UNIVERSITAT POLITÈCNICA
DE CATALUNYA

Universitat Politècnica de Catalunya
Teoria del Senyal i Comunicacions
European Master of Research
on Information and Communication Technologies

Tools for Image Retrieval in Large Multimedia Databases

Master's Final Project Dissertation

by Carles Ventura Royo

Advisors:
Verónica Vilaplana Besler
Xavier Giró Nieto

Barcelona, September 2011

Abstract

One of the challenges in the development of an image retrieval system is to achieve an efficient indexing scheme since both developers and users, who are used to make requests in order to find a multimedia element in a large database, can be frustrated due to the long computational time of the search.

The traditional indexing schemes neither fulfill the dynamic indexing requirement, which allows to add or remove elements from the structure, nor fit well in high dimensional feature spaces due to the phenomenon so called "the curse of dimensionality."

After analyzing several indexing techniques from the literature, we have decided to implement an indexing scheme called Hierarchical Cellular Tree (HCT), which was designed to bring an effective solution especially for indexing large multimedia databases. The HCT has allowed to improve the performance of our implemented image retrieval system based on the MPEG-7 visual descriptors. We have also made some contributions by proposing some modifications to the original HCT which have resulted in an improvement of its performance. Thus, we have proposed a redefinition of the covering radius, which does not consider only the elements belonging to the cell, but also all the elements holding from that cell. Since this consideration implies a much more computationally costly algorithm, we have proposed an approximation by excess for the covering radius value. However, we have also implemented a method which allows to update the covering radius to its actual value whenever it is desired. In addition to this, the pre-emptive insertion method has been adapted as a searching technique in order to improve the performance given by the retrieval scheme called Progressive Query, which was originally proposed to be used over the HCT.

Furthermore, the HCT indexing scheme has been also adapted to a server/client architecture by using a messenger system called KSC, which allows to have the HCT

loaded on a server waiting for the query requests which are launched for the several clients of the retrieval system. In addition to this, the tool used to request a search over the indexed database has been adapted to a graphic user interface, named GOS (Graphic Object Searcher), which allows the user to order the retrievals in a more friendly way.

Acknowledgments

I would like to express my greatest appreciation to my advisors, Verónica Vilaplana and Xavier Giró, and my tutor, Ferran Marqués, for their support, stimulating suggestions and encouragement throughout the course of this research work.

I would also like to thank all my colleagues and friends for their help and useful suggestions, in particular Jordi Pont and Albert Gil. Moreover, I am very grateful to my family for cheering me up constantly.

Contents

List of Figures

Chapter 1

Introduction

As a consequence of recent technology development, large image databases have been created. For example, on the Web, billions of images are uploaded in sites such as Flickr and Facebook, millions of websites including images are updated everyday and also new websites are created. But these large image databases are not only present in internet; audiovisual media companies have also all their broadcasted content stored in large private systems. In these contexts, well organized databases and efficient storing and retrieval are absolutely necessary.

Most traditional methods of image retrieval consist in adding metadata, such as keywords or textual descriptions, to the images so that retrieval can be performed on these annotations. For instance, every time you upload an image in Flickr you can assign up to 75 tags, which act as keywords. These tags are helpful to find images which have something in common. Due to the subjectivity in the choice of the keywords and the cost of manual annotation over a large database, most popular image retrieval systems, such as Google Image Search and Microsoft Live Image Search, are based on automatic keyword extraction algorithms that analyze the surrounding text [CWT08][HT09]. However, these systems do not take advantage of the visual information intrinsically contained in the image.

It was in 1992 that Kato referred to Content-Based Image Retrieval (CBIR) in order to describe his experiments [HK92]. CBIR systems are based on the automatic extraction of visual features, such as color, texture and shape, from the images, which are compared by using a similarity measure between their features. Query by Example (QbE) is the most popular query technique, which involves providing the system

with an example query image. From this image, some visual features are extracted and compared to the features of each image in the database in order to retrieve the most similar ones. Many CBIR systems, such as QBIC [NBE+93], Virage [BFG+96], Photobook [PPS95], VisualSEEk [SC96] among others, have been implemented. On the bases of MPEG-7 standard [Man02], we designed a CBIR system which uses some defined descriptors: Color Structure Descriptor, Dominant Color Descriptor, Color Layout Descriptor and Texture Edge Histogram Descriptor [Ven10].

CBIR systems suffers basically from three problems. First, there is the gap between the high level semantic concepts used by humans and the low-level visual features extracted from a computer [ZH00]. Visual features are useful to compute the similarity between two images based on color, texture... but this similarity does not always match with the human perception. Second, there is a scalability problem, i.e. CBIR systems needs incorporating indexing techniques in order to scale up well when they work over large databases. Thus, an exhaustive sequential search would not be feasible in a large database due to its linear computation time behavior. To get an idea, an exhaustive search performed on a database of 200,000 elements lasts 10 seconds by using the implemented system in [Ven10]. This is the problem we aim to solve and, therefore, an efficient indexing technique is required in order to achieve retrieval times that would be acceptable for the user. Third, there is the so-called "curse of the dimensionality" problem. Some indexing techniques, in particular spatial access methods (see Section 2.1), do not fit well with in high dimensional feature spaces. Thus, in such cases, there is no improvement in using these indexing techniques in comparison to an exhaustive search over the entire database. Therefore, an indexing technique which fulfills the following requirements is desired:

- Dynamic approach. An index structure which allows insertions and deletions of the indexed elements. Multimedia databases are not static and, therefore, we do not want an indexing technique which requires to reindex the whole database from scratch every time an element is to be either inserted or removed.

- High dimensional feature spaces. Since our implemented CBIR system works with the MPEG-7 visual descriptors, which are high-dimensional feature vectors, we need an indexing technique which does not suffer from the "curse of the dimensionality" problem.

Towards this goal, many indexing techniques have been studied in Chapter 2. The indexing technique which fits best the requirements to be fulfilled, called Hierarchical Cellular Tree (HCT) [KG07] is described in detail in Chapter 3. Furthermore, some modifications to the HCT have been proposed in Chapter 4. Next, Chapter 5 describes how the Hierarchical Cellular Tree has been implemented in our development platform. Then, the performance of the implemented indexing technique is evaluated in Chapter 6. The HCT has been built over an image database which consists of more than 200,000 elements. In order to evaluate the performance of the retrieval system, the rankings obtained by using many searching techniques over the HCT have been compared with the ranking resulting from an exhaustive search. As it was expected, there is a compromise between the computational time required for retrieval and the "goodness" of the elements retrieved, i.e. the quality of the ranking obtained by using the HCT in comparison to an exhaustive search. Some measures extracted from literature have been used in order to evaluate this "goodness" of the elements retrieved. Finally, we draw some conclusions and present future lines of work in Chapter 7.

Chapter 2

State of the Art

For the past three decades, researchers proposed several indexing techniques in order to overcome storage and specially management problems resulting from the recent technological hardware and network improvements along with the daily usage of Internet. The most traditional indexing techniques are formed mostly in a hierarchical tree structure which is used to cluster the feature space. These indexing techniques can be mainly grouped in two categories: (*i*) spatial access methods (SAMs) and (*ii*) metric access methods (MAMs). [Het] is recommended for readers which are not familiar with the basic principles of metric indexing. Furthermore, the reader is addressed to [Knu97] and [CLRS01] for an overview of the tree structure data concepts. In Section 2.1 and Section 2.2, the principles on which SAMs and MAMs are based are explained, respectively, and some indexing techniques are presented in each case. Finally, another very popular indexing technique called Locality Sensitive Hashing (LSH), which is not based on a hierarchical tree structure, is presented in Section 2.3.

2.1 Spatial Access Methods

The goal of spatial access methods is to organize spatial data in such a way that it will enable the efficient retrieval of relevant objects according to the topological properties of their spatial attributes. Researchers have proposed several SAM-based indexing techniques:

- $k-d$ tree [FBF77]. The $k-d$ tree is a generalization of the simple binary tree used for sorting and searching. The $k-d$ tree is a binary tree in which each

node represents a subset of the records in the database and a partitioning of that subset. The root of the tree represents the entire database. Each nonterminal node has two sons or successor nodes which represent the two subsets defined by the partitioning. The terminal nodes represent mutually exclusive small subsets of the data records, which collectively form a partition of the record space.

In the case of one-dimensional searching, a record is represented by a single key and a partition is defined by some value of that key. All records in a subset with key values less than or equal to the partition value belong to the left son, while those with a larger value belong to the right son. Thus, the key variable becomes a discriminator for assigning records to the two subsets.

In k dimensions, a record is represented by k keys. Any one of these can serve as the discriminator for partitioning the subset represented by a particular node in the tree. Thus, the discriminating key number can range from 1 to k. The $k-d$ tree is optimized by choosing at every nonterminal node the key with the largest spread in values as the discriminator and to chose the median of the discriminator key values as the partition.

- R-tree [Gut84]. An R-tree is a height-balanced tree with index records in its leaf nodes containing pointers to data objects. A spatial database consists of a collector of tuples representing spatial objects, and each tuple has a unique identifier which can be used to retrieve it. Leaf nodes in an R-tree contain index record entries of the form $(I, tuple-identifier)$ where $tuple-identifier$ refers to a tuple in the database and I is an n-dimensional rectangle which is the bounding box of the spatial object indexed $I = (I_0, I_1, ..., I_{n-1})$. Here n is the number of dimensions and I_i is a closed bounded interval $[a, b]$ describing the extent of the object along dimension i. Non-leaf nodes contain entries of the form $(I, child-pointer)$ where $child-pointer$ is the address of a lower node in the R-tree and I covers all rectangles in the lower node's entries.

- R*-tree [BKSS90]. The R*-tree is a R-tree variant which incorporates a combined optimization of area, margin and overlap of each enclosing rectangle in the directory. It provides a consistently better performance by introducing a policy called "forced reinsert". Thus, the R*-tree forces entries to be reinserted during the insertion routine in order to achieve dynamic reorganizations. Furthermore, forced reinsert changes entries between neighboring nodes and thus decreases the overlap which results in a decrease in the number of paths to

be traversed. As a side effect, storage utilization is improved. Higher storage utilization will generally reduce the query cost as the height of the tree will be kept low. In addition to this, less splits occur due to more restructuring. Furthermore, since the outer rectangles of a node are reinserted, the shape of the directory rectangles will be more quadratic. Thus, clustering rectangles into bounding boxes with only little variance of the lengths of the edges reduces the area of directory rectangles, which improves performance since decisions which paths have to be traversed can be taken on higher levels.

- TV-tree [LJF94]. The basis of the TV-tree is to use dynamically contracting and extracting feature vectors. As more objects are inserted to the tree, more features might be needed to discriminate among the objects. Thus, the TV-tree is based on the telescopic problem, which can be described as follows. Given an $n \times 1$ feature vector $\vec{x}$ and an $m \times n$ ($m \leq n$) contraction matrix A_m, the $m \times 1$ vector $A_m\vec{x}$ is an m-contraction of $\vec{x}$. A sequence of such matrices A_m, with $m = 1, ...$ describes a telescoping function provided that the following condition is satisfied:

  If the m_1-contractions of two vectors, $\vec{x}$ and $\vec{y}$, are equal, then so are their respective m_2-contractions, for every $m_2 < m_1$.

  In most applications, transforming the given feature vector will achieve good ordering. Ordering the features on the basis of importance is exactly what the Karhunen Lowe (KL) transform achieves. KL transform is optimal if the set of data is known in advance, thus, for static databases. In a dynamic case, data-independent transforms sucs as Discrete Cosine Transform and the Discrete Fourier Transform are used instead.

  Each node in the TV-tree represents the Minimum Bounding Region (MBR) of all its descendents. Each region is represented by a center, which is a vector determined by the telescoping vectors representing the objects, and a scalar radius. Since the center of the region is also a telescopic vector, the term Telescopic Minimum Bounding Region (TMBR) is used to denote the MBR with such a telescopic vector as a center.

- X-tree [BKK96]. The X-tree (eXtended node tree) is an index structure supporting efficient query processing of high-dimensional data. The X-tree avoids overlap whenever it is possible without allowing the tree to degenerate; other-

wise, the X-tree uses extended variable size directory nodes, so-called supernodes. The X-tree may be seen as a hybrid of a linear array-like and a hierarchical R-tree like directory.

The X-tree consists of three different types of nodes: data nodes, normal directory nodes, and supernodes. The basic goal of supernodes is to avoid splits in the directory which would result in an inefficient directory structure. The alternative to using larger node sizes are highly overlapping directory nodes which would require to access most of the son nodes during the search process. This, however, is more inefficient than linearly scanning the larger supernode. Supernodes are created during insertion only if there is no other possibility to avoid overlap. In many cases, the creation or extension of supernodes may be avoided by choosing an overlap-minimal split axis.

- NV-Tree [LJA11]. The NV-Tree is a disk-based data structure, which builds upon a combination of projections of data points to lines and partitioning of the projected space. By repeating the process of projecting and partitioning, data is eventually separated into small partitions which can easily be fetched from disk with a single disk read, and which are highly likely to contain all the close neighbors in the collection. For each pair of adjacent partitions, an overlapping partition, covering 50% of each partition, is created for redundant coverage of partition borders.

  During query processing, the search first traverses the hierarchy of inner nodes of the NV-Tree. At each level of the tree, the query descriptor is projected to the projection line associated with the current node. The search is then directed to the sub-partition with center-point closest to the projection of the query descriptor. This process of projection and choosing the right sub-partition is repeated until the search reaches a leaf node. Then, the query descriptor is projected onto the projection line of the leaf node. The two descriptor identifiers on either side of the projected query descriptor are returned as the nearest neighbors, then the second two descriptor identifiers, etc. Thus, the $k/2$ descriptor identifiers found on either side of the query descriptor projection are alternated to form the ranked k approximate neighbor of the query descriptor. Therefore, since no distance calculations are required, little CPU cost is incurred even for large collections.

In addition to the indexing techniques detailed above, several other SAM variants were proposed such as SS-tree [WJ96], SR-tree [KS97], S^2-Tree [WP01], Hybrid-Tree [CM99], A-tree [SYUK00], IQ-tree [BBJ+00], Pyramid Tree [BBK98], NB-tree [FJ03], etc.

However, SAMs have significant drawbacks for the indexing of large-scale multimedia databases. The applicability of SAMs is limited by the fact that the items have to be represented by the points in an N-dimensional feature space and the (dis)similarity measure between two points has to be based on a distance function in L_p metric such as *Euclidean* distance. Furthermore, SAMs do not scale up well to high dimensional spaces and become less efficient than sequential indexing for dimensions higher than ten [WSB98].

2.2 Metric Access Methods

MAMs give a more general approach than SAMs since they employ the indexing process by assuming only the availability of a similarity distance function that is a norm. Therefore, multimedia databases with several multidimensional features are indexed according to this similarity distance function which is usually treated as a "black box". Researchers have proposed several MAM-based indexing techniques:

- VP-tree [Yia93]. The vantage point tree (vp-tree) is based on partitioning the feature vectors (data points) into two groups according to their similarity distances with respect to a reference point, a so-called vantage point. In vp-trees, at every node of the tree, a vantage point is chosen among the data points, and the distances of this vantage point from all other points (the points which will be indexed below that node) are computed. Then, these points are sorted into an ordered list with respect to their distances from the vantage point. Next, the list is partitioned at positions to create sublists of equal cardinality.

  The structure of a binary vp-tree is very simple. Each internal node is of the form $(S_v, M, R_{ptr}, L_{ptr})$, where S_v is the vantage point, M is the median distance among the distances from S_v to all the points indexed below that node, and R_{ptr} and L_{ptr} are pointers to the right and left branches. Left branch of the node indexes the points whose distance from S_v are less than or equal to M, and right branch of the node indexes the points whose distances from S_v are

greater than or equal to M. In leaf nodes, instead of the pointers to the left and right branches, references to the data points are kept.

- MVP-tree [BO97]. Similar to the vp-tree, the mvp-tree partitions the data space into spherical cuts around vantage points. However, it creates partitions with respect to more than one vantage point at one level and keeps extra information for the data points in the leaf nodes for effective filtering of non qualifying points in a similarity search operation.

  Unlike the strategy followed in vp-trees, the vantage point does not have to be from inside the region. This means that we can use the same vantage point to partition the regions associated with the nodes at the same level. The mvp-tree takes this approach and uses more than one vantage points in the nodes for higher utilization.

  In the construction of the vp-tree structure, for each data point in the leaves, the distances between that point and all the vantage points on the path from the root node to the leaf node that keeps that data point are computed. In vp-trees, such distances are not kept. However, the mvp-tree keeps these distances for the data points in the leaf nodes in order to provide further filtering at the leaf level during search operation.

- GNAT [Bri95]. The Geometric Near-neighbor Access Tree (GNAT) is based on the philosophy that the data structure should act as a hierarchical geometrical model which reflects the intrinsic geometry of the underlying data. A k number of split points are chosen at the top level. Each one of the remaining points are associated with one of the k datasets (one for each split point), depending on which split point they are closest to. For each split point, the minimum and maximum distances from the points in the datasets of other split points are recorded. The tree is recursively built for each dataset at the next level.

- M-tree [CPRZ97]. The M-tree is a balanced and dynamic tree, which is built from bottom to top, creating a new root level only when necessary. This tree organizes the objects into fixed-size nodes, which correspond to regions of the metric space. Nodes of the M-tree can store up to M entries, which is the *capacity* of the nodes.

  For each indexed database element, one entry with format

$$entry(O_j) = [O_j, oid(O_j), d(O_j, P(O_j))]$$

is stored in a leaf node. In $entry(O_j)$, $oid(O_j)$ is the identifier of the object which resides on a separate data file, O_j are the feature values of the object and $d(O_j, P(O_j))$ is the distance between O_j and $P(O_j)$, which is the *parent* object of O_j.

An entry in an internal (non-leaf) node stores a feature value, O_r, also called a *routing object*, and a *covering radius*, $r(O_r) > 0$. The entry for a routing object O_r includes a pointer to the root of sub-tree and the distance from the parent object. The covering radius of a routing object O_r satisfies the inequality $d(O_j, O_r) \leq r(O_r)$ for each object O_j stored in the covering tree of O_r.

As any other dynamic balanced tree, M-tree grows in a bottom-up fashion. The overflow of a node N is managed by allocating a new node, N', at the same level of N, partitioning the $M + 1$ entries among the two nodes, and then promoting relevant information to the parent node, N_p. When the root splits, a new root is created and the M-tree grows by one level.

- Slim-tree [TTSF00]. The Slim-tree is a dynamic tree for organizing metric datasets in pages of fixed size. It shares the basic data structure with other metric trees like the M-tree where data is stored in the leaves and an appropriate cluster hierarchy is built on top. The Slim tree differs from previous MAMs in the following ways. First, a split algorithm based on the Minimum Spanning Tree (MST) is used which performs faster than other split algorithms without sacrificing search performance of the MAM. Second, a new insertion algorithm which leads to considerably higher storage utilization is applied to guide an insertion of an object at an internal node to an appropriate subtree. Third, an algorithm called Slim-down makes the metric tree tighter and faster in a post-processing step. This algorithm uses the "fat-factor" and the "bloat-factor" to measure the degree of overlap and improves the performance by minimizing overlaps between nodes.

- M+-tree [ZWYY03]. The M+-tree is a tree dynamically organized for large datasets in metric spaces which takes full advantages of M-tree and MVP-tree, with a new concept called *key dimension*, which effectively reduces response time for similarity search. It inherits M-tree's promotion mechanism, triangle inequality and the branch and bound technique. Furthermore, M+-tree fully utilizes the filtering twice idea used in MVP-tree and adopts the similar ideas of key dimension and the key dimension shift used in TV-tree in a novel way based

on the following concepts: (i) dimension can be ordered by their significance in a metric-space, and (ii) the active dimensions can be shift for enhancing the efficiency.

- PM-tree [TSS04]. The *Pivoting M-tree* (PM-tree) is an extension of the M-tree which explotes pivot-based ideas for metric region volume reduction. Each metric region of M-tree is described by a bounding hyper-sphere (defined by a center object and a covering radius). However, the shape of a hyper-spherical region is far from optimal since it does not bound the data objects tightly together thus the region volume is too large.

  In order to build the PM-tree, a set of p pivots P_t, which belong to the database, must be selected. This set is fixed for all the lifetime of a particular PM-tree index. Furthermore, a new attribute called HR is also included in the routing entry. This attribute is an array of p_{hr} hyper-rings ($p_{hr} \leq p$) where the t-th hyper-ring HR[t] is the smallest interval covering distances between the pivot P_t and each of the objects stored in leaves of the subtree. For the ground entries, a new attribute called PD is also included. This attribute stands for an array of p_{pd} pivot distances where the t-th distance PD[t] = d(O_i,P_t). Since each hyper-ring region defines a metric region containing all the object stored in the subtree, an intersection of all the hyper-rings and the hyper-sphere forms a metric region bounding all the objects as well. This new region is always smaller than the original M-tree region defined just by a hyper-sphere and bounds the indexed objects more tightly which, in turn, significantly improves the overall efficiency of similarity search.

However, the existing MAMs present several drawbacks for similarity-based indexing of multimedia databases. The static MAMs, for instance, do not support dynamic changes such as new insertions or deletions, whereas this is an essential requirement during the incremental construction of a multimedia database. Even though M-tree and its variants provide dynamic database access, the incremental construction of the indexing tree could lead, depending on the order of the objects or the choice of its pre-fixed parameters, to significantly varying performances during the indexing and querying phases.

2.3 Locality Sensitive Hashing

Locality Sensitive Hashing (LSH) [AI08] is an efficient indexing method to organize and query large-scale and high-dimensional database. The LSH algorithm relies on the existence of locality-sensitive hash functions. Let H be a family of hash functions mapping R^d to some universe U. For any two points p and q, consider a process in which a function h from H is chosen uniformly at random, and analyze the probability that $h(p) = h(q)$. Then, H is called (R, cR, P_1, P_2)-sensitive if for any two points $p, q \in R^d$ H satisfies the following conditions:

- if $||p - q|| \leq R$ then $Pr_H[h(q) = h(p)] \geq P_1$,
- if $||p - q|| \geq cR$ then $Pr_H[h(q) = h(p)] \leq P_2$

In order for a locality-sensitive hash family to be useful, it has also to satisfy $P_1 > P_2$. To put it simply, the principle of LSH is that nearby data points are hashed into the same bucket with a high probability while points faraway are hashed into the same bucket with a low probability.

An LSH family H can be used to design an efficient algorithm for approximate near neighbor search. However, one typically cannot use H as is since the gap between the probabilities P_1 and P_2 could be quite small. Instead, an amplification process which consists in concatenating several hash functions is needed in order to achieve the desired probabilities of collision. In particular, for parameters k and L (specified later), L functions $g_j(q) = (h_{1,j}(q), ..., h_{k,j}(q))$ are chosen, where $h_{t,j}$ $(1 \leq t \leq k, 1 \leq j \leq L)$ are chosen independently and uniformly at random from H. These are the actual functions used to hash the data points.

The data structure is constructed by placing each point p from the input set into a bucket $g_j(p)$, for $j = 1, ..., L$. To process a query q, we scan through the buckets $g_1(q), ..., g_L(q)$ and retrieve the points stored in them. Then, we compare their distance to the query point and report any point which is a valid answer to the query. Larger values of k lead to a larger gap between the probabilities of collision for close points (p_1^k) and far points (p_2^k). The benefit of this amplification is that the hash functions are more selective. At the same time, if k is large then p_1^k is small, which means that L must be sufficiently large to ensure that an R-near neighbor collides

with the query point at least once. On the other hand, a larger number of hash tables results in a heavier memory consumption, which is the significant drawback of LSH.

We want to give a solution to the K Nearest Neighbor (KNN) problem. Let q be the image query, i.e. the image used as example for searching the elements which are most similar to q. Then, the KNN problem consists in retrieving the k most similar elements from the database to the image query given. However, LSH, by its nature, does not solve the KNN problem but the ϵ-near neighbor problem. Therefore, it answers a range search, gathering points within a predefined ϵ-distance from the query point. In order to adapt LSH for K nearest neighbor retrieval it has to be configured with a significantly large ϵ, which occasionally leads to a very large number of false positives [LJA11].

Chapter 3

Hierarchical Cellular Tree

In [KG07], a novel indexing technique, called Hierarchical Cellular Tree (HCT), is presented and has been designed to bring an effective solution for indexing large multimedia databases. The elements are partitioned depending on their relative distances and stored within cells on the basis of their similarity. As its name implies, HCT is a hierarchical structure, which consists of one or more levels, and each level in turn holds one or more cells. Each cell has an element as a representative or nucleus, which is contained in a cell in the upper level except for the cell held by the top level. Figure 3.1 shows an HCT example extracted from [KG07] over a database containing 18 elements. In this example, the database elements have been divided into 6 different cells (A, B, C, D, E, F) on the ground level. Each cell's representative (d for cell A, c for cell B, e for cell C, etc.) have been clustered in 3 different cells (A, B, C) on the first level in turn. Finally, the top cell from the top level consists of each cell's nucleus (c for cell A, a for cell B and b for cell C) from the lower level.

Furthermore, HCT is a self-organized tree, which basically means that the operations (item insertion, removal, etc.) are not externally controlled, but they are carried out according to some internal rules. The indexing structure is created in a bottom-up fashion, which means that the elements are always inserted in the ground level and, due to these insertions, the cells may suffer from mitosis or their nucleus may change, so these alterations are spread towards the top of the tree.

The HCT is a MAM-based indexing technique which was developed in order to provide efficient solutions to the aforementioned shortcomings of the indexing algorithms in Chapter 2. Among all indexing structures presented, M-tree shows the highest

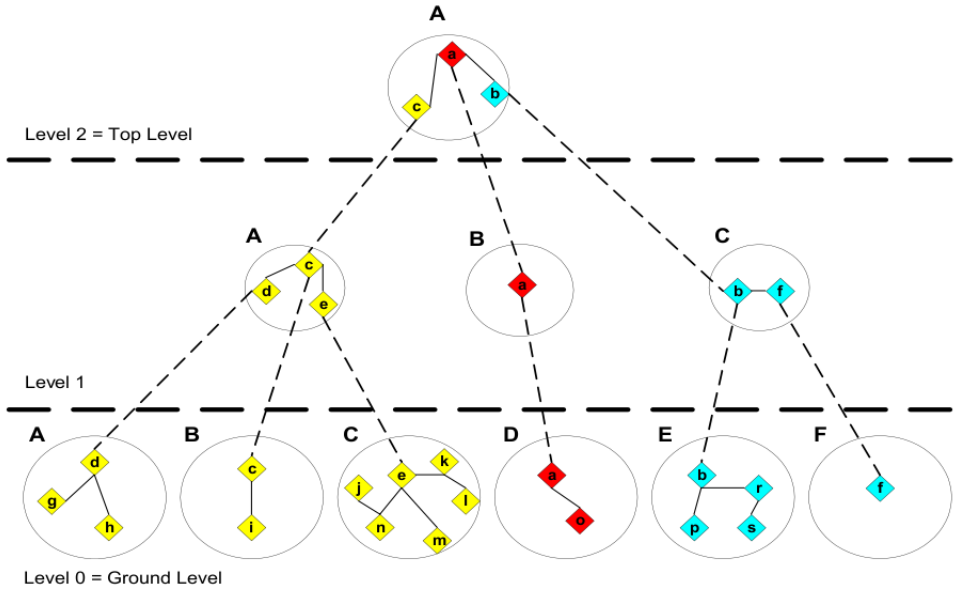


Figure 3.1: HCT example

structural similarity to HCT, such as the following:

- Both indexing schemes are MAM-based and have a similar hierarchical structure, i.e. levels.
- They are both constructed dynamically in a bottom-up fashion.
- Except the top level cell, each cell is represented by a nucleus object in the upper level.

However, there are several differences between both indexing schemes:

- HCT was designed for indexing multimedia databases where the content variation is seldom balanced and it is, therefore, an unbalanced tree optimized for achieving highly focused cells, which may exhibit variations on size and density.
- HCT does not depend on a maximum (fixed size) capacity M as the M-tree does. Therefore, its performance does not depend on a "good" choice of a prefixed parameter with respect to the database size. Thus, HCT has no limit for the cell size as long as the cell keeps a definite compactness measure.

- HCT does not measure the cell compactness by using a single nucleus item alone (the distance of the nucleus to the farthest object), but it uses all cell items and their minimum distances to the cell to define a function which represents a model for the cell compactness.

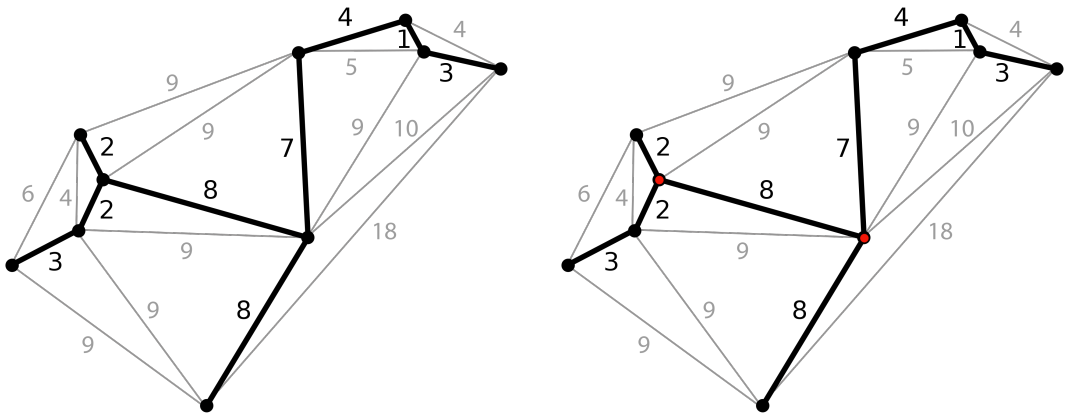
- The insertion processes differ significantly in terms of cell-search operations. M-tree insertion operation is based on "Most-Similar Nucleus" cell search whereas a Pre-emptive cell search algorithm was proposed for the HCT to perform an optimum search for the target cell especially for large databases.

- HCT has a totally dymamic approach since any operation can change the current cell nucleus to a new better one. On the contrary, M-tree has a conservative structure since the cell nucleus is not changed after an insertion or removal operation.

While Section 3.1 describes the cell structure in detail, Section 3.2 describes the level structure. Next, the different operations which can be carried out over an HCT (item insertion and removal) are explained in Section 3.3. Finally, a retrieval scheme which takes advantage of the indexing structure is detailed in Section 3.4.

3.1 Cell Structure

A cell is a basic container structure where similar database elements are stored. The ground level cells span all items in the entire database. Furthermore, each cell carries a tree structure, a minimum spanning tree (MST) [Kru56], which refers to the database objects (their database representations and basically their descriptors) as its MST nodes. If we have a connected, undirected graph and we have assigned the dissimilarity measures between each pair of elements as the weight of the edge that connects these elements, the minimum spanning tree is the subgraph that connects all the vertices together with the minimum cumulative total weight. Figure 3.2a shows an example of the spanning tree obtained from a graph containing 10 elements with their respective distances stored in the edges. This internal MST is used to keep the minimum cumulative total dissimilarity distance of each individual item to the rest of the elements in the cell. Furthermore, it is also used in order to assign the nucleus item.



(a) MST example (b) Cell Nucleus example

Figure 3.2: MST Formation (a) and possible candidates for Cell Nucleus (b)

The cell nucleus is the element which represents the owner cell on the upper level. Since these elements are used during the top-down search for item insertion in order to decide into which cell the element should be inserted or for query requests, it is essential to promote the best item for this representation in the upper level at any instant. Therefore, in [KG07] it is proposed to choose the element having the maximum number of branches (connections to other items) in the MST. According to this way of proceeding, for the MST example given in Figure 3.2a, there would be two candidates for the cell nucleus, which are those marked by a red point in Figure 3.2b, since both elements have three branches. The cell nucleus is a dynamic feature which is verified and, if necessary, updated whenever an operation is performed over a cell in order to guarantee the best representation of the dynamically changing cell.

Another dynamic cell feature is the cell compactness. This value quantifies how focused or compact the clustering for the items within the cell is. It is calculated as a function f of the following cell parameters: the mean μ_C and the standard deviation σ_C of the MST branch weights (w_C) of cell C, the covering radius (r_C), which is the distance from the nucleus to the furthest element in the cell, the maximum MST branch weight, and the number of elements N_C. Therefore, according to [KG07] an estimation of the compactness feature of the cell, CF_C, can then be formed as follows:

$$CF_C = f(\mu_C, \sigma_C, r_C, max(w_C), N_C) \geq 0 \tag{3.1}$$

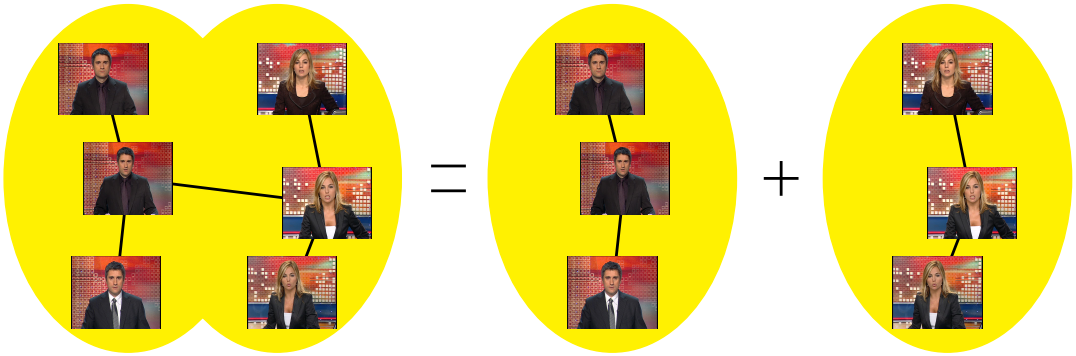


Figure 3.3: Sample mitosis operation over a mature cell

This parameter, the cell compactness, plays a key role in deciding whether or not to perform mitosis within the cell at any instant. An HCT cell can suffer from mitosis only if (i) it contains a minimum number of elements, named maturity cell size N_M, in order to have enough statistical data to be considered reliable, and (ii) it is not compact enough, i.e., the cell compactness is over the current level compactness threshold $CThr_L$ ($CF_C > CThr_L$). More details about the compactness threshold are given in Section 3.2.

Apart from the features, as it has been commented before, a cell can undergo a process called mitosis, which basically means that the cell is divided into two newborn child cells. When mitosis is granted, MST is again used to decide how to split the owner cell and the natural choice for this is to do it by breaking the branch with largest weight of the MST. Then, each of the newborn child cells is formed by using each of the MST partitions. In Figure 3.3 a sample mitosis operation is illustrated.

3.2 Level Structure

As mentioned before, the HCT has a hierarchical structure, which is formed by one or more levels. Apart from the top level, which contains only one cell, each level holds more than one cell, which have resulted from various mitosis operations that have occurred on that level. The elements belonging to a particular level are the representatives for each cell from the lower level except, obviously, the ground level, which contains the entire database. Therefore, the HCT allows a multiscale decomposition since each level is a representation of the database. The lower the level, the more detailed the representation. The tree grows one level upwards whenever a mitosis

occurs in the top level cell.

Each level is the responsible for maximizing the compactness of its cells. With this purpose, each level updates its own compactness threshold ($CThr_L$), which is updated after a number of insertion or mitosis operations by applying the median operator over the compactness values (CF_C) of its mature cells (S_M). This valued is divided by a factor k_0, called trend factor, on which the targeted enhancement will depend:

$$CThr_L = \frac{1}{k_0} Median(CF_C | \forall C \epsilon S_M)$$

Due to new elements insertion, the global compactness may suffer from degradation, but each level tries to achieve a trend of improving compactness in due time.

3.3 HCT Operations

There are two external operations that can be carried out over an HCT: item insertion and item removal. While item removal is a cell-based operation, i.e. items belonging to a same cell can be removed in a single step, item insertion is performed item by item due to the dynamical construction of the tree.

3.3.1 Item Insertion

Whenever an item insertion is performed, the first thing to do is to find the most suitable cell to which the element must be appended in the ground level. In order to perform this operation in an efficient way, a novel search algorithm called Pre-emptive cell search is proposed in [KG07], instead of an exhaustive search or the traditional cell-search technique called MS-Nucleus (Most Similar Nucleus). While an exhaustive search would imply a higher computational time, the MS-Nucleus may not retrieve the best candidate since it assumes that the closest nucleus object yields the best subtree during descend. This approach may reach a local minimum instead of the best cell to be appended. However, the MS-Nucleus perfoms the best insertion time since this technique implies the minimum number of comparison operations by simply choosing the best cell candidate at each level.

On the other hand, the proposed Pre-emptive cell search adopts a compromise between the computational time and the cell reached during descend. Thus, the best

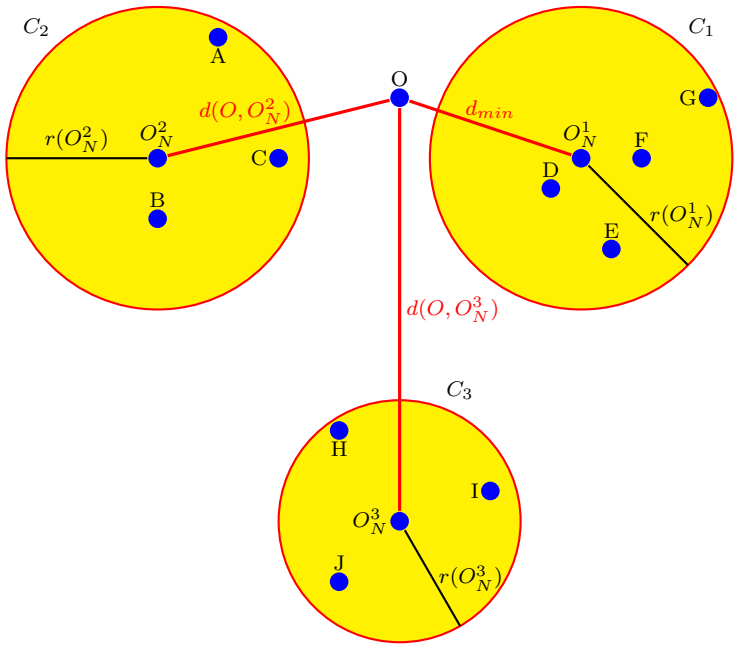


Figure 3.4: Sample Pre-Emptive cell search

owner cell to which the element must be inserted is reached although the required time is higher than the MS-Nucleus time. Furthermore, the construction of the HCT should be considered crucial because the efectiveness of the retrieval results of any posterior query over the HCT will depend on it. The Pre-Emptive cell search algorithm performs a pre-emptive analysis on the upper level to find out all possible nucleus objects which might yield the closest (most similar) objects on the lower level. If d_{min} is the distance to the closest nucleus in the upper level, then all nucleus items O_N^i that obey $d(O, O_N^i) - r(O_N^i) < d_{min}$, where O is the object to be inserted and $r(O_N^i)$ is the covering radius of the cell represented by the nucleus O_N^i, are fetched for tracking. For example, in Figure 3.4, although cell C_1 has the closest nucleus O_N^1 from the query element O, cell C_2 is not discarded because $d(O, O_N^2) - r(O_N^2) < d_{min}$ and it can also yield to the closest element. On the other hand, since $d(O, O_N^3) - r(O_N^3) > d_{min}$, cell C_3 can be discarded and it is not necessary to descend for its subtree. If MS-Nucleus was carried out instead of Pre-Emptive cell search, the algorithm would have descend for the cell C_1 subtree since it contains the closest nucleus in that level.

Pre-Emptive cell search is a recursive algorithm that terminates its recursion one level above the target level, i.e. in the first level for new elements to be inserted in the ground level. It achieves the optimum insertion in the sense that the owner cell

retrieved in the target level (ground level for external insertions) presents the closest nucleus item with respect to the item to be inserted.

Once the element is appended to the retrieved cell, this cell becomes subject to a generic post-processing check. First, the cell is examined for a mitosis operation. If the cell is mature, i.e. it contains a minimum number of elements, and its compactness is above the compactness threshold, then it is split into two newborn child cells. Therefore, the parent cell must be removed from that level and the two new child cells are inserted instead. Furthermore, the old nucleus (of the parent cell) must be removed from the upper level. The two new representatives of the child cells must be also inserted in the upper level by using the same insertion method based on the Pre-Emptive cell search but having the upper level as the target level instead of the current one. In case mitosis is not performed, it is necessary to check if the nucleus has changed due to the insertion. In such a case, the old nucleus is removed from the upper level and the new one is inserted by using the same pre-emptive technique.

The HCT construction over an entire database is the result of inserting dynamically each of its elements one after the other in the ground level. As new elements are inserted some cells may be split, resulting in a tree that grows in a bottom-up way. Whenever the top cell is split, a new top level is created above it with a new cell containing the two new nucleus items resuting from the division of the old top cell. Figure 3.5 shows an illustrative example of HCT construction, extracted from [KG07], where the elements are represented by colored circles and the similarity between two elements is given by how similar in color they are. First, all the elements are inserted in the unique cell of the HCT, which is created in the ground level, until this becomes mature and not compact enough. In this example, this happens when the yellow ball labeled as 1 is inserted. As a result, the cell is split into two new cells and a new top level is created. The new nuclei of each child cell are computed (yellow ball labeled as 1 and red ball labeled as f) and inserted in the new top cell. For each new element to be inserted, the most similar cell is retrieved by using the Pre-emptive cell search algorithm. Therefore, balls labeled as 2, 3 and 5 are inserted in the "red-blue" cell and the ball labeled as 4 in the "yellow" cell. As a result of the insertion of the ball labeled as 5, the nucleus have changed, so the old nucleus is removed from the upper level and the new one is inserted. Then, when the new element labeled as 6 is inserted in the "red-blue" cell, this is split because it is mature and not focused enough. Therefore, two newborn child cells are created and the parent cell is removed

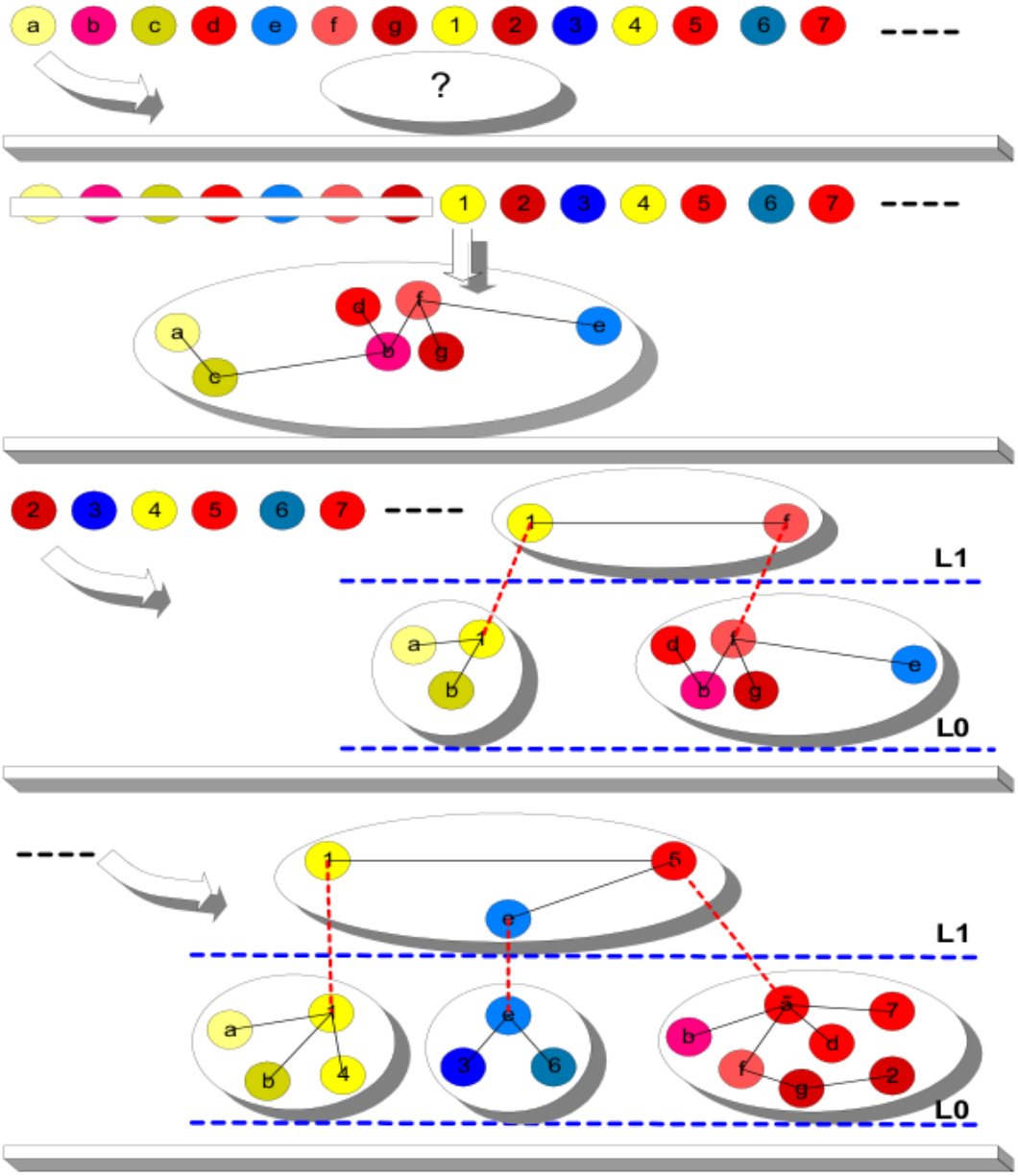


Figure 3.5: Sample HCT Construction

from the ground level. The old nucleus is removed from the upper level and the new ones are inserted instead. This process continues until all the elements of the database are inserted.

3.3.2 Item Removal

This is an operation that does not require any cell search algorithm. When some elements of the cell have to be removed, they are taken out of the cell and then the cell becomes subject to a generic post-processing check. As a result of the items removal, three cases are possible:

1. *The cell is depleted* (No elements remain in the cell). In this case, if the owner cell is the top cell, it is removed and the top level is also removed from the HCT. On the contrary, if the cell to be removed does not belong to the top level, then the old nucleus must be removed from the parent cell in the upper level.

2. *The cell is split.* An item removal can result in a cell split if the cell is still mature but no longer compact enough once the element has been removed. In this case, the old nucleus must be taken out from the upper level, and the two representatives of the two newborn child cells have to be inserted in the upper level by using the insertion algorithm explained in Section 3.3.1.

3. *The cell is not split.* In this case, it is necessary to verify the need for the cell nucleus change. In such a case, the old nucleus has to be removed from the upper level and the new representative must be inserted instead by also using the insertion algorithm from Section 3.3.1.

3.4 Retrieval scheme over HCT

In [KG07] a retrieval scheme called Progressive Query (PQ) [KG05] is proposed to be used over the Hierarchical Cellular Tree. This searching technique consists in performing periodical sub-queries over subsets of database items and allows the user to interact with the ongoing query process. The size of each subset is determined with respect to a suitable unit such as time to human perception. The order in which the comparisons are done is given by the Query Path (QP), which consists of the several subsets which the database has been divided into. Although the PQ can work within nonindexed database, the most advantageous way to perform PQ is to form the QP according to an indexing structure such as the Hierarchical Cellular Tree since an indexing scheme allows to form a QP in which the most relevant elements can be retrieved by earlier sub-queries. Next, the Query Path formation process is detailed by using an example extracted from [KG07] which is illustrated in Figure 3.6. In this

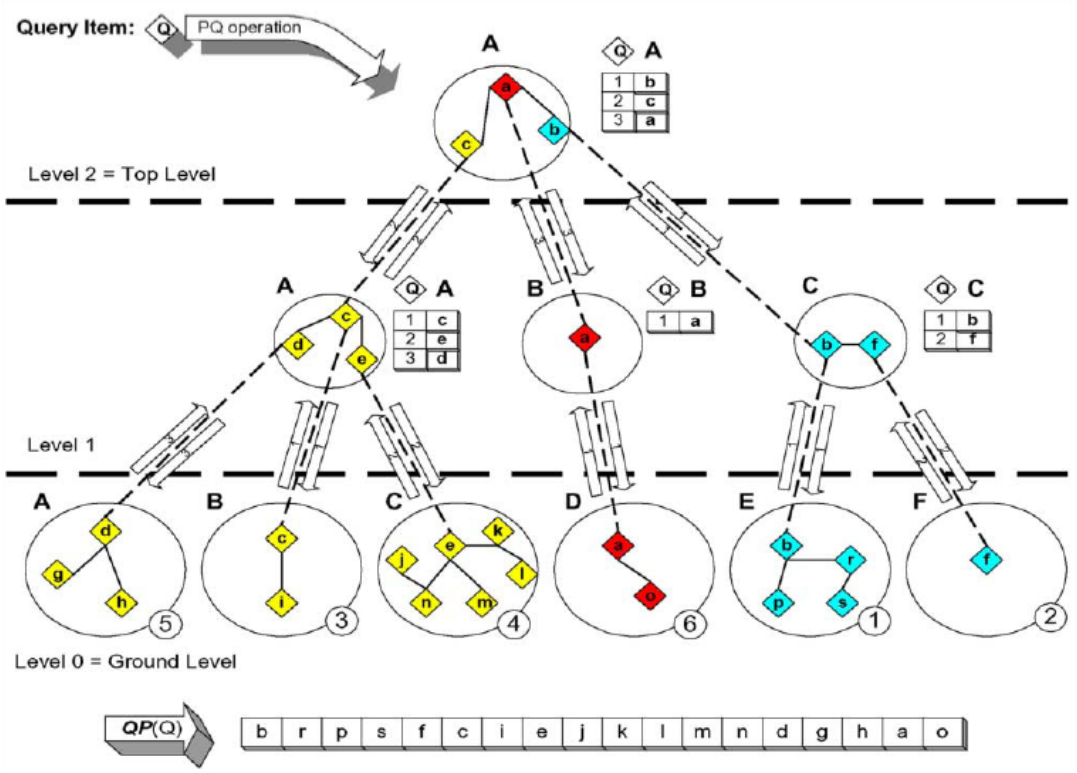


Figure 3.6: QP formation on a sample HCT body

example, the element b from the top cell is the most similar element to the query item, so we visit the cell C from the lower level. Then, the two elements hosted by this cell are compared with respect to the query item and since element b is the most similar again, the cell E from the ground level is appended to the query path. Then, we come back to the cell C from the upper level and we visit the following most similar element, i.e. element f. Therefore, the cell F from the ground level is appended to the query path. Since there are no elements left in the current cell, we come back to the top cell and visit the following most similar element, i.e. element c. Therefore, we visit cell A from the lower level. Since element c is again the most similar element, cell B from the ground level is appended. This process goes on until all the ground level are appended to the query path. In such a way, cells C, A and D from the ground level would form the QP tail.

Chapter 4

Modifications to the original HCT

In this chapter we present some modifications to the original Hierarchical Cellular Tree. First of all, we propose a redefinition of the covering radius cell value and a procedure to estimate it which are detailed in Section 4.1. Furthermore, we propose to use the Preemptive Cell Search algorithm at any level in order to make the HCT building more robust (see Section 4.2). Finally, we propose a few searching techniques and establish a criteria for determining the number of cells to be considered when an element retrieval is performed in Section 4.3.

4.1 Covering radius

The covering radius is defined in [KG07] as the distance from the nucleus to the furthest element in the cell. However, we consider that we should take into account not only the elements belonging to the corresponding cell, but also all the elements belonging to its subtree, i.e. all the elements from the ground level cells that are hanging on the afore-mentioned cell. Thus, the covering radius defined in [KG07] is an approximation by defect, i.e. the covering radius value computed is always less than or equal to its actual value. Figure 4.1 presents an example in which all the elements in the cells C_1, C_2, and C_3 should be considered in order to compute the covering radius for the cell C_9, instead of using only the elements belonging to the cell C_9. Since the computation of the exact covering radius, i.e. the distance from the nucleus to the furthest element in the subtree, can have a high computational cost

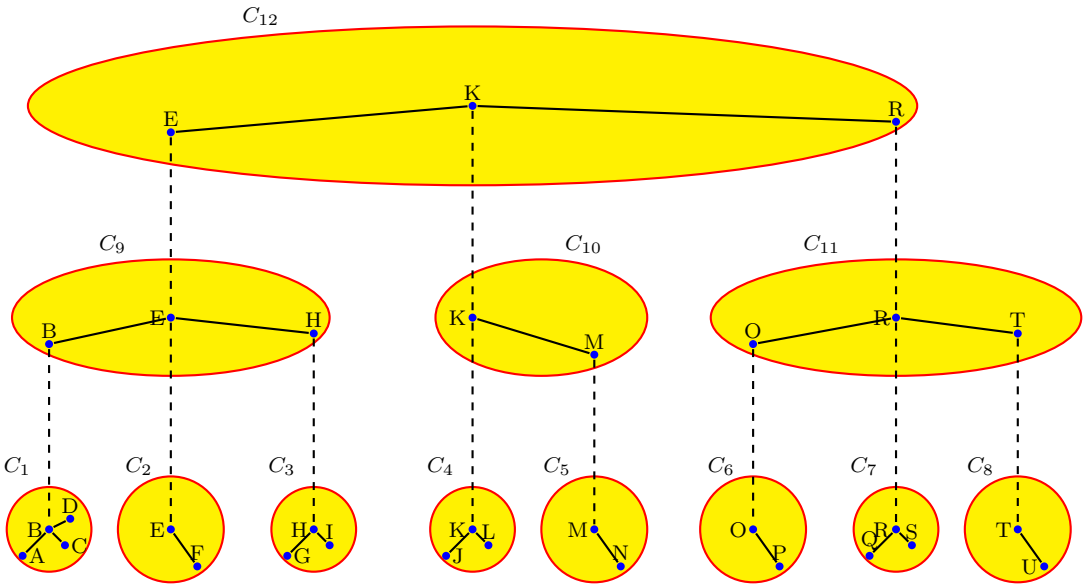


Figure 4.1: Covering radius

during the dynamic HCT construction, we have decided to approximate it according to the following equation:

$$r_C = max(r_C(S_N), d(O_1, O_N) + r_C(S_1), ..., d(O_M, O_N) + r_C(S_M)) \qquad (4.1)$$

where $r_C(S_i)$ refers to the covering radius of the son cell of element O_i with $1 \leq i \leq M$, M are the number of elements in the cell, and O_N refers to the nucleus object. Equation 4.1 gives an approximation by excess of the covering radius value, i.e. the covering radius computed is always greater than or equal to the actual covering radius value. This approximation has been also used in [CPRZ97] for M-tree and allows to keep the covering radius updated after new elements insertions with a low computational cost due to its recursivity. Each covering radius only depends on the graph of the own cell and the covering radius of its son cells from the lower level. For the cells belonging to the ground level, the covering radius is computed as the distance to the furthest element in the cells from the nucleus instead of using the approximation given by Equation 4.1. Using the same example as before (see Figure 4.1), the covering radius for the cell C_9 is computed as:

$$r_C(C_9) = max(d(B, E) + r_C(C_1), r_C(C_2), d(E, H) + r_C(C_3))$$

With this change in the covering radius definition, in spite of using the proposed approximation by defect, we can guarantee that no HCT branch will be wrongly discarded since we are using an approximation by excess for the covering radius. Therefore, the most suitable cell will be found whenever a new element is inserted. As we will see in Chapter 6, a better HCT building will result in a better performance of the image retrieval system over the HCT.

4.2 HCT building

In [KG07] it is proposed to adopt a hybrid approach to build the HCT, using the Preemptive Cell Search algorithm only in a certain number of the uppermost levels and the Most Similar Nucleus technique for the lowest ones. However, we have decided to build the HCT by using the Preemptive Cell Seach algorithm over all the levels, since we consider that the HCT construction is a keystone to guarantee the quality of the results for the future query requests to be launched on the indexed database. That is the reason why we have also considered in Section 4.1 that the covering radius for any cell has to take into account all the elements holding to that cell, not only the elements belonging to the afore-mentioned cell.

Furthermore, we propose a method for updating the covering radius to its actual value for all the cells of the HCT in order to improve the search time of the query requests based on the Preemptive Cell Search algorithm. In this way, since the covering radius is no longer an approximation by excess represented by Equation 4.1, the number of cells which will be visited is granted to be less or equal than when the approximation by excess is used. As a consequence, the number of comparisons between the query element and an element from the database is minimized, so the searching time will be also reduced. However, the approximation by excess is used again when a new element has to be inserted in order to compute the covering radius for the cells affected by the insertion process. This update method proposed for the covering radius can be performed not only after the HCT construction to reduce the retrieval times, but also during it in order to reduce the insertion time of future elements to be indexed. However, using this method every a few number of insertions results on a worse global HCT building time.

4.3 Searching techniques over HCT

We want to give a solution to the K Nearest Neighbor (KNN) problem. Let q be the image query, i.e. the image used as example for searching the elements which are most similar to q. Then, the KNN problem consists in retrieving the k most similar elements from the database to the image query given. As seen in Section 3.4, the Progressive Query (PQ) is proposed in [KG07] as a retrieval scheme over the HCT. As it will be shown in Chapter 6, the K most similar elements do not belong to the Query Path (QP) initial part and, therefore, too many progressive sub-queries are necessary to be performed in order to achieve satisfactory results. As a consequence, these progressive sub-queries result in a too much high searching time. Thus, PQ fails in solving the KNN problem in an efficient way.

That is the reason why we have implemented the following searching techniques in order to improve the PQ performance:

- Most Similar Nucleus: This technique consists on descending through the HCT branch which gives the most similar element at each level. Therefore, the nearest element to the query image from the top cell is search and its son cell from the lower level is visited discarding the rest of cells of that level. This methodology is followed until a ground level cell is reached. The advantage of this searching technique is that the image retrieval is performed very fast. On the other hand, the Most Similar Nucleus can lead us to a local optimum far away from the optimal retrieved elements. This is because descending through the HCT branch which gives the most similar element at a given level does not guarantee that the ground level cell whose nucleus is the most similar one to the query image among all the ground level cell nuclei will be reached.

- Preemptive Cell Search: This searching technique is based on the same idea as the insertion algorithm is. Therefore, the covering radius cell value is used to discard any HCT branch in which we can guarantee that the most similar element to the query image will not be found through it. It is for this searching technique that the covering radius redefinition proposed in Section 4.1 becomes essential, as well as the method for updating the covering radius to its exact value in order to minimize the number of visited cells and, therefore, reduce the searching time. By using this searching technique, the ground cell whose

nucleus is the nearest one to the query image is always found. However, this fact does not mean than the most nearest element from the database will be retrieved since this element can belong to another cell. Anyway, this problem is inherent to any clustering algorithm, but the probability of not retrieving the most similar elements can be reduced by considering not only one cell but a few cells. On the other hand, regarding the searching time, the Preemptive Cell Search performance can never be better than the Most Similar Nucleus performance since the number of HCT branches analyzed is always greater than or equal to the number when the Most Similar Nucleus is applied, which only descends for one of the HCT branches. Therefore, since more cells are visited at each level, more comparison operations are also necessary, which results in a worse retrieval time. It is only when at each level all the cells except the one with the nearest nucleus can be discarded by using the covering radius that both searching techniques gives the same performance.

- Hybrid: This searching technique is a hybrid of the two afore-mentioned techniques: (*i*) the Most Similar Nucleus and (*ii*) the Preemptive Cell Search. This technique tries to take advantage of the strong points of each one, i.e. the searching time of the Most Similar Nucleus and the retrieved results of the Preemptive Cell Search. With this objective, the latter technique is performed over the uppermost levels, in which an error can be critical for the search performance, whereas the Most Similar Nucleus is applied on the lowest levels, in which the cells are expected to be more focused and, therefore, their nucleus are expected to lead to the cell hosting the most similar elements to the query image. When this technique is used, the number of levels over which the Preemptive Cell Search algorithm will be performed must be specified.

- Exhaustive: The exhaustive search technique consists in a linear search over all the elements of the database. Only this technique can guarantee that all the nearest elements to the query image will be found. On the other hand, this technique also would result in a linear search time (with respect to the number of images in the database) and, therefore, cannot be considered for large databases unless the user does not mind the retrieval time. In such a case, we would not take advantage of the hierarchical clustering. However, this searching technique can also be used on an given level and be combined with other techniques on the lowest levels as in the Hybrid approach.

When a search is performed over the HCT, the user selects the number of results expected to be retrieved. Since the query element can be located near a cell border, the cell whose nucleus is the nearest element to the query may not contain the most similar elements. That is the reason why we propose considering a minimum number of cells (MIN_C) regardless of the searching technique used. For example, if we set the minimum number of cells to be visited to 3 and the user expects to retrieve 10 elements, the 3 cells whose nucleus are the 3 nearest elements to the query will be taken into account although the most similar cell may contain more than 10 elements.

We also propose that the number of cells being considered depends on the number of expected results (K). Therefore, the cells considered must host at least twice the number of elements expected (C_{2K}). The greater the number of elements considered, the higher the likelihood of the exact K nearest neighbours to be found. On the other hand, considering more elements will also result in an increase of the searching time.

According to the two above considerations, the number of cells (N_C) considered will be determined by the following equation:

$$N_C = max(MIN_C, C_{2K})$$

Therefore, we have two cases:

- Case 1: The number of cells hosting at least twice the number of elements expected, i.e. C_{2K}, is smaller than the minimum number of cells to be considered. In such a case, MIN_C cells are taken into account.
- Case 2: The number of cells hosting at least twice the number of elements expected, i.e. C_{2K}, is greater than the minimum number of cells to be considered. Therefore, C_{2K} cells are taken into account.

Finally, the elements hosted by the considered N_C cells are sorted according to the distance from each element to the query element. The results given to the user, therefore, no longer keep the cellular structure. The motivation is that we consider the clustering as a method for speeding up the searching process. The most important thing from the user point of view is how good the retrieval system is, and not how well the elements have been clustered.

Chapter 5

Implementation

The Hierarchical Cellular Tree (HCT) has been implemented in C++ language in a development platform of the Video and Image Processing Group called ImagePlus. The HCT implementation in this platform involved respecting the different traits for which ImagePlus is characterized:

- Cross-platform: Linux, Windows (MinGW) and MacOSX (darwin).
- 64 bits ready platform
- Modular structure for a faster and better development
- Use of a Unit Test Framework based on the Boost Test Library as a release validation system
- Configurable build system based on SCons that also allows to execute Unit Tests in build-time
- Well formated documentation based on Doxygen
- Open to external libraries: Boost C++ Libraries
- Generic programming techniques

The implementation of the HCT has been divided into three classes: (*i*) Cell, (*ii*) Level, and (*iii*) HCT. These different classes are result of the intrinsic HCT structure since it consists of different cells that are hierarchically organized in different levels.

Each of these elements is detailed in Section 5.1, Section 5.2, and Section 5.3 respectively. In the implementation we have used STL containers, such as vectors, sets and maps, whenever it has been feasible.

5.1 Cell Implementation

As explained in Section 3.1, a cell is a basic container of similar elements. First of all, it is necessary to know (*i*) the data type of the elements we are working with (*VDModel*) and (*ii*) the (dis)similarity measure that is used for comparing the elements (*distance_functor*). Thus, the class Cell depends on these two templates. Although the Hierarchicall Cellular Tree will be used in this project with some visual descriptors (and its respective similarity distances) defined in MPEG-7, this class has been implemented in this way in order to be useful for any other data type. Therefore, it will also allow to index a cloud of k dimensional points according to the Euclidean distance for instance.

To represent the relationships between the elements of the cell we have used the Boost Graph Library [SLL01]. Thus, the most important member of a cell is the graph, which consists of vertices and edges. Each vertex represents an element of the database. In order to avoid a great number of I/O operations not only during the construction, but also after for each query request, the own elements have been stored in the vertices. Therefore, we have used different vertex properties for different kind of information:

- property_index1. Except for the ground level, each element of a cell represents an entire cell of the lower level. Thus, we store the pointer to the cell son for each element of the cell in this vertex property.
- property_index2. This is the vertex property where the element value is stored. Its type is the same as the referred by the *VDModel* template.
- property_name. In general, this vertex property will be used to identify the element. In our context, it refers to the uri name of the XML containing the visual descriptors of the respective element, as well as the uri name of the image filename among other additional information. Although we have the descriptor values stored in the vertex, these uri names are used for returning the results after a query request.

- property_key. In general, this property is not necessary and can take the same value as the property_name. It is an identifier for the element. In our context, this property is used to store the identifier of each element for an external (not local) database. In particular, it will be useful when the Fedora Commons Repository [LPSW06] has to be used, since each digital object of this tool has associated a unique Persisten ID (PID) [GiNVPT$^+$10].

Apart from the vertices, the edges are the other basic element of a graph. They connect each pair of vertices in each cell and have an associated value which represents how similar each pair of elements in the cell are. This similarity value, which is stored in each edge, is obtained by computing the similarity measure given by the *distance_functor* template between the two vertices belonging to the given edge.

Another member of the cell is the Minimum Spanning Tree (MST). Once we have the graph, we can obtain it by applying the *kruskal_minimum_spanning_tree* method implemented in the Boost Graph Library [SLL01]. Then, we propose as nucleus the element which has the maximum number of branches (connections to other elements) in the MST. This element will be the representative of the cell in the upper level. The Minimum Spanning Tree is also used for computing the mean and variance of the edge weights, and the maximum edge weigth, all of them used for obtaining the cell compactness. According to Equation 3.1, the lower the edge weights are, the lower the compactess is and, therefore, the lower the probability of the cell to be split. As seen in Section 3.1, this parameter also depends on the covering radius and the number of elements belonging to the cell.

The covering_radius is a cell member which plays an important role in the HCT building since it is used by the Preemptive Cell Search Algorithm when a new element is inserted to the indexed structure. Recall that we have proposed a redefinition of the covering radius which takes into account all the ground level cells belonging to the subtree which has the given cell as root and an approximation by excess of the actual value (see Section 4.1). Therefore, we have implemented both the original covering radius (defined in [KG07]) and the proposed approximation in order to compare both versions in Chapter 6.

Another element which is stored in each cell is a pointer to its parent cell except for the top cell since there are no more levels above it. Therefore, in each cell there is

also hierarchical information. Each element from a cell knows which cells represents from the lower level and each cell knows which is the cell from the upper level which its nucleus belongs to. According to Figure 4.1, from cell C_9, vertex B has a pointer to cell C_1, vertex E to cell C_2 and vertex H to cell C_3. Furthermore, cell C_9 has a pointer to cell C_{12}, which is its parent cell.

Apart from a default constructor for an empty cell, a constructor for a cell with one element has been also implemented. The prototype of this constructor is:

```
Cell(VDModel& new_elem, std::string uri_name,
     std::string key = "-", Cell* son = NULL)
```

where `new_elem` is the new element to be indexed, `uri_name` is the pathname of the element inserted, `key` is the element identifier, and `son` is a cell pointer to the son cell. This parameter is used when a parent cell is created after the splitting of a cell in the lower level. In this case, the son cell creates the parent cell and gives it information about which cell is represented by this new element in the parent cell. In this particular case, a graph with only a vertex is build and the covering radius depends on the level which the cell belongs to. Thus, the covering radius is 0 if the cell is from the ground level otherwise it is the son cell's covering radius.

Since the Hierarchical Cellular Tree allows a dynamic approach in which the elements are inserted one after the other, the cell also needs an insertion method. Thus, when a new element has to be inserted in the cell, a vertex is added to the graph including the element's content (the data type value, the uri name, the key identifier and a pointer to the son cell if it has one), i.e. the vertex properties. Once the vertex has been added, the similarity distance from this new element to each element belonging to the cell is computed and, therefore, the graph is completed by adding the edges with these computed weights. Then, the Minimum Spanning Tree (MST) is updated by recomputing it over the new graph. Due to the variation of the MST, all the cell parameters which are related to the MST (the mean and variance of edge weights, the maximum edge weight, the nucleus, the covering radius, and the compactness) must be recomputed.

Finally, a method for updating the cell parameters has also been implemented. This method is called when either an element has been removed from a cell or a cell has

been divided into two newborn cells. In the former case, after being the vertex and its edges removed from the graph, the MST must be recomputed and also all the cell parameters depending on it. In the latter case, once the MST has been split by removing the edge with the maximum weight, two new cells with two new MST are obtained so the parameters for each newborn cell have to be also updated.

5.2 Level Implementation

A level is a basic container of cells that allows the HCT to be built in a hierarchical way. We have implemented it as a set of pointers to the cells which belong to the same level in order not to duplicate the same information. The data information is only stored at the cell scale. The two higher level classes, i.e. level and HCT classes, always work with pointers to the most basic container, i.e. the cell. Thus, a Level object will depend on a CellModel as a template. For example:

```
typedef Cell<ColorStructure<InputType>, cs_dist> CellColorStructureType;
Level<CellColorStructureType> level;
```

where `CellColorStructureType` is a type of cell including elements which are ColorStructure descriptors, `cs_dist` is the distance function used to compute the similarity between two elements described by the ColorStructure feature, and `level` is a Level object which depends on the previous defined Cell type.

Apart from the set of cell pointers, there are another three parameters:

- Maturity size: This parameter represents the minimum number of elements to be held by a cell to consider it mature.
- Number of insertions: It represents the number of elements that have been inserted on that particular level so far.
- Compactness threshold: Each time an element is inserted in a mature cell, the updated compactness value of the cell is compared with the compactness threshold in order to decide whether or not the cell must be split. As said before in Section 3.2, the compactness threshold is updated after a number of insertion operations.

A constructor with the maturity size as an input parameter has been implemented. This constructor is called each time the top cell is split and a new top level is to be created. There is also a method that allows changing the maturity size of the level since in [KG07] it is proposed to have a different value for the top level. Therefore, each time a new top level is created, the maturity size of its lower level, i.e. the former top level, must be changed. There is no need of an insertion or removal method since the STL container set has its own methods (insert and erase) which are used whenever a cell must be either inserted or removed from a level.

5.3 HCT Implementation

The Hierarchical Cellular Tree has been implemented as a vector (STL container) of Level objects. The HCT constructor has only two input parameters: (*i*) the maturity size of an inner level (`m_size`), and (*ii*) the maturity size of the top level (`m_size_top_level`):

```
HCTree(uint64 m_size = 7, uint64 m_size_top_level = 7)
```

This section has been divided into four subsections according to the different operations which can be performed over an HCT: (*i*) the insertion methods, (*ii*) the removal methods, (*iii*) the fitness check operations, and (*iv*) the proposed update method for the covering radius.

5.3.1 Insertion methods

Once the HCT object has been created, the Hierarchical Cellular Tree is built by calling an insertion method in a dynamic way for each element of the database to be indexed. However, we have to differentiate between two insertion methods:

- A public method used for inserting new elements to be indexed. The element is inserted in a cell from the ground level. The syntaxis is the following one:

```
void insert(typename CellType::VDType& new_elem,
            std::string uri_name, std::string key = "-")
```

where `new_elem` is the new element to be indexed, `uri_name` is the pathname of the element inserted, and `key` is the element identifier.

- A private method which cannot be controlled externally. This method is called automatically when a cell nucleus has to be inserted in the upper level due to either a cell split or a nucleus change. The syntaxis is the following one:

```
void insert(typename CellType::VDType& new_elem,
            std::string uri_name, uint64 level_num,
            CellTypePointer son, std::string key = "-")
```

  where `new_elem`, `uri_name`, and `key` are the same parameters as before, `level_num` is the level number in which the element is to be inserted, and `son` is a pointer to the cell from the lower level whose nucleus is the element being inserted.

The insert methods (both public and private) are based on the Preemptive Cell Search algorithm proposed in [KG07]. This algorithm consists in searching the optimal cell from a given level (the ground level if it is an external insertion) for the new element to be inserted. The optimal cell, which will be called owner_cell, is considered as the cell whose nucleus is the nearest one to the element being inserted. Beginninng from the top level, the most similar element at the current level is found, which is d_{min} far from the new element. Then, for each element, if the distance to the new element minus the covering radius of the son cell is greater than d_{min} the son cell and the whole subtree can be discarded. In this way, we have the certainity that the optimal cell will be found and the computational cost is expected to be lower than an exhaustive search.

As said in Section 4.1, we proposed an approximation by excess for the covering radius instead of using the exact value. This approximation has as advantage that the optimal cell is never discarded and that the computational cost of keeping updated an approximated value for the covering radius is much lower than doing it with the exact value. On the other hand, using this approximation by excess can result in an increment of the visited cells during future insertions or query requests. Next, there is how the Preemptive Cell Search algorithm is called from the insert method:

```
//! Type to refer to a pointer to CellModel
typedef CellModel* CellTypePointer;

CellTypePointer owner_cell;
std::vector<CellTypePointer > cells;
for(typename std::set<CellTypePointer >::iterator it=
    _tree_container[top_level_num].cells.begin();
    it!=_tree_container[top_level_num].cells.end(); ++it)
    {
        cells.push_back(*it);
    }
owner_cell = _preemptive_cell_search(cells, new_elem,
                                     top_level_num, level_num);
```

where `_tree_container` is the STL container vector of Level objects, `cells` is a parameter which consists of the cells we have to consider from the current level during the search (in this case, since the search starts on the top level, only the top cell is pushed back), `new_elem` is the element being inserted, `top_level_num` is the level on which the search starts, and `level_num` is the level on which the owner cell for the new element has to be found.

The searching process for the optimal cell will not be necessary when the element has to be inserted on the top level or on a level that is above the top one. In the former case, there is only one cell on the top level so the element will be inserted on it:

```
owner_cell = *(_tree_container[top_level_num].cells.begin());
```

This happens when the HCT consists of only one level or when a cell from the second top level has changed its nucleus or has suffered from a mitosis operation and the new nucleus or nuclei has to be inserted on the top level. In the latter case, when the top cell is split due to an insertion, then a new top level is created where the two nuclei from the two newborn cells are inserted on it:

```
Level<CellType> new_top_level(_maturity_size_top_level);
CellTypePointer new_top_cell = new CellType(new_elem, uri_name,
                                            key, son);
new_top_level.cells.insert(new_top_cell);
_tree_container.push_back(new_top_level);
_tree_container[level_num].num_insertions = 1;
```

Once the owner_cell is found, if the pointer to the son cell is not NULL, i.e. the insert method has been called internally for promoting a new nucleus on the upper level, then the parameter parent_cell from the son cell will be pointing to the owner_cell. Then, the element is appended to this cell by calling the insert method from the cell object and the parameter num_insertions from that level is incremented:

```
  if(son!=NULL)
  {
      son->parent_cell = owner_cell;
  }
 owner_cell->insert(new_elem, uri_name, key, son);
 _tree_container[level_num].num_insertions++;
```

Once the element has been appended to the owner_cell, this cell is submitted to a post-processing check:

```
 _post_processing(owner_cell, old_nucleus, level_num);
```

where `old_nucleus` is the cell nucleus before being post-processed and `level_num` is the level which the owner_cell belongs to. First of all, it is checked if the compactness threshold has to be updated by comparing the number of insertions performed on the current level with the updating period, i.e. every k insertions the compactness threshold is updated, where k has been set to 10 in our experiments. The compactness threshold is computed by using Equation 3.1 from Section 3.2, where k_0 has been set to 1.25 since this value is recommended in [KG07]. Then, the number of elements which belong to owner_cell is compared with the mature size of the level level_num to check if the cell is mature. On that point, we can have two different cases:

- Case 1: The cell is not mature or the cell is mature but compact enough. Therefore, the cell is not split. We have to check if the owner_cell's nucleus has changed. In that case, if we are not at the top cell, the old nucleus is removed from the upper level and the new representative is inserted in it:

```
_remove(old_nucleus, owner_cell->parent_cell, level_num+1);
insert(new_desc, new_uri, new_key, level_num+1, owner_cell);
```

  where `new_desc` is the new nucleus to be inserted in the upper level (`level_num+1`) with `new_uri` pathname and `new_key` identifier, and `owner_cell` is the pointer to the son cell. On the other hand, if the nucleus has not changed we have to check if the covering radius has been modified. In such a case, the parent_cell's covering radius may have also changed. We check it by calling the function:

```
_check_parent_covering_radius(owner_cell, level_num);
```

  which is called recursively until either the top cell or a parent cell whose covering radius has not changed is reached. This function updates the covering radius for the visited parent cells.

- Case 2: The cell is mature but not compact enough. We look for the edge with the largest weight by iterating through the edges of the owner_cell's graph. Two newborn cells (`left_cell` and `right_cell`) will be constructed by splitting the MST at the retrieved edge. In order to know to which cell belongs each vertex, the edge is removed from the MST and from each one of its two vertices (one will belong to the `left_cell` and the other one to the `right_cell`) we iterate through the edges of the MST and add each new vertex to the respective newborn cell. When we visit an MST edge we can have two different cases: (*i*) none of its two vertices belongs to one of the two newborn cells yet or (*ii*) one of its vertices belongs to one of the two newborn cells. In the former case, the edge is skipped for the time being. In the latter case, if one vertex of the edge in which we are belongs to, let's say, the `left_cell`, then the other vertex is also appended to the `left_cell` and the edge is erased from the MST. We go on iterating through the MST edges until there is no edge left. Once all the vertices have been appended to either the `left_cell` or the `rigth_cell`, the two new graphs are built. The edge weights of the two new graphs are not recomputed

but they are got from the owner_cell's graph. Then, the update_parameters cell method is called for both cells in order to update the different parameters which characterize a cell (mean_weight, var_weight, max_weight, covering_radius and compactness). Next, the two newborn cells are appended to the set (STL container) of cells belonging to the level given by `level_num`. In addition to this, the parent_cell parameter of the son cells pointed by the property_index1 from every vertex of the two new cells are updated by pointing the respective cell instead of the original owner_cell. Then, if we are not at the top cell, the owner_cell's nucleus (`old_nucleus`) is removed from the upper level:

```
_remove(old_nucleus, owner_cell->parent_cell, level_num+1);
```

Finally, the cell pointer to the owner_cell is erased from the set of cells of the current level, the content pointed by the owner_cell is deleted and the two nucleus from the newborn cells are inserted in the upper level:

```
_tree_container[level_num].cells.erase(owner_cell);
delete owner_cell;
insert(desc_left, uri_left, key_left, level_num+1, left_cell);
insert(desc_right, uri_right, key_right, level_num+1, right_cell);
```

where `desc_left` and `desc_right` are the two nucleus to be inserted in the upper level (`level_num+1`) with the respective pathname (`uri_left` and `uri_right`), identifiers (`key_left` and `key_right`) and the pointer to the son cell (`left_cell` and `right_cell`).

5.3.2 Removal methods

The Hierarchical Cellular Tree construction also allows the removal of items from the indexing structure. As in the insertion case, we also have to differentiate between two removing methods:

- A public method used for removing an element from the indexed database. The syntaxis is the following one:

```
void remove(std::string elem_to_remove)
```

where `elem_to_remove` is the pathname of the element to be removed.

- A private method which cannot be controlled externally. This method is called automatically when a cell nucleus has to be removed from the upper level due to a cell split, a nucleus change or a item removal from the indexed database. The syntaxis is the following one:

```
void _remove(std::string uri_name, CellTypePointer owner_cell,
             uint64 level_num, bool from_depleted_cell = false)
```

  where `uri_name` is the pathname of the element to be removed, `owner_cell` is the cell which hosts this element, `level_num` is the level number from which the element has to be removed, and `from_depleted_cell` is a boolean value which indicates whether the son cell from the lower level had become depleted after the previous item removal.

The public remove method is called whenever an element from the database is not wanted to belong to the indexing structure any longer. We have two different cases depending on the number of elements belonging to the cell from which the element will be removed:

- Case 1: There is only one element. Therefore, the cell will be depleted after the element removal. If there is only one level which only contains the top cell, then the cell is erased from the ground (and top) level's set of cells and the level is also pop back from the HCT's vector of levels (_tree_container). In such a case, the whole HCT would be depleted. On the other hand, if there is more than one level, it means that the cell from which the element has to be removed does not belong to the top level. In such a case, the vertex is cleared and removed from the graph, the cell pointer is erased from the current level and the element, which was the nucleus of the depleted cell, must be removed from the upper level by calling the private remove method with the parameter `from_depleted_cell` set to true:

```
_remove(parent_uri, parent_cell, 1, true);
```

where `parent_uri` is the same pathname as `elem_to_remove` and `parent_cell` is the cell from the level 1 (level above the ground level) to which the `parent_uri` belongs.

- Case 2: There is more than one element. Therefore, the cell will not be depleted after the item removal. In this case, the vertex is cleared and removed from the graph and the MST is recomputed. Next, the cell parameters are updated by calling the cell method _update_parameters. Finally, the cell is subject to a generic post processing as the explained before in the description of the method _post_processing, which is also called by the insertion algorithm.

As said before, an element has to be removed from a cell not only when it has been removed from the database. There are other situations in which an element is removed internally due to the dynamic approach for the HCT building. The cells can split and, therefore, the old nucleus has to be removed from the upper level where the two new representatives will be inserted. Or simply a cell nucleus may have changed and it is necessary to change it in the upper level. These cases can occur after both an item insertion and removal. The way to proceed of the private remove method is very similar to the public one. The main differences are detailed next:

- If an element is removed from the top cell and this cell becomes depleted, then the new top level is the lower level and, therefore, its maturity size has to be changed.

- If an element is removed from the top cell which has only two elements and the element to be removed is the nucleus of a depleted cell from the lower level, then the whole top cell has to be removed since we have the certainity that the element has not been removed due to a nucleus change and, therefore, only one element would belong to the top cell.

5.3.3 Fitness Check operations

We have also implemented one of the fitness check operations proposed in [KG07] which is called Outliers Check. The objective of this operation is to minimize the corruption, which might have occurred due to the order in which the elements have been inserted, by reinserting the elements which belong to minor cells, i.e. cells with

only one or a few items in it. So what we expect is that some mature or not minor cells may host these elements. However, if the element insertion in the most suitable cell results in a significant degradation on the cell compactness the cell may suffered from a mitosis operation and the reinserted element may be hosted by a minor cell again. In such a case, this means that the given element is an actual outlier and belongs to a minor cell not due to the insertion order of the items. This operation is performed for all levels in decreasing order from top to bottom except for the top level since there is only one cell in it and can be performed periodically during or after the HCT building.

We have implemented this method only for minor cells hosting one element, but not for cells hosting a few items, since we consider that this is the worst case. The way to proceed is the following one:

1. We iterate through the cells of the current level until a minor cell hosting only one element is found. If there are no minor cells, we descend to the lower level.

2. Once a minor cell is found, its element is removed from the cell (so is the cell, since becomes depleted) and is reinserted in the HCT at the current level. The identifier of the reinserted element is stored in a temporal vector in order to detect the possible real outliers.

3. We go on iterating through the cells and repeating Step 2 until there are no minor cells or all the existing minor cells host elements which have been already reinserted. Then, if we are at the ground level the algorithm has finished. Otherwise, we descend to the lower level and go back to Step 1.

5.3.4 Update method for covering radius

Finally, the update_covering_radius method presented in Section 4.2 has been implemented. This method consists in searching, for each nucleus of each cell and at each level, the farthest element which is hosted by a ground level cell which belongs to the subtree which has the afore-mentioned cell as root. To illustrate it better, let us suppose that we have the HCT represented in Figure 5.1. If the update_covering_radius method is called over this HCT, the covering radius will be updated for the cells C_9, C_{10}, C_{11} and C_{12}. When the covering radius for the cell C_9 has to be updated, a method which returns the ground level cells belonging to the subtree which has the

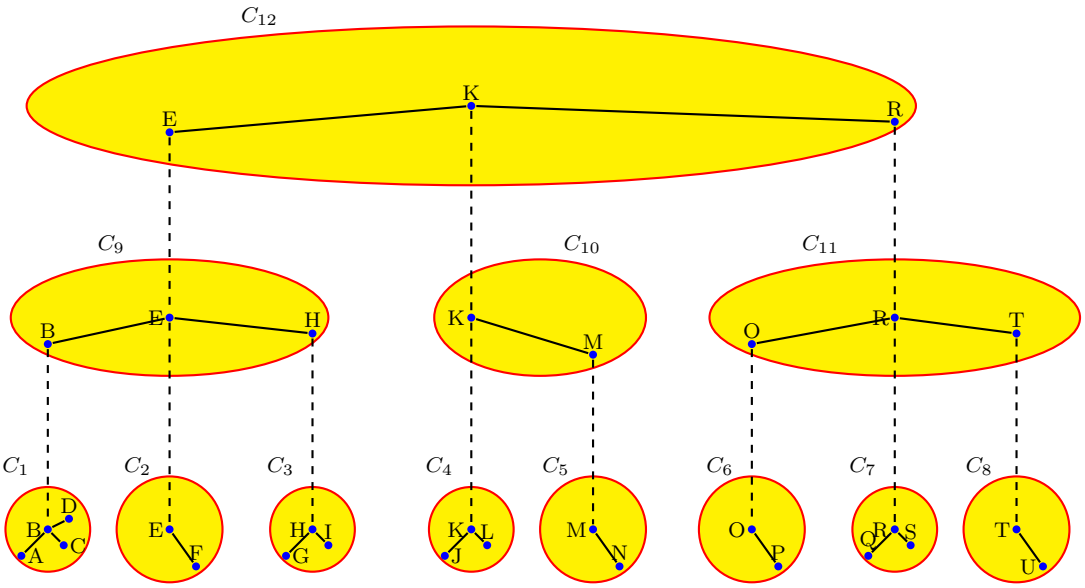


Figure 5.1: Updating covering radius

cell C_9 as root is called. Therefore, this method returns the cells C_1, C_2 and C_3. Then, the elements from these cells are compared to the nucleus of the cell C_9, i.e. the element E, and the distance given by the farthest element becomes the covering radius. For the covering radius of the cell C_{10} and C_{11}, the elements from the cells C_4 and C_5, and C_6, C_7 and C_8 are compared with the elements K and R respectively. Finally, the elements from all the ground level cells are all compared with K in order to update the covering radius for the top cell C_{12}.

5.4 Write/Read HCT operations

The HCT building over a large database is a high cost operation. That is the reason why the implementation of input and output operations for storing the indexing structure in the hard disk and for reading the HCT from it is essential. Otherwise, any modification on the HCT or any unpredicted query request would imply to rebuild the HCT from scratch as long as we are not working over a server/client architecture. In such a case, we would have a process which would be continously listenning for possible query requests, new element insertions or element removals. However, if the HCT was not stored at disk we would also run the risk of a server failure and the HCT would need being rebuilt again from scratch.

The WriteHCTree class allows to write the Hierarchical Cellular Tree at disk by calling

its operator << as shown following:

```
WriteHCTree wtree(outfile);
wtree << tree_cs;
```

where first the `WriteHCTree` constructor have been called, which have the pathname `outfile` where the HCT will be stored, and then the operator << writes the HCT `tree_cs` at disk in the given file. When this operator is called over an HCT, a TXT file is created where the first line represents the maturity size of the inner levels and the maturity size of the top level:

```
maturity_size top_maturity_size
```

Except for the first line, each line from the TXT file represents all the information hosted by a cell of the HCT with the following format:

```
level_num kernel kernel_uri kernel_key num_vertices vertex#0
vertex#0_uri vertex#0_key vertex#0_data_value ... vertex#i
vertex#i_uri vertex#i_key vertex#i_data_value ... vertex#K
vertex#K_uri vertex#K_key vertex#K_data_value source_edge#0
target_edge#0 weight_edge#0 ... source_edge#i target_edge#i
weight_edge#i ... source_edge#K-1 target_edge#K-1 weight_edge#K-1
covering_radius mean_weights var_weights max_weight compactness
```

where `level_num` is the level to which the current cell belongs; `kernel`, `kernel_uri` and `kernel_key` are the nucleus vertex, its path_name and its identifier respectively; `num_vertices` refers to the number of vertices (K+1) of the current cell; `vertex#i`, `vertex#i_uri`, `vertex#i_key` and `vertex#i_data_value` are the ith vertex, its path_name, its identifier and its *VDModel* data type value; `source_edge#i`, `target_edge#i` and `weight_edge#i` are the two vertices the ith edge consists of and the distance between them; `covering_radius`, `mean_weights`, `var_weights`, `max_weight` and `compactness` are the statistical cell parameters.

Once the information related to all the cells have been written to disk, an ending line which consists of the number -1 is appended to the output file. All the HCT data

need to be written at disk in order to rebuild perfectly an HCT in which to insert new elements or to make query requests.

On the other hand, the complementary ReadHCTree class is the responsible for the HCT reading. In this case, the constructor has the filename from which the HCT has to be read as an input parameter. Then, the operator >> is used for loading the HCT in it:

```
ReadHCTree rtree(infile);
rtree >> tree_cs;
```

This operator creates a new cell for each line read. An empty graph is built and the different vertices and edges are added to it in order to fill the cell. Then, the cell is appended to the corresponding level of the HCT container, which consists of a vector of Level objects each of which contains a set of cell pointers. The HCT is rebuilt in descending order (from top to bottom). In this way, except for the top cell, each time a cell has been filled with the data taken from the TXT file, we look for the cell nucleus identifier at the upper level and once found we set both the parent_cell pointer from the cell at the current level and the property_index1 parameter of the vertex from the upper level, i.e. the cell pointer to the son cell. This piece of information is not stored at disk since it is intrinsically contained and, therefore, can be obtained as explained before. We do the same for the MST, since its construction only needs the graph, which has been already built. Finally, both maturity_size and maturity_size_top_level global parameters are set.

5.5 A tool for image database indexing

The tool database_indexing allows the user to index a database by using the Hierarchical Cellular Tree technique in a handier way. Although the HCT has been implemented for indexing any element type, this tool is no longer generic and is used to index image database elements which are represented by some MPEG-7 visual descriptors and are compared by using (dis)similarity measures defined also in MPEG-7 standard for each one of them. The user is asked to pre-compute the MPEG-7 visual descriptors for each image from the database to be indexed and stored them in XML files before using this tool. These XML files containing the descriptors can be gen-

erated by using a tool named bpt_population which was developed for the retrieval system implemented in [Ven10]. Once computed, the image database to be indexed is given as a TXT file containing the image visual descriptor filenames (XML) and a unique identifier for each one:

```
pathname#1 id#1
pathname#2 id#2

   ⋮          ⋮

pathname#N id#N
```

where `N` is the number of elements to be indexed. If more than one visual descriptor is to be used, a different HCT is computed for each one of them. This tool also allows the user to load from disk a database which has been previously indexed by using the ReadHCTree class and add new elements to be indexed. In this way, the HCT does not need to be recomputed from scratch every time that new items have to be inserted and indexed. The resulting indexed structure is always saved as a TXT file by using the WriteHCTree class.

Before detailing the different input parameters, we have to differentiate between the three kinds of parameters of a tool implemented in the ImagePlus development platform:

- Arguments. These ones are mandatory when the tool is called by the user. It does not make sense to execute the program without them.
- Options. These ones are optional input parameters. They have a default value in case the user has not specified any of them. An *option_name* option is used by adding it to the call:

  ```
  --option_name=option_value
  ```

  where `option_value` is the value given by the user to the `option_name` option.

- Flags. These ones are as the options but their data type is a boolean value. Since there are only two possibles values (false or true), it is not necessary to specify its value. Its default value is false and if we want to change it we only have to add it to the call:

  ```
  --flag_name
  ```

Next, the different input parameters for the database_indexing tool are detailed:

- database: TXT filename containing the image visual descriptor filenames (XML) and a unique identifier for each one. Argument type.
- results: Directory where the HCT will be stored at disk in a TXT file. Argument type.
- TopMaturitySize: Value of the maturity size cell parameter for the top level. Once the top cell has a number of elements greater than or equal to the TopMaturitySize, the cell may suffer from a mitosis operation if it is not compact enough. It is an integer value. Option type. Its default value is 7.
- MaturitySize: Value of the maturity size cell parameter for all levels except for the top level. When a cell has a number of elements greater than or equal to the MaturitySize it becomes mature and may suffer from a mitosis operation. It is also an integer value. Option type. Its default value is 7.
- VDColorStructure: Whether Color Structure descriptor is considered or not for HCT building. Flag type.
- VDColorLayout: Whether Color Layout descriptor is considered or not for HCT building. Flag type.
- VDDominantColor: Whether Dominant Color descriptor is considered or not for HCT building. Flag type.
- VDEdgeHistogram: Whether Texture Edge Histogram descriptor is considered or not for HCT building. Flag type.
- PreviousHCTreeColorStructure: TXT filename containing an HCT which has been built according to the Color Structure descriptor and is wanted to be

loaded in order to add new elements to the indexed structure. If this parameter is used, then the VDColorStructure flag must be also used. Option type. Its default value is "-".

- PreviousHCTreeColorLayout: TXT filename containing an HCT which has been built according to the Color Layout descriptor and is wanted to be loaded in order to add new elements to the indexed structure. If this parameter is used, then the VDColorLayout flag must be also used. Option type. Its default value is "-".

- PreviousHCTreeDominantColor: TXT filename containing an HCT which has been built according to the Dominant Color descriptor and is wanted to be loaded in order to add new elements to the indexed structure. If this parameter is used, then the VDDominantColor flag must be also used. Option type. Its default value is "-".

- PreviousHCTreeEdgeHistogram: TXT filename containing an HCT which has been built according to the Texture Edge Histogram descriptor and is wanted to be loaded in order to add new elements to the indexed structure. If this parameter is used, then the VDEdgeHistogram flag must be also used. Option type. Its default value is "-".

Next, we give an example of a database_indexing tool call:

```
database_indexing /imatge/cventura/.../database.txt
/imatge/cventura/.../results/ --TopMaturitySize=24
--MaturitySize=6 --VDColorStructure
```

where the HCT would be built according to the Color Structure descriptor. The file where the HCT is stored in the *results* directory is named tree-cs.txt after the descriptor name the HCT building is based on. In the same way, if the HCT is built based on the Color Layout, the Dominant Color or the Edge Histogram, the HCT is stored in a file named tree-cl.txt, tree-dc.txt or tree-eh.txt respectively.

If we wanted to add new elements to the previous generated HCT, we would call the database_indexing tool in the following way:

```
database_indexing /imatge/cventura/.../new_elements_database.txt
/imatge/cventura/.../results/ --TopMaturitySize=24
--MaturitySize=6 --VDColorStructure
--PreviousHCTreeColorStructure=/imatge/cventura/.../results/tree-cs.txt
```

In such an example, the old Hierarchical Cellular Tree would be rewritten since we have given the same *results* directory as before.

5.6 A tool for image retrieval over HCT

The hct_query tool allows the user to carry out a search on an image database given an example, a technique which is called Query by Example. This image database has to be previously indexed using the Hierarchical Cellular Tree technique to reduce the high computational cost time which would imply to perform an exhaustive search in the database. Therefore, the database_indexing tool has to be previously called to index the image database. In the hct_query tool, the user selects the query image by giving the filename of the XML containing its MPEG-7 visual descriptors, which have been previously extracted by using the bpt_population tool. Instead of an XML file, the user is also allowed to give an image filename, but the searching time will increase slightly due to the compution of the necessary visual descriptors. Furthermore, this tool also allows the user to perform several retrievals by giving a TXT file including the different images query filenames (XML or not). The user also selects on which descriptors the search is based and which searching technique is to be used. There will be a compromise between the searching time and the quality of the retrieved images depending on the chosen searching technique. Finally, the user is asked if he wants to perform a new retrieval by changing some configuration parameters (query, used descriptors, number of results...) without waiting for the HCT being load again.

Next, the different input parameters for the database_indexing tool are detailed:

- query: Query image visual descriptors filename (XML), query image filename (JPEG,PNG...) or TXT file including several query images filenames (XML or JPEG,PNG...). Argument type.
- results: Directory where the results of the search will be stored at disk in an XML file. The format of this output file is compatible with the graphic interface

GOS (Graphic Object Searcher) [CY09] to visualize the results in a handier way. Argument type.

- num_results: Number of retrieved images expected to be retrieved for the user. Option type. Its default value is 10.

- PreviousHCTreeColorStructure: TXT filename containing an HCT which has been built according to the Color Structure descriptor and is wanted to be loaded in order to perform query requests over it. Option type. Its default value is "-".

- PreviousHCTreeColorLayout: TXT filename containing an HCT which has been built according to the Color Layout descriptor and is wanted to be loaded in order to perform query requests over it. Option type. Its default value is "-".

- PreviousHCTreeDominantColor: TXT filename containing an HCT which has been built according to the Dominant Color descriptor and is wanted to be loaded in order to perform query requests over it. Option type. Its default value is "-".

- PreviousHCTreeEdgeHistogram: TXT filename containing an HCT which has been built according to the Texture Edge Histogram descriptor and is wanted to be loaded in order to perform query requests over it. Option type. Its default value is "-".

- VDColorStructure: Whether Color Structure descriptor is considered or not for the search. If this parameter is used, then an HCT based on that descriptor must be loaded by using the PreviousHCTreeColorStructure option. Flag type.

- VDColorLayout: Whether Color Layout descriptor is considered or not for the search. If this parameter is used, then an HCT based on that descriptor must be loaded by using the PreviousHCTreeColorLayout option. Flag type.

- VDDominantColor: Whether Dominant Color descriptor is considered or not for the search. If this parameter is used, then an HCT based on that descriptor must be loaded by using the PreviousHCTreeDominantColor option. Flag type.

- VDEdgeHistogram: Whether Texture Edge Histogram descriptor is considered or not for the search. If this parameter is used, then an HCT based on that

descriptor must be loaded by using the PreviousHCTreeEdgeHistogram option. Flag type.

- ColorStructureWeight: Weight for Color Structure descriptor. It is used when more than one visual descriptor is considered. It is a float number. Option type. Its default value is 1.
- ColorLayoutWeight: Weight for Color Layout descriptor. It is used when more than one visual descriptor is considered. It is a float number. Option type. Its default value is 1.
- DominantColorWeight: Weight for Dominant Color descriptor. It is used when more than one visual descriptor is considered. It is a float number. Option type. Its default value is 1.
- EdgeHistogramWeight: Weight for Texture Edge Histogram descriptor. It is used when more than one visual descriptor is considered. It is a float number. Option type. Its default value is 1.
- interactive: Mode in which the user can perform a new retrieval by changing some configuration parameters once the original query request has been completed. Flag type.

Next, we give an example of an hct_query tool call:

```
hct_query /imatge/.../image-vd.xml /imatge/.../results/ --num_results=20
--PreviousHCTreeColorStructure=/imatge/.../tree-cs.txt --VDColorStructure
```

where the HCT based on the Color Structure descriptor and stored in the tree-cs.txt file is loaded and the 20 most similar images to the query image (its descriptor is read from the `image-vd.xml` file) retrieved by the preemptive search technique are stored in an XML file of the *results* directory.

5.7 HCT over a server/client architecture

The image retrieval system based on the Hierarchical Cellular Tree indexing technique has been also implemented over a server/client architecture which can support several

clients who are allowed to launch query requests over an image database. Therefore, a daemon program which loads an indexed database and is constantly running waiting for new query requests is necessary. Furthermore, we also need a means of communication between the receiver of the query requests (the server) and the sender (the client).

With this purpose, a messaging system called KSC (Keni Socket Communication) [Jor09] has been selected. This system emerged from the need of an asynchron messaging system which allowed different modules to communicate each other in the CHIL (Computers in the Human Interaction Loop) European project [Chi]. In a KSC system, there is only one server and several clients. The objective of the server is to control the registration of the clients, to find out to which message type the clients subscribe and to forward the messages to the subscribed clients. There are two kinds of clients: (*i*) writers and (*ii*) readers. There can be only one writer for each message type whereas there is no restriction on the number of readers. When a writer creates a message, this message is sent to the server since the writer does not know which clients are subscribed to this message type. That is the reason why the server establishes the communication between the writer and the readers by forwarding the messages and manages the subscription queries. In Figure 5.2, a scheme of the KSC messaging system extracted from [Jor09] is represented.

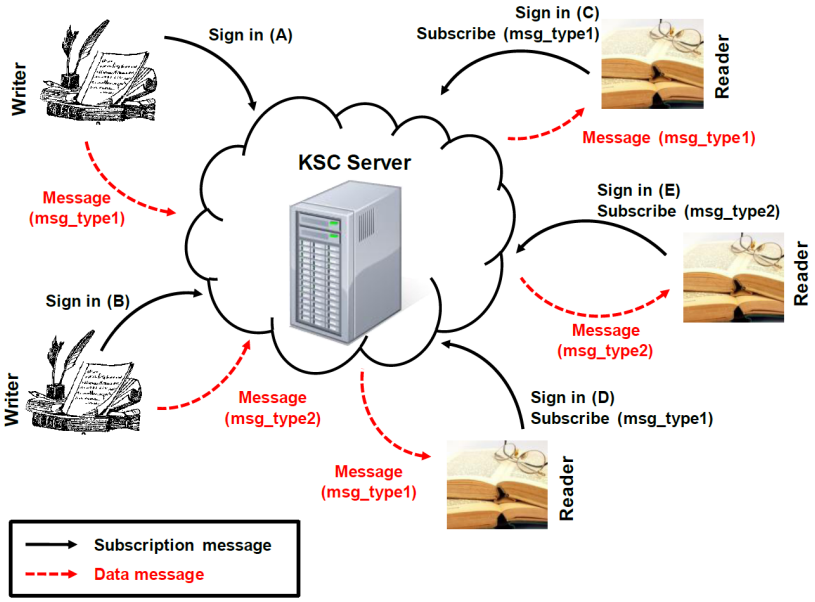


Figure 5.2: Scheme of the KSC messaging system

Based on this scheme, two tools have been implemented: (*i*) the hct_building tool and (*ii*) the query_request tool. The former is the responsible for indexing the image database and carry out the query requests sent by the query_request tool. Thus, once the image database has been indexed, the hct_building subscribes as a receiver to a message type named SearchKSCMessage. It is the query_request tool which subscribes as a sender to this kind of message in order to communicate with the hct_building tool trough the KSCenter and define the parameters of the search. The SearchKSCMessage has the following structure:

```
class SearchKSCMessage
{
   public:
     char query[256];
     char results[256];
     uint64 num_results;
     bool VDColorStructure;
     bool VDColorLayout;
     bool VDDominantColor;
     bool VDEdgeHistogram;
     bool fusion;
     float64 ColorStructureWeight;
     float64 ColorLayoutWeight;
     float64 DominantColorWeight;
     float64 EdgeHistogramWeight;
     bool exhaustive;
     bool preemptive;
     uint64 exhaustive_level;
     char file_type[256];
};
```

where each SearchKSCMessage member has the same meaning as the configuration parameters of the hct_query tool. These parameters are set by the user through the query_request tool input parameters, which are used in the same way as the hct_query tool. There are two additional parameters in order to establish the communication with the KSCenter:

- host: the host where the KSC server has been launched. Option type. Its default value is the local host.
- port: the port on which the KSC server is listening for requests. Option type. Its default value is 4444.

These two parameters allow our architecture to have different servers communicated with several clients each one. In this way, we can have different image database indexed each one on a different host or port. Thus, the user can perform his query requests over different image database as long as they have been indexed by the hct_building tool each one on a different port or host, which the user must know.

As said before, the hct_building is the responsible for indexing the database. This tool has to be executed before launching any query request. Its input parameters are the same as the database_indexing ones, except the two additional parameters (host and port) which are necessary for the KSC communication.

Apart from these two tools, one of them subscribing as receiver and the other one as sender of the SearchKSCMessage type, it is also necessary the KSC server, which is the first to be executed since it supplies the communication mean between the two former ones. The KSC server is launched by command line:

```
>$ KSCenter port
```

Where `port` is the port number on which we want to establish the communication between the hct_building and query_request tools. After launching the KSC server, the following message shoud appear on the console:

```
>$ KSC started. Listening for request on port port
```

Both hct_building and query_request tools also subscribe to another message type named SearchCompletedKSCMessage. This time it is the hct_building which subscribes as a sender and the query_request as a receiver. This message is used to inform that the searching process has been properly completed. A scheme of the KSC architecture for our image retrieval system is illustrated in Figure 5.3.

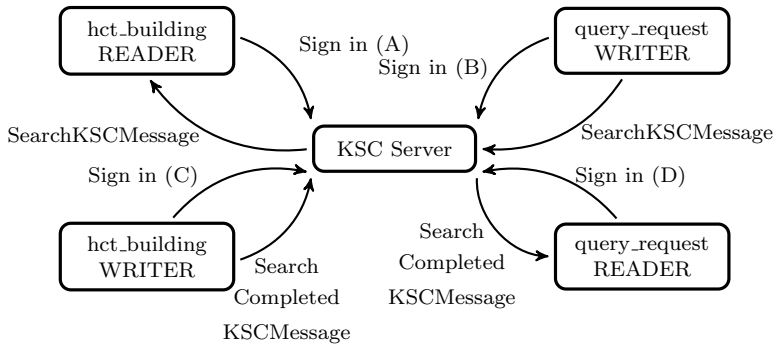


Figure 5.3: Our KSC architecture scheme

An example of use of the HCT over the server/client architecture is shown following:

1. First of all, the KSC server is executed on the 147.83.50.77 host:

```
>$ KSCenter 4444
```

2. The hct_building tool is called to index the desired image database over which the query requests will be performed:

```
>$ hct_building --database=/imatge/cventura/.../database.txt
--VDColorStructure --host=147.83.50.77 --port=4444
```

3. Finally, the query_request tool is launched each time the user wants to perform a new image retrieval:

```
>$ query_request /imatge/.../image-vd.xml /imatge/.../results/
--num_results=20 --VDColorStructure --preemptive --port=4444
--host=147.83.50.77
```

As in the hct_query tool, the format of this output file, which is created by the hct_building once the requested search has been completed, is compatible with the graphic interface GOS (Graphic Object Searcher) to visualize the results in a handier way. This interface manages the user interaction with the query by example system implemented in [Ven10], allowing the user to retrieve some images similar to the query in an attractive and intuitive way. Furthermore, thanks to the support of the

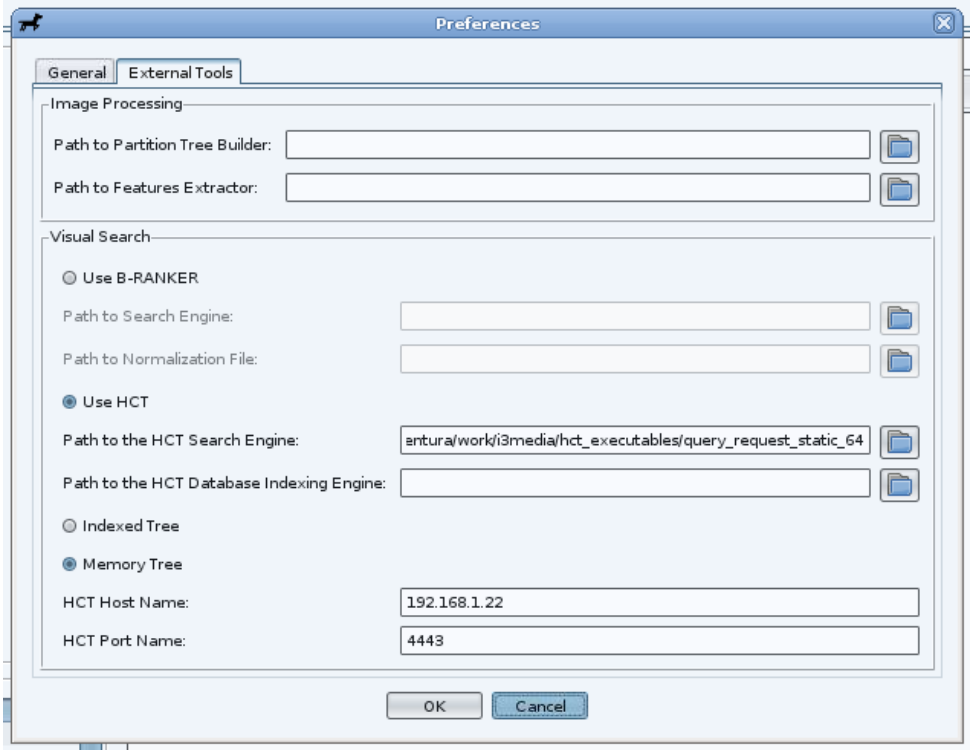


Figure 5.4: Configuration of the HCT parameters in the GOS

image researching group, the GOS has been also adapted in order to launch the query requests over the HCT directly from this graphic user interface. Thus, if we access to *File* > *Preferences* we can configure the GOS so that the search are performed over an indexed image database as it is shown in Figure 5.4. We have to choose the option *Use HCT* and specify the path of the *query_request* executable in the *Path to the HCT Search Engine* field. Furthermore, we choose the option *Memory Tree* and specify the values of the *Host* and *Port* parameters in which the *hct_building* tool is running waiting for new query requests. These configuration parameters are saved for future retrievals.

Chapter 6

Experimental results

Once the Hierarchical Cellular Tree and several searching techniques have been implemented, we proceed to their evaluation in order to know whether the proposed modifications improve the original implementation. First of all, in Section 6.1 we set the experiments which have been carried out. Thus, the different parameters to be analyzed are defined as well as the image database over which the experiments have been carried out. Next, the HCT building is evaluated in Section 6.2 by analyzing its intrinsic HCT parameters as well as the time required for the HCT to be built and the time required for some post processing methods. It is in Section 6.3 where the retrieval system is evaluated according to several retrieval performance measures defined in the literature. Finally, many illustrative query requests examples from the image database are shown in Section 6.4.

6.1 Setting the experiments

In this section we are going to define the framework for the experiments, i.e. which factors are going to be analyzed, which image database has been indexed for evaluating the retrieval system and which criteria have been used in order to compare the database elements. Basically, we have different kinds of HCT to be evaluated since the tree construction depends on several factors:

- The covering radius. The HCT can be built according to either the original covering radius proposed in [KG07] or our proposed approximation by excess (see Section 4.1).

- The update method for the covering radius (see Section 4.2). Whether this method is applied once the HCT has been built in order to have the exact values for the covering radius or not.

- The outliers check method (see Section 5.3.3). Whether this method is applied once the HCT has been built in order to minimize the number of minority cells or not.

Therefore, we have 8 (2^3) differents HCTs. Furthermore, we also want to evaluate how the maturize size has an influence on the results evaluation. Thus, we have set 3 differents values for this parameter:

- We have set the values proposed in [KG07], which are 6 and 24 for maturity_size and top_maturity_size, respectively.

- In order to evaluate whether setting the same value to the top level has any influence on the performance, we have set both the maturity_size and the top_maturity_size to 7

- In order to evaluate how the HCT performes when a larger maturity_size value is used, we have set both the maturity_size and the top_maturity_size to 12.

These HCTs have been built over a database which consists of 216,317 images. These images are keyframes which have been extracted from the video content given by the *Corporació Catalana de Mitjans Audiovisuals* (CCMA), which is involved in the CENIT Buscamedia project [Bus]. The image database used is divided in several assets which represent different video sequences. The content of these videos is generic since we can find news programs, sport events such as soccer matches or Formule One races, cultural programs, political debates, etc. In Figure 6.1, some images from the database are presented.

For all these images, the MPEG-7 Color Structure Descriptor has been computed and has been stored in XML files. We have chosen this descriptor for the experiments since the Color Structure gave the best performance for image retrieval in [Ven10]. Therefore, the HCTs have been built based on this descriptor and the Jeffrey divergence [Ven10] as its dissimilarity measure since this distance gave the best results.

Figure 6.1: Sample images from CCMA database

6.2 HCT building evaluation

Although the main objective of these experiments is to evaluate how fast the retrieval system is and how good the obtained results are depending on the searching technique and the HCT used, we also want to analyze the intrinsic parameters of the HCT, which are independent from the retrieval system. In other words, the HCT parameters reflect how well performed the hierarchical clustering is. Thus, for each HCT, the following parameters have been studied:

- Number of levels: Number of levels into which the HCT has been hierarchically divided.
- Number of cells: Number of cells belonging to the ground level.
- Mean cell size: A cell from the ground level consists of the number of elements on average.
- Variance cell size: Variance of the number of elements belonging to each ground level cell.
- Number of mature cells: Number of cells from the ground level which are mature, i.e. the cells which consist of more elements than the maturize_size.
- Percentage of mature cells: Relative number of mature cells in the ground level.

- Percentage of elements in mature cells: Relative number of elements belonging to mature cells.
- Covering radius mean and variance: Mean and variance of the covering radius values from the ground level cells.
- Compactness mean and variance: Mean and variance of the compactness values from the ground level cells.

All these parameters have been analyzed independently from the covering radius update method since it has no influence on their values. When the method for updating the covering radius is used, only the covering radius values from the cells which do not belong to the ground level can change and, therefore, also the compactness values do. Table 6.1 shows these HCT parameter statistics.

As we expected, the larger the maturity_size value, the smaller the number of levels of the HCT. This is because the cells can host more elements without becoming mature and, therefore, the mean cell size is increased as can be seen in the table. This increase in the number of elements by cell does not mean that the cells become more focused. On the contrary, when we increase the maturity_size value, we are allowing the cells to host more elements without any compactness requirement. That is the reason why the mean covering radius and the mean compactness values become worse when the maturity_size values are increased. As a consequence, the number of cells in the ground level is reduced and, therefore, there are fewer elements (the cell's nuclei) to be clustered in the upper level. With the same reasoning for the rest of levels, it is clear the reason why the HCT with the largest maturity_size value is the HCT which has the smallest number of levels.

Furthermore, the selection of either the proposed covering radius or the original one has also an influence on the HCT parameters. Since the original covering radius can not guarantee that the most suitable cell is found whenever a new element is inserted, the selection of a cell which is not the most appropriate results in a worse global compactness. As a consequence, the mean covering radius and the mean compactness values are better when the proposed covering radius is used.

	proposed covering radius			original covering radius	
	6/24	7/7	12/12	6/24	7/7
Num. Levels	11	12	9	11	12
Num. Cells	65941	59096	42503	66006	59161
Mean cell size	3.28	3.66	5.09	3.28	3.66
Variance cell size	54.92	68.72	109.42	7.18	19.65
Mature cells (%)	0.48	1.04	0.80	0.37	0.85
Elems in mature cells (%)	2.31	4.43	4.27	1.83	3.82
Mean covering radius	0.19	0.21	0.26	0.23	0.24
Variance covering radius	0.04	0.04	0.06	0.06	0.06
Mean compactness	0.05	0.07	0.40	0.10	0.10
Variance compactness	0.06	0.11	0.69	3.06	0.73

Table 6.1: HCT parameter statistics

There are also two more parameters to be considered:

- Construction time: the time which is necessary for the HCT to be built. This parameter will be considered jointly with the retrieval system evaluation in Section 6.3 to discern which HCT behaves better.
- Inserting time: the time which is necessary for a new element to be inserted in the HCT. It will also be evaluated jointly with the retrieval system performance in Section 6.3.

The time required for building the HCT over the whole image database by using the original covering radius is 50,867.9 seconds (14.13 hours). In Figure 6.2 the evolution

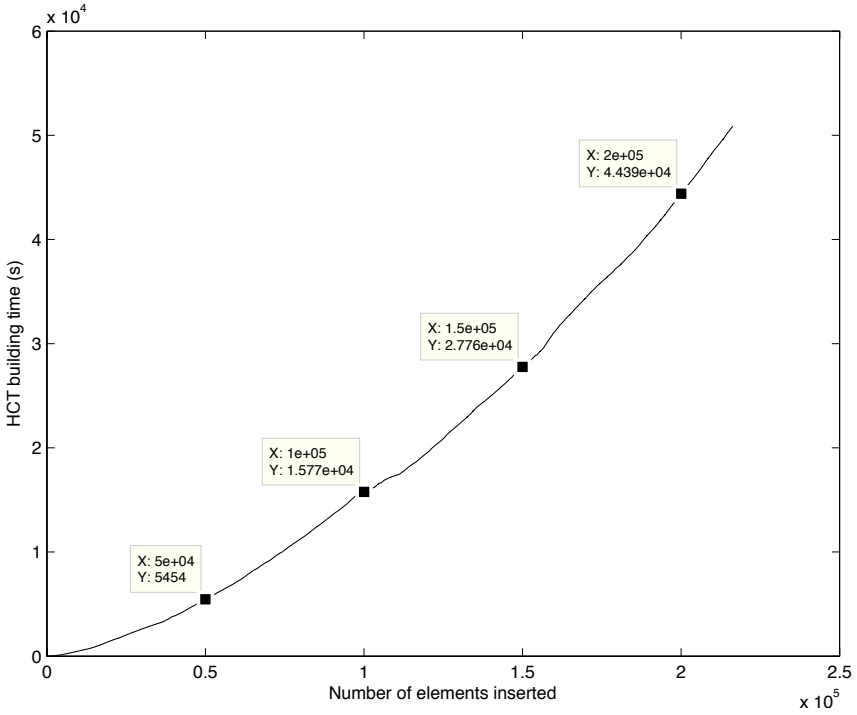


Figure 6.2: HCT building time for original covering radius

of the building time in function of the number of elements inserted is shown. We can see that the building time does not behave linearly but exponentially. This behaviour can also be seen in Figure 6.3 where the insertion time for a new element has been represented as a function of the elements previously inserted in the HCT. Each insertion time has been obtained by averaging the insertion time of 1,000 elements. We can see that the greater the number of elements previously indexed, the longer the insertion time required for a new insertion. Thus, the mean insertion time is 0.1758 seconds when 50,000 elements have been indexed, whereas the required time for a new insertion when more than 200,000 elements have been inserted is 0.3818 seconds. This is because the greater the number of elements, the more complex the HCT structure and, therefore, the more time the Preemptive Cell Search algorithm used by the insertion process requires.

On the other hand, when our proposed approximation by excess of the covering radius is used, the time required for building the HCT over the whole image database is 472,534 seconds (131.26 hours), which means more than 9 times the time required by the original covering radius. For the proposed covering radius, the evolution of the

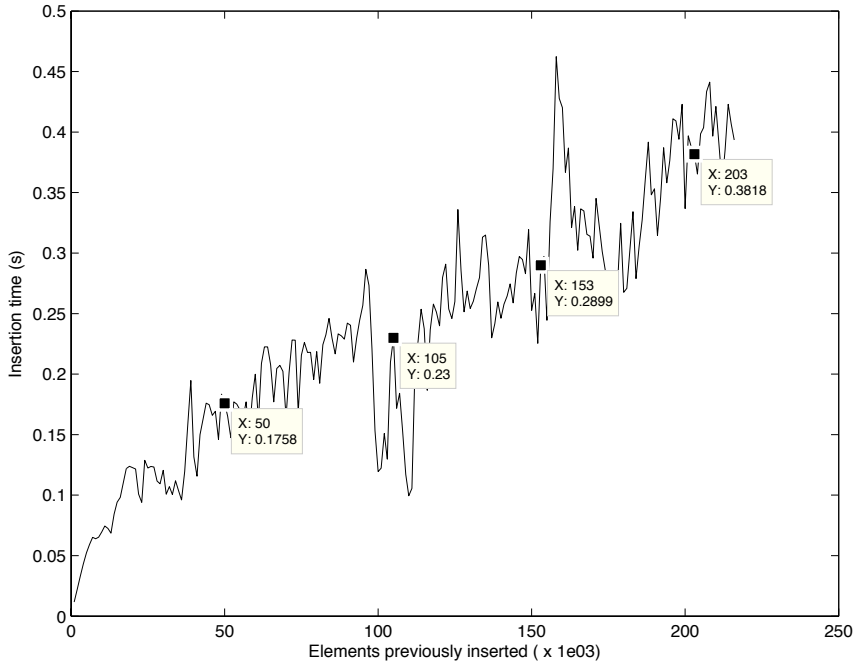


Figure 6.3: HCT insertion time for original covering radius

building time in function of the number of elements inserted is shown in Figure 6.4. As when the original covering radius is used, the building time also behaves in an exponential way. As before, the time required for a new insertion in function of the elements previously inserted is also represented in Figure 6.5. We can see that the mean insertion time required is far longer than in the original covering radius case, taking from 1.173 seconds when 50,000 elements have been indexed (in comparison to the 0.1758 seconds for the original covering radius) to 3.973 seconds when more than 200,000 elements have been previously inserted (0.3818 seconds for the original covering radius). This larger building time for the proposed covering radius in comparison to the original one is because a number of cells larger than the strictly necessary are being visited and, therefore, the correctness of the HCT construction is guaranteed, whereas the original covering radius does not guarantee the right insertion of the elements in the sense that they may not be inserted in the most suitable cell.

Finally, we are going to analyze the time required for the post processing methods which have been implemented, i.e. the outliers check method proposed in [KG07] and the update method for the covering radius (see Table 6.2). These methods are going to be jointly evaluated with the retrieval system in order to know whether it is

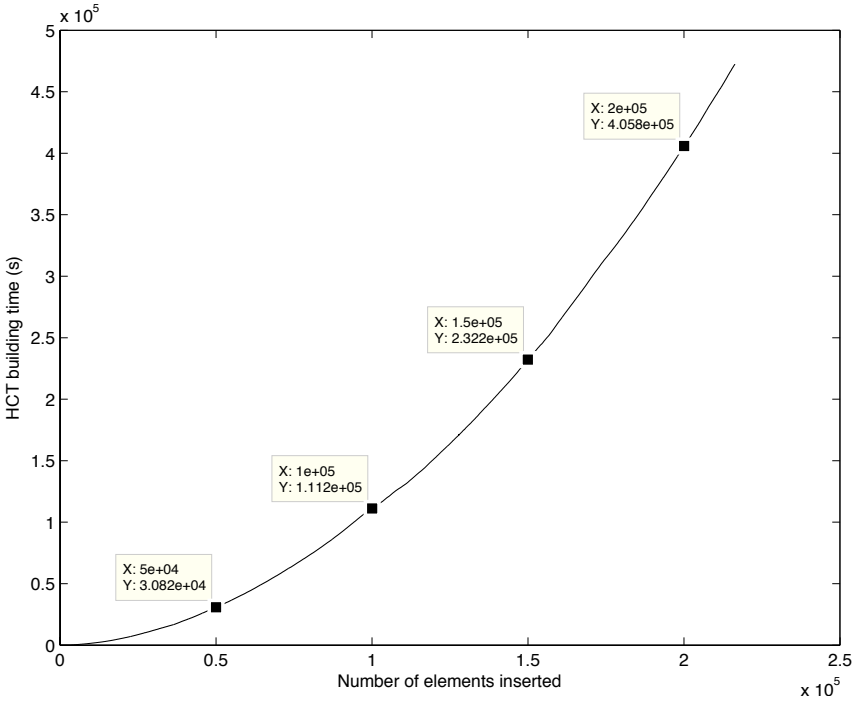


Figure 6.4: HCT building time for proposed covering radius

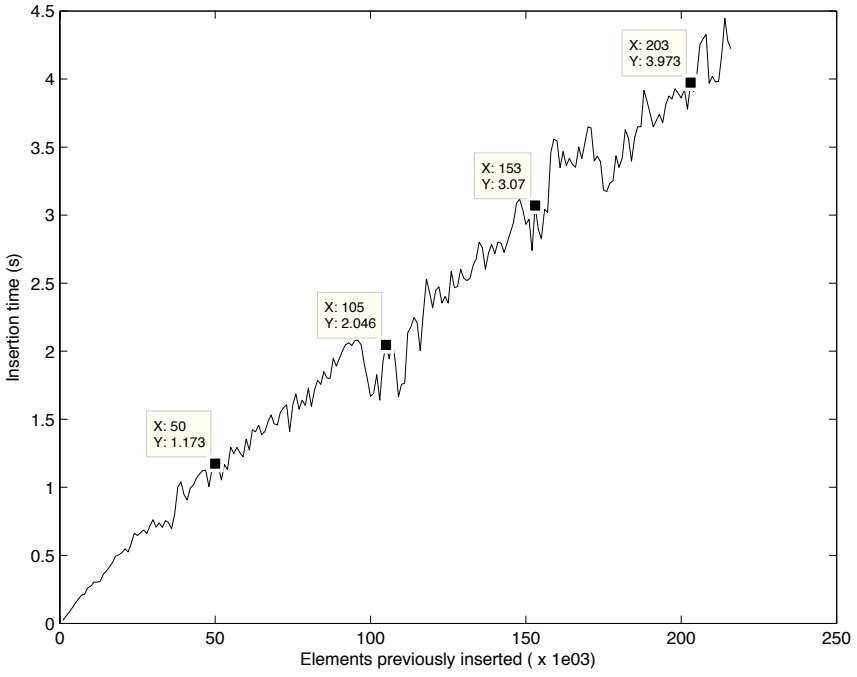


Figure 6.5: HCT insertion time for proposed covering radius

	proposed covering radius	original covering radius
Outliers check required time (s)	47225.8	56760.2
Covering radius update time (s)	98.03	95.42

Table 6.2: Required time for post processing methods

worth or not applying them. Up to now we come to the conclusion that the outliers check method is far more computationally costly than the update method for the covering radius. Furthermore, we have also analyzed which influence has the update method when it is applied periodically. Figure 6.6 shows the evolution of the HCT building time when the update method for the proposed covering radius is applied every 25,000, 10,000 and 5,000 elements in comparison to when this method is not applied. We can see that the building time is reduced when the update method is applied with respect to the proposed covering radius but it is still worse than the building time for the original covering radius. On the contrary, the building time for the original covering radius is increased when the update method is used. This is because the number of cells visited during the insertion process once the update method is applied increases since the covering radius is no longer an approximation by defect of its actual value. Therefore, a number of cells smaller than the necessary were being visited before the update method was applied. This behavior is shown in Figure 6.7, in which the update method has been applied after 200,000 elements have been inserted. As a result, the mean insertion time gets remarkably worse. On the other hand, as it will be seen in the next section, applying the update method will result in a better quality of the retrieved results.

6.3 Retrieval system evaluation

Once the HCT parameter statistics have been analyzed, we proceed to evaluate our retrieval system. Since there is no large-enough image-database available with a ground truth which would allow us to evaluate the whole system, i.e. both the indexing technique and the visual descriptors used for computing the distance between any pair of elements, we have decided to evaluate only the indexing technique. In other words, we are going to evaluate how worse is the search along the indexed

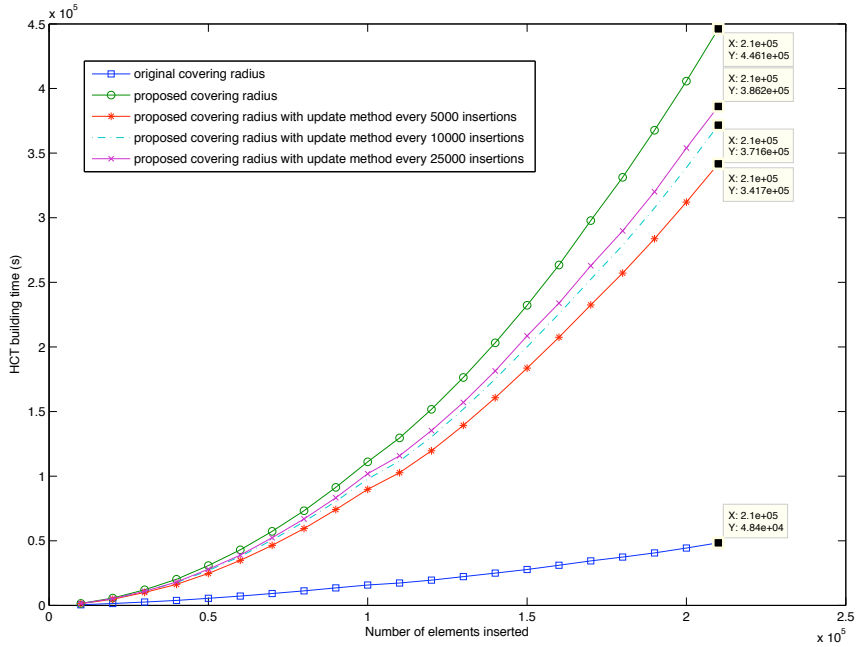


Figure 6.6: HCT building time comparison for update method of the covering radius

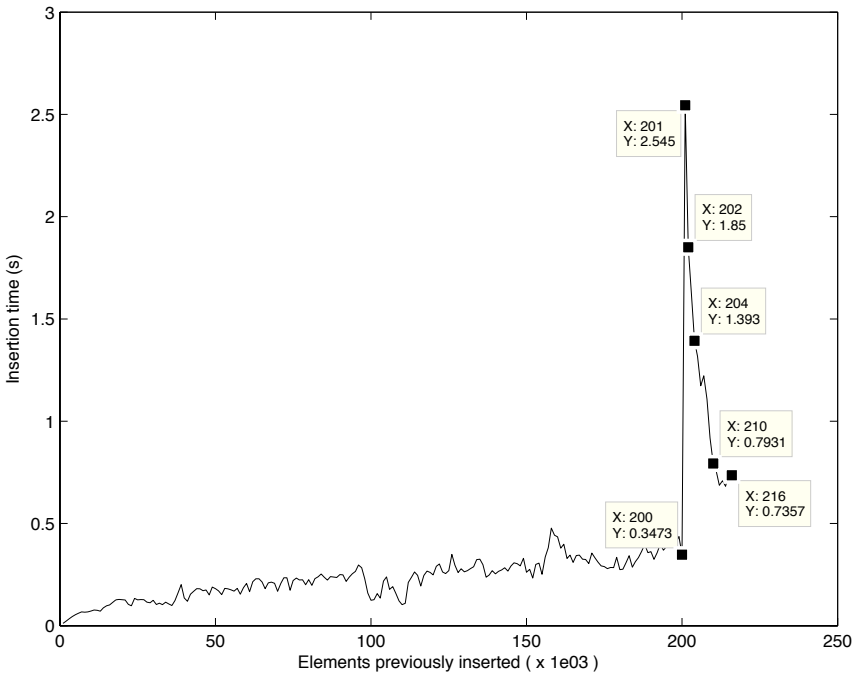


Figure 6.7: HCT insertion time for original covering radius when the update method of the covering radius is applied after 200,000 elements have been inserted

database with respect to an exhaustive search. On the other hand, the retrieval times are expected to be better by using the searching techniques over the HCT than a linear search over the whole image database, which results in a retrieval time of 10 seconds approximately. Thus, as in all systems of this kind, we expect a compromise between the retrieval time and the quality of the retrieved images. First of all, we are going to evaluate independently these two factors for each proposed searching technique and each HCT built.

In order to evaluate the retrieval system, we have chosen randomly 1,082 images from the database which have been used as queries. They have been chosen by taking an image out of every 200 images. Then, for each query image, an exhaustive search has been carried out to know which are the most similar images to the query one. Next, for each HCT built and for each searching technique a new ranking is obtained. These new rankings consist in approximations of the nearest neighbours for each image query. Therefore, these new rankings must be compared with the ranking obtained by the exhaustive search. Since in both the exhaustive and the approximate rankings the elements are not sorted at the cellular scale but at the element scale, the elements belonging to both rankings, i.e. the elements in common, will be sorted in the same order. Thus, the approximated ranking can be formed by taking the exhaustive ranking and removing the elements which have not been found by the searching technique used over the HCT.

In order to compare the approximate rankings with the exhaustive ranking, we use three different measures which have been found in the literature for comparing top k lists:

- Mean Competitive Recall [CPR$^+$07][SMR04][Ger07]. Let k be the number of images we want to retrieve and $A(k, q, E)$ the set of k images retrieved by the searching technique A (*Preemptive Cell Search*, *Most Similar Nucleus* and *Hybrid*) for an image query q on our indexed image database E. Let $GT(k, q, E)$ be the set of the k nearest neighbours to the image query q which has been obtained through an exhaustive search. Then, the competitive recall $CR(A, q, k)$ is defined as the number of elements belonging to the instersection of both sets, i.e. $CR(A, q, k) = |A(k, q, E) \cap GT(k, q, E)|$. Note that competitive recall is an integer number in the range $[0, ..., k]$ and a higher value indicates higher quality. The Mean Competitive Recall $\overline{CR}$ is the average of the competitive recall over

a set of queries Q:

$$\overline{CR}(A, Q, E) = \frac{1}{|Q|} \sum_{q \in Q} CR(A, q, k) \tag{6.1}$$

In our experiments, Q is the set of 1,082 query images chosen randomly from the image database.

- Mean Normalized Aggregate Goodness [SMR04][Ger07]. Let $FS(k, q, E)$ be the set of k images in the image database E farthest from the image query q. Let $W(k, q, E)$ be the sum of distances of the k farthest elements from q:

$$W(k, q, E) = \sum_{p \in FS(k,q,E)} d(q, p)$$

where $d(p, q)$ is the distance from the image query q to the image p which belongs to the set FS. In our experiments, this distance is the (dis)similarity measure for the MPEG-7 Color Structure Descriptor. Then, the Normalized Aggregate Goodness (NAG) is defined as:

$$NAG(k, q, A) = \frac{W(k, q, E) - \sum_{p \in A(k,q,E)} d(p, q)}{W(k, q, E) - \sum_{p \in GT(k,q,E)} d(p, q)}$$

where, as in the Mean Competitive Recall, $A(k, q, E)$ is the set of k images retrieved by the searching technique A (*Preemptive Cell Search*, *Most Similar Nucleus* and *Hybrid*) for an image query q on our indexed image database E and $GT(k, q, E)$ is the set of the k nearest neighbours to the image query q which has been obtained through an exhaustive search. Note that the NAG is a real number in the range [0,1] and a higher value indicates higher quality. The NAG is 1 only when the k nearest elements are retrieved and is 0 only when the k farthest elements are retrieved. Thus, the Aggregate Goodness is normalized with respect to the worst possible result. Then, the Mean Normalized Aggregate Goodness results from the average of the NAG over a set of queries Q:

$$\overline{NAG}(A, Q, E) = \frac{1}{|Q|} \sum_{q \in Q} NAG(A, q, k) \tag{6.2}$$

- Kendall distance [FKS03]. This distance is a variation of the standard Kendall's tau metric between permutations [Ken70]. A permutation σ is a bijection from a set D (in our case the image database) onto the set $[n] = \{1, ..., n\}$ where n is

the size of $|D|$. Therefore, in our context, a permutation represents the order in which the images from the database have been sorted with respect to the query image, i.e. the ranking over the whole image database. For a permutation σ, let $\sigma(i)$ be the position of the element i in the rank. We say that i is ahead of j in σ if $\sigma(i) < \sigma(j)$. Let $P = \{\{i, j\}|i \neq j \text{ and } i, j \in D\}$ be the set of unordered pairs of distinct elements. Let σ_1 and σ_2 be two permutations to be compared. Then, for each pair $\{i, j\} \in P$ of distinct members of D, if i and j are in the same order in both permutations there is no penalization, i.e. $K_{i,j}(\sigma_1, \sigma_2)$=0; otherwise, $K_{i,j}(\sigma_1, \sigma_2)$=1. Finally, Kendall's tau is given by:

$$K(\sigma_1, \sigma_2) = \sum_{\{i,j\} \in P} K_{i,j}(\sigma_1, \sigma_2)$$

Kendall's tau turns out to be equal to the number of exchanges needed in a bubble sort to convert one permutation to the other. This distance is modified in [FKS03] not to compare permutations but top k lists. The main difference to be considered is that two top k lists τ_1 and τ_2 over the same database can have different elements, i.e. an element from the database can appear only in one of the two top k lists. Now, D is defined as the union of the elements belonging to at least one permutation, i.e. $D = D_{\tau_1} \cup D_{\tau_2}$, and P as the set of all unordered pairs of elements in $D_{\tau_1} \cup D_{\tau_2}$. Now, for each pair $\{i, j\} \in P$ of distinct members of D, there are 4 different cases:

- Case 1 (i and j appear in both top k lists). If i and j are in the same order there is no penalty, i.e. $K_{i,j}(\tau_1, \tau_2) = 0$; otherwise, $K_{i,j}(\tau_1, \tau_2) = 1$. In our experiments, this latter case is not possible since the elements are sorted according to the same criteria on both top k lists.
- Case 2 (i and j both appear in one top k list (say τ_1), and exactly one of i or j, say i, appears in the other top k list (τ_2). If i is ahead of j in τ_1 there is no penalty; otherwise, $K_{i,j}(\tau_1, \tau_2) = 1$. In our experiments, this case happens when an element of the exhaustive rank has not been found by the searching technique over the HCT and does not appear in the approximated ranking. On the contrary, when two elements belong to the approximated ranking and exactly only one of them belongs to the exhaustive ranking, the element which appears in both lists is always the best placed in the approximated ranking and, therefore, there is no penalty.

- Case 3 (i, but not j, appears in one top k list (say τ_1), and j, but not i, appears in the other top k list (τ_2)). In such a case, there will be always penalty, so $K_{i,j}(\tau_1, \tau_2) = 1$. In our retrieval system, this case is given when we have a pair consisting of one element from the exhaustive ranking which have not been retrieved by the used searching technique and one element from the bottom of the approximated ranking which does not appear in the exhaustive ranking.
- Case 4 (i and j both appear in one top k list (say τ_1), but neither i nor j appears in the other top k list (τ_2)). In such a case, since the elements do not belong to one of the top k lists, we cannot know whether the order in which the elements would appear is the same or not. The user is allowed to define the penalty for this case. In our experiments, this case occurs when we have a pair consisting of either two elements from the exhaustive rank which have not been retrieved by the used searching technique or two elements from the bottom of the approximated rank which do not appear in the exhaustive rank. We have set the penalty to 0, which corresponds with the *minimizing Kendall distance* defined in [FKS03].

Apart from the these measures for the retrieval system evaluation and the retrieval time, we have also computed the percentage of success in retrieving the query image, i.e. how often the own query image can be found among the results of the retrieval system. There are several cases to analyze depending on the following factors:

- The covering radius. The HCT can be build according to either the original covering radius proposed in [KG07] or our proposed approximation by excess.
- The update method for the covering radius. Whether this method is applied once the HCT has been built in order to have the exact values for the covering radius or not.
- The outliers check method. Whether this method is applied once the HCT has been built in order to minimize the number of minority cells or not.
- The maturity size. We have given 3 different values to this parameter: (i) maturity_size = 6 and top_maturity_size = 24, (ii) maturity_size = top_maturity_size = 7, and (iii) maturity_size = top_maturity_size = 12.

- The searching technique. We have used the proposed searching techniques: (*i*) Most Similar Nucleus, (*ii*) Preemptive Cell Search, and (*iii*) Hybrid. In the latter case, we have set to 7, 8 and 9 the number of levels in which the Preemptive Cell Search algorithm is used.

- The number of results asked by the user. We have set this value to 10, 20, 40 and 80.

First of all, we want to focus the results evaluation on the covering radius and also the proposed update method for it. Table 6.3 shows the results when the outliers check method is not used, the maturity_size = 6 and the top_maturity_size = 24, the searching technique is the Preemptive Cell Search and the number of results asked is 40.

	proposed covering radius		original covering radius	
	non updated	updated	non updated	updated
Mean retrieval time (s)	1.2386	0.8319	0.1058	1.0095
Variance retrieval time (s)	0.1886	0.1466	0.0057	0.1994
Retrieved queries (%)	99.26	99.26	49.26	97.04
$\overline{CR}$	28.09	27.51	12.35	26.42
NAG	0.9970	0.9967	0.9814	0.9965
Kendall	295.24	313.87	934.14	356.92

Table 6.3: HCT results evaluation without outliers check method, maturity_size = 6 and top_maturity_size = 24, Preemptive Cell Search and 40 results

Analyzing the results from Table 6.3 we can see that the retrieval system behaves by far the worst when the original covering radius [KG07] is used without the implemented method for updating the covering radius since there are only 12.35 out of 40 elements in common between the rankings on average. In addition to this, the query image is retrieved only in the 49.26% of the query requests. We come to the same conclusion by analyzing the Normalized Aggregate Goodness (0.9814 is the lowest value) and the Kendall distance (934.14 is the greatest value). On the other hand, the retrieval time is far better than in the other cases. It takes only 0.1058

seconds on average. This is because the original covering radius is an approximation by defect of its exact value and, therefore, a lower number of cells are visited, which means a lower number of comparisons. As consequence, not all the cells which should be visited according to the exact covering radius are visited, which results in a bad performance. This performance for the original covering radius is improved when the update method for the covering radius is used. This is because the covering does not take into account only the elements in the current cell but the whole subtree holding from that cell once this update method is applied. On the other hand, the searching time gets far worse than before (1.0095 seconds on average). The reason why the quality measures on the retrieved results for the proposed covering radius are slightly better when the update method is not used is that more cells are visited and, therefore, more elements can be taken into account for generating the ranking. On the other hand, this results on a far worse searching time (1.2386 seconds on average). Therefore, for the experiment reflected on Table 6.3, our proposed approximation by excess for the covering radius and using the update method gives the best performance considering a compromise between the quality measures and the searching time. In addition to this, we have to consider that the time required for the updated method (see Table 6.2) can be considered negligible in comparison with the time required for the HCT construction.

Now, we are going to analyze the same experiment but using the Most Similar Nucleus technique instead of the Preemptive Cell Search. The results are shown in Table 6.4. For this technique, the covering radius is not used during the search so it does not matter whether the update method for the covering radius is used or not.

Analyzing the results from Table 6.4, we come to the conclusion that the Most Similar Nucleus technique gives a bad performance since the query image is retrieved only in the 5.00% of the query requests and there are only 1.35 out of 40 elements in common between the rankings on average. These results are even worse when the original covering radius is used. On the other hand, the retrieval time is only 0.0075 seconds on average. Since the retrieval time is so short, we have tried incrementing the number of elements considered in the sorting process to obtain a better performance on the resulting ranking. Therefore, instead of considering the number of cells which include at least twice the number of elements desired, i.e. 80 elements, we have tried considering 400 (10 times), 800 (20 times), 1000 (25 times), 2000 (50 times), 5000 (125 times), 10000 (250 times) and 20000 (500 times) elements to generate the top

40 list. The obtained results are shown in Table 6.5.

As we expected, the quality of the results improves when the number of elements being considered for the sorting is increased. On the other hand, the greater the number of elements considered, the worse the retrieval time. This is because the number of comparisons done is directly proportional to the number of elements considered. Anyway, we come to the conclusion that even if we consider 20000 elements, which means a retrieval time of 1.3695 seconds, the quality of the results is worse than the quality obtained when the Preemptive Cell Search algorithm is used. The former gives 12.33 out of 40 elements in common between the rankings on average whereas

	proposed covering radius	original covering radius
Mean retrieval time (s)	0.0072	0.0075
Variance retrieval time (s)	2.01e-05	1.91e-05
Retrieved queries (%)	5.00	3.70
$\overline{CR}$	1.35	0.84
NAG	0.9087	0.9075
Kendall	1530.93	1554.52

Table 6.4: HCT results evaluation without outliers check method, maturity_size = 6 and top_maturity_size = 24, Most Similar Nucleus and 40 results

Elements	400	800	1000	2000	5000	10000	20000
Mean retrieval time (s)	0.0575	0.0567	0.0700	0.1380	0.3438	0.6871	1.3695
Variance retrieval time (s)	3.10e-05	3.21e-05	4.20e-05	8.20e-05	0.0004	0.0014	0.0054
Retrieved queries (%)	7.21	8.23	9.15	12.01	18.39	22.83	31.61
$\overline{CR}$	2.24	2.82	3.16	4.26	6.53	8.61	12.33
NAG	0.9309	0.9387	0.9416	0.9517	0.9646	0.9709	0.9776
Kendall	1490.57	1462.89	1447.66	1395.03	1289.14	1192.19	1025.71

Table 6.5: HCT results evaluation without outliers check method, maturity_size = 6 and top_maturity_size = 24, Most Similar Nucleus and 40 results

the latter gives 27.51 elements in common in a shorter retrieval time (0.8319 seconds). This conclusion is also corroborated by the other quality measures, i.e. the Normalized Aggregate Goodness and the Kendall distance. Furthermore, this experiment allows us to come to conclude that the first progressive subqueries proposed in [KG07] for the *Progressive Query* over a *Query Path* would not give a performance so good as the performance obtained when the Preemptive Cell Search is applied.

Let us consider the Hybrid technique which combines the two searching techniques analyzed above. As explained in Section 4.3, the Preemptive Cell Search is applied over the uppermost levels (a given number of levels) and the Most Similar Nucleus is used over the lowest levels. We have carried out the experiments by setting to 7, 8 and 9 the number of levels over which the Preemptive Cell Search is applied. The results of the evaluation are shown in Tables 6.6, 6.7 and 6.8 respectively.

	proposed covering radius		original covering radius	
	non updated	updated	non updated	updated
Mean retrieval time (s)	0.2484	0.2172	0.0442	0.2123
Variance retrieval time (s)	0.0002	0.0009	0.0003	0.0006
Retrieved queries (%)	37.99	37.99	23.75	30.13
$\overline{CR}$	9.84	9.83	5.86	7.69
NAG	0.9727	0.9727	0.9629	0.9686
Kendall	1105.96	1106.08	1292.37	1199.37

Table 6.6: HCT results evaluation without outliers check method, maturity_size = 6 and top_maturity_size = 24, Hybrid with Preemptive Cell Search over the 7 uppermost levels and 40 results

From the results showed in Tables 6.6, 6.7 and 6.8, we come to the conclusion that we expected. Thus, the larger the number of levels over which the Preemptive Cell Search is applied, the longer the retrieval time. Therefore, for the Hybrid searching technique, we have a retrieval time which is always shorter than the Preemptive Cell Search retrieval time and longer than the Most Similar Nucleus retrieval time. For example, for the proposed covering radius we have:

$$0.0072 \text{ seconds} \leq \text{Retrieval time Hybrid} \leq 0.8319 \text{ seconds}$$

where 0.0072 seconds is the retrieval time for the Most Similar Nucleus technique and 0.8319 is the retrieval time for the Preemptive Cell Search technique. On the

	proposed covering radius		original covering radius	
	non updated	updated	non updated	updated
Mean retrieval time (s)	0.5338	0.4144	0.0612	0.4241
Variance retrieval time (s)	0.0039	0.0093	0.0010	0.0081
Retrieved queries (%)	57.95	57.95	31.61	45.19
$\overline{CR}$	14.23	14.22	7.84	11.24
NAG	0.9824	0.9824	0.9704	0.9782
Kendall	883.36	883.51	1184.59	1016.11

Table 6.7: HCT results evaluation without outliers check method, maturity_size = 6 and top_maturity_size = 24, Hybrid with Preemptive Cell Search over the 8 uppermost levels and 40 results

	proposed covering radius		original covering radius	
	non updated	updated	non updated	updated
Mean query time (s)	0.9818	0.6835	0.0829	0.7413
Variance query time (s)	0.0495	0.0569	0.0026	0.0595
Retrieved queries (%)	84.20	84.20	42.70	73.01
$\overline{CR}$	20.85	20.81	10.62	18.29
NAG	0.9918	0.9917	0.9779	0.9894
Kendall	579.14	580.43	1042.70	686.00

Table 6.8: HCT results evaluation without outliers check method, maturity_size = 6 and top_maturity_size = 24, Hybrid with Preemptive Cell Search over the 9 uppermost levels and 40 results

other hand, the larger the number of levels over which the Preemptive Cell Search is performed, the better the quality measures are. Therefore, the performance of the retrieval system by using the Hybrid technique is always better than the performance obtained by the Most Similar Nucleus and worse than the performance given by the Preemptive Cell Search. For example, for the proposed covering radius and the Mean Competitive Recall measure we have:

$$1.3475 \leq \overline{CR} \text{ Hybrid} \leq 27.512$$

where 1.3475 is the Mean Competitive Recall when the Most Similar Nucleus technique is used and 27.512 is the value for the Preemptive Cell Search. Given a number of preemptive levels, we come to the same conclusion than for the other searching techniques, i.e. the original covering radius without the update method gives by far the worst performance according to the quality measures whereas the proposed covering radius with the updated method gives the best one.

Now, we are going to evaluate which influence the maturity size parameter has on the performance of the retrieval system. We have three different scenerios: (i) maturity_size = 6 and top_maturity_size = 24, (ii) maturity_size = 7 and top_maturity_size = 7, and (iii) maturity_size = 12 and top_maturity_size = 12. We have set the other variables as before, i.e. without the outliers method and 40 results, and we have used the Preemptive Search Cell technique and the proposed covering radius with the update method since they gives the best performance. The results are shown in Table 6.9.

From the results showed in Table 6.9, we come to the conclusion that setting the maturize size to a different value for the intermediate levels and the top level has hardly any influence on the performance. On the other hand, increasing the maturity size value results in a slightly better retrieval time (0.7444 seconds on average) but the quality measures show that the results are slightly worse.

Regarding the outliers check method, we are going to evaluate whether the results improve or not. Table 6.10 shows the performance evaluation depending on whether this method is applied or not for both the original and the proposed covering radius. On both HCTs, the update method for the covering radius has been also applied before the outliers check method.

	maturity_size = 6 top_maturity_size = 24	maturity_size = 7 top_maturity_size = 7	maturity_size = 12 top_maturity_size = 12
Mean retrieval time (s)	0.8319	0.8131	0.7444
Variance retrieval time (s)	0.1466	0.1356	0.0954
Retrieved queries (%)	99.26	98.43	97.60
$\overline{CR}$	27.51	27.64	25.70
NAG	0.9967	0.9969	0.9959
Kendall	313.87	311.71	380.35

Table 6.9: HCT results evaluation without outliers check method, with the proposed covering radius and the update method, the Preemptive Cell Search and 40 results

	proposed covering radius		original covering radius	
	non outliers check method	outliers check method	non outliers check method	outliers check method
Mean retrieval time (s)	0.8319	0.9155	1.0095	1.2212
Variance retrieval time (s)	0.1466	0.1119	0.1994	0.1720
Retrieved queries (%)	99.26	99.35	97.04	96.67
$\overline{CR}$	27.51	28.14	26.42	26.51
NAG	0.9967	0.9972	0.9965	0.9964
Kendall	313.87	292.65	356.92	356.03

Table 6.10: HCT results evaluation of the outliers check method over the HCT with the update method, the Preemptive Cell Search and 40 results

From the results showed in Table 6.10 we can observe a slight improvement of the performance when the check outliers method is applied whereas the retrieval time gets worse. In addition, taking into account the computational cost of this method (see Table 6.2) we come to the conclusion that it is not worth applying the outliers check method.

Finally, we are going to analyze which influence the number of results asked by the user has on the performance of the retrieval system. Table 6.11 shows the performance

evaluation depending on the number of results: (*i*) 10, (*ii*) 20, (*iii*) 40, and (*iv*) 80. As before, these results have been obtained with the Preemptive Cell Search over the HCT build with maturity_size = 6 and top_maturity_size = 24, the proposed covering radius with the update method and without applying the outliers check method.

Num. results	10	20	40	80
Mean retrieval time (s)	0.8514	0.8301	0.8319	0.8357
Variance retrieval time (s)	0.1464	0.1466	0.1466	0.1467
Retrieved queries (%)	95.75	98.15	99.26	99.72
$\overline{CR}$	7.21	14.09	27.51	53.67
NAG	0.9977	0.9974	0.9967	0.9956
Kendall	18.33	75.23	313.87	1313.06

Table 6.11: HCT results evaluation without outliers check method, maturity_size = 6 and top_maturity_size = 24, with Preemptive Cell Search over the HCT with the proposed covering radius and the update method

From Table 6.11, we can see that the number of results asked by the user has hardly influence on the retrieval time. This is because it is during the search among the intermediate level that most of the comparison operations are performed. The number of comparisons over the elements of the ground level, which depends on the number of results asked by the user, is negligible with respect to the number of comparison over the whole HCT body. The percentage of success in retrieving the query image increases with the number of results asked since the number of elements taken into account during the sorting process also does (the number of elements considered is at least twice the number of results asked). The Mean Competitive Recall cannot be directly compared since its value depends on the number of results. We have 7.21 out of 10 elements in common on average, which means 72.10% of the elements, 14.09 out of 20 (70.35%), 27.51 out of 40 (68.78%) and 53.67 out of 80 (67.09%). Therefore, the percentage of elements in common between the exact ranking and the approximated one gets slightly worse when the number of elements asked by the user increases. We come to the same conclusion analyzing the Normalized Aggregate Goodness since it also get slighty worse when the number of results increases. Finally, the behavior of Kendall distance is not linear behaviour but quadratic. In other words, the distance

computed over a ranking τ_1, which consist of twice the elements of a ranking τ_2, is expected to be four times the distance computed over τ_2. Therefore, since $18.33 \times 4 = 73.32 < 75.23$, $75.23 \times 4 = 300.92 < 313.87$ and $313.87 \times 4 = 1255.48 < 1313.06$ we also come to the same conclusion that the results get slightly worse when the number of results increases.

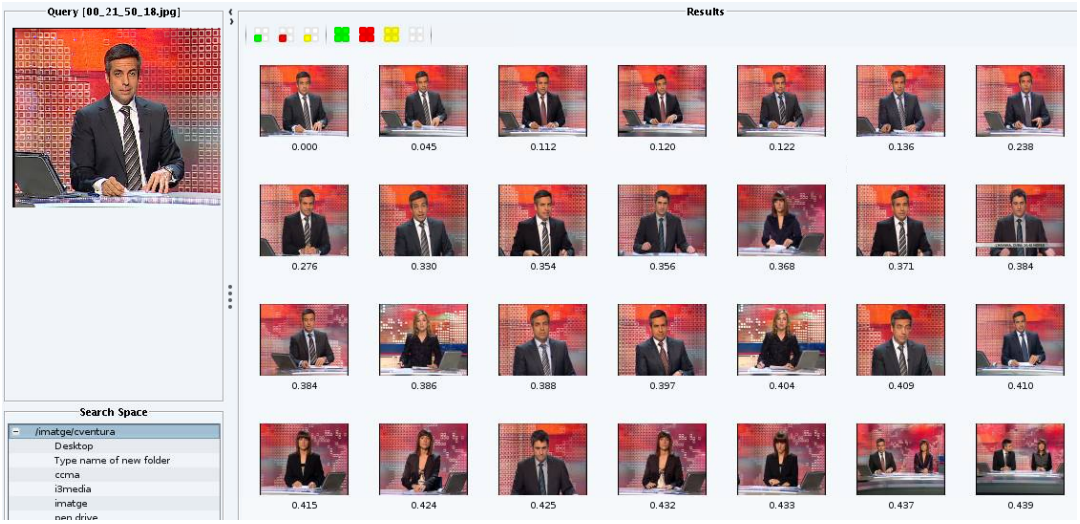
6.4 Query request examples

Now, we want to illustrate the performance of our retrieval system through several query request samples which have been carried out by using the GOS interface. We are going to show examples from different kinds of content which have been extracted from the CCMA database. In particular, the results to be shown have been taken from the set of 1,082 queries used in the retrieval system evaluation in Section 6.3.

The first example showed in Figure 6.8 is divided into two figures. The upper one represents the results returned by the retrieval system when the Preemptive Cell Search is used over the HCT (see Figure 6.8a). In particular, these results have been obtained by using the HCT with maturity_size = 6 and top_maturity_size = 24, with the updated method, without the outliers check method and 40 results asked by the user. We have decided these values for the maturity_size and the top_maturity_size since these ones were proposed in [KG07]. Moreover, the other assessed values did not change the performance of the system. On the contrary, the update method and the Preemptive Cell Search adapted as a searching technique result in a far better performance than Progressive Query, which is also proposed in [KG07] and is based on the Most Similar Nucleus technique. Therefore, this configuration for the retrieval system over the HCT will be also used for the rest of the examples. The query image is shown on the top-left corner in which there is an anchorperson. The lower figure (see Figure 6.8b) represents the results of an exhaustive (linear) search over the whole image database, i.e. the ranking with which the results obtained over the HCT are compared. The images marked with a green rectangle have been retrieved by the Preemptive Cell Search algorithm, i.e. they appear in the upper image, whereas the ones which are marked with a red rectangle are the missing ones in the approximated ranking.

From now on only the lower figure will be shown in the rest of examples. This is because this figure by itself gives information of both the exhaustive search and the

Preemptive Cell Search. Thus, all the images showed (it does not matter which is the color of the rectangle which are marked with) represent the exhaustive ranking whereas the images marked with a green rectangle form the top part of the approximated ranking obtained by the Preemptive Cell Search. In this way, we are not showing the elements which form the bottom part of the approximated ranking, i.e. the elements which do not belong to the exact K nearest neighbors but have been retrieved by the Preemptive Cell Search. However, we want to highlight the elements



(a) Results obtained by Preemptive Cell Search over the HCT

(b) Results obtained by a linear search over the image database

Figure 6.8: Comparison between the results obtained by using or not the indexing structure

Figure 6.9: Comparison between rankings from an anchorperson query image

which have not been retrieved by the retrieval system over the HCT since these elements give us an idea of how relevant is what the user is missing. Furthermore, we consider that the elements retrieved by the Preemptive Cell Search which are not shown in the lower figure are not so relevant since the user will discard quickly in his mind due to the following reasons: (*i*) they are the least similar elements to the query images among the results returned, and (*ii*) they belong to the bottom of the approximate ranking and the user tends to focus on the top of the ranking. Figure 6.9 shows an example in which another anchorperson is used as a query and where both exhaustive and approximate rankings are the same.

Figures 6.10, 6.11, 6.12, 6.13 and 6.14 show examples in which images from sports events have been used as queries. In particular, the image queries consist of a close-up and a pan shot from a soccer match in Figure 6.10 and Figure 6.11 respectively, a pan shot from a handball match in Figure 6.12 and a close-up and a pan shot from Formule One in Figures 6.13 and 6.14 respectively.

Next, Figures 6.15 and 6.16 show two more examples of the retrieval system in which two images from a political debate have been used as queries.

Finally, Figure 6.17 shows an example in which a forecast weather program is used as a query image whereas the query images used in Figures 6.18, 6.19 and 6.20 consist of images extracted from entertainment TV programs. Figure 6.21 shows a query request from TV series.

Figure 6.10: Comparison between rankings from a soccer match close-up query

Figure 6.11: Comparison between rankings from a soccer match pan shot query

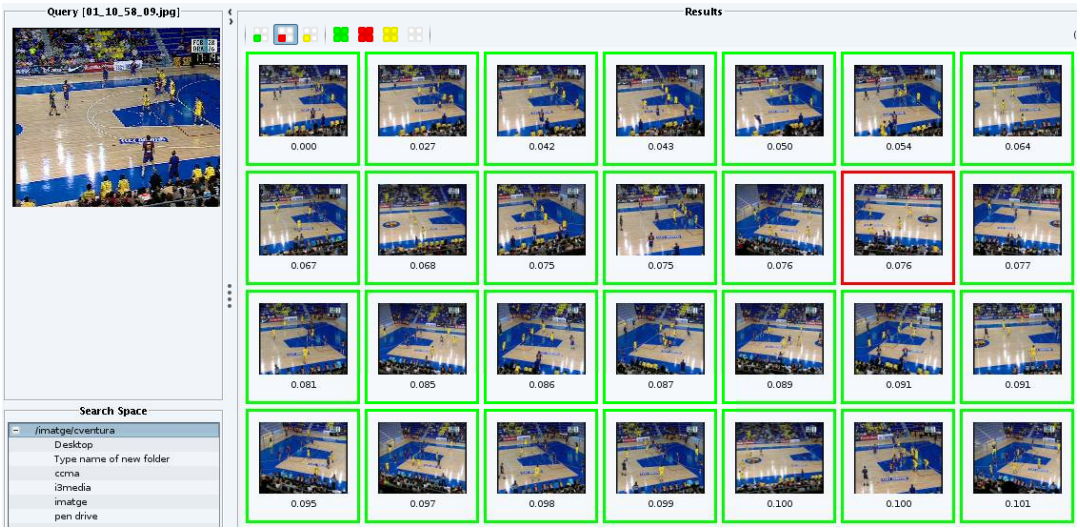


Figure 6.12: Comparison between rankings from a handball match pan shot query

Figure 6.13: Comparison between rankings from a Formule One close-up query

Figure 6.14: Comparison between rankings from a Formule One pan shot query

Figure 6.15: Comparison between rankings from a political debate query

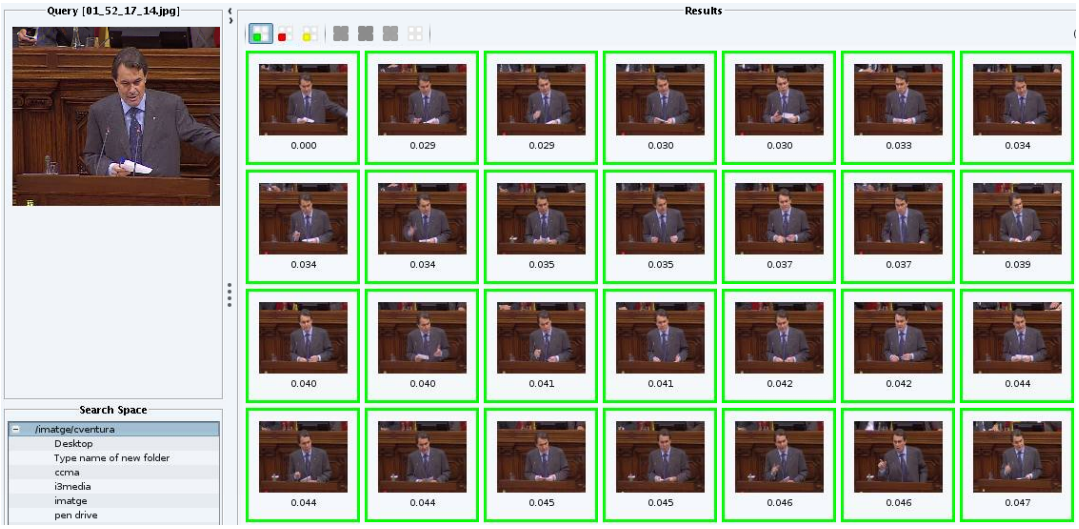


Figure 6.16: Comparison between rankings from a political debate query

Figure 6.17: Comparison between rankings from a forecast weather program query

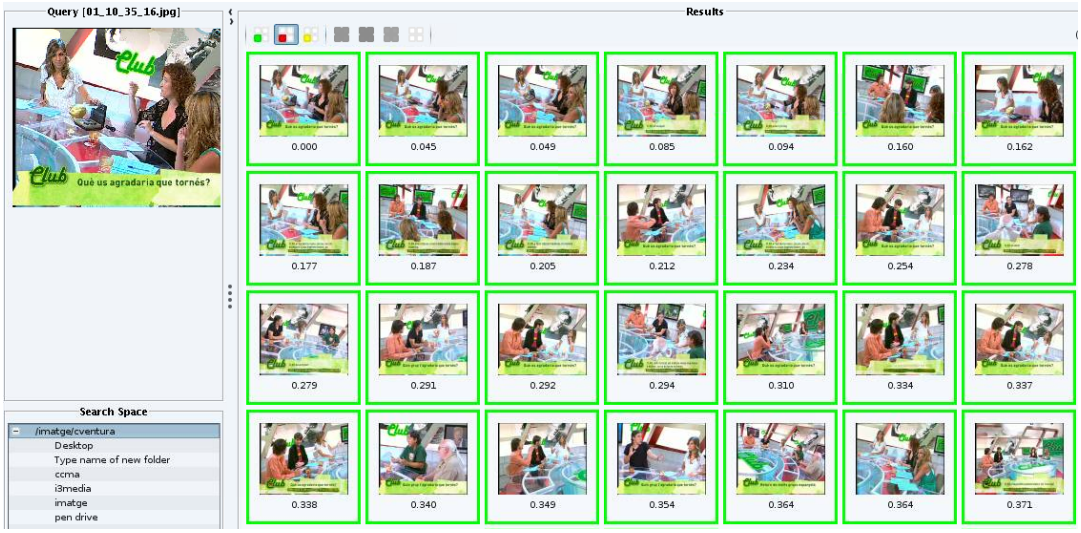


Figure 6.18: Comparison between rankings from an enternainment TV program

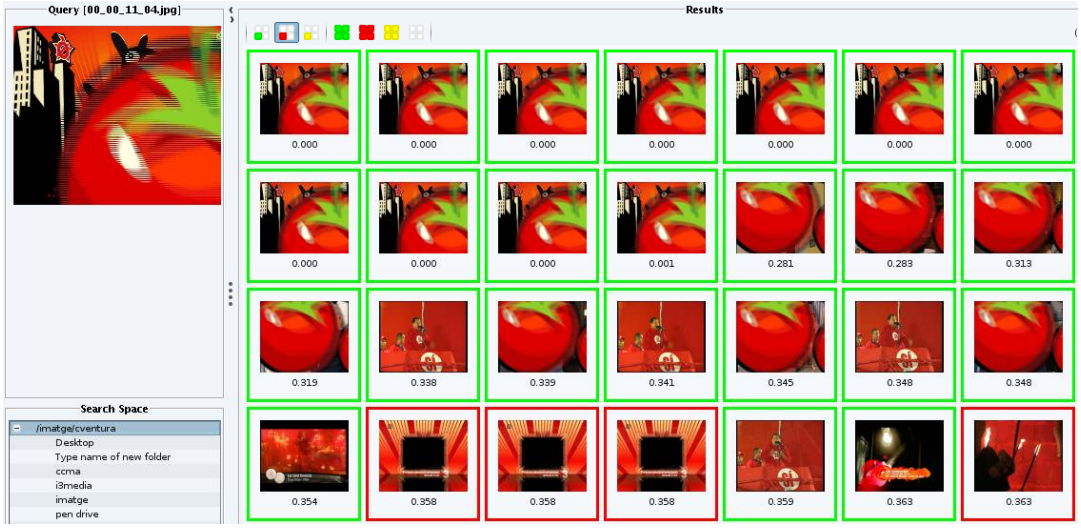


Figure 6.19: Comparison between rankings from an enternainment TV program

Figure 6.20: Comparison between rankings from an enternainment TV program

Figure 6.21: Comparison between rankings from a TV series

Chapter 7

Conclusions and Future Work

In this project, an indexing technique called Hierarchical Cellular Tree (HCT) [KG07] has been implemented in order to improve the retrieval times of the retrieval system based on some MPEG-7 visual descriptors which had been previously implemented in [Ven10]. HCT is a MAM-based indexing technique with a dynamical approach which allows changes in the indexed database. Therefore, new elements can be inserted whenever we want and existing elements can be removed from the database without having to rebuild the HCT from scratch.

Although the Hierarchical Cellular Tree has been chosen to be used in a particular scenario, i.e. a CBIR system based on some specific features, its implementation allows to index any type of data and, therefore, the HCT is useful for indexing a generic database as long as there is a function to measure the dissimilarity between any pair of elements.

Furthermore, some modifications have been proposed to the original HCT. Thus, (i) a redefinition of the covering radius by giving an approximation by excess and (ii) the implementation of a method which allows to update the covering radius to its actual value have resulted in an improvement of the quality of the retrieved elements (27.51 out of 40 elements retrieved on average and 99.26% of success in retrieving the query element) with respect to the retrieval system based on the original covering radius (12.35 out of 40 elements retrieved on average and 49.26% of success in retrieving the query element), which was an approximation by defect. In addition to this, the use of the Preemptive Cell Search algorithm, which is the basic method for the insertion of new elements, as a searching technique is another important contribution. Thus,

the retrieval system performance is far better in both retrieval time and retrieved results aspects when this technique is applied (a retrieval time of 0.8319 seconds and 27.51 out of 40 elements retrieved on average) in comparison to the Progressive Query system (a retrieval time of 1.3695 seconds and 12.33 out of 40 elements retrieved on average), which was proposed in [KG07].

Since there are no large databases with a ground truth for evaluating our retrieval system, a great effort has been done in order to evaluate it in the most possible objective way. Thus, the results obtained by the searching techniques over the HCT for a set of queries have been compared with the top k list containing the k nearest elements resulting from an exhaustive search over the whole database by using some contrasted measures between top k lists extracted from the literature. Therefore, the HCT has been built over a database which consists of 216,317 images obtained from a public service broadcaster.

According to the experimental results, we conclude that the proposed covering radius gives the best performance as long as the update method is used (a retrieval time of 0.8319 seconds and 27.51 out of 40 elements retrieved on average). Although the original covering radius gives by far the best retrieval time (0.1058 seconds on average), it is also the one which gives by far the worst retrieved results (12.35 out of 40 elements retrieved on average). However, the performance of the retrieval system based on the original covering radius is remarkably improved when the update method is applied once the HCT has been built (26.42 out of 40 elements retrieved on average) to the detriment of the retrieval time (1.0095 seconds on average), which gets remarkably worse than before the update method was applied. On the other hand, the main disadvantage of the proposed covering radius lies in the computational time required for building the HCT (472,534 seconds) in contrast to the time required when the original covering radius is used (50,868 seconds). However, this time can be reduced by applying periodically the update method for the covering radius during the HCT construction.

This project opens the door to new future work lines. The main work line consists in adapting the HCT implementation for working with very large databases which do not allow to be indexed by only using the main memory. Therefore, it is necessary to implement a new approach in which both the main memory and the hard disk are used. Thus, when a new element needs to be indexed or a search over the indexed

database is requested, the uppermost levels of the HCT will be loaded at the main memory and the lowest levels will remain stored at the hard disk. Then, when a cell from the lowest level of the HCT loaded at the main memory is reached, the subtree holding from that cell is also loaded at the main memory, whereas the upper levels are freed. Other approaches consisting in using data pointers instead of storing the visual descriptors in the HCT structure must be analyzed, although the computational time is expected to be increased since each comparison will result in an access to hard disk.

The implementation of this indexing technique will also allow to improve the retrieval time of any region-based CBIR system. In such a system, we are interested in searching images which contain a region similar to the query region. Thus, each image from the database is divided into a set of regions and the visual descriptors are computed independently for each region. Therefore, the number of elements to be compared is remarkably increased. For instance, if each image is represented by a set of 100 regions, an image database of 1,000 images results in 100,000 elements, which now can be seen as a large database. It is in this context that the HCT can be very useful. Thus, each element from the HCT will no longer represent an image, but a region, and the cells will contain similar regions according to the criteria used, which now can be based on new features such as shape descriptors.

Another future work line resulting from this project is the implementation of a browser for an image database as the HCT Browsing application proposed in [KG07]. Thus, HCT can provide a basis for accomplishing an efficient browsing scheme. The hierarchic structure of the HCT is quite appropriate to give an overview to the user about what lies under the current level. Therefore, a browser based on the HCT can turn out to be a guided tour among the database items if it is well supported via an user-friendly graphic interface. Thus, the browser can be also used as an alternative way to retrieve elements from a database.

Finally, in this project, the search operations have been always based on only one visual descriptor. However, for some query requests maybe more than one descriptor is required. For instance, the user may want to perform a search based not only in color but also in texture. Furthermore, in a region-based CBIR system, the visual descriptors based on shape play an important role. Therefore, several visual descriptors will be required by the user. As a consequence, it is necessary to study how to work with several descriptors at the same time. There are two options: (i) to build a different

HCT for each possible combination of descriptors, and (*ii*) to build one HCT for each descriptor and then fusion the results of the queries performed independently on each HCT. The former solution has a drawback: the number of HCTs to be built increases exponentially with the number of descriptors which can be used. The second solution builds one HCT for each descriptor. However, it has a potential drawback: if only the K nearest elements are retrieved over each HCT, the intersection of the obtained rankings could be empty. Therefore, it is necessary to develop a more sophisticated technique which deals with this problem.

Bibliography

[AI08] Alexandr Andoni and Piotr Indyk, *Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions*, Communications of the ACM **51** (2008), no. 1, 117.

[BBJ+00] S. Berchtold, C. Bohm, H.V. Jagadish, H.-P. Kriegel, and J. Sander, *Independent quantization: an index compression technique for high-dimensional data spaces*, Data Engineering, 2000. Proceedings. 16th International Conference on, 2000, pp. 577 –588.

[BBK98] Stefan Berchtold, Christian Böhm, and Hans-Peter Kriegal, *The pyramid-technique: towards breaking the curse of dimensionality*, Proceedings of the 1998 ACM SIGMOD international conference on Management of data (New York, NY, USA), SIGMOD '98, ACM, 1998, pp. 142–153.

[BFG+96] J. Bach, C. Fuler, A. Gupta, A. Hampapur, B. Horowitz, R. Humphrey, R. Jain, and C. Shu, *The virage image search engine: An open framework for image management*, SPIE Conference on Storage and Retrieval for Image and Video Databases IV, vol. 2670, March 1996, pp. 76–87.

[BKK96] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel, *The x-tree: An index structure for high-dimensional data*, Proceedings of the 22th International Conference on Very Large Data Bases (San Francisco, CA, USA), VLDB '96, Morgan Kaufmann Publishers Inc., 1996, pp. 28–39.

[BKSS90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger, *The r*-tree: an efficient and robust access method for points and rectangles*, SIGMOD Rec. **19** (1990), 322–331.

[BO97] Tolga Bozkaya and Meral Ozsoyoglu, *Distance-based indexing for high-dimensional metric spaces*, Proceedings of the 1997 ACM SIGMOD international conference on Management of data (New York, NY, USA), SIGMOD '97, ACM, 1997, pp. 357–368.

[Bri95] Sergey Brin, *Near neighbor search in large metric spaces*, Proceedings of the 21th International Conference on Very Large Data Bases (San Francisco, CA, USA), VLDB '95, Morgan Kaufmann Publishers Inc., 1995, pp. 574–584.

[Bus] *CENIT Buscamedia Project*, `www.cenitbuscamedia.es`.

[Chi] *Computers in the Human Interaction Loop (CHIL project*, `http://gps-tsc.upc.es/imatge/_JosepRamon/CHIL/CHIL.html`.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*, Second ed., Section 10.4: Representing rooted trees, pp. 214–217, MIT Press and McGraw-Hill, 2001.

[CM99] K. Chakrabarti and S. Mehrotra, *The hybrid tree: an index structure for high dimensional feature spaces*, Data Engineering, 1999. Proceedings., 15th International Conference on, mar 1999, pp. 440 –447.

[CPR+07] Flavio Chierichetti, Alessandro Panconesi, Prabhakar Raghavan, Mauro Sozio, Alessandro Tiberi, and Eli Upfal, *Finding near neighbors through cluster pruning*, Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (New York, NY, USA), PODS '07, ACM, 2007, pp. 103–112.

[CPRZ97] P. Ciaccia, M. Patella, F. Rabitti, and P. Zezula, *Indexing metric spaces with mtree*, Proc. Quinto convegno Nazionale SEBD, 1997, pp. 67–86.

[CWT08] Jingyu Cui, Fang Wen, and Xiaoou Tang, *Real time google and live image search re-ranking*, Proceeding of the 16th ACM international

conference on Multimedia (New York, NY, USA), MM '08, ACM, 2008, pp. 729–732.

[CY09] Silvia Cortés Yuste, *Interfaz gráfica de usuario para la búsqueda de imágenes basada en imágenes: Gos- graphic object searcher*, Master's thesis, Escola d'Enginyeria de Terrassa, 2009.

[FBF77] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel, *An algorithm for finding best matches in logarithmic expected time*, ACM Trans. Math. Softw. **3** (1977), 209–226.

[FJ03] Manuel J. Fonseca and Joaquim A. Jorge, *Indexing high-dimensional data for content-based retrieval in large databases*, Proceedings of the Eighth International Conference on Database Systems for Advanced Applications (Washington, DC, USA), DASFAA '03, IEEE Computer Society, 2003, pp. 267–.

[FKS03] Ronald Fagin, Ravi Kumar, and D. Sivakumar, *Comparing top k lists*, Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms (Philadelphia, PA, USA), SODA '03, Society for Industrial and Applied Mathematics, 2003, pp. 28–36.

[Ger07] Filippo Geraci, *Fast clustering for web information retrieval*, Ph.D. thesis, Universita degli Studio di Siena, Facoltá di Ingegnieria, Dipartaminto di Ingegnieria dell'Informazione, 2007.

[GiNVPT+10] Xavier Giro-i Nieto, Carles Ventura, Jordi Pont-Tuset, Silvia Cortes, and Ferran Marques, *System architecture of a web service for content-based image retrieval*, Proceedings of the ACM International Conference on Image and Video Retrieval (New York, NY, USA), CIVR '10, ACM, 2010, pp. 358–365.

[Gut84] Antonin Guttman, *R-trees: a dynamic index structure for spatial searching*, SIGMOD Rec. **14** (1984), 47–57.

[Het] Magnus Lie Hetland, *The Basic Principles of Metric Indexing*, Norwegian University of Science and Technology.

[HK92] Kyoji Hirata and Toshikazu Kato, *Query by visual example - content based image retrieval*, Proceedings of the 3rd International Conference

on Extending Database Technology: Advances in Database Technology (London, UK), EDBT '92, Springer-Verlag, 1992, pp. 56–71.

[HT09] Gang Hua and Qi Tian, *What can visual content analysis do for text based image search?*, Proceedings of the 2009 IEEE international conference on Multimedia and Expo (Piscataway, NJ, USA), ICME'09, IEEE Press, 2009, pp. 1480–1483.

[Jor09] Javier Vidal Jordana, *Sistema de control de múltiples cámaras activas en el entorno de una smart room*, Master's thesis, Escola Tècnica Superior d'Enginyeria de Telecomunicació de Barcelona, 2009.

[Ken70] Maurice G. Kendall, *Rank correlation methods [by] maurice g. kendall*, 4th ed. ed., Griffin, London,, 1970 (English).

[KG05] S. Kiranyaz and M. Gabbouj, *Novel multimedia retrieval technique: progressive query (why wait?)*, Vision, Image and Signal Processing, IEE Proceedings - **152** (2005), no. 3, 356 – 366.

[KG07] ———, *Hierarchical cellular tree: An efficient indexing scheme for content-based retrieval on multimedia databases*, Multimedia, IEEE Transactions on **9** (2007), no. 1, 102 –119.

[Knu97] Donald E. Knuth, *The Art of Computer Programming: Fundamental Algorithms*, Third ed., Section 2.3: Trees, pp. 308–423, Addison-Wesley, 1997.

[Kru56] Jr. Kruskal, Joseph B., *On the shortest spanning subtree of a graph and the traveling salesman problem*, Proceedings of the American Mathematical Society **7** (1956), no. 1, pp. 48–50 (English).

[KS97] Norio Katayama and Shin'ichi Satoh, *The sr-tree: an index structure for high-dimensional nearest neighbor queries*, Proceedings of the 1997 ACM SIGMOD international conference on Management of data (New York, NY, USA), SIGMOD '97, ACM, 1997, pp. 369–380.

[LJA11] Herwig Lejsek, Björn Thór Jónsson, and Laurent Amsaleg, *Nv-tree: nearest neighbors at the billion scale*, Proceedings of the 1st ACM International Conference on Multimedia Retrieval (New York, NY, USA), ICMR '11, ACM, 2011, pp. 54:1–54:8.

[LJF94] King Ip Lin, H. V. Jagadish, and Christos Faloutsos, *The tv-tree: an index structure for high-dimensional data*, The VLDB Journal **3** (1994), 517–542.

[LPSW06] Carl Lagoze, Sandy Payette, Edwin Shin, and Chris Wilper, *Fedora: an architecture for complex objects and their relationships*, Int. J. Digit. Libr. **6** (2006), 124–138.

[Man02] B. S. Manjunath, *Introduction to mpeg-7, multimedia content description interface*, John Wiley and Sons, Ltd., Jun 2002.

[NBE+93] Wayne Niblack, Ron Barber, William Equitz, Myron Flickner, Eduardo H. Glasman, Dragutin Petkovic, Peter Yanker, Christos Faloutsos, and Gabriel Taubin, *The qbic project: Querying images by content, using color, texture, and shape*, Storage and Retrieval for Image and Video Databases, 1993, pp. 173–187.

[PPS95] A. Pentland, R. W. Picard, and S. Sclaroff, *Photobook: Content-based manipulation of image databases*, 1995.

[SC96] John R. Smith and Shih-Fu Chang, *Visualseek: a fully automated content-based image query system*, Proceedings of the fourth ACM international conference on Multimedia (New York, NY, USA), MULTIMEDIA '96, ACM, 1996, pp. 87–98.

[SLL01] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine, *Boost graph library, the: User guide and reference manual*, Addison-Wesley Professional, 2001.

[SMR04] Pavan Kumar C. Singitham, Mahathi S. Mahabhashyam, and Prabhakar Raghavan, *Efficiency-quality tradeoffs for vector score aggregation*, Proceedings of the Thirtieth international conference on Very large data bases - Volume 30, VLDB '04, VLDB Endowment, 2004, pp. 624–635.

[SYUK00] Yasushi Sakurai, Masatoshi Yoshikawa, Shunsuke Uemura, and Haruhiko Kojima, *The a-tree: An index structure for high-dimensional spaces using relative approximation*, 2000, pp. 516–526.

[TSS04] J. Pokorny T. Skopal and V. Snasel, *Pm-tree: Pivoting metric tree for similarity search in multimedia databases*, Local proceedings of the 8th East-European Conference on Advances in Databases and Information Systems (ADBIS) (Budapest, Hungary), 2004, pp. 99–114.

[TTSF00] Caetano Traina, Agma Traina, Bernhard Seeger, and Christos Faloutsos, *Slim-trees: High performance metric trees minimizing overlap between nodes*, In 7th International Conference on Extending Database Technology (EDBT, Springer-Verlag, 2000, pp. 51–65.

[Ven10] Carles Ventura, *Image-based query by example using mpeg-7 visual descriptors*, Master's thesis, Escola Tècnica Superior d'Enginyeria de Telecomunicació de Barcelona, 2010.

[WJ96] David A. White and Ramesh Jain, *Similarity indexing with the ss-tree*, Proceedings of the Twelfth International Conference on Data Engineering (Washington, DC, USA), ICDE '96, IEEE Computer Society, 1996, pp. 516–523.

[WP01] Haixun Wang and Chang-Shing Perng, *The s2-tree: An index structure for subsequence matching of spatial objects*, Proceedings of the 5th Pacific-Asia Conference on Knowledge Discovery and Data Mining (London, UK), PAKDD '01, Springer-Verlag, 2001, pp. 312–323.

[WSB98] Roger Weber, Hans-Jörg Schek, and Stephen Blott, *A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces*, Proceedings of the 24rd International Conference on Very Large Data Bases (San Francisco, CA, USA), VLDB '98, Morgan Kaufmann Publishers Inc., 1998, pp. 194–205.

[Yia93] Peter N. Yianilos, *Data structures and algorithms for nearest neighbor search in general metric spaces*, Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms (Philadelphia, PA, USA), SODA '93, Society for Industrial and Applied Mathematics, 1993, pp. 311–321.

[ZH00] Xiang Sean Zhou and Thomas S. Huang, *Cbir: From low-level features to highlevel semantics*, Proc. SPIE Image and Video Communication and Processing, 2000, pp. 24–28.

[ZWYY03] Xiangmin Zhou, Guoren Wang, Jeffrey Xu Yu, and Ge Yu, *M+-tree: a new dynamical multidimensional index for metric spaces*, Proceedings of the 14th Australasian database conference - Volume 17 (Darlinghurst, Australia, Australia), ADC '03, Australian Computer Society, Inc., 2003, pp. 161–168.