Speeding up IDM without degradation of retrieval quality

Michael Springmann, Heiko Schuldt
University of Basel
{michael.springmann, heiko.schuldt}@unibas.ch

Abstract

The Image Distortion Model (IDM) has shown good retrieval quality in previous runs of the medical automatic annotation task of previous ImageCLEF workshops. However, one of its limitations is computational complexity and the resulting long retrieval times. We applied several optimizations, in particular the use of an early termination strategy for the individual distance computations, the use of optimized data structures for the intermediate results, and the proper use of multithreading on state-of-the-art hardware.

With these extensions, we were able to perform the IDM P2DHMDM down to 1.5 second per query. Moreover, we extended the possible displacements to an area of 7x7 pixels, using a local context of either 5x5 or 7x7 pixels. We also introduced a classifier, that exploits the hierarchical structure of the IRMA code.

The results of the extendeded IDM P2DHMDM have been submitted to this year's medical automatic annotation task and achieved rank 19 to 25 out of 68. More importantly, the used techniques are not limited strictly to IDM but are also applicable to other expensive distance measures.

Categories and Subject Descriptors

H.3 [**Information Storage and Retrieval**]: H.3.1 Content Analysis and Indexing; H.3.3 Information Search and Retrieval; I.5 [**Pattern Recognition**]: I.5.4 Applications

General Terms

Algorithms, Measurement, Performance, Experimentation

Keywords

Content-Based Image Retrieval, Image Distortion Model, Medical Image Retrieval, Classification

1 Introduction

The Image Distortion Model (IDM) has shown very good retrieval quality in the Medical Automatic Annotation Task at ImageCLEF 2005 [6] and was still ranked in the top 10 results of ImageCLEF 2006 [12]. It is used as a local feature, this means, the individual values of the feature are bound to a pixel or an area of the image rather the image as a whole. IDM performs a pixel-by-pixel gray value comparison of downscaled versions of the image, in which some displacements of pixels are accepted. P2DHMDM extends IDM to consider not only a single pixel, but e.g. a 3x3 area for displacements – which improves the query results, but is at the same time associated with significantly higher complexity. In contrast to most of the successful techniques from the object recognition track that performed better in ImageCLF 2006, it is not strictly limited to

the area of classification, but applicable to a wider range of image retrieval problems. A limiting factor of IDM as a similarity measure is the computational complexity and, consequently, its long retrieval times. According to [15], the execution of a single query in a collection of 8'728 images took about 5 minutes on a standard Pentium PC with 2.4 GHz – which is clearly not even close to interactive response times. The authors therefore proposed to use a sieve function to reduce the number of expensive IDM computations to the nearest neighbors until a cutoff position $c = 500$ based on the less expensive Euclidean distance. By this, they reduced the time for a single query to 18.7 seconds, but this is only possible with some degradation in retrieval quality. Using such an approach therefore requires to find c that achieves a good tradeoff between speed an quality.

We propose an approach that increases the speed of IDM without any negative impact on the retrieval quality. In our experiments, we used the parameters for IDM as used in [9] with the Pseudo-2D Hidden Markov Distortion Model (P2DHMDM) [10], but apply an early termination condition in the individual distance computation. By this, we can reduce the execution time for a single query on similar hardware to 4h 28m for the full run of 1'000 queries, therefore about 16 seconds per query. On modern hardware which is better suited for multithreading, we could further reduce this time to 1h 11m on a Core 2 Duo with 2.4 GHz (4.2 seconds per query) and only 24 minutes on a 8-way Xeon MP with 2.8 GHz enabled (1.5 seconds per query).

The main improvement in our approach is based on the following idea: As a result of the query, we want to retrieve a small set of reference images, that are the k nearest neighbors of the query image within a potentially big collection. This set is passed to a kNN classifier to determine the class of the query image. For this, we need the rank and the distance of the k nearest neighbors. It is important to notice that ranks and distances of images are not needed, as long as we can guarantee, that they do not belong to the set of those k nearest neighbors. Therefore we can terminate any individual distance computation of the query image and a reference image as soon as a certain threshold is exceeded. This threshold can be derived out of the best k images that have been found so far. This means, that only for the first k images of the collection we have to compute the exact distance. Then we can derive the threshold and can apply it to subsequent distance computations. We start processing the next feature vector of an image value-by-value. As soon as we find out, that it must exceed the threshold, we abort the computation for this image. If we processed the feature vector until its end and the image did not exceed the threshold, it also must have a distance that is smaller than some of the k best images we found before. By inserting its distance in a ordered list of the distances we found, we can decrease the value of the threshold to the k-th value in this list. We can now continue the query with a smaller threshold and therefore might be able to abort the following computations even earlier. We reapply this technique until we processed all images of the collection. Such a technique has already been used in the implementation of [19], which is a part of ISIS [11, 3] Further improvements have been achieved by using specialized data structures and making excellent use of multi-threading.

For our runs in the ImageCLEF Medical Automatic Annotation Task, we extended the warp range of IDM to three pixels instead of only two and used a P2DHMDM with 5x5 and 7x7 pixel areas instead of only 3x3 as in [10]. When applied to the development data set, we have achieved better scores. Notice that extending the area results in significantly longer computation times. Therefore, the above mentioned optimizations we have applied are essential.

The remainder of this paper is organized as follows: Section 2 describes in depth the used distance measure and the chosen parameters. Section 3 describes the algorithmic optimizations, that achieved most of the performance improvements in the similarity computation. Section 4 adds more details about the implementation, that contributed in smaller performance improvements. Section 5 introduces the classifier that has been used to exploit to the hierarchical IRMA code. The results of all these modifications are presented in Section 6. Finally, Conclusion and outlook are given in Section 7.

2 The IDM Distance Measure and Parameters of the Submitted Runs

The implementation used in the submitted runs is based on [9]. In contrast to FIRE[1], the implementation of used for the winning run of the medical automatic annotation task in 2005 [7], which has been implemented in C++ and some Python for the UI, we used Java as the implementation language of choice. Due to the algorithmic nature of our proposed modifications, the implementation language has only little effect of the relative execution time of the unoptimized vs. the optimized solution.

2.1 Starting Point: IDM with P2DHMDM

The Image Distortion Model (IDM) is deformation model [10] which is used as a measure of distance or dissimilarity between two images. By using a k-Nearest Neighbor (kNN) classifier, it can be applied to the automatic annotation required in this task.

To determine the k nearest neighbors, each image of the collection needs to be compared to the query image using a distance function or distance measure. More formally, a distance function is a function that computes for $Q, R \in I$ with I being the set of all images and Q being the query image and R being one reference image out of the collection: $D : I \times I \rightarrow [0, \infty)$ The Euclidean distance is an example for an elementary distance measure that can be applied to two dimensional images.

$$D_{euclid}(Q,R) = \sqrt{\sum_{x=0}^{height} \sum_{y=0}^{width} (Q(x,y) - R(x,y))^2} \tag{1}$$

with *height* and *width* being the height and width of the query image Q. $R(x,y)$ and $Q(x,y)$ denote the gray value of the pixel in row x and column y. The k nearest neighbors are the k images who have lowest distance to the query image.

IDM then allows for some displacements of each individual pixel on both axes within the range of the warping distance w. Each displacement may get penalized with some costs that are associated with this displacement which are computed using the cost function C. Out of the possible displacements of a single pixel, always the one is chosen that results in the smallest distance.

$$D_{IDM}(Q,R) = \sqrt{\sum_{x=0}^{height} \sum_{y=0}^{width} \min_{\substack{x' \in [x-w, x+w], \\ y' \in [y-w, y+w]}} ((Q(x,y) - R(x',y'))^2 + C(x-x', y-y'))} \tag{2}$$

For means of clarity and generalization, we introduce the more generic model

$$D(Q,R,p) = \sqrt{\sum_{x=0}^{height} \sum_{y=0}^{width} p(Q,R,x,y)} \tag{3}$$

with

$$p_{euclid}(Q,R,x,y) = (Q(x,y) - R(x,y))^2 \tag{4}$$

and

$$p_{IDM}(Q,R,x,y) = \min_{\substack{x' \in [x-w, x+w], \\ y' \in [y-w, y+w]}} ((Q(x,y) - R(x',y'))^2 + C(x-x', y-y')) \tag{5}$$

as pixel distance computation function. Every pixel distance computation function evaluates the distance[2] of a single pixel (x,y) of the query image with its corresponding pixel in the reference image – and as for the distance function, the resulting values must be out of the interval $[0, \infty)$.

[1] FIRE - Flexible Image Retrieval Engine, `http://www-i6.informatik.rwth-aachen.de/~deselaers/fire.html`

[2] Actually, this does not return the distance value since the square root still needs to be computed.

Q	...	the query image
R	...	a reference image
I	...	set of all images
$Q(x, y)$	...	gray value of pixel in row x and column y
$height$	...	height of query image
$width$	...	width of query image
k	...	number of nearest neighbors for classification
w	...	warp range of IDM
$C(dr, dc)$	...	cost function of IDM for pixel displacement in row (dr) and column (dc)
h	...	HMM range of P2DHMDM
t	...	threshold for distance achieved by single pixel
c	...	cutoff position in sieve function
$d(k)$	...	distance achieved by k nearest neighbor
$\hat{d}(k)$	...	distance achieved by current k nearest neighbor candidate
$cl(l, i)$	...	class of i-th entry of list l
$dl(l, i)$	...	distance of i-th entry of list l

Figure 1: Overview of frequently used symbols

Our experiments showed that it can improve the retrieval quality if the distance that can be achieved by any single pixel is limited by a threshold t. We introduced an intermediate function p_{pt} , that can be applied on any pixel distance computation function:

$$p_{pt}(Q, R, x, y) = \begin{cases} p(Q, R, x, y) & \text{for } p(Q, R, x, y) < t^2 \\ t^2 & \text{for } p(Q, R, x, y) \geq t^2 \end{cases} \tag{6}$$

We use t^2 instead of t here in order to be in line with the square root that will be taken in Equation 3.

To improve IDM, we need a method that does not only take into account single pixels, but the local context which is defined by an area of pixels around the central pixel that differ in their row and column value by not more than the HMM range h. P2DHMDM computes the average distance between those pixels in the area with the corresponding pixels of the reference image.

$$p_{P2DHMDM}(Q, R, x, y) = \min_{\substack{x' \in [x-w, x+w], \\ y' \in [y-w, y+w]}} \frac{\sum_{\hat{x}=-h}^{h} \sum_{\hat{y}=-h}^{h} ((Q(x+\hat{x}, y+\hat{y}) - R(x'+\hat{x}, y'+\hat{y}))^2 + C(x-x', y-y'))}{(2h+1)^2} \tag{7}$$

The distortion model is commonly applied to grayscale images, which may have been scaled down to a common size that is sufficient for the task of finding nearest neighbors for classification. If images differ in size and/or aspect ratio, corresponding pixels need to be identified at the time of comparison. For pixels at the boundary, only deformations are allowed, where pixels remain inside the image boundary. For P2DHMDM the local context of such pixels consists only of the pixels inside the image. For instance at position $x = 0, y = 7$ with $h = 1$, the resulting 3x3 area is not entirely inside the image: Only six pixels can be inside the image since a row index of $x = -1$ is not allowed. In such a case, only the pixels inside the image will be considered and the average distance will get computed accordingly. The entire model may either be applied to the image directly or on the image after it has been processed by a Sobel filter to detect edges. It is also possible to treat both versions of the image as layers of the images and compute the pixel distance on either or both layers.

2.2 Modifications of Parameters

In [15], images have been scaled to a common height of 32 pixels while preserving the aspect ratio. This means, that images in portrait orientation contain significantly less pixels after scaling than images in landscape orientation. In order to reduce the effect of this with regard of the absolute value of the summed distance over all pixels as defined in Equation 3, we decided to scale the images to 32 pixels on the longer side.

The recommended parameters in [10] are: $w \in \{1, 2\}$, allowing a deformation in a warp range of 3x3 or 5x5 pixels and a HMM range $h = 1$, hence using another 3x3 pixel area for the P2DHMDM. For our runs, we used $w = 3$, therefore allowing displacements in an area of 7x7 pixels which showed better results on the development data set. Notice that we used only IDM with P2DHMDM, which is –even if allowing some deformations– still a local feature, whereas in other approaches might get combined with global features, like Tamura texture features and aspect ratio in [15]. Therefore the more relaxed displacement rule might be less important when combining IDM with features, that take into account the entire image already. However, also the winning run in 2005 was using IDM alone [7], so they also did not combine it with any global feature. We increased also the HMM range to $h = 2$ for most runs, our best run used $h = 3$. We applied a cost function with two different sets of parameters, out of which the one with higher costs and therefore higher penalty for displacements achieved slightly better results.

We used both layers, the one with the gray values of down scaled images directly and one using the Sobel-filtered image. An implementation detail here is, that for generating the second layer, we filtered the original image and scaled this down separately, rather than applying the edge detection on the very small image where one would expect to find more or less only hard edges since areas of pixels of same color are less likely to be found in small images. We did not weight both layers equally; on the training dataset and generated cross validation sets, giving the gray values twice the importance of the Sobel-filtered version achieved better results.

Finally, we used a kNN classifier that takes not only the class of the nearest neighbors into account, but also weighs the class information based on the distance computed using IDM, with $k \in \{3, 5\}$ and made further use knowledge about the IRMA code. In [10], $k = 1$ has been used. In our experiments, $k = 3$ performed best, 4 slightly worse. 1 was preferable over 5 if distances were ignored; when the distances were taken into account and the IRMA code was exploited, it outperformed $k = 1$ in most cases, but couldn't outperform $k = 3$ with or without exploiting IRMA code.

3 Algorithmic Optimization

In the following, we report on the algorithms that have been designed to implement the model introduced in Section 2 in an efficient way.

3.1 Maximum Sum

As a matter of fact, only the ordering and distance of the k nearest neighbors will be used in the classification step. Therefore the exact distance of any image with a rank $> k$ is unimportant and can safely be approximated. Since the rank of those images will not be important either, it is sufficient to approximate the distance to ∞. Let $d(r)$ be the exact distance of the image with rank r, then we can rewrite equation 3 to a thresholded function similar to equation 6:

$$D(Q, R, p) = \begin{cases} \sqrt{\sum_{x=0}^{height} \sum_{y=0}^{width} p(Q, R, x, y)} & \text{if } \sqrt{\sum_{x=0}^{height} \sum_{y=0}^{width} p(Q, R, x, y)} < d(k) \\ \infty & \text{otherwise} \end{cases} \quad (8)$$

On first glance, this did not improve the performance yet. But due to the fact that we compute sums of potentially many non-negative values, we can safely abort this computation as soon as the sum we aggregated so far exceeds $d(k)$. Since $d(k)$ can only be known after *all* images in

```
DISTANCE(Q, R, p)
 1  sum ← 0                                        ▷ initialize to sum to 0
 2  maxSum ← d̂(k)²                                 ▷ initialize maxSum
 3  for x ≤ height
 4        do
 5           for y ≤ width
 6                 do
 7                    sum ← sum + p(Q, R, x, y)   ▷ perform computation for this pixel
 8                    if sum ≥ maxSum             ▷ early termination condition reached
 9                       then return ∞            ▷ terminate and return worst distance
10  return √sum                                    ▷ terminate regularly
```

Figure 2: Distance computation using maximum sum

the collection have been compared to Q, we have to further approximate $d(k)$ with $\hat{d}(k)$, that is, the distance of the k nearest neighbor identified so far. To ensure correctness, we only need to assure $d(k) \leq \hat{d}(k)$ at any time of the process, which requires as a consequence that if we did not yet compute the similarity of at least that k images, then $\hat{d}(k)$ must return ∞. Thus, after some optimization to delay the calculation of the square root, our approximation of equation 8 is:

$$D(Q,R,p) = \begin{cases} \sqrt{\sum_{x=0}^{height} \sum_{y=0}^{width} p(Q,R,x,y)} & \text{if } \sum_{x=0}^{height} \sum_{y=0}^{width} p(Q,R,x,y) < \hat{d}(k)^2 \\ \infty & \text{otherwise} \end{cases} \quad (9)$$

A simple, yet efficient pseudo-code for this function is presented in figure 3.1. In worst case, where k is the size of the entire collection, every image in the collection is so similar that each of them achieves the same distance, or images in the collection are ordered in exactly the reverted order of there similarity with regard to Q, the only overhead generated by this approach is a couple of very cheap comparison and the assignment in line 2. The ordered list of all nearest neighbors seen so far, which is needed to implement $\hat{d}(k)$ would be required to some extend anyway for computing the final set of kNN. But in common cases, in particular when k is small like $k \leq 5$ as for our classifier, this method is likely to terminate early for most of the images.

Since the algorithm is orthogonal to the used pixel distance computation function, it can be applied to IDM, P2DHMDM as well as the Euclidean distance that might be used in a sieve function. In our own implementation, we tried out also distance functions where there is no need to compute the square root. For our implementation, we did not separate the code of the generic distance function from the pixel distance computation function and therefore implemented this kind of behavior directly in every combined function, which also gave the chance to better use the locality of some variables.

In order to keep the amount of main memory occupied acceptable, $\hat{d}(k)$ is also used as a threshold to filter results: Only results with a distance less than $\hat{d}(k)$ will be kept and added to the list of results in main memory. Details on options how to implement this efficiently are presented in Section 4.2.

3.2 Filter–and–Refinement

The sieve function as described in [15] is similar to a top operator in some database systems, that performs a simple cutoff at position c of the ascendingly ordered list. Finding a value c that makes a good tradeoff between the reduced running time and the reduced retrieval quality is very important in such a case. It looks appealing, that if the retrieval takes too long, c simply is reduced until retrieval times are acceptable again. Unfortunately, in order to validate if the retrieval quality is still close enough to the potential of the distance measure, it actually requires the full run for comparison.

We therefore explored possibilities to filter the candidates without any loss of retrieval quality. This is possible using a fiter–and–refinement algorithm using upper and lower bounds as used in [17, 14]. For finding a valid upper bound, we use the fact that our cost function C is designed such that $C(0,0) = 0$ – or in other words: If the pixel has not been displaced, there is no penalty for that. Combining this with the minimality constraint in equation 5, one can easily conclude that if $p_{euclid'}$ is a variant of p_{euclid} that is using the same the scaling / mapping in case of images that differ in there number of pixel in width and/or hight as p_{IDM} and also all layers with the same weights, then $p_{euclid'}(Q,R,x,y) \geq p_{IDM}(Q,R,x,y)$ holds for every $Q,R \in I$ and $x \in [0, height], y \in [0, width]$ and therefore is a valid upper bound for the distance. For finding a upper bound for $p_{P2DHMDM}$ with $w = 3$ as in our experiments, the straight forward choice is to use $p_{P2DHMDM}$ with $w = 0$ and therefore also disallowing displacement.

As a safe lower bound, we can allow displacements all over the image without penalty. This is equal to retrieving the distance to the closest pixel value within the histogram of the reference image. We need only the gray value of such a pixel, therefore we did not compute a full histogram, but simply extracted all gray values that appear at least one time. We store the found gray values in an ordered list, such that binary search can be performed for finding the most similar value to each pixel in the query image. For 8-bit grayscale images, even a simple lookup table can be used to speed up this task. This histogram lookup can be used directly as the lower bound for p_{IDM}; for a very quick, but not very tight lower bound for $p_{P2DHMDM}$, we still need to divide this value by the number of pixels in the area defined by the HMM range h which is $(2h+1)'^2$, since all surrounding pixels in this area of the query image might be exactly equal to the corresponding ones of the reference image and there contribution to the sum therefore 0.

The retrieved candidates are stored in a priority queue which is ordered ascendingly on minimal lower bound distance. For being a candidate, an image needs to achieve a lower bound similarity smaller than the smallest k upper bounds (or exact similarity values) seen so far. Unfortunately, the lower bound turned out to be close to zero for almost any two images, since the number of pixels and the type of images resulted in each gray value being present in almost any image. Therefore the filter stage was not selective enough and the run times did not improve, since basically all exact distances still needed to be computed. In other setups, this approach might achieve better results.

3.3 Multithreading

Within the last years, multi-core CPUs became very popular and affordable. Therefore it becomes more and more important to design applications in a way that they can use multiple threads in order to utilize all the capabilities provided by current hardware.

In image retrieval tasks, there are commonly two distinct tasks that which might be CPU-intinsive: a) the feature extraction of entire collections and b) the actual retrieval. For the first one, it is very common to separate this task not among several threads, but frequently also using many nodes in a network as in [18]. For the task of the actual search, parallel index structures [16] and the coordination [13] becomes more important since different execution speeds of parts of the query may limit the overall speed and prohibit in worst case interactive answering times.

Since communication between threads is much cheaper than the communication over network, we decided to not assign ranges as workloads to individual threads, but individual similarity computations. A dispatcher takes the computed result, updates the $\hat{d}$ as described in Section 3.1 and assigns the next task to the thread. Though this we could achieve almost linear speedup on multi-core CPUs, since IDM is much more CPU-bound than I/O-bound and accesses to the disk get serialized through the dispatcher and therefore the concurrent execution does not lead to slow concurrent disk accesses.

4 Implementation Details

4.1 Feature Storage

In order to keep the I/O costs as low as possible, in particular to avoid random seeks on disk, we store all extracted features in a single file. For the time being, we use the Java object serialization to store the individual feature vectors in this file. The resulting file for all 10'000 feature vectors occupies only about 60 MB for the image scaled down to 32 pixels on the longer side together with the Sobel-filtered version. Since this is not much comparing state-of-the-art hard disks with several Gigabytes up to Terabyte storage capacity, we generated separate files in case we want to use either just the gray values without the Sobel-filtered version or the other way round.

Notice that we use $p_{euclid'}$ in case of sieve function or the filter–and–refinement algorithm presented in Section 3.2 and can use for this the same feature vectors as for IDM with or without P2DHMDM. This is possible since we adapted the Euclidean distance to perform the same scaling and therefore we i.) get better approximations since the filter uses a closer bound, ii.) do not need to store separate feature vectors for the 32x32 representation of images. The situation could be slightly different if we employed more features and corresponding distance functions. But even then it might be beneficial to perform a feature fusion on-the-fly to aggregate all features used in a multi-feature single-object query in main memory [2].

The feature file gets sequentially read, either for each query performed or just a single time and then remains cached entirely in memory. This is easily possible for the current collection since 60 MB are not much compared to the RAM that current computers are equipped with. In any case, in order to reduce subsequent needs to re-read the features or meta-information about an image file during the execution of a query, we keep the entire feature vector in memory as long as the image remains a valid candidate of being among the k nearest neighbors. By this, we only need to keep and pass references rather than copying bytes in memory and can make excellent use of Java's garbage collection.

More sophisticated index structures like the VA-File [19], M-Tree [4], or R*-Tree [1] have not been used for several reasons: First of all, the dimensionality of the feature vectors is high (2 layers with up to 32 x 32 pixels = 2048 dimensions) and worse, are of variable length. Second, neither IDM, P2DHMDM, nor the modified Euclidean distance are metrics. This is due to the preservation of the aspect ratio, that results in varying width and height and therefore the feature vector length of each image may differ. Since the query image is determining the number of pixels used for distance computation, the symmetry condition $D(X,Y) = D(Y,X)$ does not hold for any of these distance functions. For similar reason, the triangle inequation $D(X,Z) \leq D(X,Y) + D(Y,Z)$ may not be satisfied.

4.2 Bounded Priority Queue

Most of the improvements achieved in Section 3 are based on the fact, that we can easily and inexpensively find or refer at any time to the k minimal distances that have been computed for this query up to this moment. This means that we need some ordering of distance scores. We also need an ordering of candidate images in a different context, but this has in detail two different requirements.

First, we need a priority queue to get the best candidate in the filter-and-refinement algorithm as presented in Section 3.2 and also to perform the sieve function more efficiently. In both cases, we add many (potentially all) images of the search collection to the queue, but intent to take out only a small fraction (the cutoff value c in case of the sieve function) from the top. We therefore do not have to entirely sort the queue. Sun provides in Java version 1.5 a very decent implementation of a priority queue based on a heap. But in addition, since we are only interested in keeping good candidates in the list, it would be ideal to be able to remove previously inserted items and thus reduce memory requirements and also speed up other operations that modify the priority queue. We are not interested which items get removed, as long as we do not accidentally remove one that would become one of the k nearest neighbors. For selecting victims for deletion, we can apply

again the bound $\hat{d}(k)$ defined for computing the maximum sum in Section 3.1. Of course, such deletion should be cheap, because otherwise it would slow down the organization of the priority queue more than it improves it. We therefore used a binary minimum heap implementation of our own. Due to the heap property it is possible to delete any leaf node without violating the heap property and therefore can perform this efficiently. The binary heap is further organized such that it is either perfectly balanced or if the last level of the tree is not filled entirely, it is always filled from left to right. We implemented a fast yet simple heuristic in which, before each new insertion, we check if the right-most leave node still is better than $\hat{d}(k)$. If not, we will delete it and continue deleting until we find a better leaf node. We will place the new entry to the right of the this item. Of course, we will start this insertion only, if the new item is better than $\hat{d}(k)$. Although this does cannot guarantee that the heap is free of "bad candidates", it may reduce significantly the number of entries – depending on the value of $\hat{d}(k)$.

As a second requirement, in order to determine and maintain $\hat{d}(k)$ efficiently, we need a sorted list of a fixed size to which we can add new distance scores. In case the list is already full, the data structure does not need to grow, but simply drop the worst value and insert the new value at the appropriate position. This data structure is used to get the value that a distance may not exceed in order to become a candidate. It is checked frequently, in particular to decide whether an item should be added to the priority queue described in the first requirement. Therefore we actually need a list of the best values so far, hence a list of ascending distances, but need to be able to retrieve worst score very efficiently – not the best value as it is the property of priority queues. Common priority queues provide very efficient access to the best value, but less efficient or even no direct access to all other values. So we rather need a fully sorted data structure. In addition, we need the ability to store duplicate values since two different nearest neighbors may have exactly the same distance and -in contrast to range queries- the number of items is very important for nearest neighbors searches. These properties together cannot be easily met with e.g., SortedSet out of the Java Class Library or FixedSizeList of Jakarta Commons Collections. Therefore, we implemented our own data structure. This list is fixed in size and backed by a simple float array. Insertion points can be identified by a simple binary search and shifting of huge parts of the list in case of insertion close to the head can be performed by comparably cheap system calls.

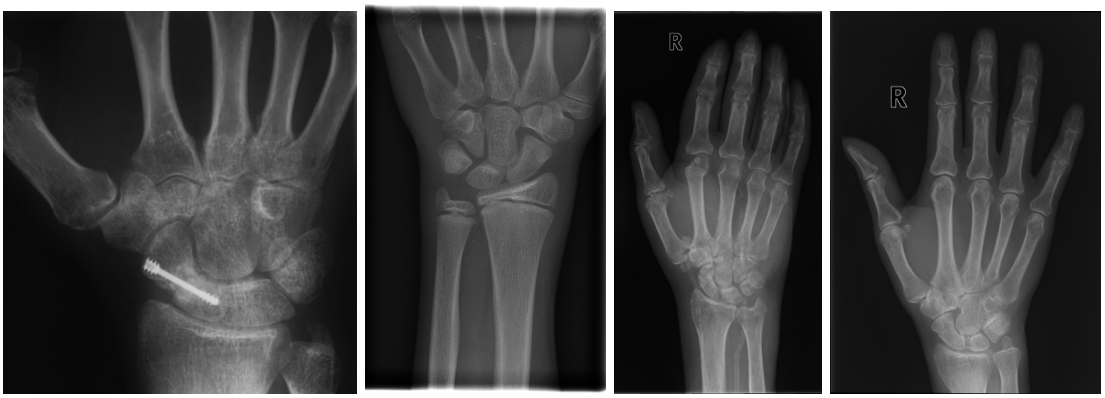
For the filter-and-refinement algorithm, we need these two requirements at the same time, but on different lists – one priority queue for the images based on their lower bound distances and the sorted list of upper bound (or exact) distances. But for the sieve function as well as a plain sequential scan, those two requirements are actually always applied to the same elements. Therefore we extended the concept of the sorted list of fixed size to also act as a "priority list", which is bounded in size and automatically drops all elements with ranks greater than the fixed size. With this, we were able to reduce the minimal memory requirement to k elements in case of a plain sequential scan and $k + c$ elements for the sieve function.[3] The deletion heuristics to limit the size of a priority heap sketched above is no longer required here, since the work to erase "bad candidates" is performed already to be able to determine $\hat{d}(k)$.

5 IRMA-Code-aware Classifier

Since the goal of this task is the automatic annotation of medical images by assigning them classes, the step following the generic retrieval is the classification. For this, we started with a weighted kNN classifier that uses the inverse square of the distance [8]. Let l be the ordered list of nearest neighbors and $dl(l, i)$ be the distance of the i-th nearest neighbor in the list and $cl(l, i)$ the class, respectively. For each class $a \in l$ we compute the score s:

$$s(a, l) = \sum_{i=0}^{k} \begin{cases} \frac{1}{dl(l,i)^2} & \text{if } cl(l, i) = a \\ 0 & \text{if } cl(l, i) \neq a \end{cases} \tag{10}$$

[3] For the sieve function, we need to store in addition to the result also temporarily the top-c results of the less expensive distance used for filtering in a separate data structure.



(a) Query: 1121-120-413-700 (b) 1121-120-421-700 (c) 1121-110-415-700 (d) 1121-110-415-700

Figure 3: Query image and three nearest neighbors and there IRMA codes

The function $kNN(l)$ then simply returns the class with the maximum score.

The IRMA code is hierarchical and uncertainty / a "don't know" character is allowed at any level in the hierarchy in submission. Therefore it can be better to express uncertainty at some level which will receive an error of 0.5 at this level rather then submitting a false guess, which will receive an error of 1.0. We implemented a function $scode(l)$ that given a set of classes generates the code containing the "don't know" symbol * at any position the codes of the classes differ. Example as shown in Fig. 3: Image number 372250 (Fig. 3(a)) was the query image for training and the images 9532 (Fig. 3(b), dist= 648.25), 10372 (Fig. 3(c), dist=649.07), and 12305 (Fig. 3(d), dist=651.49) are the retrieved three nearest neighbors. Notice that the correct classification would be 1121-120-413-700 to which none of the found nearest neighbors belongs. Using kNN we would get the class 1121-110-415-700, which results according to [5] in an error of 0.186. If we use only the class information instead and generate the $scode(l)$ of the 3-NN with "don't know" symbols, we get the class 1121-1**-4**-700 with an error of only 0.122.

Using only $scode(l)$ will frequently result in codes with high uncertainty, even if the $kNN(l)$ would give very good guesses. This is particularly true for big k. So we limit $scode(l, i)$ to use only the i nearest neighbors for generating the IRMA code, with $i = 2$ being the best choice we found so far. As a rule of thumb, $kNN(l)$ gives good results if the nearest neighbors are close enough to the query image. As pointed out in Section 4.1, P2DHMDM is not symmetric, because its values depend on the number of pixels in the query image. This makes it harder to determine when the nearest neighbors are close enough to rely on $kNN(l)$. As a simple way to normalize the distance, we divided it by the number of pixels of the query image. In experiments we tried several values as thresholds and it turned out that the best value would be around 1. So we came up with the following classifier:

$$kNN_{IRMA}(l) = \begin{cases} knn(l) & \text{if } \frac{dl(l,1)}{width*height} < 1.0 \\ scode(l,2) & \text{otherwise} \end{cases} \tag{11}$$

In order to directly compare many classifiers using different parameter sets, we wrote a small program to perform the nearest neighbor search with the biggest k required by any of the classifiers and then feeding all of them with only the number of top-k results that are needed for the classifier. By this, we avoided to repeat the costly nearest neighbor search for each classifier individually.

6 Results

6.1 Testing Infrastructure

For our experiments, we used a couple of machines in order to be able to verify the expected performance improvements on different CPUs and operating systems. The used machines are listed in Figure 4.

P4	Compaq Evo D51C/P2, Intel Pentium 4 w/o HT, 2.4 GHz, 512 MB RAM, Windows XP, Java: Sun JDK 1.6.0_02-b06
Xeon	IBM xSeries 445, 8x Intel Xeon MP 2.8 GHz, 2 MB Cache each, 8 GB RAM, Windows 2003 SP2, Java: Sun JDK 1.6.0_02-b06 Xeon_{HT} means HyperThreading enabled; disabled if Xeon is without $_{HT}$
C2D	Fujitsu-Siemens Celsius M450 Workstation, Intel Core 2 Duo E6600, 2.4 GHz, 2x2 MB Cache, 4 GB RAM, Ubuntu 7.04 GNU/Linux 2.6.20-16-generic SMP, Java: Sun JDK 1.6.0_02-b05 64 bit

Figure 4: Infrastructure used for measuring execution times

Due to time limitations, not all experiments could be performed on all machines. P4 has a single 32-bit processor, the other machines have multiple CPUs / cores, and therefore require multiple threads to fully utilize all CPUs. The number of parallel threads used in an experiment are written behind the used machine as an exponent, e.g. Xeon^{8} means that 8 worker threads were used - in this case this would use all 8 CPUs, whereas Xeon^{16}_{HT} would fully use all CPUs in case HyperThreading is enabled, for which each CPU appears to the OS as two separate ones. The Core 2 Duo processors is a 64-bit architecture, therefore a 64-bit software environment (OS, Java) has been installed.

6.2 Submitted Runs

For the runs submitted, we used an image distortion model with a warp range of $w = 3$ and two different cost matrices, 4812 and 369, where the one assigning higher costs (4812) performed a little better when using only a 5x5 pixel area ($h = 2$) in the P2DHMDM. For $h = 3$ we submitted only a single run using the 369 matrix - which turned out to be the best of our runs. A "c" appended to the number of k nearest neighbors indicates that the IRMA-code aware classifier shown in Equation 11 was used with a threshold of 1.0 on the normalized distance and that the two nearest neighbors will be used to generate the code. We used a per-pixel threshold of $t = 96$ in Equation 6 in all cases. As mentioned in Section 5, whenever there were a couple of runs that differ only in the in the used classifier, we performed the nearest neighbor search only once with the biggest k and passed the result to all appropriate classifiers. This is the reason why several runs took exactly the same time – even when k was different.

Rank	Run id - UNIBAS-DBIS-...	w	h	k	Score	Execution time
19	IDM_HMM_W3_H3_C	3	3	3c	58.14	3h 41m on Xeon^{16}_{HT}
20	IDM_HMM2_4812_K3	3	2	3	59.84	2h 38m on Xeon^{8}
21	IDM_HMM2_4812_K3_C	3	2	3c	60.67	2h 38m on Xeon^{8}
22	IDM_HMM2_4812_K5_C	3	2	5c	61.41	2h 38m on Xeon^{8}
23	IDM_HMM_369_K3_C	3	2	3c	62.83	2h 23m on Xeon^{16}_{HT}
24	IDM_HMM_369_K3	3	2	3	63.44	2h 23m on Xeon^{16}_{HT}
25	IDM_HMM_369_K5_C	3	2	5c	65.09	2h 23m on Xeon^{16}_{HT}
(not submitted)	IDM P2DHMDM	2	1	1	n/a	24m 25s on Xeon^{16}_{HT}

Figure 5: Execution times of performed runs

Features were cached in main memory, execution time is for the entire run on 1'000 images. Each image has been processed one after another by first extracting the features of the image, then performing the nearest neighbor search and finally wait until all classifiers (if more than one) have finished. We did not apply further optimizations for the throughput of the entire run, e.g. extracting the features of the next image while the classifier for the last image is still running. Therefore one can simply divide the runtime of the entire run by 1'000 to get the average execution time per query. This means for $h = 2$, a single query took on average less than 8.6 seconds with HyperThreading enabled, less than 9.5 seconds without HyperThreading, and 13.3 seconds when $h = 3$. For comparison: Our best run with $w = 3$ and $h = 3$ took 16h 18m on Xeon^{16}_{HT} when no early termination based on the maximum sum and no checking on insertion to the priority queue was used. This long duration even increased to entire 5 days, 17h and 32m on the same machine when we limited the number of used threads to a single one – as it was the case in our starting point of the implementation. This means that our optimizations achieved a speedup of 4.42 and 37.34, respectively.

We also performed runs with the parameters proposed in [10]: IDM with a deformation in 5x5 pixels ($w = 2$) and a P2DHMDM of a local context of 3x3 pixels ($h = 1$) and the nearest neighbor decision rule ($k = 1$). On the standard Pentium 4 PC, this run finished within 4 hours and 28 minutes (16.0 seconds per query) – without any sieve function. The same run was finished on the 64-bit Core 2 Duo machine in 1 hour and 11 minutes (4.2 seconds per query) and on the 8-way Xeon^{16}_{HT} with HyperThreading enabled within 24 minutes and 45 seconds (1.5 seconds per query). When we turned off just the early termination, the durations increased to 19 hours 21 minutes on P4 (69.77 seconds per query, factor 4.33), 4 hours 14 minutes on C2D (15.0 seconds per query, factor 3.58), and 1 hour 59 minutes on Xeon^{16}_{HT} (7.1 seconds per query, factor 4.86).

6.3 Short summary of other experiments

- More experiments need to be performed to verify the benefit o the IRMA-code aware classifier. The official runs did not achieve the same clear picture as we would have expected from the results on the development set.
- Filter-and-refinement could be implemented, but did not improve the speed because the lower bound was not tight enough.
- The sieve function as proposed in [15] can also reduce the execution times significantly, but only at the cost of loss of retrieval quality. In our experiements, the loss was smaller if the Euclidean distance has been replaced by the Manhattan distance. The Manhattan distance as a pixel distance computation function is $p_{manhattan}(Q, R, x, y) = |Q(x, y) - R(x, y)|$, but requires that $D(Q, R, p)$ may not take the square root of the sum if $p = p_{manhattan}$.
- We could speedup also the sieve function. Since the cutoff position $c = 500$ is much bigger than $k \in 1, 3, 5$, the early termination strategy could not achieve a speedup of approximately 4, but at least more than half of the pixel distance computations could be omitted. The more computationally intensive part occurs after the sieve function performed the cutoff – and here the smaller k is used to compute the exact distance out of the c candidates. Also for this stage the early termination strategy paid off.
- Not caching the entire feature file in memory results in additional work to (re-)read sequentially 60 MB of features for each query. The required time for this depends heavily on the hard disk I/O speed of the system and varied in our case between ≤ 1 second on the server hardware and 2 to at most 5 seconds for the old hard disk of the standard PC. Iterating over the feature vectors is integrated in the retrieval process. How much this slows down the query execution depends on the complexity of the distance measure. For features that are fast to compute like the Euclidean distance, the I/O speed determines the overall performance. For P2DHMDM, the overhead is rather small, e.g. on C2D^2, the additional time spend for reading the features from disk for an entire run of 1'000 was only about 2.5 – 5 minutes.

7 Conclusion and Outlook

In this paper, we propose an early termination strategy, which has proven to successfully speed up the expensive Image Distortion Model with Pseudo-2D Hidden Markov Distorion Model by factors in the range of 3.5 to 4.9 in our experiments. Using appropriate data structures for storing intermediate sets of candidates and ordered lists of best distances achieved so far, we could reduce the computation time to perform a single query in the collection with 10'000 reference images to approximately 16 seconds on a standard Pentium 4 PC. Making proper use of multithreading, we could even perform a single query within 1.5 seconds on a 8-way Xeon MP server.

This improved speed opened enough resources, to allow for local displacements of IDM in an area of 7x7 pixels instead of 5x5 and also increase local context of the HMM range from a 3x3 area to 5x5 or 7x7. Even with these parameters that cause significantly more computations, we could perform this on the Xeon server in maximum 13.3 seconds per query.

By this we were able to achieve rank 19 – 25 among 68 runs that have been submitted to the Medical Automatic Annotation Task of ImageCLEF 2007. Further experiments shall provide an in-depth analysis of the applied optimizations. So far, some parameters like the cost function and the chosen classifier have been set manually. IDM suffered until now from the lack of support for setting those values automatically by means of machine learning and automated verification through cross-validation since the runtimes were simply too long. With our optimizations we expect to be able to perform those within a reasonable time.

Acknowledgments

Part of the implementation are based on the work performed by Andreas Dander for his BSc project at UMIT, Hall i.T., Austria. Based on his idea we tried out the Manhatten distance instead of Euclidean distance in the sieve function, which performed significantly better in all our experiments. He also adapted the Filter–and–Refinement Algorithm for the use in this context.

References

[1] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: an efficient and robust access method for points and rectangles. In *SIGMOD '90: Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pages 322–331, New York, NY, USA, 1990. ACM Press.

[2] Klemens Böhm, Michael Mlivoncic, Hans-Jörg Schek, and Roger Weber. Fast evaluation techniques for complex similarity queries. In *VLDB*, pages 211–220, 2001.

[3] Gert Brettlecker, Paola Ranaldi, Hans-Jörg Schek, Heiko Schuldt, and Michael Springmann. Isis & osiris: a process-based digital library application on top of a distributed process support middleware. In *DELOS Conference 2007*, February 2007.

[4] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *VLDB*, pages 426–435, 1997.

[5] Thomas Deselaers, Jayashree Kalpathy-Cramer, Henning Müller, and Thomas M. Deserno. Hierarchical classification for imageclef 2007 medical image annotation, 2007.

[6] Thomas Deselaers, Henning Müller, Paul Clogh, Hermann Ney, and Thomas M Lehmann. The clef 2005 automatic medical image annotation task. *International Journal of Computer Vision*, 74(1):51–58, August 2007.

[7] Thomas Deselaers, Tobias Weyand, Daniel Keysers, Wolfgang Macherey, and Hermann Ney. Fire in imageclef 2005: Combining content-based image retrieval with textual information retrieval. In *CLEF*, pages 652–661, 2005.

[8] Martin Esters and Jörg Sander. *Knowledge Discovery in Databases*. Springer, Berlin, 2000.

[9] D. Keysers, C. Gollan, and H. Ney. Classification of medical images using non-linear distortion models, 2004.

[10] Daniel Keysers, Thomas Deselaers, Christian Gollan, and Hermann Ney. Deformation models for image recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(8):1422–1435, August 2007.

[11] Michael Mlivoncic, Christoph Schuler, and Can Türker. Hyperdatabase infrastructure for management and search of multimedia collections. In *DELOS Workshop: Digital Library Architectures*, pages 25–36, 2004.

[12] Henning Müller, Thomas Deselaers, Thomas Lehmann, Paul Clough, and William Hersh. Overview of the imageclefmed 2006 medical retrieval and annotation tasks. In C. Peters, P. Clough, F. Gey, J. Karlgren, B. Magnini, D.W. Oard, M. de Rijke, and M. Stempfhuber, editors, *Evaluation of Multilingual and Multi-modal Information Retrieval – Seventh Workshop of the Cross-Language Evaluation Forum, CLEF 2006*, volume 4730 of *LNCS*, pages 595–608, 2007.

[13] Hans-Jörg Schek and Roger Weber. Higher order databases and multimedia information. In *Swiss/Japan Seminar Advances in Database and Multimedia*, February 2000.

[14] Michael Springmann. A novel approach for compound document matching. *Bulletin of the IEEE Technical Committee on Digital Libraries (TCDL)*, 2(2), 2006.

[15] Christian Thies, Mark Oliver Güld, Benedikt Fischer, and Thomas M. Lehmann. Content-based queries on the casimage database within the irma framework: A field report. In C. Peters, P. Clough, J. Gonzalo, G. J. F. Jones, M. Kuck, and B. Magnini, editors, *Multilingual Information Access for Text, Speech and Images: 5th Workshop of the Cross-Language Evaluation Forum*, volume 3491/2005 of *LNCS*, pages 781–792. Springer Berlin, Heidelberg, 2005.

[16] Roger Weber, Klemens Böhm, and Hans-Jörg Schek. Interactive-time similarity search for large image collections using parallel va-files. In *ICDE*, page 197, 2000.

[17] Roger Weber and Michael Mlivoncic. Efficient region-based image retrieval. In *CIKM '03: Proceedings of the twelfth International Conference on Information and Knowledge Management*, pages 69–76. ACM Press, 2003.

[18] Roger Weber and Hans-Jörg Schek. A distributed image-database architecture for efficient insertion and retrieval. In *Multimedia Information Systems*, pages 48–55, 1999.

[19] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases*, pages 194–205. Morgan Kaufmann, 1998.