learning to hash with binary reconstructive embeddings. In *Advances in neural information processing systems*, pages 1042–1050, 2009.
 - [23] Vincent Lepetit, Pascal Lager, and Pascal Fua. Randomized trees for real-time keypoint recognition. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 2, pages 775–781. IEEE, 2005.
 - [24] Stefan Leutenegger, Margarita Chli, and Roland Y Siegwart. Brisk: Binary robust invariant scalable keypoints. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 2548–2555. IEEE, 2011.
 - [25] David G Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.
 - [26] D.G. Lowe. Method and apparatus for identifying scale invariant features in an image and use of same for locating an object in an image, March 23 2004. US Patent 6,711,293.

-
- [27] Tomás Mardones, Héctor Allende, and Claudio Moraga. Graph fusion using global descriptors for image retrieval. In *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications*, pages 290–297. Springer, 2015.
 - [28] Jiri Matas, Ondrej Chum, Martin Urban, and Tomás Pajdla. Robust wide-baseline stereo from maximally stable extremal regions. *Image and vision computing*, 22(10):761–767, 2004.
 - [29] Marius Muja and David G Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. *VISAPP (1)*, 2, 2009.
 - [30] David Nister and Henrik Stewenius. Scalable recognition with a vocabulary tree. In *Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Volume 2*, CVPR '06, pages 2161–2168, Washington, DC, USA, 2006. IEEE Computer Society.
 - [31] Aude Oliva and Antonio Torralba. Modeling the shape of the scene: A holistic representation of the spatial envelope. *International journal of computer vision*, 42(3):145–175, 2001.
 - [32] Florent Perronnin, Yan Liu, Jorge Sánchez, and Hervé Poirier. Large-scale image retrieval with compressed fisher vectors. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 3384–3391. IEEE, 2010.
 - [33] James Philbin, Ondřej Chum, Michael Isard, Josef Sivic, and Andrew Zisserman. Object retrieval with large vocabularies and fast spatial matching. In *Computer Vision and Pattern Recognition, 2007. CVPR'07. IEEE Conference on*, pages 1–8. IEEE, 2007.
 - [34] Edward Rosten and Tom Drummond. Machine learning for high-speed corner detection. In *European conference on computer vision*, pages 430–443. Springer, 2006.
 - [35] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. Orb: An efficient alternative to sift or surf. In *2011 International conference on computer vision*, pages 2564–2571. IEEE, 2011.
 - [36] Ruslan Salakhutdinov and Geoffrey Hinton. Semantic hashing. *RBM*, 500(3):500, 2007.
 - [37] Jorge Sánchez, Florent Perronnin, Thomas Mensink, and Jakob Verbeek. Image classification with the fisher vector: Theory and practice. *International journal of computer vision*, 105(3):222–245, 2013.
 - [38] Ali Sharif Razavian, Hossein Azizpour, Josephine Sullivan, and Stefan Carlsson. Cnn features off-the-shelf: an astounding baseline for recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 806–813, 2014.
 - [39] Jianbo Shi and Carlo Tomasi. Good features to track. In *Computer Vision and Pattern Recognition, 1994. Proceedings CVPR'94., 1994 IEEE Computer Society Conference on*, pages 593–600. IEEE, 1994.

-
- [40] Josef Sivic and Andrew Zisserman. Video google: A text retrieval approach to object matching in videos. In *Computer Vision, 2003. Proceedings. Ninth IEEE International Conference on*, pages 1470–1477. IEEE, 2003.
 - [41] Antonio Torralba, Rob Fergus, and Yair Weiss. Small codes and large image databases for recognition. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pages 1–8. IEEE, 2008.
 - [42] Antonio Torralba, Rob Fergus, and Yair Weiss. Spectral hashing. pages 1–8, 2008.
 - [43] Shiliang Zhang, Qingming Huang, Gang Hua, Shuqiang Jiang, Wen Gao, and Qi Tian. Building contextual visual vocabulary for large-scale image applications. In *Proceedings of the 18th ACM international conference on Multimedia*, pages 501–510. ACM, 2010.
 - [44] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM computing surveys (CSUR)*, 38(2):6, 2006.