University “POLITEHNICA” of Bucharest

Automatic Control and Computers Faculty,
Computer Science and Engineering Department

A Scalable Algorithm for Similar Image Detection

Scientific Advisers:

Prof. Dr. Ing. Nicolae Ţăpuş
Ing. Ştefan-Teodor Crăciun

Author:

Andrei-Bogdan Pârvu

Contents

Abstract

In this paper, I propose an algorithm for detecting similar images in a large scale dataset, with the goal of obtaining a method of determining the appearance of copyright infringement. After considering various methods as invisible watermarking and image feature detection, I decided upon using the *Harris corner detection* and the *Scale Invariant Feature Transform* keypoints and descriptors for analyzing the properties of an image.
In order to be able to handle a large amount of images and associated data, I have decided to maintain the descriptors of the images in a KD-tree structure for fast querying. This allows an initial filtering of the data set, reducing the number of images which should be analyzed. Thus, I will use two algorithms, one of which has a lower time complexity, but only gives semi-accurate results, and a second one, which performed a more in-depth analysis of the images.
I will describe the mathematical aspects of the two mentioned algorithms, and I will describe the adjustments made to the algorithms in order to perform better on our given problem.
The architecture of the application also important, because it highly influences the distribution of data, and how the algorithm performs on a large data set, with a high number of queries and possible updates on the initial image set. I will describe this architecture and the way the data should be distributed so that an efficient use of multiple machines can speed up the algorithm.

Chapter 1

Introduction

Image processing has been a very important research domain the last years, the ever increasing number of images present on the Internet becoming more and more challenging to index, store and analyze.
Privacy and copyright have also been big issues which have been discussed and assessed for a long time, major users in the photography industry wanting to know at all times when someone is using or modifying one of their photos.
The problem discussed in this paper is considered a very difficult one, because the algorithm needs to provide both accurate results and have a decent running time. This is a continuous research problem and with a very high potential, so the study of such an algorithm can easily be extended and continued beyond this paper.

1.1 Requirements

We want to design a scalable algorithm which can maintain a large set of images, and perform queries of finding a highly similar image with a given input one.
The algorithm should be focused on copyright detection, so it should be able to determine if the input image is one of the images in the dataset, with possible transformations applied upon it:

- watermarks
- scaling
- cropping
- various filters

As said above, the granularity of the algorithm should be able to distinguish between two lightly similar images (e.g. two different pictures of the Eiffel Tower, made by two different persons) and two images, one of which is obtained from the other.

The running time of the algorithm is also a major factor which should be seriously taken into consideration. It should be able to handle a large number of queries and provide a small response-time per query. Of course, there is a close connection between the running time of the algorithm and its precision, connection which should be closely analyzed in order to create a balance between the two.

In Figure 1.1, we can see an example usage of the algorithm. The large database is queried with the image in the bottom-left corner, and the algorithm is capable is returning the original image (even if the query image has been cropped, rotated and a gray filtered applied).

Figure 1.1: Basic usage of our algorithm

Chapter 2

Related Work

There are a lot of algorithms which focus on image similarity and key feature detection, my main goal being to select specific ones on which we can control the sensitivity of the matches, but also which can be easily and efficiently distributed on several machines for a large input set.
I have focused on two main algorithms, the *Harris corner detector* and the *Scale Invariant Feature Transform*.

2.1 Harris Corner Detection

The main idea of the *Harris corner detector* algorithm is that, given an input image, the most predominant features that a human eye recognizes and memorizes are corners. A corner is considered to be an intersection of two edges, so, selecting a small area around the point and shifting it should result in a large variation in the intensity of the pixels in that area.
Therefore, each area in an image can be classified in three categories:

- flat, in which intensities do not vary in either direction (as see in Figure 2.1)
- edge, in which intensities don't vary in the direction of the edge (as see in Figure 2.2)
- corner, in which intensities vary in all directions (as see in Figure 2.3)

In order to determine in which category a certain area with a size of (w, h) belongs to, we will compute the variation of intensity: $E(w,h) = \sum_{x,y} w(x,y) * [I(x+w, y+h) - I(x,y)]^2$, where w is a window function, which assigns weights to pixels, and I is the intensity of a certain pixel of the grayscale image.
In order to determine the corner areas, we have to maximize the function $\sum_{x,y}[I(x+w, y+v) - I(x,y)]^2$, which using *Taylor* expansion and representing in a matrix form can be written as $E(w,h) \approx \begin{bmatrix} w & h \end{bmatrix} * \left(\sum_{x,y} \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} \right) * \begin{bmatrix} w \\ h \end{bmatrix}$, and, furthermore, using a substitution $E(w,h) \approx \begin{bmatrix} w & h \end{bmatrix} * M * \begin{bmatrix} w \\ h \end{bmatrix}$.
Using this equation, the score of a certain area is computed as $R = det(M) - k * (trace(M))^2$. A higher score of R denotes a higher probability of the area being a corner.

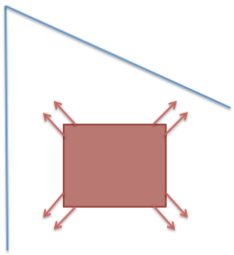

Figure 2.1: Flat Area

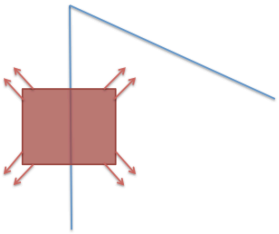

Figure 2.2: Edge Area

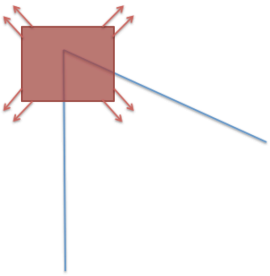

Figure 2.3: Corner Area

2.2 Scale Invariant Feature Transform

2.2.1 Keypoint localization

Although the *Harris corner detection* algorithm presented in the previous section is immune to rotation transformations of an image, it does not perform well if the image is scaled, because a high intensity change in an area of size (w, h) of an image might vary if the dimensions of the image change, but the size of the area remains the same.
Thus, D. Lowe, in 2004, presented a new algorithm for extracting keypoints and computing their descriptors, named *Scale Invariant Feature Transform*.
At first, a Gaussian distribution is applied on the analyzed image, which depending of the standard deviation, σ, blurs the image with a certain amount: $G(x, y) = \frac{1}{2*\pi*\sigma^2} * e^{-\frac{x^2+y^2}{2*\sigma^2}}$.
Then, the Laplacian of the image is computed, in order to highlight the regions of rapid intensity changes: $L(x, y) = \frac{\delta^2 I}{\delta x^2} + \frac{\delta^2 I}{\delta y^2}$. Combined with the previous Gaussian filter, we obtain the so called Laplacian of Gaussian: $LoG(x, y) = -\frac{1}{\pi*\sigma^4} * \left(1 - \frac{x^2+y^2}{2*\sigma^2}\right) * e^{-\frac{x^2+y^2}{2*\sigma^2}}$.
Because the LoG has a high computational cost, it is approximated with a Difference of Gaussians, which is a difference of two Gaussians with two different σ deviations, representing two different scaled images. The local extrema of the computed DoG are considered potential keypoints.

2.2.2 Computing the Descriptors

Once we have the keypoints, the corresponding descriptors are computed by taking a 16×16 neighborhood around the keypoint, and creating a 8 bin histogram for each sub-block of 4×4 size of the initial neighbourhood. Thus, a keypoint descriptor will contain 128 values.

Chapter 3

Design of the Algorithm

3.1 Analysis of an image

3.1.1 Using Harris corner detector

The analysis of an image begins with applying the Harris detector on it. It will compute a given score for each pixel of the image, the higher the score, the greater the chance that pixel represents a corner.
Using these values we will have to determine a subset of pixels that will represent the Harris mask of the image. Experimentally, I have established that all the points which have a value greater than $0.01 * max\_image\_value$ are to be part of the subset.
As an example, Figure 3.1 shows a normal image of a woman's face, while Figure 3.2 shows the corresponding pixels that form the corner mask.

Figure 3.1: Sample image

Figure 3.2: Harris corner mask applied

As it can be seen from the image, the corner mask centers around the interest points in the image, the eyes and the eyelashes, while leaving the smooth surfaces (the skin) unmarked.

Another observation is that possible watermarks can be present in the image (as seen above), which, of course, the mask detects (they are center pieces of the image, and the corner algorithm cannot determine that they have no real connection with the image).

Furthermore, in some cases the watermark might contain all (or at least a vast majority of) the points in the mask (as seen in Figure 3.3). To avoid such a behavior, I have split the image into 9 sub images (three rows and three columns of identical dimensions) and the Harris mask is applied to each of these.

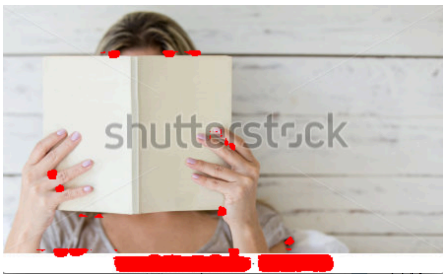

Figure 3.3: Single image

Figure 3.4: 9 subimages

This determines the corner mask to also contain pixels from the center of the image (some of which, unfortunately, are also a watermark).

3.1.2 Using SIFT keypoints and descriptors

As we have seen, the SIFT algorithm computes the keypoints of an image, and then the descriptors of these keypoints. Due to the high similarity nature of our problem, we do not want to compute the keypoints for the entire image, but filter them based on the corner mask determined in the previous section (as observed in Figure 3.5 and Figure 3.6)

Figure 3.5: SIFT keypoints for the entire image

Figure 3.6: SIFT keypoints for corner mask

The SIFT descriptors are then computed for each keypoint located in the corner mask and this will be the information stored for a certain image.

3.2 Analysis of a pair of images

In order to analyze a pair of images, we shall use the SIFT descriptors determined in the previous section. The two sets of descriptors are compared in order to obtain the best matches between pairs of keypoints. A distance is computed between each pair of keypoint descriptors, which is the Euclidian norm between the SIFT descriptors of the keypoints. Experimentally, I

have concluded that a match between two keypoints has a high similarity if the Euclidian norm is less than 100.
For a pair of images, we first find the set of corner-mask keypoints and then compute the best matches between these keypoints. We shall keep only the best 10 matches, and compute the arithmetic mean between the distances of these matches. As stated before, if this mean is smaller than 100, the two images are considered similar. We shall name this algorithm the *pair similarity algorithm*.
Figure 3.7 shows the corresponding matches between two images with two different watermarks, one of which is rotated 90^o clockwise.

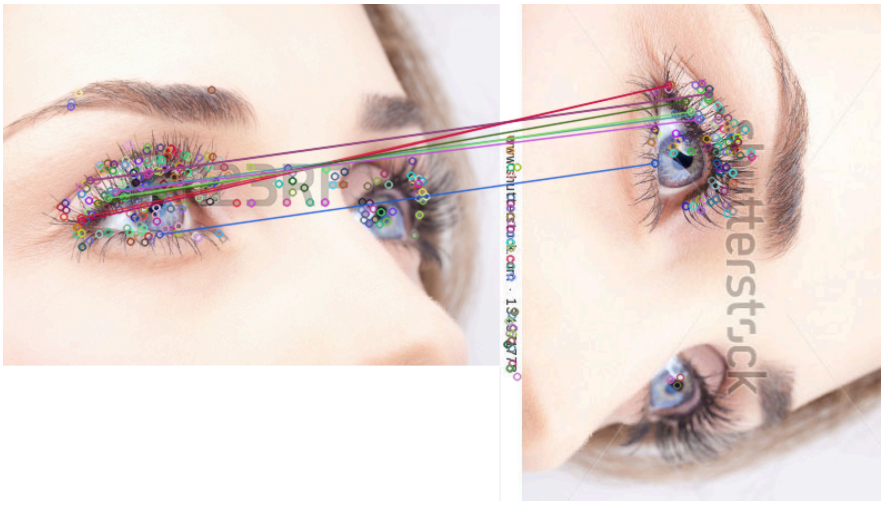

Figure 3.7: Comparison between two images

3.3 Analysis of a set of images

Suppose that we have a set of images, and we want to compare a test image with the set and detect whether we have a similar image within the set. Of course, the first possibility is the brut force one: we iterate through all the images and apply the *pair similarity algorithm* described in the previous section. Although this provides a correct result, it has a complexity of $O(number\_of\_images * image\_match\_time)$. We shall name this basic algorithm as the *linear algorithm*. Although this algorithm is very straightforward, its complexity is undesirable if the number of images becomes large. Moreover, most of the images in our set will likely have a big similarity distance with our searched image, so maybe we don't want to apply the full *pair similarity algorithm*.
Thus, we need to determine an efficient algorithm which can filter the initial set of images to a smaller set which contains very likely matching candidates with our test image.
The filtering algorithm is implemented as follows: we will maintain a maximum number of M descriptors for each image in the initial set and create a KD-tree with the set of descriptors of the initial images. When a query for a test image arrives, we will compute its descriptors and then perform a N nearest-neighbor search on our KD-tree. Then we will select the top T images with the most descriptors returned by the KD-tree and perform the *linear algorithm*. Experimentally, I have determined that the optimal values for M, the number of descriptors, N, the number of neighbors and T, the number of images is 40, 5 and 10. We shall name this algorithm the *kdtree algorithm*.

Chapter 4

Architecture

4.1 Basic structure

To illustrate the functionality of the described algorithm, I have designed a basic application which maintains a data base of images and, for a given query image, can retrieve the best match between the query and the image set.
The basic structure of the application is shown in Figure 4.1: all the queries are received by the Load Balancer, which acts like a front end processor and distributes in a round-robin fashion the queries to several Map Reduce servers. When these servers receive a query, they distribute it to the associated Image Servers, which contain the descriptors for various sets of images. The Image Servers compute the top T most similar images and then they send it back to the Map Reducer, which extracts the best images from the received responses and sends them back to the Load Balancer. There are several advantages of separating the image servers from the FEP. First of all, it allows load balancing: the FEP can maintain a queue of requests and the server with the least work to do can pick it up and process it. Secondly, it allows different types of image servers to be tested, by easily inserting and removing the image server module.
The querying instance type is not particularly important, it can be either a simple web page or a browser extension (e.g. Google Chrome extension).

4.2 Load Balancer

The Load Balancer acts as a classical server. It uses socket communication in order to receive the queries and send them in a round-robin order to the connected Map Reducers. When a response is received from the Map Reducer the Load Balancer sends it back to the entity that has made the given query.

4.3 Map Reducer

The Map Reducer maintains a set of connections to a number of associated Image Servers to which it broadcasts a received query. The Map Reducer waits for the response from the servers (in the meantime it does not process other queries), and combines the received results, by selecting the images with the highest *image pair score* from all the images returned by the Image Servers.

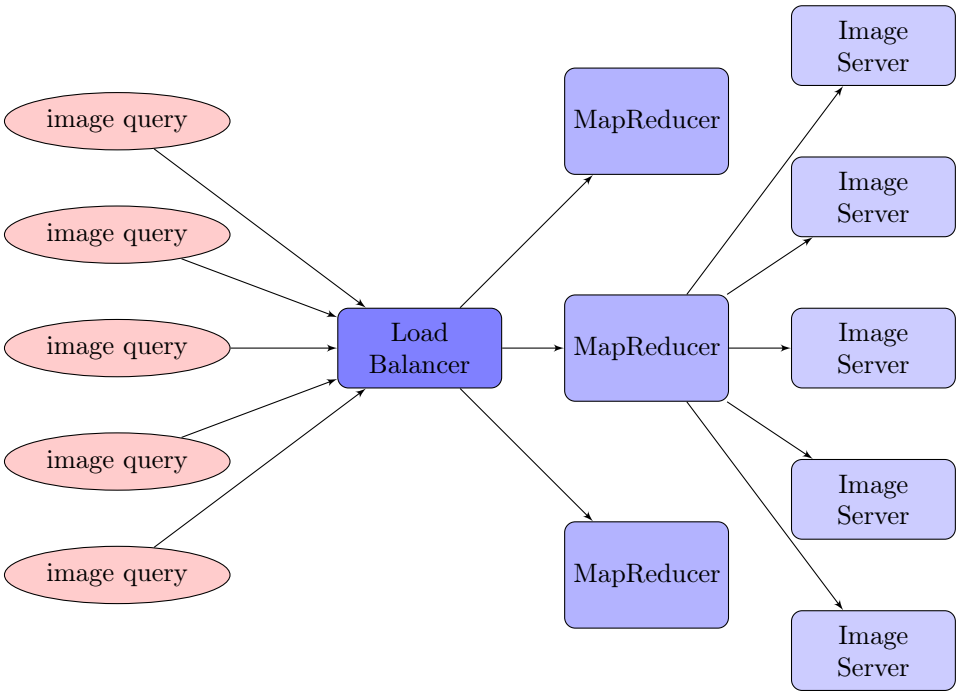

Figure 4.1: Basic structure

4.4 Image Server

4.4.1 Linear Server

The first type of server uses the *linear algorithm* described in Design of the Algorithm. It stores the list of images that form our database and applies the *linear algorithm* a certain query arrives from the Map Reducer.

4.4.2 KD-tree Server

The second type of server uses the *kdtree algorithm* described in Design of the Algorithm. At initialization time the server reads a given list of images from our database, computes its descriptors and applies the above mentioned algorithm when a query arrives from the Map Reducer. The resulting KD-tree is maintained in memory throughout the incoming queries and it it used to find the top T most similar images and return them to the Map Reducer.

4.5 Technologies

The majority of our code, Load Balancer, Map Reducer and Image Serves, is written in C++, because of the high computation nature of our problem. For the image processing and associated algorithm we have used OpenCV version 2.4.8.

4.5.1 Image Storing

We determined that it is better to store only the urls of the images used in out database (not the actual images), the Image Servers downloading an image whenever they need to use it (at initialization time or at query time). The downloading is done using a CURL library for C++. Also, to avoid computing the SIFT descriptors for all the images every time we start an Image Server we opted to retain those descriptors in our database, and only load them when needed. We maintain a Redis server on a machine which stores all the urls and the descriptors of the images, and which we can query using a specific C++ library.

4.5.2 Process Communication

For the process communication, we opted to use sockets so that the various servers can reside on different machines. ZeroMQ sockets looked liked the best option, because of their robustness, small running time, auto-handling of connection failures and different types (DEALER, PUB-SUB, PUSH-PULL, ROUTER, etc.).

Chapter 5

Results

I have tested the two algorithms described in Design of the Algorithm using the structure given in Architecture.
There are two major parts which should be analyzed: the correctness of the algorithm and it's running time.

5.1 Correctness

We have collected a set of 4500 images from the Internet, which we have classified in about 1600 similarity sets, each set being composed of up to five images. The images within a similarity set differ in size, having various watermarks and filters applied (these sets are known to be correct beforehand). We have inserted these images in a larger set of 100.000 images taken from the Internet and performed a queries for each of the 4500 images of the similarity sets, getting the top 10 similar images. We want to observe:

- if the algorithm finds the exact match, i.e. the image that has been queried with
- if the algorithm finds the other images which are part of the same similarity set
- the mean running time of a query
- how varying the metrics described above influences the results of the query

We evaluate the response to a query, by looking at the indices of the images from the current similarity set in the list of results returned by the query. Thus, the *index score* for a certain query can be computed as the sum of these indices; the smaller the sum, the closer in the response list are the images we want to find.
For the set of 100.000 images we use $P = 20$ KD-trees, each KD-tree storing the descriptors for $N = 5000$ images.
The results of this test can be seen in the following two tables. The first one describes, for similarity sets of size 3, 4 and 5, and for the metrics described in **??**, the average number of how many of these images are found in the list of 10 returned images.

	max nr descs	min avg	max dist to avg
3	2.47	2.37	2.52
4	3.26	3.21	3.38
5	3.98	3.63	4.14

The second table shows the *index score* (described above) for the same queries and metrics.

	max nr descs	min avg	max dist to avg
3	8.32	9.29	7.84
4	12.69	13.15	11.76
5	18.35	21.36	17.10

It can be observed that the third metric, the largest distance from the descriptors of an image to the average of all found descriptors, provides the best results.

Also, we have constructed a single KD-tree which contains the 4500 images from the similarity set in order to test the three different metrics described in **??** for selecting the filtered images. We retained the *image pair score* of the returned similar images, and evaluated these three metrics by computing the following *correctness scores*:

- the mean between the *image pair scores*
- the sum of the differences between the *pair scores* of two consecutive similar images in the returned list
- the maximum *image pair score*

The goal is to minimize each of these *correctness scores*.

The results of this test are shown in the next table:

	max nr descs	min avg	max dist to avg
mean	661880	693745	647355
sum of diff	867877	1021543	856521
max score	1162618	1134470	1150899

As it can be seen, the third metric, largest distance from the descriptors of an image to the average of all found descriptors, performs the best out of the three metrics.

5.2 Running Time

We have tested our algorithm on a machine with 55GB of RAM, 16 quad-core processors with a frequency of 2.4GHz.

The first experiment that we did was comparing the runtimes of the *linear algorithm* with the *kdtree algorithm* by running them on sets of $5, 10, 20, 50, 200, 350$ and 500 images. The corresponding running times are shown in Figure 5.1.
The red line shows the corresponding running times for the *linear algorithm*, while the green one shows the times for the *kdtree algorithm*. As it can be seen, the *kdtree algorithm* outperformes the linear one, with the same returned image (so it does not give different results). The non-ascending running times of the *kdtree algorithm* can be explained by the fact that the KD-tree data structure is traversed heuristically, so depending on the given input a certain query can execute with varying running times. For a KD-tree of 5000 images and 5000 queries, the mean running time of a search is 1.36 seconds.
Of course, the *kdtree algorithm* does require an initialization time, which is the price that has to be paid in order to perform fast queries. Figure 5.2 shows the sum between the initialization time and query time of a linear image server and an image server which constructs a KD-tree.

As it can be seen, the initialization time for the KD-tree grows linearly, but this gets compensated by the small query time. For a 500-size image set, the initialization and a query on the KD-tree takes 36.771 seconds, which is less than a query on the same image set using the *linear algorithm*, which takes 77.331 seconds.

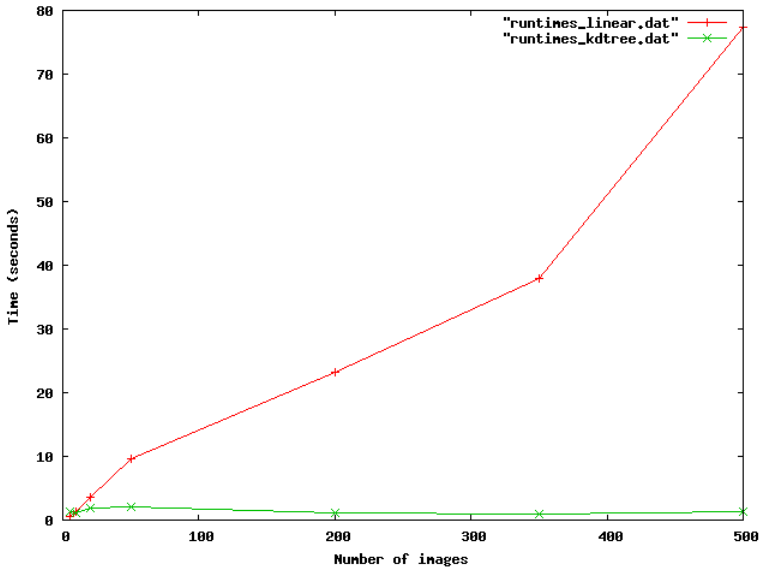

Figure 5.1: Runtime of the two algorithms

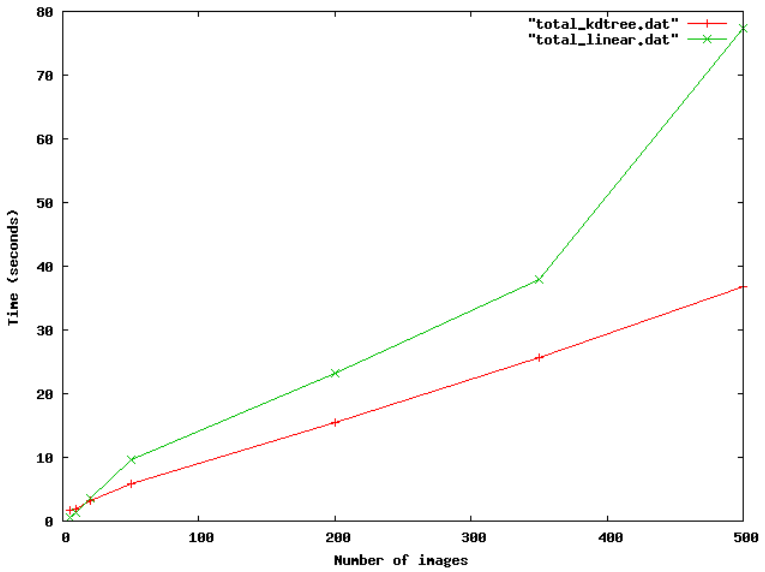

Figure 5.2: Total query time for *linear algorithm* and *kdtree algorithm*

The second set of tests have been concentrated on analyzing the behavior of large scale KD-trees. As stated above, our main concern was the initialization time of a KD-tree, which is divided into two steps: the computation of the descriptors for the images, and the construction of the actual KD-tree.
In Figure 5.3 we can see the total initialization time for a KD-tree, and the time needed only for the construction of the KD-tree data structure (presuming that the descriptors are already computed).

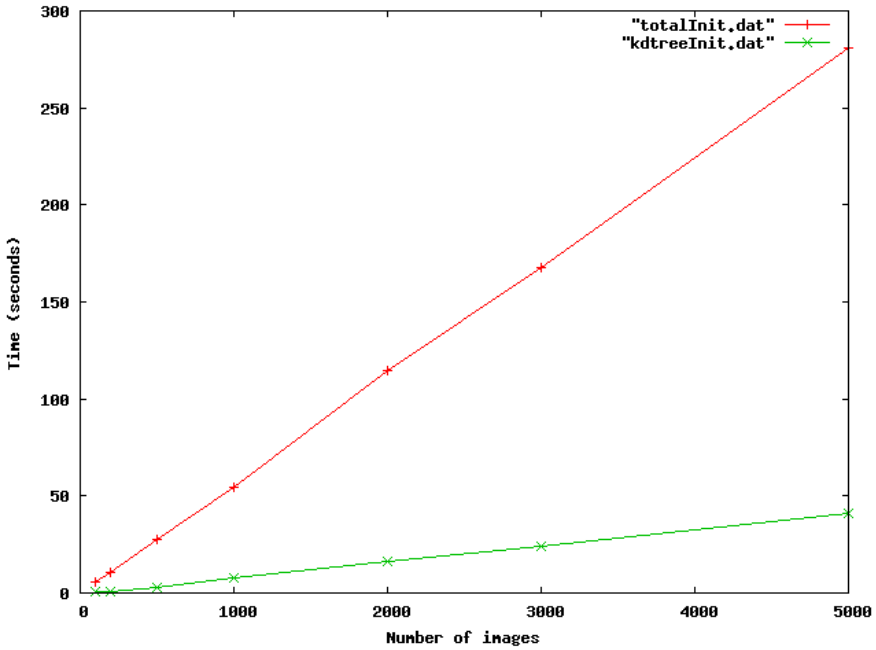

Figure 5.3: Initialization runtimes

number of images	total init (seconds)	kdtree init (seconds)
100	5.34	0.48
200	10.53	0.99
500	27.71	3.17
1000	54.52	7.72
2000	114.27	16.26
3000	167.83	24.31
5000	280.67	41.20

Because the total number of images is divided into KD-trees of fixed dimension (in our case 5000 images), inserting a new image into our database is constant, because it just implies creating a new KD-tree (or expanding a current one, if its dimension doesn't exceed 5000 images).

Chapter 6

Conclusions

In this paper I have designed a scalable algorithm that is capable of detecting a highly similar images in a large database.
I have succeeded in obtaining a small response time for querying the database in finding a highly similar image with a given one, maintaining at the same time a good quality of the responses.

6.1 Further Work

This algorithm can be further extended for handling a dynamic database, in which operations as adding and removing an image can be present along with the query operations previously described.
Also, the structure and size of the KD-tree data structure can be further analyzed, in order to determine the moment in which new data should be used to create a new KD-tree.
The management and distribution of the queries between the FEP and the image server should be considered as a possible improvement, possibly establishing several filtering layers instead of a single one (as described in Architecture).