University “POLITEHNICA” of Bucharest

Automatic Control and Computers Faculty,
Computer Science and Engineering Department

BACHELOR THESIS

A Scalable Algorithm for Similar Image Detection

Scientific Advisers:

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

Author:

Andrei-Bogdan Pârvu

First of all, I would like to thank Professor Ţăpuş for his support throughout the years since we have met and for his assistance and advice in writing this thesis. Also, I would like to thank my colleagues at Adobe Systems Romania, especially Ştefan Crăciun and Virgil Palanciuc, who have provided continuous feedback and suggestions, and have stood beside me during my work.

Last but not least, I would like to thank all the professors, teaching assistants and colleagues with whom I have interacted during my four years at POLITEHINCA University and who have influenced me in a positive manner over the period of time that I was a student.

Abstract

In this paper, we propose an algorithm for detecting similar images in a large scale dataset. After considering various methods as invisible watermarking and image feature detection, we developed a method based on the *Harris corner detection* and the *Scale Invariant Feature Transform* keypoints and descriptors for analyzing the properties of an image.
Our goal is that the proposed algorithm be resilient to watermarked, scaled or cropped images, and provide a fast running time for our large dataset.
In order to be able to handle a large amount of images and associated data, we have decided to maintain the descriptors of the images in a set of KD-tree structures for fast querying. This allows an initial filtering of the data set, reducing the number of images which should be analyzed. Thus, we 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.
We will describe the mathematical aspects of the two mentioned algorithms, and 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. We will detail this architecture and the way the data should be distributed so that an efficient use of multiple machines can speed up the algorithm.

Contents

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.
Copyright infringement and image detection have also been big issues which have been discussed and studied 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 be at the same time accurate, fast and scalable.

1.1 Requirements

Our goal was to design a scalable algorithm which can index information about 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, our 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.
We 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* [6], [7] 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.

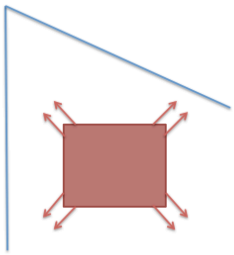

Figure 2.1: Flat Area

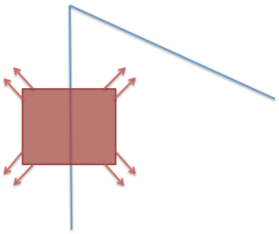

Figure 2.2: Edge Area

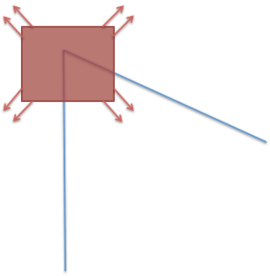

Figure 2.3: Corner Area

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, David Lowe, in [1], 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, we have to compute a unique fingerprint for a given keypoint, which should be invariant to scaling, rotation and luminance [5].
A 16×16 window is selected around a keypoint, which is divided into 16 4×4 blocks. In each of these blocks, the gradient magnitude and orientation is computed for each of the 16 elements. These gradients are then inserted in a 8 bin histogram. The histogram is computed using the

following principles:

1. the gradients are placed in the corresponding bin after their orientation: a gradient in range $0^o - 44^o$ is placed in the first bin, a gradient in range $45^o - 89^o$ is placed in the second, etc.
2. the value added to the corresponding bin depends on the gradient magnitude
3. also, the amount added to the corresponding bin depends on the distance from the keypoint. This is done using a gaussian weighting function
4. to achieve rotation independence we have to subtract the keypoint's orientation of the orientation of the histograms.

After this computation, we remain with 16 8 bin histograms, so we get a total of 128 numbers. 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 neighborhood. Thus, a keypoint descriptor will contain 128 values. These values are then normalized, and the descriptor of the current keypoint is obtained.

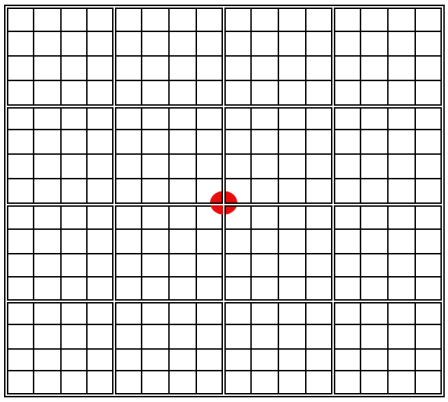

Figure 2.4: 16×16 window surrounding a keypoint

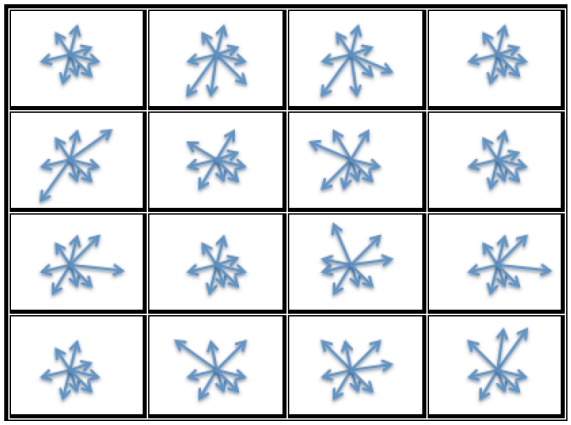

Figure 2.5: Orientation histograms for the 4×4 window

2.3 Speeded Up Robust Features

SURF (Speeded Up Robust Features) is an algorithm, which was introduced in 2006 as a speeded-up version of SIFT [2]. Insead of approximating the Laplacian of Gaussian with Difference of Gaussian, as SURF does, It approximates it with the Box Filter.
It uses wavelet responses [8], [9] on the X and Y axes to determine orientation assignment, which are plotted in a 2D space and summed up in a sliding window of 60^o.
The wavelet responses are used again to compute the descriptors for the keypoints: a 4-dimensional vector $(\sum d_x, \sum d_y, \sum |d_x|, \sum |d_y|)$ is computed for each block of size 4x4, thus obtaining a final descriptor of size 64.
We were interested in this descriptor, because of its similar performance with the SIFT descriptor, but lower dimension and faster computation.

2.4 Descriptor Compression

In [3], a method is proposed for compressing SIFT and SURF descriptors, reducing their dimension and creating a metric which could be used to compute the distance between two compressed descriptors without requiring their decompression.
This method has four parts:

1. a matrix which normalizes each row of a certain descriptor
2. a distance lookup matrix
3. a weight vector, which assigns each row of a descriptor a certain weight depending on how much it will contribute to the final distance
4. a tree coding scheme, similar to Huffman trees, which compresses the descriptors

This is particular interesting for us because it can refuse the dimensionality of our problem, and provide a similar query response with a smaller running time.

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.

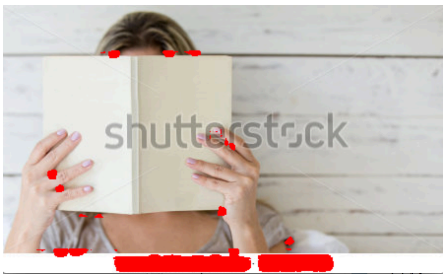

Figure 3.3: Single image

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 with 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.

3.1.3 Using SURF keypoints and descriptors

As stated in Related Work, SURF descriptors have a similar performance as the SIFT ones, but are faster and have a lower dimensionality, so we might want to do a comparison between the two. In Figure **??** and Figure 3.8 we can see an example of the SURF keypoints applied on the above image, and the corner-filtered SURF keypoints.

Figure 3.7: SURF keypoints for the entire image

Figure 3.8: SURF keypoints with corner mask

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 3.9 shows the corresponding matches between two images with two different watermarks, one of which is rotated 90^o clockwise.

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 (i.e more than 500). 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 the best possible candidates in terms of visual similarity.
The filtering algorithm is implemented as follows: we will maintain a maximum number of M

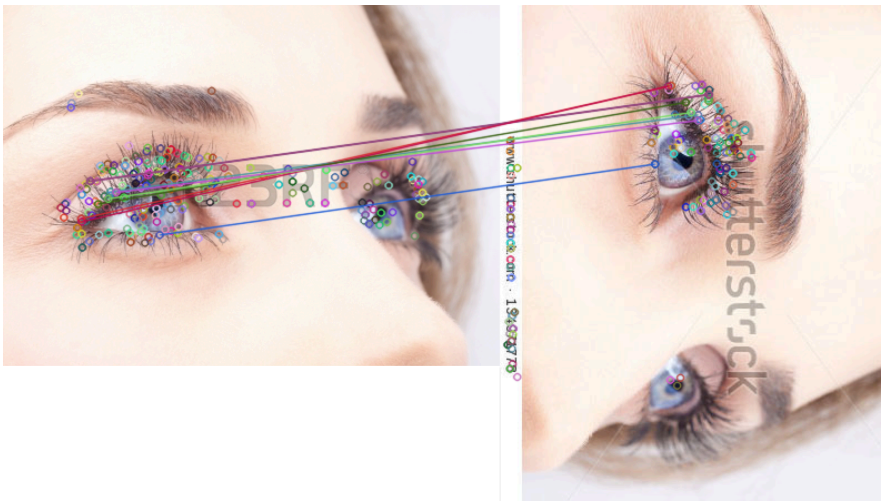

Figure 3.9: Comparison between two images

descriptors for each image in the initial set and create P KD-trees 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 K nearest-neighbor search on each of our KD-trees. Then we will select the top T images returned from the nearest neighbor searches and perform the *linear algorithm*. We shall name this algorithm the *kdtree algorithm*.
Of course, the important factors in the *kdtree algorithm* are:

- the values of the numbers M, T and K from the above description
- the metric used in selecting the top T images from the nearest-neighbor search
- the metric used in selecting the M descriptors from the input images to form the KD-tree
- N, the number of images that form a KD-tree
- P, the number of KD-trees, which can be determined from the total number of images and N.

We propose three metrics in for selecting the filtered images from the KD-tree based on the nearest-neighbor search:

- the images with the largest number of descriptors returned from the search
- the images with the smallest average of the distance of the found descriptors
- the images with the largest distance between its descriptors and the average of all found descriptors (from all the images)

For the initial attempts, we have set the values of M, T and K to 40, 10 and 5 and used the largest number of found descriptors as metric for selecting the filtered images from the KD-tree.

Chapter 4

Architecture

4.1 Basic structure

To illustrate the functionality of the described algorithm, we 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 Map Reducer from the Load Balancer. First of all, it allows load balancing: the Load Balancer can maintain a queue of requests and the Map Reducer 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).

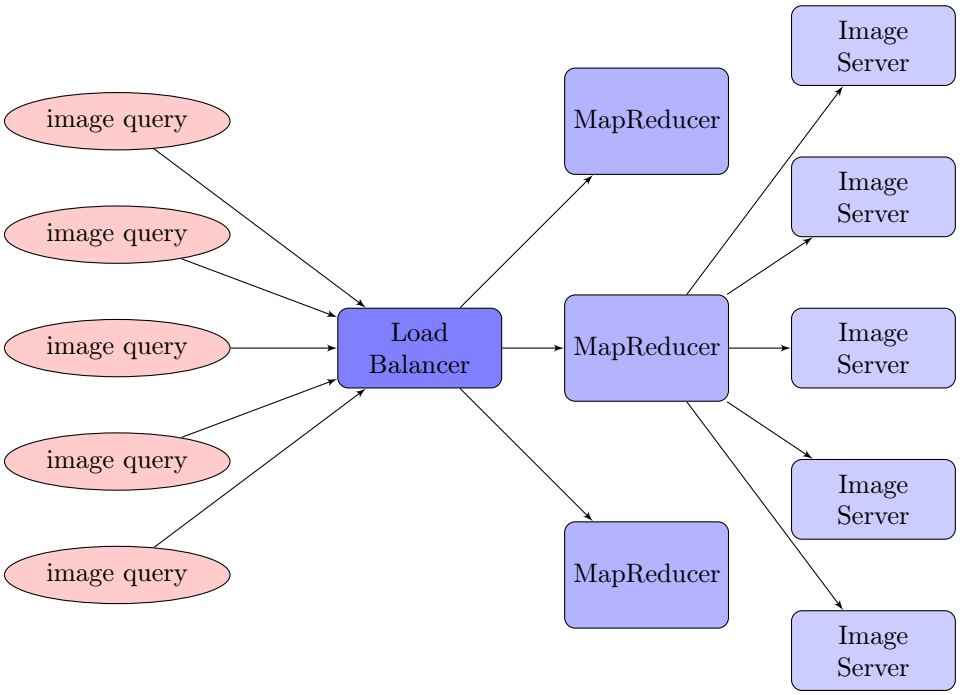

Figure 4.1: Basic structure of service

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.

```
1 while (true):
2   msg = receiveMessage()
3   if msg is a Query:
4         send to next Map Reducer
5   else if msg is response from Map Reducer:
6         send response back to querying entity
```

Listing 4.1: Load Balancer pseudocode

The command to run the Load Balancer is:

```
1 make run_load_balancer PORT_FRONT=10000 PORT_BACK=10001
```

where PORT_FRONT is the port on which the queries are received and PORT_BACK is the port on which they are passed along to the Map Reducers.

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.

```
while (true):
  query = receiveQueryFromLoadBalancer()

  for server in imageServers:
    send(query, server)

  for server in imageServers:
    response = receiveResponse(server)
    allResponses.append(response)

  sendToLoadBalancer(top T images from allResponses)
```

Listing 4.2: Map Reducer pseudocode

The command to run the Map Reducer is:

```
make run map_reducer PORT_RECV=10001 PORT_PUB=11000 PORT_RSP=11001
    CLIENTS=5
```

where PORT_RECV is the same as PORT_BACK from the Load Balancer, PORT_PUB is the port on which the query is transmitted to all Image Servers and PORT_RSP is the port used to receive the responses from the Image Servers. CLIENTS indicated the number of Image Servers connected to the Map Reducer and, implicitly, how many responses must the Map Reducer wait for after publishing a query.

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.

```
def doLinearProcessing(queryImage, imageList):
  for image in imageList:
    allScores.append(computePairImageScore(queryImage, image))

    return allScores.sort()
```

Listing 4.3: Linear Image Server pseudocode

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.

```
1 def doKDTreeSearch(queryImage, kdTree):
2   filteredImages = kdTree.findNearestNeighbors(queryImage)
3
4   return doLinearProcessing(queryImage, filteredImages)
```

Listing 4.4: KD-Tree Image Server pseudocode

The main problem when storing the descriptors in the KD-tree is in what form to have the images during the computation. We have three possibilities:

- keep them in the original size
- resize all the images to a fixed dimension (eg. 400×300)
- resize all the images to a fixed width and scale the height

Multi-threaded server

In order to support a large number of queries on our service, we have implemented the KD-tree Server as a multi-threaded one, each thread being able to process multiple queries. The computation of the KD-tree remains on the main thread and, after it is finished, several threads are created with the purpose of responding to queries from the Map Reducers.
As we mentioned before, the Map Reducer is able to process only one query at a time - it receives a query from the Load Balancer, submits it to the Image Servers, and then waits for the responses from these servers. So, to take advantage of the multi-threaded implementation of the Image Servers, we have to create multiple Map Reducers, each of them connected to a certain thread in the Image Servers and able to submit and process queries.
Figure 4.2 shows how to Image Servers with 3 threads interact with 3 Map Reducers.

4.4.3 Multiple KD-trees

Because of the relative constant time of querying a KD-tree, correlated with the fact that this query is an heuristic one (so the quality of the response may get smaller as the size of the KD-tree grows), we have opted to maintain multiple KD-trees in one KD-tree Image Server. We will use the variable name R to denote the number of KD-trees per Image Server
These KD-trees can be searched in a linear fashion, one at a time, or in parallel, with multiple threads (using OpenMP, for example).

```
1 def doMultipleKDTreeSearch(queryImage, kdTrees):
2         for current_kdTree in kdTrees:
3                 filteredImages.append(current_kdTree.
                      findNearestNeighbors(queryImage))
4
5         return doLinearProcessing(queryImage, filteredImages)
```

Listing 4.5: Multiple KD-Tree Image Server pseudocode

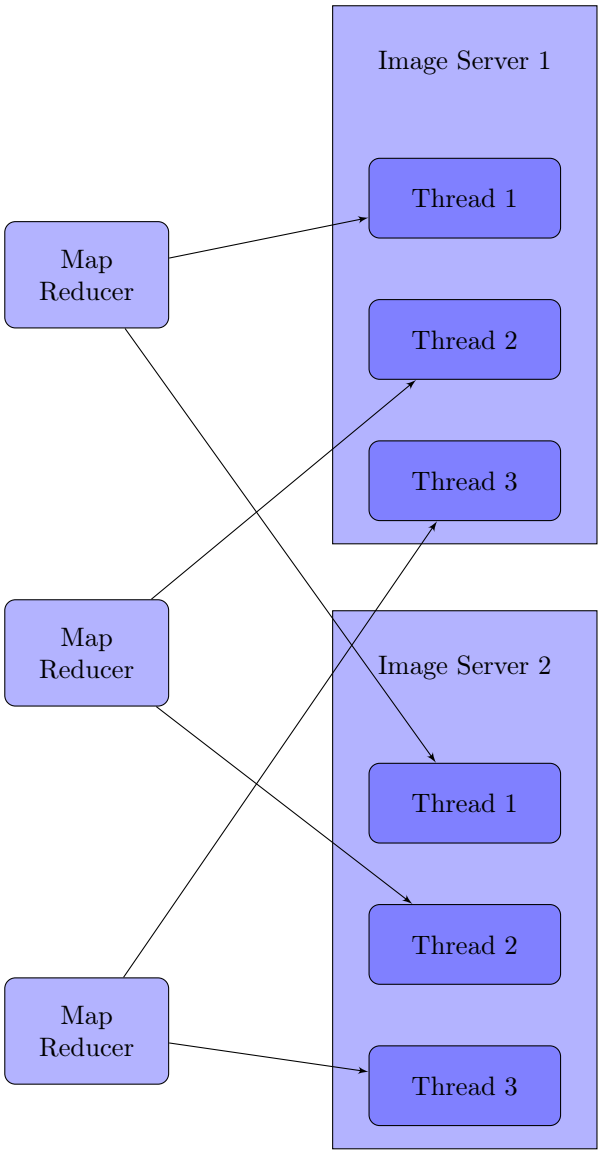

Figure 4.2: Image Servers with 3 threads and associated Map Reducers

4.4.4 Running the Image Server

The command used to run an Image Server is:

```
1 make run_images KEY=data_urls START=0 END=25000 TYPE=linear|kdtree
    IMAGES_KDTREE=2500 NUM_THREADS=1 PORT_FILE=port_file.json COMP=
    yes|no
```

The parameters are:

- KEY: the key in the Redis database which stores the urls to be loaded
- START, END: the interval of urls from the KEY to be loaded
- TYPE: this can be either *linear* or *kdtree* depending on the Image Server type

- IMAGES_KDTREE: the number of images per KD-tree: the total number of KD-trees per Image Server is computed as the total number of images (END - START + 1) divided by IMAGES_KDTREE
- NUM_THREADS: the number of threads per Image Server
- PORT_FILE: a JSON file containing the ports used by each thread for communicating with the associated Map Reducer. An example of port file is listed below, with each two consecutive ports representing a communication channel between a thread and a Map Reducer:

```
1   ["11000", "11001", "11100", "11101", "11200", "11201"]
```

Listing 4.6: port_file.json

- COMP: this can be either *yes* or *no*, depending if we want to compute the descriptors for the current images, or if we want to load them from the database

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 [10].

4.5.1 Image Storing

At first, we stored all our images in a directory on the hard disk, and we maintained a file with all the names of the images we wanted to load in our Image Servers. This started to be a problem when the number of images started to grow, because keeping all the images in a single directory increased the access time to the respective files (finding, opening and reading from it).

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++ [15].
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 [11] 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 [12] looked liked the best option, because of their robustness, small running time, auto-handling of connection failures and multiple types.
The ZeroMQ sockets that we used in our service are:

- ROUTER-DEALER. We used this pair of sockets to implement the Load Balancer. The ROUTER socket can accept multiple connections, add a header to the message, to remember which of the connected clients sent it and forwards it to the DEALER.

The DEALER receives the messages from the ROUTER and distributes them in a round-robin fashion to all the connected clients (which in fact are the Map Reducers). When a reply comes to the DEALER, it forwards it to the ROUTER, which, by inspecting the message's header, identifies the original client and sends it back.

- REP. The Map Reducer maintains this type of socket in its communication with the Load Balancer. It allows consecutive operations of receive and send, which suits very well our "receive query - send answer" protocol
- PUB-SUB. This pair of sockets forms the connection between a Map Reducer and its associated Image Servers. The PUB (publish) socket sends it message to all SUB (subscribed) sockets, making it ideal for the process of map reducing.
- PULL-PUSH. This pair of sockets implements the response communication between the Image Server and the Map Reducer. The PUSH (Image Server) sends its response, and the PULL (Map Reducer) receives it.

4.5.3 Submitting a query and image retrieval

For submitting a query, we use a JSON-encoded string, which is sent via socket communication to the LoadBalancer, which then forwards it in the above mentioned way. An example of a JSON query string is given below:

```
1 {
2   "name" : "http://webpa-mp:8080/image/98070.jpg",
3   "num_responses" : 10,
4   "type_experiment" : 0
5 }
```

Listing 4.7: Query JSON

The *name* key represents a url with an associated image, the *num_responses* key represents the number of similar images which we want to be returned, and the *type_experiment* key represents one of the 3 metrics used (described in Design of the Algorithm) to be used for extracting the images from the KD-tree.

A JSON response string is shown below:

```
1 {
2   "name": "http://webpa-mp:8080/image/141579.jpg http://webpa-mp
       :8080/image/223221.jpg http://webpa-mp:8080/image/116303.jpg
       http://webpa-mp:8080/image/265219.jpg http://webpa-mp:8080/
       image/216601.jpg http://webpa-mp:8080/image/100891.jpg",
3   "scores": "0 18 22 51 79 200 ",
4   "time": 1.25,
5   "num_responses": 10,
6   "type_experiment": 0
7 }
```

Listing 4.8: Response JSON

The *name* filed now contains the list of similar images, *scores* the associated *pair image scores* and *time* the time of the query on the KD-trees The coding and decoding of JSON arrays is done using a special C++ library [14].

4.5.4 Web Server

For the purpose of a web server, which we need for the submission of the queries and local image retrieval, we have used WebPY [13]. This server hosts our local images - which can be retrieved by accessing a certain URL, and also is responsable for receiving the queries, forwarding them to the Load Balancer and then generating the HTML for the result page.

4.5.5 User interface

For testing purposes, we have implemented a Google Chrome extension [16] which allows a user to select an image from his browser, right click on it, and search our database for similar images. An example of such a search is shown below, where we want to find similar images of the pirate-skull image from the top-right corner:

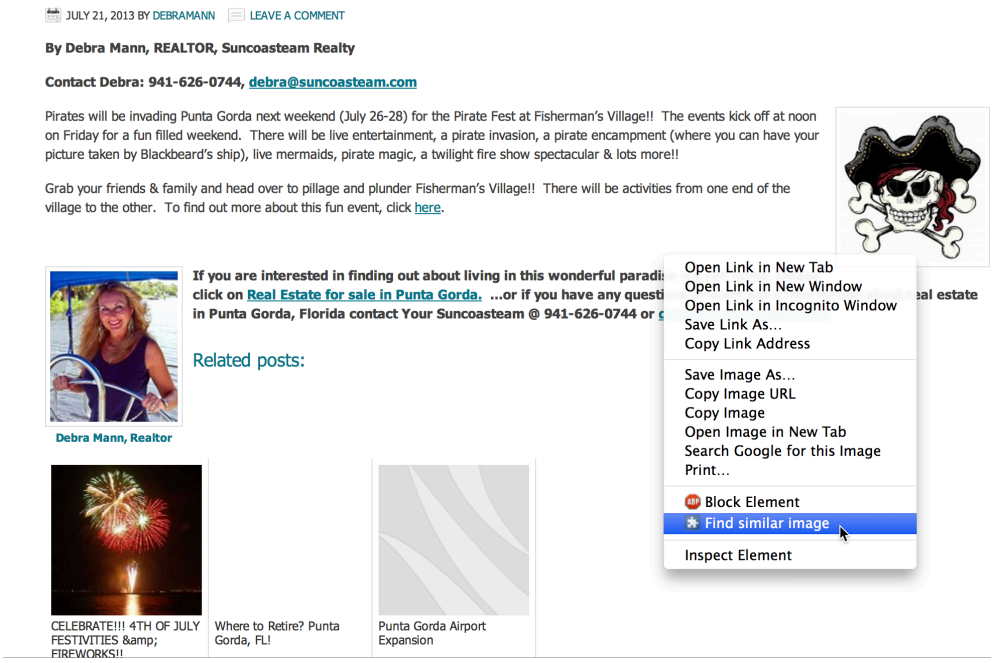

Figure 4.3: Using the Google Chrome extension

The results from the image search can be seen below, where we have the top 3 most similar images from our database, along with the total query time and the associated *pair image scores*. We have considered a score of less than 50 represents a highly similar image (marked by a green color), a score less than 100 a possible similar image (marked by an orange color) and a score higher than 100 an improbable match (marked with red).

The overall architecture of the entire system is shown in the Figure 4.5. The general workflow of the application is:

1. The querying instance (Google Chrome extension) sends the query to the web.py server
2. The server redirects the query to the Similar Image Service
3. The service gets the URLs and descriptors from the database
4. The service may attempt to download images from the web.py server
5. The server generates the result page, with the found images

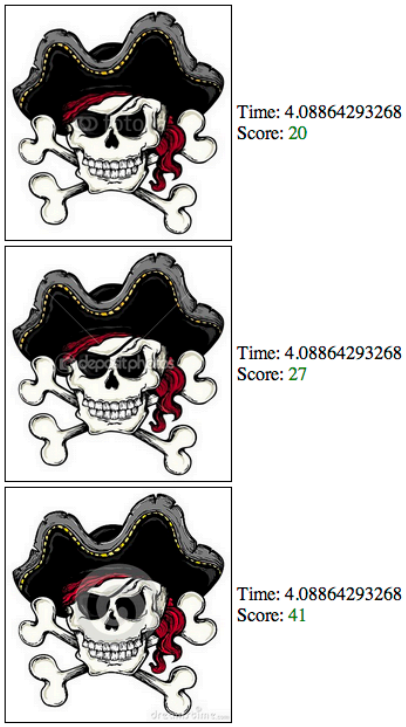

Figure 4.4: Google Chrome extension result

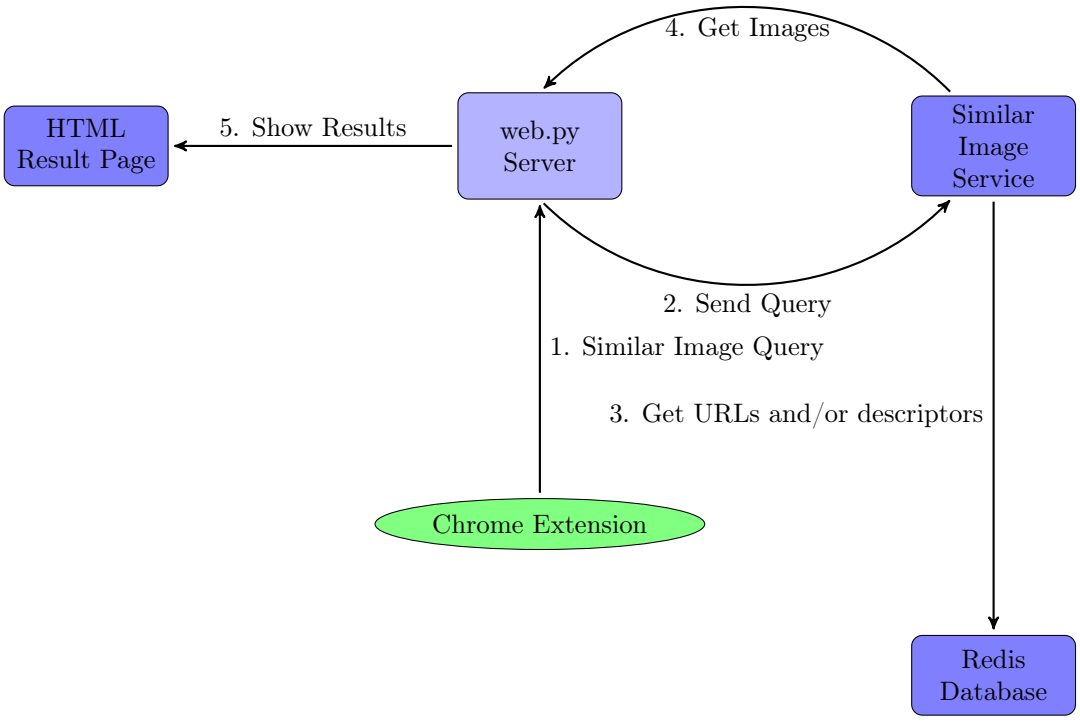

Figure 4.5: System architecture

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 its 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 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 in Design of the Algorithm 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 descriptors are computed for images which are resized to keep the aspect ratio (the third storing possibility described in Architecture).

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 Design of the Algorithm, 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	2.85	2.63	2.86
4	3.85	3.55	3.85
5	4.65	4.13	4.64

Table 5.1: Average number of found images for $P = 20$ and $N = 5000$

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	4.86	6.85	4.78
4	7.87	10.24	7.85
5	13.47	17.48	13.35

Table 5.2: *Index scores* for $P = 20$ and $N = 5000$

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.

5.1.1 Fixed dimension image storing

We computed the descriptors for images resized to a fixed value (height 400 and width 300), and analyzed the same metrics and scores described above. The results can be seen in tables 5.3 and 5.4. These results confirm that this image storing performs worse than the aspect-ratio one, so we have decided to continue using the first one.

	max nr descs	min avg	max dist to avg
3	2.69	2.08	2.73
4	3.55	2.60	3.62
5	4.17	2.80	4.32

Table 5.3: Average number of found images for $P = 20$, $N = 5000$ and fix dimension storing

	max nr descs	min avg	max dist to avg
3	6.32	11.93	5.97
4	10.37	18.46	9.84
5	16.8	28.35	15.67

Table 5.4: *Index scores* for $P = 20$, $N = 5000$ and fix dimension storing

5.1.2 Varying the number of images per KD-tree

We also wanted to see how the number of images within a KD-tree influences the above metrics, so we separated the 100.000 images into $P = 40$ KD-trees, each of then storing 2500 images. We applied the same correctness test described above, the average number of found images and the *index score* is shown in the tables below.

	max nr descs	min avg	max dist to avg
3	2.87	2.64	2.88
4	3.82	3.55	3.84
5	4.65	4.13	4.66

Table 5.5: Average number of found images for $P = 40$ and $N = 2500$

	max nr descs	min avg	max dist to avg
3	4.83	6.80	4.81
4	8.20	10.41	8.04
5	13.4	17.32	13.4

Table 5.6: *Index scores* for $P = 40$ and $N = 2500$

As it can be seen, a smaller number of images per KD-tree provides a higher quality response, which is what we expected, due to the high multi-dimensionality of our problem (a descriptor is composed out of 128 numbers). For the overall algorithm, we will have to determine the optimum number of images per KD-tree, in order to strike a balance between the total number of processes and the quality of the response.

5.1.3 Single KD-tree analysis

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 Design of the Algorithm 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 Table 5.7:

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

Table 5.7: *Correctness scores* for KD-tree with $N = 4500$

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.

5.1.4 Multiple KD-trees in an Image Server

As stated in Architecture, we decided to maintain multiple KD-trees in one Image Server in order to improve the quality of the heuristic search on one such KD-tree and reduce the total number of processes.
On the same set of 100.000 images, we created $P = 40$ Image Servers, with each server having 5000 images and $R = 2$ KD-trees (that means $N = 2500$ images per KD-tree), so that we could compare the *correctness scores* between this implementation and the one with $R = 1$.
These are shown in the tables below:

	max nr descs	min avg	max dist to avg
3	2.86	2.5	2.87
4	3.83	3.35	3.85
5	4.6	4.75	4.62

Table 5.8: Average number of found images for $P = 40$, $N = 2500$ and $R = 2$

	max nr descs	min avg	max dist to avg
3	4.76	8.03	4.70
4	7.96	11.92	7.82
5	13.56	20.35	13.49

Table 5.9: *Index scores* for $P = 40$, $N = 2500$ and $R = 2$

It can be seen that the scores are comparable with the ones obtained from running the algorithm it the two cases shown in the previous subsections, so it can be confirmed that using multiple KD-trees of a fixed (and lesser size) is in our advantage when the overall dataset becomes larger, in order to avoid a large number of Image Servers.

5.1.5 Internet Search

Besides running the algorithm on our similarity sets, we also took some images from our 100.000 set and used Google Image Search or TinEye to find replicas of those images on the Internet. Then we did a reverse search of those images with our algorithm to see if it finds the original images used in the query.
We had some interesting results, shown in Figure 5.1 and Figure 5.3.

Figure 5.1: Image from website

Figure 5.2: Image from database

Figure 5.3: Image from website

Figure 5.4: Image from database

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.

5.2.1 Comparison between *kdtree algorithm* and *linear algorithm*

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 5.5.
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 nearest-neighbor 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 5.6 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.

5.2.2 Large scale KD-trees

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 5.7 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

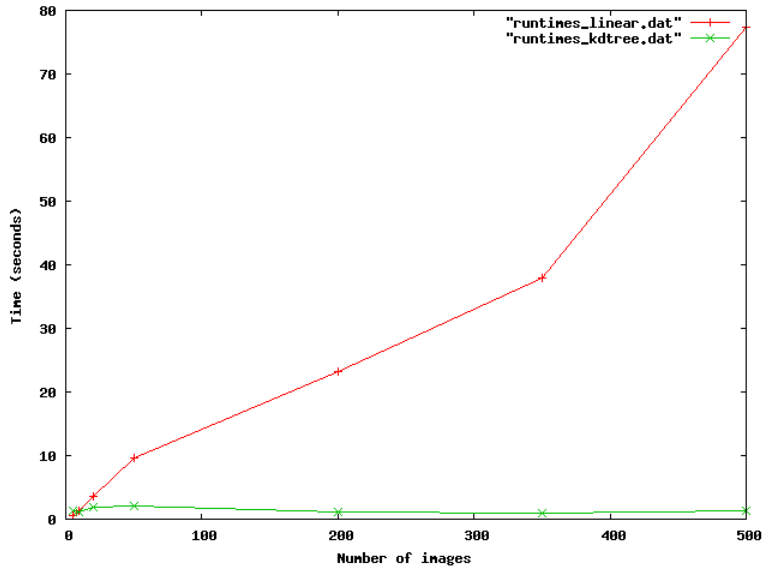

Figure 5.5: Query runtime of the two algorithms

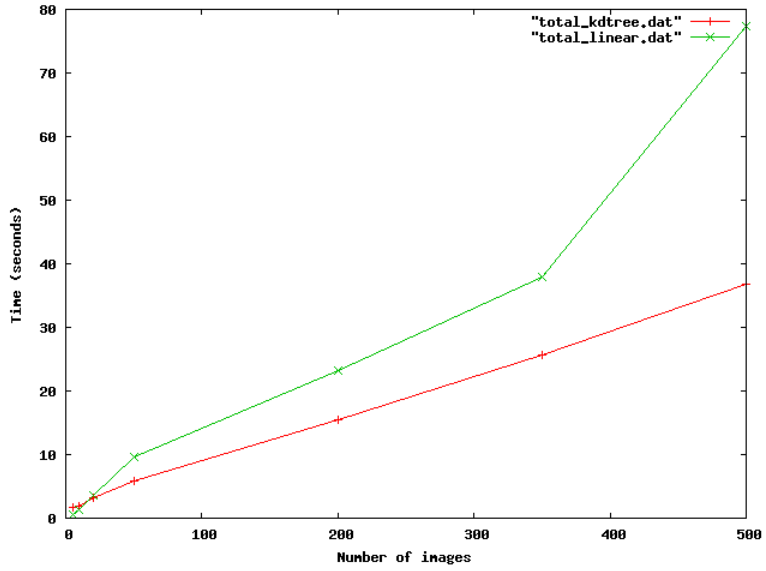

Figure 5.6: Total init and query time for *linear algorithm* and *kdtree algorithm*

computed).

number of images	*linear algorithm* time	*kdtree algorithm* time
5	0.60	1.391
10	1.23	1.21
20	3.54	1.90
50	9.58	2.08
200	23.28	1.05
350	38.00	0.96
500	77.33	1.39

Table 5.10: Query runtime

number of images	*kdtree algorithm* time
5	1.68
10	1.81
20	3.24
50	5.80
200	15.51
350	25.58
500	36.77

Table 5.11: Init and query runtime

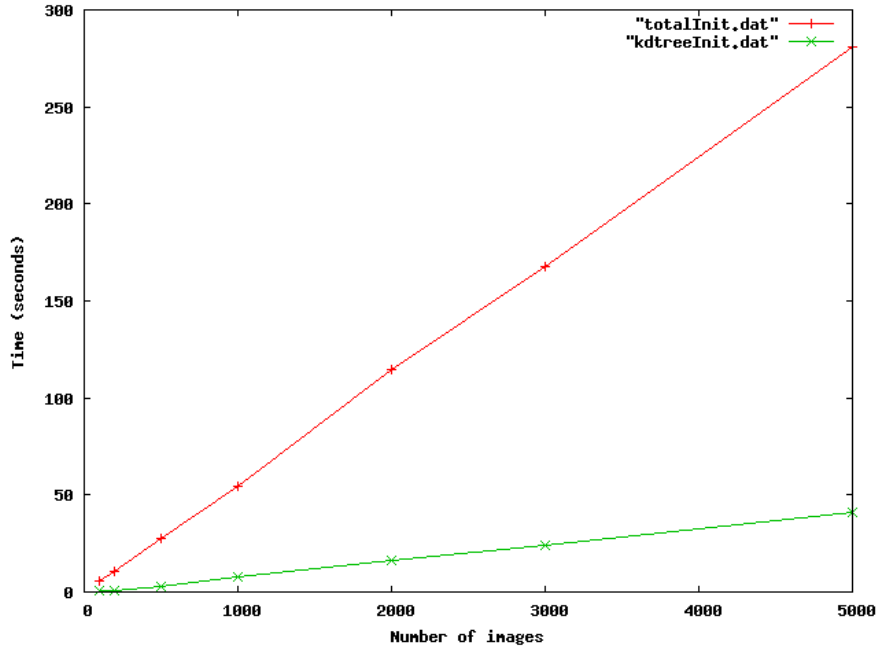

Figure 5.7: Initialization runtimes

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

Table 5.12: Descriptor computation and KD-tree init time

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).

5.2.3 Multiple KD-tree servers

As shown above, the running time of a nearest-neighbor search is constant, due to its heuristic nature, the overall query time varies depending on the number of processes involved in the search. This is due to the increasing socket communication between the Map Reducer and the growing number of processes. We illustrate this overall running time in Figure 5.8, where each process holds a KD-tree of 1000 images (a higher number of images would not have influenced the process). For each number of processes 20 queries have been run and the mean execution time of them is listed in table Execution time for increasing number of KD-tree servers, as well as in the figure.

number of KD-tree servers	query time (seconds)
5	2.85
10	3.24
20	3.53
40	5.50
600	6.38
800	8.23
100	8.74

Table 5.13: Execution time for increasing number of KD-tree servers

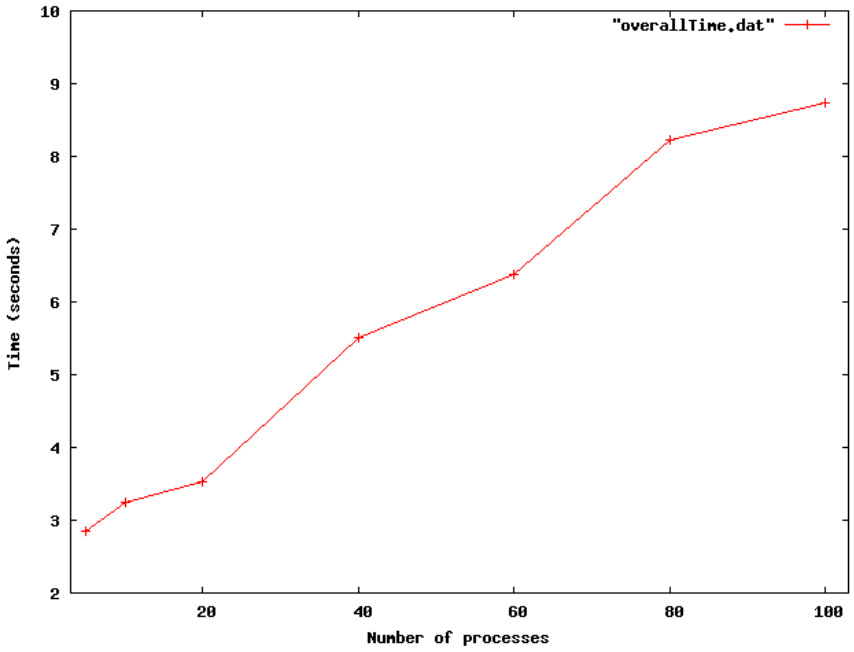

Figure 5.8: Overall query times

5.2.4 Multi-threaded servers

As stated in Architecture, in order to be able to handle a large number of queries at the same time, the KD-tree servers are multi-threaded, with each separate thread being connected to a different Map Reducer and, thus, being able to support multiple queries simultaneously.
Table 5.14 shows the running time for a number of 10 KD-tree servers, each with 2500 images and respective number of threads and queries.
It can be easily observed that there is a big running time improvement when the number of queries gets bigger and the number of threads increases. In the case of a smaller number of queries, an bigger number of threads does not improve the running time, partially because of the overhead caused by the creation of multiple threads and the overhead of multiple Map Reducers and synchronization between them.
Also, because a service with a large number of images will have a considerable amount of Image Servers running (and, implicitly, number of processes), creating more threads will certainly increase the load of the system and not have the desired effect.
Inside an Image Server, the KD-tree structure is shared between processes, so, during multiple queries done by multiple threads, the nearest neighbor search performed on the KD-tree may be a computational bottle neck, all the threads accessing the same data structure. To avoid this, in case we use multiple KD-trees inside the same Image Server, it could be a good idea to perform the nearest neighbor search in a random order, thus time-multiplexing the access to the data structures.

number of threads \ number of queries	20	50	100	200
1	74.13	176.28	-	-
2	38.28	94.77	-	-
3	28.15	77.51	-	-
5	23.81	57.04	95.90	217.64
10	-	-	91.43	183.12
20	-	-	90.23	180.20

Table 5.14: Running time (seconds) for a given number of processes and queries

5.2.5 Multiple KD-trees per Image Server

We analyzed how a certain KD-tree Image Server with a varying number of linear processed KD-trees is performs when submitting a query, in order to determine a viable value for the R parameter (the number of KD-trees that form a KD-tree Image Server).
In Figure **??** and in Table 5.15 we can observe these running times as we increase the number of KD-trees for with $N = 2500$ images.

R, number of KD-trees	running time (seconds)
2	2.45
3	2.75
5	2.84
10	3.01
20	3.44
40	4.41

Table 5.15: Running time with various number of KD-trees per KD-tree Image Server

By looking at these running times, we can determine that we can easily increase the number of KD-trees per Image Server, as long as we maintain a decent number of images as input for the *linear algorithm*, which is used after the actual KD-tree search.

5.3 Interpretation of results

By looking both at the running time and correctness of the algorithm, along with the varied parameters which compose it, we have determined the best way to run it when handling a large amount of images:

- store images maintaining aspect ratio
- store beforehand the computed image descriptors
- use the third metric for extracting images from KD-trees: "the images with the largest distance between its descriptors and the average of all found descriptors (from all the images)"
- keep a relatively small number of Image Servers
- keep a moderate number of threads per Image Server, especially if there isn't a large number of queries
- increase the number of KD-trees per Image Server

Chapter 6

Conclusions

In this paper we have developed a scalable algorithm for finding similar images in a large database, using the Harris corner transformation, the SIFT descriptors and multiple KD-trees for storing the image data.
We have designed a basic architecture for our service, consisting of a back-end side, formed by a Load Balancer, multiple Map Reducers and Image Servers, and a front-end side, represented by a Google Chrome extension.
We have explored several metrics for selecting images out of the KD-trees and have analyzed how these different metrics influence the images returned by a query.
We have analyzed the running time of the algorithm, varying the dimensions of the image database, the number of images stores inside a KD-tree, the number of KD-trees associated with a Image Server and the overall number of Image Serves. Also, we tested our service with a high number of queries and tried to reduce the overall running time by doing a parallel computation of the queries.
We tested the service on a large number of images and already computed similarity sets and considered various metrics for testing the correctness of the query responses.
In the end, we have succeeded in obtaining a service which implements a scalable algorithm that extracts similar images from a large dataset, providing good quality responses and a small running time. The main advantage of our algorithm is, beyond the small query time and quality of responses, the relatively reduced initialization time (computation of descriptors and construction of KD-trees). If using a neural network for object recognition, although the similarity would be rather a semantic one than a visual one, the training of the network would take a far longer time than our initialization process.

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. Because of the fixed dimensions of the KD-trees, this insertion and removal should take a constant amount of time with respect to the overall number of images from the database. 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.
Testing the algorithm with different kind of descriptors (such as SURF or GIST), or with a descriptor compression algorithm [3], can be a further research topic, in order to determine how they behave in our service, both as running time and as correctness.
This algorithm can also be used for automatic keywording. If all the images in our database have associated keywords, and we want to find possible keywords for a new image, we can use

our algorithm to take the keywords of similar images and suggest them as possible ones of the new image.
Not least, we should increase the overall number of images from our database and distribute the Google Chrome extension to various people, so that they can test it with various images found on the Internet. The feedback receives from the users can help us see the possible weak spots of our algorithm, try to fix them and improve the overall results.

Bibliography

[1] D.G. Lowe, *Distinctive Image Features from Scale-Invariant Keypoints*, 2004

[2] H Bay, T. Tuytelaars, L. Van Gool, *SURF: Speeded Up Robust Features*, 2006

[3] M. Johnson, *Generalized Descriptor Compression for Storage and Matching*, 2005

[4] O. Chum, J. Philbin, M. Isard, A. Zisserman, *Scalable Near Identical Image and Shot Detection*, 2008

[5] A. Vedaldi, *An implementation of SIFT detector and descriptor*, 2006

[6] M Trajkovi, M Hedley, *Fast Corner Detection*, 1998

[7] C Harris, M Stephens, *A combined corner and edge detector*, 1988

[8] P. Porwik, A. Lisowska, *The HaarWavelet Transform in Digital Image Processing: Its Status and Achievements*, 2004

[9] E. J. Stollnitz, T. D. DeRose, D. H. Salesin, *Wavelets for Computer Graphics*

[10] Open Source Computer Vision Library, *http://opencv.org/*

[11] Redis Data Structure Server, *http://redis.io*

[12] Zero Message Queues, *http://zeromq.org/*

[13] Web.PY Server, *http://webpy.org/*

[14] JsonCpp, *http://jsoncpp.sourceforge.net/*

[15] Lib CURL, *http://curl.haxx.se/libcurl/*

[16] Google Chrome Extensions, *https://developer.chrome.com/extensions*