# *Hierarchical Cellular Tree*: An Efficient Indexing Scheme for Content-based Retrieval on Multimedia Databases

Serkan Kiranyaz and Moncef Gabbouj

***Abstract*— One of the challenges in the development of a content-based multimedia indexing and retrieval application is to achieve an efficient indexing scheme. The developers and users who are accustomed to making queries to retrieve a particular multimedia item from a large scale database can be frustrated by the long query times. Conventional indexing structures cannot usually cope with the requirements of a multimedia database, such as dynamic indexing or the presence of high-dimensional audiovisual features. Such structures do not scale well with the ever increasing size of multimedia databases whilst inducing corruption and resulting in an over-crowded indexing structure. This paper addresses such problems and presents a novel indexing technique, *Hierarchical Cellular Tree*, which is designed to bring an effective solution especially for indexing large multimedia databases. Furthermore it provides an enhanced browsing capability, which enables user to make a guided tour within the database. A pre-emptive cell search mechanism is introduced in order to prevent corruption, which may occur due to erroneous item insertions. Among the hierarchical levels that are built in a bottom-up fashion, similar items are collected into appropriate cellular structures at some level. Cells are subject to mitosis operations when the dissimilarity exceeds a required level. By mitosis operations, cells are kept focused and compact and yet, they can grow into any dimension as long as the compactness is maintained. The proposed indexing scheme is then used along with a recently introduced query method, the *Progressive Query*, in order to achieve the ultimate goal, from the user point of view that is retrieval of the most relevant items in the earliest possible time regardless of the database size. Experimental results show that the speed of retrievals is significantly improved and the indexing structure shows no sign of degradations when the database size is increased. Furthermore, *HCT* indexing body can conveniently be used for efficient browsing and navigation operations among the multimedia database items.**

***Index Terms*— content-based retrieval, metric access methods, multimedia databases, similarity-based indexing.**

## I. Introduction

It is a known fact that recent technological hardware and network improvements along with the daily usage of Internet have caused a rapid increase in the size of digital audio-visual information that is used, handled and stored via several applications. Besides several benefits and usages, such massive collection of information has brought storage and especially management problems. In order to overcome such problems several content-based indexing and retrieval techniques and applications have been developed such as MUVIS system [18], [19], [24], Photobook [28], VisualSEEk [34], Virage [39], and VideoQ [9], all of which are designed to bring a framework structure for handling and especially the retrieval of the digital multimedia items such as images, audio and/or video clips. In such frameworks, database primitives are mapped into some high dimensional feature domain, which may consist of several types of features such as visual, aural, etc. From latitude of low-level features, careful selection of the feature sets to be used for a particular application may capture the semantics of the database items in a content-based multimedia retrieval (CBMR) system. In this way the similarity between two database items can be estimated by calculating the (dis-) similarity distance between their feature vectors. Such distances produce a ranking order of similar multimedia items within the database. This is the general query-by-example (QBE) scenario, which on the other hand is costly and CPU intensive especially for large multimedia databases. This fact brought a need for indexing techniques, which will organize the database in such a way that the query time and I/O operations could be reduced. The indexing techniques can be mainly grouped in two categories: Spatial and Metric Access Methods (SAMs and MAMs). However, both types have significant drawbacks for the indexing of large-scale multimedia databases. SAMs are, by nature, not suitable for this purpose due to strict assumptions and several well-known limitations they present. For instance the applicability of SAMs is limited by the fact that items have to be represented by the points in *N* dimensional feature space and the (dis-)similarity distance between two points has to be based on a distance function in $L_p$ metric such as *Euclidean* distance. Furthermore SAMs, while providing good results on low dimensional feature space do not scale up well to high dimensional spaces due to the phenomenon so called "the curse of dimensionality". Recent studies [37] show that most of the SAM-based indexing schemes even become less efficient than sequential indexing for dimensions higher than 10. Especially large multimedia databases might contain many visual and aural features exceeding this limit multiple times. A more general approach can be obtained by MAMs, which basically comes from the fact that any MAM employs the indexing process by assuming only the availability of a similarity distance function that is a norm. Therefore, in a multimedia database with several multi-dimensional features, as long as a similarity distance function that is usually treated as a "black box" by the underlying MAM, exists the database can be indexed by any MAM. Yet the existing MAMs present several drawbacks for similarity-based indexing of multimedia databases. The static MAMs, for instance, do not support dynamic changes (new insertions or deletions); whereas this is an essential requirement during the incremental construction of a multimedia database. Even though M-tree [10] 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.

In order to overcome such problems and provide efficient solutions to the aforementioned shortcomings of the indexing algorithms for the multimedia databases, we develop a MAM-based, dynamic and self-organized indexing scheme, the *Hierarchical Cellular Tree* (*HCT*). As its name implies, *HCT* has a hierarchic structure, which is formed into one or more levels. Each level is capable of holding one or more cells. A cell corresponds to a node in an M-tree. The reason for the different name is because each cell further contains a tree structure, a Minimum Spanning Tree (MST), which refers to the database objects (their database representations and basically their descriptors) as its MST nodes. Among all indexing structures available, M-tree shows the highest structural similarity to *HCT*, such as:

- Both indexing schemes are MAM-based and have a similar hierarchical structure, i.e. levels.
- They are both created dynamically, in a bottom-up fashion. The tree grows one level upwards whenever a split occurs in the top level cell.
- Except the top level cell, each cell is represented by a nucleus (routing) object in the higher level.

However, there are several major differences in their design philosophies and objectives:

- M-tree is designed to achieve a balanced tree with a low I/O cost in large data set. *HCT* is on the other hand 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 significant variations on size and density.
- M-tree depends on a maximum (fixed size) capacity M. Therefore, its performance depends on a “good” choice of this parameter with respect to the database size and thus, M-tree construction significantly varies with it. However, for multimedia databases the database size is dynamic and its content may vary significantly. *HCT*, on the other hand, has no limit for the cell size as long as the cell keeps a definite “compactness” measure.
- In M-tree the cell compactness is only measured with respect to distance of the routing (nucleus) object to the farthest object that is so called the covering radius. Due to the aforementioned reasons of unreliability on such single measure for the cell compactness, *HCT* uses all cell items and their minimum distances to the cell (instead of a single nucleus item alone) to define a regularization function that represents a dynamic model for the cell compactness. During the lifetime of the *HCT* body (i.e. item insertions, removals, fitness checks, post-reactions, etc.) this function dynamically updates the current cell compactness feature, which is then compared to a certain statistically driven level threshold value to decide whether or not the cell should be split (mitosis).
- The split policies and objectives are also different between M-tree and *HCT*.
- The insertion processes differ significantly in terms of cell search operations. M-tree insertion operation is based on “Most-Similar Nucleus” (MS-Nucleus in this article) cell search, which depends on a simple heuristics which assumes that the closest nucleus item (aka “routing object”) yields the best sub-tree during the descend and finally the best (target) cell to be appended. In this paper, we will show that this is not always a valid assumption and it is a potential cause for corruption since it may lead to sub-optimum insertions especially for large databases due to the “crowd effect”. *HCT* is designed to perform an optimum search for the target cell to which the incoming item should belong. This search, so called Pre-emptive cell search, during descend at each level verifies all possible paths that are likely to yield a better nucleus item (and hence a better cell at a lower level) in an iterative way. By this way, along with the mitosis operation this search algorithm further improves the cell compactness factor at each level.
- M-tree has a conservative structure that might cause degradations in due time. For example, the cell nucleus (routing object) is not changed after an insertion or removal operation even though another item might now be a more suitable candidate for being the cell nucleus. On the contrary, *HCT* has a totally dynamic approach. Any operation (insertion, removal or mitosis) can change the current cell nucleus to a new (better) one.

The rest of this paper is organized as follows: Section 2 presents the related work in the area of indexing and retrieval. In Section 3 we introduce the generic *HCT* design philosophy and implementation details. Section 4 is devoted to QBE operations over *HCT* indexing structure. A novel browsing scheme, *HCT* Browsing, is discussed in Section 5. Section 6 presents the experimental results. Finally, Section 7 concludes the paper and discusses some future research topics.

## II. Related Work

For the past three decades, researchers proposed several indexing techniques that are formed mostly in a hierarchical tree structure that is used to cluster (or partition) the feature space. Initial attempts such as KD-Trees [2] used space-partitioning methods that divide the feature space into predefined hyperplanes regardless of the distribution of the feature vectors. Such regions are mutually disjoint and their union covers the entire space. In R-tree [12] the feature space is divided according to the distribution of the database items and region overlapping may occur as a result. Both KD-tree and R-tree are the first examples of Spatial Access Methods (SAMs). Afterwards several enhanced SAMs have been proposed. R*-tree [1] provides a consistently better performance by introducing a policy called “forced reinsert” than the R-tree and R+-tree [32]. R*-tree also improves the node splitting policy of the R-tree by taking overlapping area and region parameters into consideration. Lin et al. proposed TV-tree [25], which uses so-called telescope vectors. These vectors can be dynamically shortened assuming that only dimensions with high variance are important for the query process and therefore low variance dimensions can be neglected. Berchtold et al. [5] introduced X-tree, which is particularly designed for indexing higher dimensional data. X-tree avoids overlapping of region bounding boxes in the directory structure by using a new organization of the directory and as a result, X-tree outperforms both TV-tree and R*-tree significantly. It is 450 times faster than R-tree and between 4 to 12 times faster than the TV-tree when the dimension is higher than two and it also provides faster insertion times. Still bounding rectangles can overlap in higher dimensions. In order to prevent this, White and Jain proposed the SS-tree [38], an alternative to R-tree structure, which uses minimum bounding spheres instead of rectangles. Even though SS-tree outperforms R*-tree, the overlapping in the high dimensions still occurs. Thereafter, several other SAM variants are proposed such as SR-tree [14], $S^2$-Tree [36], Hybrid-Tree [8], A-tree [31], IQ-tree [3], Pyramid Tree [4], NB-tree [11], etc. The aforementioned degradations and shortcomings prevent a wide spread usage of SAM based indexing structures especially on multimedia collections. In order to provide a more general approach to similarity indexing for multimedia databases, several MAM-based indexing techniques have been proposed. Yianilos [40] presented vp-tree that is based on partitioning the feature vectors (data points) into two groups according to their similarity distances with respect to a reference point, so called vantage point. Bozkaya and Ozsoyoglu [6] proposed an extension of vp-tree, so-called mvp-tree (multiple vantage point), which basically assigns m vantage points to a node with a fan out of $m^2$. They reported 20% to 80% reduction of similarity distance computation compared to vp-trees. Brin [7] introduced Geometric Near-Neighbor Access Tree (GNAT) indexing structure, which chooses k number of split points at the top level and each of the remaining feature vectors are associated with the closest split points. GNAT is then built recursively and the parameter *k* is chosen to be a different value for each feature set depending on its cardinality. Koikkalainen and Oja introduced TS-SOM [20] that is used in PicSOM [22] as a CBIR indexing structure. TS-SOM provides a tree-structured vector quantization algorithm. Other similar SOM-based approaches are introduced by Zhang and Zhong [41], and Sethi and Coman [33]. All SOM-based indexing methods rely on training of the levels using the feature vectors and each level has a pre-fixed node size that has to be arranged according to the size of the database. This brings a significant limitation, that is, they are all static indexing structures, which do not allow dynamic construction or updates for a particular database. Retraining and costly reorganizations are required each time the content of the image database changes (i.e. new insertions or deletions), that is indeed nothing but rebuilding the whole indexing structure from scratch. Similarly the rest of the MAMs so far addressed present several shortcomings. Contrary to SAMs, these metric trees are designed only to reduce the number of similarity distance computations, paying no attention to I/O costs (disk page accesses). They are also intrinsically static methods in the sense that the tree structure is built once and new insertions are not supported. Furthermore, all of them build the indexing structure from top to bottom and hence the resulting tree is not guaranteed to be balanced. Ciaccia et al. [10] proposed M-tree to overcome such problems. M-tree is a balanced and dynamic tree, which is built from bottom to top, creating a new root level only when necessary. The node size is a fixed number, M, and therefore, the tree height depends on M and the database size. Its performance optimization concerns both CPU computational time for similarity distances and I/O costs for disk page accesses for feature vectors of the database items. Recently, Traina et al. [35] proposed Slim-tree, an enhanced variant of M-trees, which is designed for improving the performance by minimizing overlaps between nodes. They introduced two parameters, "fat-factor" and "bloat-factor", to measure the degree of overlap and proposed the usage of Minimum Spanning Tree (MST) [21], [29], for splitting the node. Another slightly enhanced M-tree structure, so-called M+-tree, can be found in [42].

Along with the indexing techniques addressed so far, certain query techniques have to be used to speed up a query process within indexed databases. The most common query techniques are as follows:

- *Range Query*: Given a query object, Q, a maximum similarity distance range, ε, and a non-negative similarity distance function SD, the range query selects all indexed database items, $Q_i$, such that SD (Q, $Q_i$) < ε.
- *kNN Query*: Given a query object, Q, and an integer number k > 0, kNN query selects the k database items, which have the shortest similarity distance from Q.

Both query techniques may not provide efficient retrieval scheme from the user point of view due to their parameter dependency. For instance, range queries require a distance parameter, ε, where the user may not be able to provide such a number prior to a query process since it is not obvious how to find out a suitable range value if the database contains various types of features and feature subsets. Similarly, for a *kNN* query the parameter k might be hard to determine since if chosen too small the database may provide a large number of similar (relevant) items than required, and if too big, unnecessary CPU time might have been wasted for that query process if only a much smaller number was in fact needed. In general, both query techniques require several trials to converge to a successful retrieval result and this might remove the speed benefit of the underlying indexing scheme, if there is any.

In order to eliminate such drawbacks and provide a faster query scheme, recently a novel retrieval scheme, the *Progressive Query* (*PQ*), has been proposed [15]. *PQ* is a retrieval (via query) technique, which can be performed over the databases with or without the presence of an indexing structure. When the database has an indexing structure, PQ can replace *kNN* and range queries whenever a *Query Path* (*QP*) over which *PQ* proceeds, can be formed. Instead of relying on some unknown parameters such as k or ε, *PQ* provides periodic query results along with the query process and allows the user to stop the query in case the results obtained so far are satisfactory. Therefore, the proposed (*HCT*) indexing technique has been designed to work in harmony with *PQ* in order to evaluate the retrieval performance in the end, i.e. how fast the most relevant items can be retrieved or how efficient *HCT* can provide a *QP* for a particular query item.

## III. *HCT* OVERVIEW

*HCT* is a dynamic, cell–based and hierarchically structured indexing method, which is purposefully designed for *PQ* operations and advanced browsing capabilities within large multimedia databases. It is mainly a hierarchical clustering method where items are partitioned depending on their relative distances and stored within cells on the basis of their similarity proximity. The similarity distance function implementation is a *black-box* for the *HCT*. Furthermore, *HCT* is a self-organized tree, which is implemented via genetic programming principles. This basically means that the operations are not externally controlled; instead each operation such as item insertion, removal, mitosis, etc. are carried out according to some internal rules within a certain level and their outcomes may uncontrollably initiate some other operations on other levels. Yet all such "reactions" terminate in a limited time, that is, for any action (i.e. an item insertion), its consequent reactions will not last indefinitely due to the fact that each of them can occur only in a higher level and any *HCT* body has naturally a finite number of levels. In the following sub-sections, we will detail the basic structural components of the *HCT* body and then explain the indexing operations in an algorithmic way.

## A. *Cell Structure*

A cell is the basic container structure, in which similar database items are stored. The ground level cells contain the entire database items. Each cell further carries a MST whose nodes span all items in the cell. This internal MST is used to keep the minimum (dis-) similarity distance of each individual item to the rest of the items in the cell. So this scheme resembles MVP-tree [6] structure; however instead of using some (pre-fixed) number of items, all cell items are now used as vantage points for any (other) cell item. These item-cell distance statistics are mainly used to calculate the cell compactness. In this way we can have a better idea about the similarity proximity of any item instead of comparing it only with a single item (i.e. the cell nucleus) and hence a better compactness feature. The compactness algorithm is a *black-box* implementation. Here, we use a regularization function obtained from the statistical analysis using the MST and some cell data. This dynamic feature can then be used to decide whether or not to perform mitosis within the cell at any instant. If mitosis is granted, MST is again used to decide where the split should occur and the longest branch of the MST is the natural choice for this. Furthermore, MST is used to update cell nuclei to the most suitable item after any operation is completed within the cell.

In *HCT*, the cell size is kept entirely flexible and varies with no upper bound. However, similar to organic cells, *HCT* cells are not allowed to undergo mitosis before reaching a certain level of maturity. Otherwise one cannot obtain reliable information whether or not the cell is ready for mitosis since there is simply not enough statistical data that are gathered from the cell items and its MST. Therefore, a maturity cell size (e.g. $N_M \geq 5$) is set for all cells in *HCT* body (level independent) except the top level. Since the top level is the unique level hosting a single cell, the latter may be allowed to have a moderate maturity cell size (i.e. $N_M^T \geq 10$), possibly set as a user preference since the top level (cell) can be thought as a "Table of Contents" of the database whilst giving a summary of the overall *HCT* body. On the other hand, the maturity cell size should not be confused with parameter *M* for M-tree where *M* is used to enforce mitosis for a cell with size *M* irrespective of the cell condition (i.e. compactness) is. In *HCT*, we set minimum size as a pre-requisite condition for a cell to undergo mitosis. This is not a significant parameter, which neither affects the overall performance of *HCT* nor needs to be proportional to the database size or any other parameter, as is the case for M-tree.

*1) MST Formation:* Let $G=\{N,B\}$ be a connected and weighted graph, where $|N|=n$ nodes (vertices) and $|B|=b$ branches (edges). Let $w_i$ represents the $i^{th}$ branch weight. A spanning tree of *G* is a subgraph $S=\{N,B_S\}$ where $B_S \subseteq B$. The overall weight of S can be defined as the cumulative weight of its branches, i.e. $W_S=\sum_{B_S} w_i$. MST of *G* can then be defined as the (unique) spanning tree with minimum cumulative (total) weight. There are several MST construction algorithms, such as Kruskal's [21] and Prim's [29]. Those algorithms are, however, static algorithms, that is, all MST branches with their weights should be known beforehand. Since MST nodes represent database items, this requires a priori calculation of the relative similarity distances and hence yields a $O(n^2)$ computational cost. In *HCT* cells and their MST should be constructed dynamically (incrementally) since items can be inserted any time and it would be infeasible to re-construct MST from scratch each time a new item is inserted since such an operation would require $O(n^3)$ computations. Therefore, an incremental MST construction algorithm is adopted based on leaf node (vertex) pruning and branch (edge) contraction [13]. This is a sequential algorithm and has $O(n)$ computational complexity per (incoming) item and hence $O(n^2)$ overall cost as desired.

*2) Cell Nucleus:* Cell nucleus is the item, which represents the owner cell on the higher level(s). Since during the top-down cell search for an item insertion, these nucleus items are used to decide the cell into which the item should be inserted, it is therefore essential to promote the best item for this representation on any instant. When there is only one item in the cell, it is obviously the nucleus item of that cell. Otherwise the nucleus item is assigned by using the cell MST as the item having the maximum number of branches (connections to other items). This heuristics makes sense since it is the unique item to which most of the items have the closest proximity to it (according to the MST optimality on the minimal branch weights). Contrary to static nucleus assignment of the some other MAM-based indexing schemes such as M-tree, the cell nucleus is dynamically verified and if necessary updated for any *HCT* cell whenever an operation is performed over the cell in order to maintain the best representation of the (dynamically changing) cell and there is no computational cost for this so far since it can be extracted directly from the "ready" MST (branch) data.

*3) Cell Compactness:* Cell compactness quantifies how tight (focused) the clustering for the items within the cell. Furthermore, the regularization function implementation for the calculation of the cell compactness value is in general a *black box* for *HCT*. In this sub-section we will present the statistical parameters of this function used in the experiments.

Due to "semantic gap" the discrimination power of the low-level visual or aural features can be quite limited. Consequently, high variations might occur among the similarity distances calculated between a single item (i.e. a vantage point) and a group of "similar" items and this naturally creates a major problem if the compactness measure would be based on a single nucleus item. This is the main reason why instead of using a single (nucleus) item to find out the similarity of a new (incoming) item, multiple vantage points, i.e. all cell items for a *HCT* cell, are used. Once a cell reaches maturity (a pre-requisite for evaluating cell compactness) reliable first order statistics can thus be obtained from the branch weights of cell MST. Using also the covering radius, a regularization function, *f*, providing a model for the compactness feature of the cell, $CF_C$, can then be formed as follows:

$$CF_C = f(\mu_C, \sigma_C, r_C, \max(w_C), N_C) \geq 0 \qquad (1)$$

where $\mu_C$ and $\sigma_C$ are the mean and standard deviation of the MST branch weights, $w_C$, of cell *C*. $r_C$ is the covering radius, that is the distance from the nucleus to the farthest item in the cell and $N_C > N_M$ is the number of items in the cell *C*. The regularization function can then be formed in such a way that higher values of all the statistical parameters are to be penalized since a better compactness can be achieved via minimizing all whilst $N_C$ increases gradually with the item insertions. In the limit, the highest compactness is achieved when $CF_C = 0$ which means that all cell items are identical.

Similar to continuous updates for the nucleus item, the $CF_C$ value is also updated (recalculated) each time an operation is performed over the cell *C*. The new (updated) $CF_C$ is then compared with the current level compactness threshold, $CThr_L$, that is dynamically calculated within each level and if the cell is mature but not compact enough, i.e. $CF_C > CThr_L$, mitosis is therefore, granted for that cell.

*4) Cell Mitosis:* As explained earlier there are two conditions necessary for a mitosis operation: maturity (i.e. $N_C > N_M$**)** and cell compactness (i.e. $CF_C > CThr_L$). Both conditions are checked after an operation (e.g. item insertion or removal) occurs within the cell in order to signal a mitosis operation. Due to the presence of MST within each cell, mitosis has no computational cost in terms of similarity distance calculations. The cell is simply split by breaking the longest branch in MST and each of the newborn child cells is formed using each of the MST partitions. A sample mitosis operation is illustrated in Figure 1.

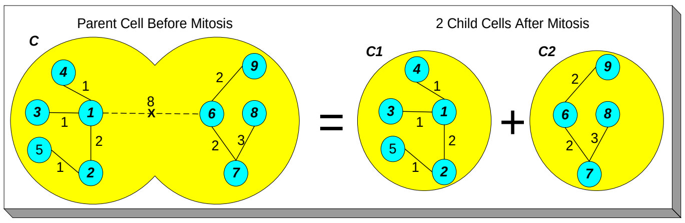

**Figure 1: A Sample Mitosis operation over a mature cell C.**

## B. *Level Structure*

*HCT* body is hierarchically partitioned in one or more levels, as one sample example shown in Figure 5. In this example there are three levels that are used to index 18 items. Apart from the top level, each level contains various number of cells that are created by mitosis operations, which have occurred on that level. The top level contains a single cell and when this cell splits, a new top level is created above this level. As mentioned earlier, the nucleus item of each cell on a particular level is represented on the higher level.

Each level logs the operations performed on it, such as the number of mitosis operations and the compactness of the cells. Note that each level tries to dynamically maximize the compactness of their cells. This however is not a straightforward process since incoming items may not exhibit a close similarity to the items present in the cells, and therefore, such dissimilar item insertions will cause a temporary degradation of the overall (average) compactness of the level. So each level, while analyzing the effects of the (recent) incoming items on the overall level compactness, should employ necessary management steps to provide a *trend* of improving compactness in due time (i.e. with future insertions). Within a period of time (i.e. during a number of insertions or after some number of mitosis operations), each level updates its compactness threshold according to the compactness of mature cells, into which items were inserted. In our earlier work [17], where an initial *HCT* indexing scheme is first designed, we used a simple, average-based setting for $CThr_L$, such as:

$$CThr_L = \frac{k_0}{P} \sum_{C \mid N_C > N_M}^{C \in S_P} CF_C = k_0 \mu_{CF_C} \quad \forall C \in S_P \qquad (2)$$

where $S_P$ is the set of mature cells on level *L*, upon which *P* insertions have recently been performed and $k_0 \geq 0$ is the inverse of compactness trend factor, which determines how much enhancement will be targeted for the next *P* insertions beginning from the moment of the latest $CThr_L$ setting. Although this function gives fairly good results for most of the cases, it is significantly effected by the extreme cases where $CF_C$ is too high or too low for some cells during *P* insertions. Therefore, it might show a noisy behavior due to random item insertions and the danger of over- or under-splitting cells emerges. A robust and more convergent $CThr_L$ function can be expressed in Eq. (3)

$$CThr_L = \frac{1}{k_0} Median\,(CF_C \mid \forall C \in S_M) \qquad (3)$$

where $S_M$ is the set of mature cells present in the current *HCT* body and $k_0 > 0$ is the compactness trend factor, which determines how much flexibility can be allowed for incoming insertions starting from the moment of the latest $CThr_L$ setting. If $k_0 = 1$, the trend is built upon keeping the current level of compactness intact and so no enhancement will be targeted for future insertions. On the other hand, when $k_0 \to \infty$ then the cells will split each time they reach maturity and in this case *HCT* split policy will be identical to M-tree. The *Median* operator keeps the extreme cases out from the $CThr_L$ calculation for future insertions and hence continuously tracks a median cell maturity level. Its convergence behavior can be seen in Figure 8 (top) for a sample incremental *HCT* formation experiment.

## C. *HCT Operations*

There are mainly three *HCT* operations: cell mitosis, item insertion and removal. Cell mitosis can only happen as a post processing after any of the other two *HCT* operations occurs. Both item insertion and removal are generic *HCT* operations that are identical for any level. Item insertion is performed as one item into one level at a time; whereas, item removal is a cell-based operation meaning that items belonging to the same cell can be removed in a single step. In the following sub-sections, we will present the algorithmic details of both operations.

1) *Item Insertion Algorithm for HCT:* Let *nextItem* be the item to be inserted into a target level indicated by a number, *levelNo*. Accordingly, the **Insert** algorithm can be expressed as follows:

```
Insert (nextItem, levelNo)
  ➢ Let top level number: topLevelNo and the single cell in top level: cell-T
  ➢ If(levelNo > topLevelNo) then do:
      o Create a new top level: level-T with number = topLevelNo+1
      o Create a new cell in level-T: cell-T
      o Append nextItem into cell-T.
      o Return.
  ➢ Let the Owner (target) cell in level levelNo: cell-O
  ➢ If(levelNo = topLevelNo ) then do:
      o Assign cell-O = cell-T
  ➢ Else do:
      o Create a cell array for Pre-emptive cell search: ArrayCS[], put cell-T into it
      o Assign cell-O = PreEmptiveCellSearch (ArrayCS[], nextItem, topLevelNo)
  ➢ Append nextItem into cell-O.
  ➢ Check cell-O for Post-Processing:
      o If cell-O is split then do:
          ▪ Let item-O, item-N1 and item N2 be old nucleus item (parent) and new nucleus items.
          ▪ Remove( item-O, levelNo+1)
          ▪ Insert(item-N1, levelNo+1)
          ▪ Insert(item-N2, levelNo+1)
      o Else if nucleus item is changed within cell-O then do:
          ▪ Let item-O and item-N be old and new nucleus items.
          ▪ Remove( item-O, levelNo+1 )
          ▪ Insert( item-N, levelNo+1 )
  ➢ Return.
```

The insertion algorithm, **Insert** (*nextItem*, *levelNo*), first performs a novel search algorithm, the *Pre-emptive* cell search, which recursively descends *HCT* from top to the target level in order to locate the most suitable cell for *nextItem*. Once the target cell is located, the item is inserted into the cell and then the cell becomes subject to a generic post-processing check. First the cell is examined for a mitosis operation and as explained earlier if the cell is mature and yields a worse compactness than required (i.e. $CF_C > CThr_L$), then mitosis is applied to produce two new (child) cells on the same level. The parent cell is thus removed from the cell queue of the level and two child cells are inserted instead. Accordingly, the old nucleus item is removed from the upper level and two new nucleus items are inserted into the upper level by consecutively calling **Insert** (*nextItem*, *levelNo+1*) function for both of the (nucleus) items. This is a particular genetic algorithm example where an independent process deterministically calls another process in an iterative way. Note that these processes are independent from each other but the outcome of one may initiate the other. In case mitosis is not performed (for instance the cell is still compact enough after insertion) another post processing step is performed to verify the need for the cell nucleus change. In such a case, first the old nucleus is removed from the upper level and the new one is inserted. Item insertion is a level-based operation and is implemented per item at a time.

**PreemptiveCellSearch** implements the *Pre-emptive* cell search algorithm for finding the target (owner) cell on the level where insertion should occur. The traditional cell search technique, *MS-Nucleus* used in M-Tree and its derivatives, depends on a simple heuristics, which assumes that the closest nucleus (routing) object yields the best sub-tree during descend and finally the best (owner) cell to be appended. Let *d*( ) be the similarity distance function, ***O*** the object to be inserted, $O_N^i$ and $r(O_N^i)$ the nucleus object and its covering radius for the *i*th cell, $C_i$, respectively. Particularly in M-tree, the rationale used is divided into two distinct cases:

**Case 1.** If no nucleus item for which $d(O,O_N^i) \le r(O_N^i)\,\forall C_i$ exists, goal becomes to minimize the increase of the covering radius, i.e. $\Delta_i = d(O,O_N^i) - r(O_N^i)\,\forall C_i$, among all the nucleus objects that are in the owner cell *C*.

**Case 2.** If there exists a nucleus item for which $d(O,O_N^i) \le r(O_N^i)\,\forall C_i$, then its sub-tree is tracked on the lower level. If multiple sub-trees (nucleus objects) with this property exist, then the one to which the object ***O*** is the closest, is chosen.

Both cases fail to track the closest (most similar) objects on the lower level as the sample illustration shows in Figure 2. In this figure, $O_N^1$ and $O_N^2$ are the nucleus (routing) objects representing the lower level cells $C_1$ and $C_2$ on the upper level. In both cases, the *MS-Nucleus* technique tracks down the sub-tree of $O_N^2$, that is, the cell $C_2$ as a result of the cases expressed above. However, on the lower level the closest (most similar) object is item *c* (since $d_1 < d_2$), which is a member of $C_1$.

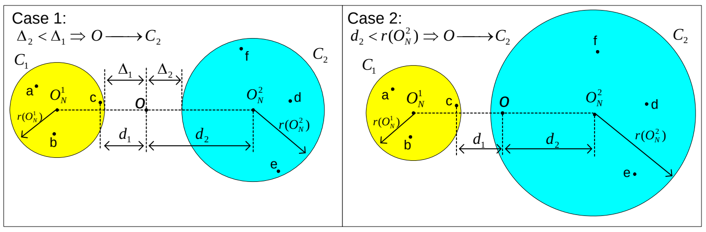
Case 1:
$\Delta_2 < \Delta_1 \Rightarrow O \longrightarrow C_2$
$C_1$
$C_2$
a
b
c
$O_N^1$
$r(O_N^1)$
$\Delta_1$
$\Delta_2$
O
$d_1$
$d_2$
$O_N^2$
$r(O_N^2)$
d
e
f
Case 2:
$d_2 < r(O_N^2) \Rightarrow O \longrightarrow C_2$
$C_1$
$C_2$

**Figure 2: M-Tree rationale used to determine the most suitable nucleus (routing) object for two possible cases. Note that in both cases the rationale fails to track on the closest nucleus object on the lower level.**

*Pre-emptive* cell search algorithm in *HCT* 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. Note that on the upper level we have no information about the items in cells $C_1$ and $C_2$, yet we can set appropriate pre-emptive criteria to fetch all possible nucleus items whose cells should be analyzed to track on the closest item (item *c* in this particular example) in the lower level. Let $d_{\min}$ be the distance to the closest nucleus item (in the upper level). Then pre-emptive cell search rationale can be expressed as follows:

**Case 1.** If no nucleus item for which $d(O,O_N^i) \le r(O_N^i)\,\forall C_i$ exists, then fetch all nucleus items whose cells on the lower level may provide the closest object, i.e. $\Delta_i = d(O,O_N^i) - r(O_N^i) \le d_{\min} \quad \forall C_i$, among all the nucleus objects that are in the owner cell *C*.

**Case 2.** If there exists one or more nucleus item(s) for which $d(O,O_N^i) \le r(O_N^i) \; \forall C_i$, then fetch all of them since their owner cells on the lower level may provide the closest object.

Since Case 1 implies Case 2, Case 1 can be used as the one and only criterion to fetch all nucleus items for tracking. At each level descending towards the target level, using such a pre-emptive analysis that fetches all nucleus items whose owner cells may provide the "most similar" nucleus item for the lower level and so on, *Pre-emptive* cell search terminates its recursion one level above the target level and presents the (final) most similar nucleus item with its owner cell on the target level into which the *nextItem* should be inserted. This achieves an optimum insertion scheme in the sense that the owner cell found on the target level presents the closest nucleus item with respect to the item to be inserted (i.e. *nextItem*). As a natural consequence of this, *Pre-emptive* cell search based item insertion algorithm increases the likelihood of achieving a better cell compactness along with the mitosis operations. Accordingly the *Pre-emptive* cell search algorithm, **PreemptiveCellSearch,** can be expressed as follows:

```
PreemptiveCellSearch (ArrayCS[], nextItem, curLevelNo)
➢ By searching $\forall O_N^i \mid O_N^i \in C_i \wedge \forall C_i \in ArrayCS \rightarrow$ Find
  the most similar item, item-MS and $d_{\min}$.
➢ If(curLevelNo = levelNo + 1) then do:
    o Let the owner cell of item-MS: cell-MS in the (target)
      level (with level number: levelNo)
    o Return cell-MS
➢ Create a new array for cell search: NewArrayCS[] = $\varnothing$
➢ For $\forall O_N^i \mid O_N^i \in C_i \wedge \forall C_i \in ArrayCS$, do:
    o If( $\Delta_i = d(O,O_N^i) - r(O_N^i) \le d_{\min}$ ) then do:
        ▪ Find the owner cell of (nucleus) item $O_N^i$ in the
          lower level: $cell - C_N^i$
        ▪ Append $cell - C_N^i$ into NewArrayCS[]
➢ End loop.
➢ Return PreemptiveCellSearch (NewArrayCS[], nextItem,
  curLevelNo-1)
```

Experimental results show that *Pre-emptive* cell search is effective especially on the upper levels to find out the correct track, which yields the best target cell; however, the computational cost increases significantly especially on the lower levels. In order to find a trade-off, a *hybrid* cell search algorithm can be used especially for very large databases. From the top level till a certain depth (say *PECS_DEPTH*), *Pre-emptive* cell search is applied to guarantee to follow the right track and from this level downwards *MS-Nucleus* is applied. In this way the overall computational cost can be significantly reduced whilst causing only a minimal corruption. Note also that although *hybrid* mode is enabled during the incremental construction of any database, when the database height is below *PECS_DEPTH*+1, only *Pre-emptive* cell search will be used, and afterwards the *hybrid* cell search mechanism is used.

*2) Item Removal Algorithm for HCT:* This is another level-based operation, which does not require any cell search operation. However upon its completion it may cause several post-processing operations, affecting the overall *HCT* body. As explained earlier, if multiple items need to be removed at a particular (target) level, then they are removed one subgroup at a time where items in a subgroup belong to the same cell. Therefore, without loss of generality we will introduce the algorithmic steps assuming that all items to be removed belong to a single cell. Let *ArrayIR[]* be the array for the items (which belong to an owner cell, say *cell-O*) to be removed from (target) level, *levelNo*. The **Remove** algorithm can then be expressed as follows:

```
Remove (ArrayIR[], levelNo)
➢ Let top level number: topLevelNo and the single cell in top level:
  cell-T
➢ Let the Owner (target) cell in level levelNo: cell-O
➢ Remove items in ArrayIR from cell-O
➢ Check cell-O for Post-Processing:
    o If cell-O is depleted (cell-death) then do:
        ▪ If( levelNo = topLevelNo ) then do:
            • Remove cell-O=cell-T
            • Remove the top level from HCT body
        ▪ Else do:
            • Let item-O be the old nucleus item
            • Remove (item-O, levelNo+1)
    o Else if cell-O is split then do:
        ▪ Let item-O, item-N1 and item N2 are old nucleus item
          and two new nucleus items.
        ▪ Remove (item-O, levelNo+1)
        ▪ Insert (item-N1, levelNo+1)
        ▪ Insert (item-N2, levelNo+1)
    o Else if nucleus item is changed within cell-O then do:
        ▪ Let item-O and item-N be old and new nucleus items.
        ▪ Remove (item-O, levelNo+1)
        ▪ Insert (item-N, levelNo+1)
➢ Return.
```

## D. *HCT Indexing*

*HCT* can index a multimedia database using any set of available features, as long as a fusion mechanism and a similarity measure are provided. There are mainly two distinct operations for *HCT* indexing. The incremental construction of the *HCT* body and an optional periodic fitness check operation over it. In the following sub-sections, we will present the algorithmic details of both operations.

*1) HCT Incremental Construction:* Let *G* represent the indexing genre (visual and/or aural) for a multimedia database, *D*. Let *ArrayI<G>* be the item array containing items that are to be appended to *D*. Initially, *D* may or may not have a *HCT* indexing body. If not then all the (valid) items within *D* will be inserted into *ArrayI<G>* and a new *HCT* body is constructed; otherwise, the available *HCT* body is first loaded and updated for the newcomers present in $ArrayI<G>$. Accordingly, the *HCT* indexing body construction algorithm, **HCTIndexing,** can be expressed as follows:

```
HCTIndexing ( $ArrayI<G>$, G, D)
➢ Load and activate HCT indexing body in genre G for database
  D.
➢ For $\forall O_G^i \mid \forall O_G^i \in ArrayI<G>$, do: // For all items
  in the array, perform incremental insertion.
    o Insert ( $O_G^i$ , 0) // Insert ith item into HCT body.
➢ End loop.
```

2) *HCT Fitness Check*: The fitness check is an optional operation that can be performed periodically during or after the indexing operation. It aims to minimize the corruption, which might have occurred due to the only uncontrollable factor during the formation of the *HCT* body that is the order of item insertions. In general multimedia database items are inserted in any order, which might yield an ever-growing corruption if not handled appropriately. Fitness check is implemented with two distinct operations, namely **Outliers Check** and **Cell Merging**, which are presented next.

### I. Outliers Check

The objective of this operation is to reduce the "crowd effect" by removing redundant minority cells (i.e. cells with only one or a few items in it) from the *HCT* body. Due to the insertion order of items, one or some minor group of items may form a cell at the initial stages of the *HCT* construction operation. Later on, some other major cells may become more suitable for hosting those items, which have already been trapped in the minor cells. Note that such minority cells create an over-crowded scheme on their level as well as on the upper levels since each one of them has a representative (nucleus) item hosted by a cell on the upper level. So the idea is to get rid of such cells and feed their items back to the system, expecting that some other mature cells might now host them. Note that after they are inserted to the most suitable cell on the level, the host cell may still refuse them if their insertion results in a significant degradation on the cell compactness and hence causing the cell to split. In such a case, the original part of the host cell and the new item will be assigned to one of two newborn cells. This is the case where they are in fact the outliers that no other (similar) cell exists yet to host them and thus they only have the privilege to stay in a minority cell; whereas the others are successfully hosted by mature cells.

Once completed the primary expectation from this operation is a percentage increase for the mature cells along with their item coverage on a particular level without causing significant degradations in the overall compactness. This operation is performed for all levels in decreasing order (top to bottom) except the top level. The reason for such ordering is because the (incremental) insertion operation on a particular level requires a cell search (*Pre-emptive*) operation performed on all higher levels. So performing an Outliers Check operation first on upper levels is likely to improve the performance of fitness check operations performed on lower levels.

### II. (Mature) Cell Merging

Another consequence of uncontrolled order of item insertion is the erroneous splitting of cells during the early stages of *HCT* body formation. Such cases occur especially when incoming items cannot form a focused cell initially due to the lack of items present (to make the cell compact or dense enough) or a distinct set of items initially inserted and more than one cell was needed to achieve the required compactness level. As an illustrative example shown in Figure 3, such an initial cell splitting decision might have been reasonable and necessary for the current set of items so far present in the cell; however, with the arrival of the newcomers, the two cells can be conveniently merged into a single cell, which still achieve a sufficient compactness level.

Cell merging operation traces the items on the upper level, using the MST branch information of each cell. The closest (minimal) distance eliminates the need for searching the most suitable candidate cells for merging on the lower level. Let *d* be the distance (branch weight) of two nucleus items on the upper level with covering radii, $r_C^1$ and $r_C^2$. If $d \le \left| r_C^1 - r_C^2 \right|$ then merging can directly be granted since one cell can cover the other cell items. In a generic case, a more flexible condition can be applied, such as $d \le k.\left| r_C^1 - r_C^2 \right|$ where $k > 1$. If the merged cell cannot provide a compactness value that its level requires, it will be subject to a *mitosis* operation anyway during the post-processing stage performed after the merging operation. Otherwise the post processing operation removes both the (old) cells and their nucleus items from the *HCT* body and inserts the new (merged) cell and its nucleus item instead.

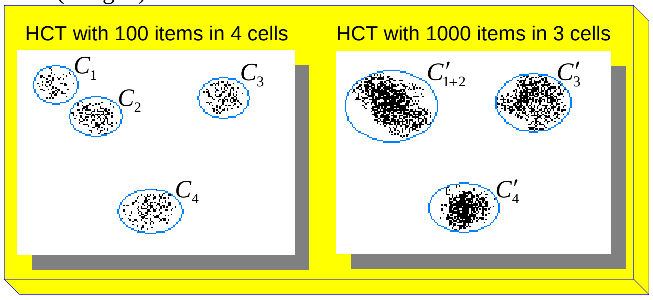

**Figure 3: Merging operation is applied over cells $C_1$ and $C_2$.**

Due to space limitations, the algorithmic details of both Outliers Check and Cell Merging are skipped in this article.

## IV. *PQ* OVER *HCT*

*Progressive Query (PQ)* [15] is a recently proposed retrieval scheme. It basically presents *Progressive Sub-Query* (*PSQ*s) retrieval results periodically to the user and allows the user to interact with the ongoing query process. Among other traditional query techniques such as exhaustive search based *Normal Query* (*NQ*), *kNN* and range queries, *PQ* presents the following significant innovative features:

- It is an efficient technique, which works within both (similarity) indexed and non-indexed (meaning that no similarity indexing method applied) databases onto which it is the unique query method that may provide "faster" retrievals (than *NQ* and requires less memory and CPU power.
- The most important advantage is that it provides user interaction with the ongoing query operation. The user can browse the *PSQ* results so far obtained, can perform "relevance feedback", and can stop the query operation if satisfactory results are obtained so far.
- It can also be applied to (similarity) indexed databases efficiently (to get the most relevant retrieval results in the fastest possible way) and in this case it shows "dynamic *kNN/range* query" like behavior where *k* (or $\varepsilon$) increases gradually with time and hence the user can have the advantage of assigning it by seeing (and judging) the results.

Due to these advantages, we use *PQ* to perform similarity query operations over *HCT*. Before focusing on the details of *PQ* operations over *HCT* we first present a brief overview about *PQ* in the next sub-section.

### A. *PQ Overview*

Basically *PQ* performs over a series of sub-queries, each of which is a fractional query process performed over a sub-set of database items. The items within a sub-set can be chosen by any convenient manner such as randomly or sequentially but the size of each sub-set is determined with respect to a suitable (to human perception) unit such as time (period, $t = t_p$ ).

*PQ* can be performed over any indexed database as long as a *query path* (*QP*) can be formed over the clusters (partitions) of the underlying indexing structure. The most advantageous way to perform *PQ* is to form *QP* according to indexing structure so that the most relevant items can be retrieved in earlier periodic updates of *PQ* as it proceeds over *QP*. More detailed information about *PQ* along with a hypothetical *QP* formation can be found in [15].

## B. *PQ Operation over HCT*

When an indexing structure is available for a database, the most advantageous way to perform *PQ* is to use the indexing information so that the most relevant items can be retrieved in earlier *PSQ* steps. As an example, Figure 4 shows a hypothetical clustering scheme and the formation of the *query path* (*QP*) over which *PQ* will proceed during its run-time. This sample illustration shows 4 clusters (partitions or nodes), which contain a certain number of items (features) and the *QP* is formed according to the relative (similarity) distance to the queried item and its parent cluster. Therefore, *PQ* will give the priority to cluster A (the host), then B (the closest), C, D, etc. Note that the *QP* might differ from the final retrieval result depending on the accuracy of the indexing scheme. For instance, query path gives priority to item B2 on the search with respect to item C4 but item C4 may have more similarity (relevancy) with respect to the queried item A2. When the retrieval results are formed it will eventually be ranked higher and presented earlier to the user by *PQ*. Even though *PQ* corrects this misleading result due to the erroneous indexing (note that in this case item C4 should have belonged to cluster B, not C), as a possible consequence of this, the retrieval of C4 might be delayed to the next periodic PSQ retrieval.

*PQ* operation over *HCT* is executed synchronously over two parallel processes: *HCT* tracer and a generic process for *PSQ* formation using the latest *QP* segment. *HCT* tracer is a recursive algorithm, which traces among the *HCT* levels in order to form a *QP* (segment) for the next *PSQ* update. When the time allocated for this operation is completed, this process is paused and the next *PSQ* retrieval result is formed and presented to the user. Then *HCT* tracer is re-activated for the next *PSQ* update and both processes remain active unless the user stops *PQ* or the entire *PQ* process is completed.

As mentioned earlier, *QP* is formed segment by segment for each *PSQ* update. Once a *QP* segment is formed, then the periodic sub-query results are obtained within this segment (group of items) and then this result (the sorted list of items) is fused with the last *PSQ* update to form the next *PSQ* retrieval result. Starting from the top level, *HCT* tracer algorithm recursively navigates among the levels and their cells according to the similarity of the cell nucleuses. This is similar to the *MS-Nucleus* cell search process, only this time it will not stop its execution when it finds the "most similar" cell on the ground (target) level but continues its sweep by visiting the 2nd most similar, then 3rd and so on, while inserting all visited cell items on the ground level to the current *QP* segment. Starting from the top level, each cell it visits on an intermediate level (any level except the ground level), *HCT* tracer forms a priority (item) queue, which ranks the cell items according to their similarity with respect to the query item. Note that these items are nothing but the nuclei on the lower level. When the tracing operation is completed on the lower level, *HCT* tracer retreats to the upper level (cell) where it came from. The process is terminated when the priority queue of the top level (cell) is depleted, which means, the whole *HCT* body has been traced. Within the implementation of *HCT* tracer, we further develop an internal structure that prevents redundant similarity distance calculation, that is, the similarity distances between the items of the cells in intermediate levels are calculated only once and used in the lower levels whenever needed. In fact this is a general property of overall *PQ* operation, all the (computationally) costly operations such as similarity distance calculations, loading the features from disc to the system memory, etc. are performed only once and shared between the processes whenever needed.

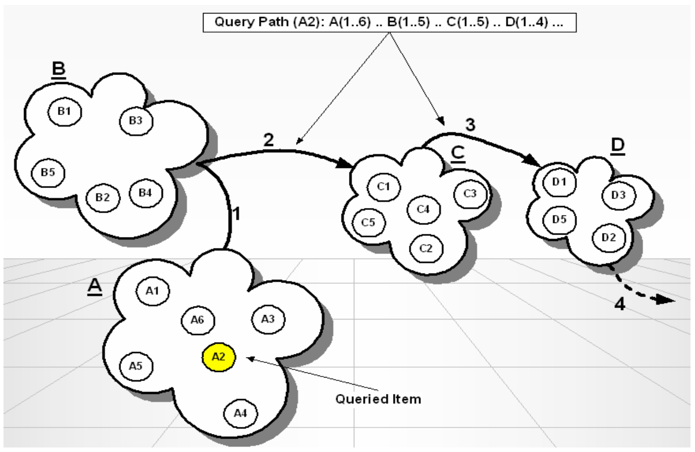
Query Path (A2): A(1..6) .. B(1..5) .. C(1..5) .. D(1..4) ...
B
B1
B3
B5
B2
B4
2
3
C
C1
C3
C4
C5
C2
D
D1
D3
D5
D2
1
A
A1
A6
A3
A2
A5
A4
Queried Item
4

**Figure 4: Query path formation in a hypothetical indexing structure.**

The following **HCTtracer** algorithm implements *HCT* tracer operation, which basically extracts the next *QP* segment into a generic array, *ArrayQP[]*. It is initially called with the top-level number (*topLevelNo*) and an item (*item-MS)* from the single cell on the top level.

Let *item-MS* be the (next) most similar item to the query item, *item-Q*, on the (target) level indicated with a number, *levelNo*. **HCTtracer** algorithm can then be expressed as follows:

```
HCTtracer (ArrayQP[], levelNo, item-MS)
➢ Let cell-MS be the owner cell of item-MS.
➢ If (levelNo = 0) then do: // if this is ground level
    o Append all items in cell-MS into ArrayQP[].
    o Return.
➢ Else do: // if this is an intermediate level
    o Create the priority queue of cell-MS: queue-MS.
    o For $\forall O_N^i \in queue-MS$, do: // for all sorted (nucleus) items do:
        ▪ HCTtracer (ArrayQP[] , levelNo-1, $O_N^i$ )
➢ Return.
```

Note that this algorithm is executed as a separate process (thread) and can be paused externally from the main *PQ* process when the time comes for the next *PSQ* update. An example *HCT* tracer process for an external query item, Q, is illustrated in Figure 5.

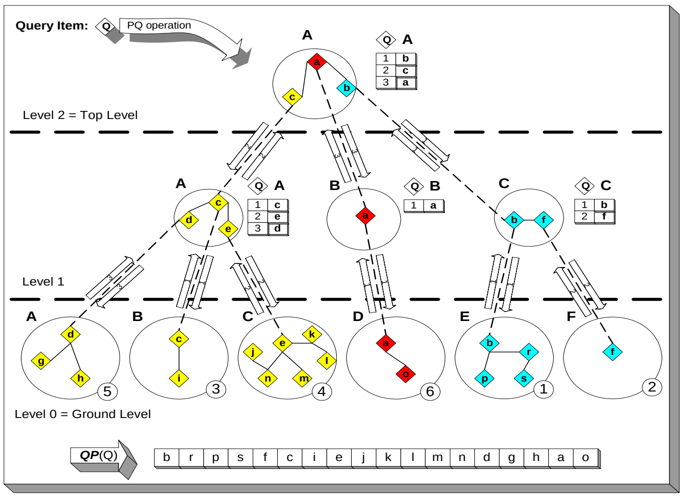

**Figure 5: *QP* formation on a sample *HCT* body.**

## V. *HCT* Browsing

Generally speaking, there are two ways to retrieve items from a (multimedia) database: through a query process such as query by example (QBE) and browsing. In the previous section, an efficient query method (*PQ*) implementation over the proposed indexing scheme, *HCT* was presented. Moreover, *HCT* can provide a basis for accomplishing an efficient browsing scheme, namely *HCT* Browsing [16]. The hierarchic structure of *HCT* is quite appropriate to give an overview to the user about what lies under the current level so that if well supported via user friendly GUI, *HCT* Browsing can turn out to be a guided tour among the database items. The details of *HCT* Browsing and the necessary GUI support within MUVIS framework can be found in [16].

Two examples of *HCT Browsing* with inter-level navigations are shown in Figure 6. In both illustrations, the user starts the browsing from the 3rd level within a 5-level *HCT* body and, due to the space limitations only some portion of *HCT* body (where the browsing operation is performed) is shown. Note that in both examples, *HCT* indexing scheme provides more and more "narrowed" content in the descending order of the levels. For example, the user chooses an "outdoor, city, architecture" content on the third level where it yields "outdoor, city, beach and buses" content carrying cell on the second level. The user then chooses a multi-color "bus" and then navigating down to the first level, it yields a cell, which owns mostly "buses" with different colors, and finally choosing a "red bus" image (nucleus item) yields the cell of "red buses" on the ground level. Similar series of examples can also be seen in the sample *HCT Browsing* operation within a texture database. The cells are getting more and more compact (focused) in the descending order of level and the ground level cells achieve a "clean" clustering of texture images showing high similarity.

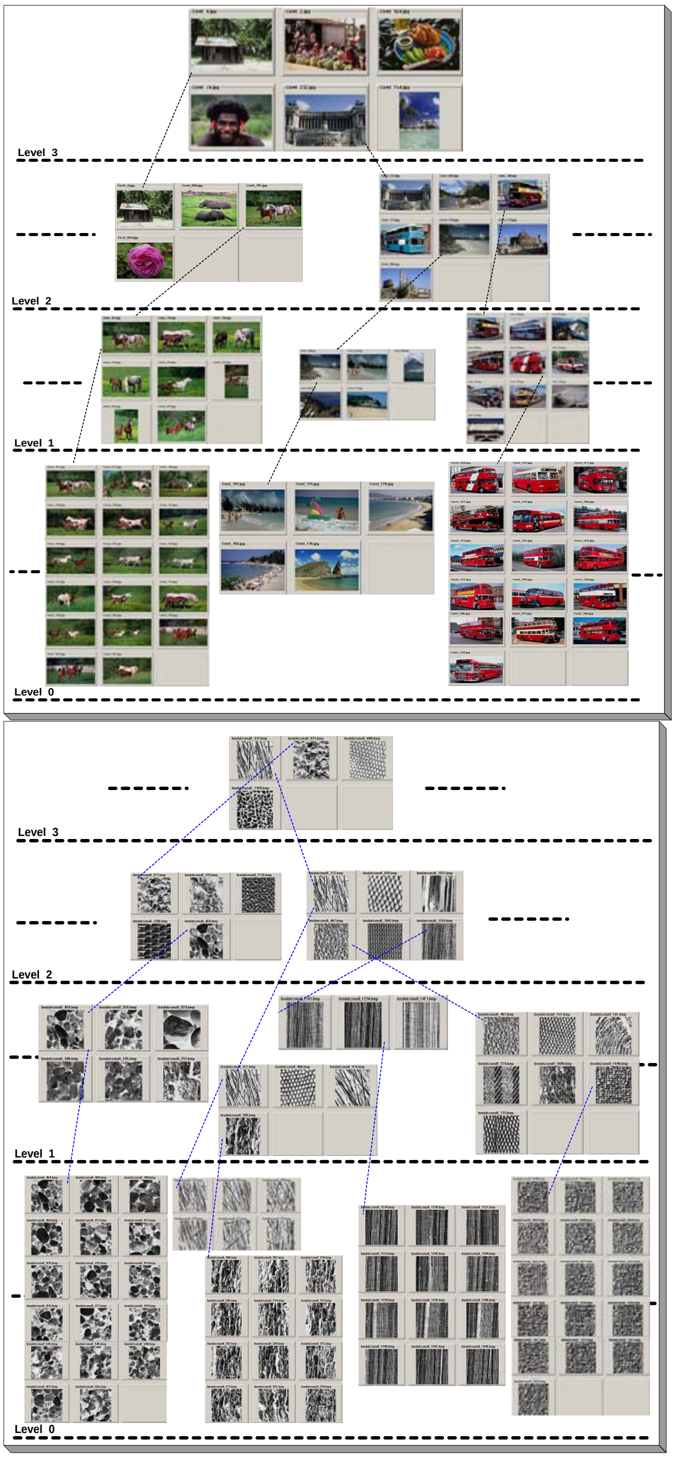

**Figure 6: Two *HCT* Browsing examples both of which start from the third level within Corel_1K (top) and Texture (bottom) databases. The user navigates among the levels shown with the lines through ground level.**

## VI. Experimental Results

This section is divided into three sub-sections, each includes several experiments performed to test the clustering, indexing and retrieval (via *PQ*) capabilities of *HCT* and perform comparative evaluation with M-Tree. It is, however, not straightforward to do a direct performance comparison between *HCT* and M-Tree due to the strict parameter dependency and various internal modes of M-Tree. For instance an M-Tree body (index structure) with *M*=10 will be completely different than the one with *M*=11. Similarly using one of different *split* policies (*Balanced*, *Generalized Hyperplane*, etc.) or *promote* methods (*m_RAD, mM_RAD, M_LB_DIST, RANDOM*, etc.) [10] will result in a completely different indexing body than using another. Therefore, we do the partial comparisons between major M-Tree and *HCT* properties such as fixed (with a certain *M*) versus flexible cell size (*HCT*) policy and *MS-Nucleus* versus *Pre-Emptive* cell search algorithms. Section A presents the clustering performance of *HCT* on synthetic databases, which contain a certain number of natural clusters varying in size, form, density and shape. Computational complexity and clustering accuracy of *HCT* will be presented and especially the “cost vs. accuracy” analysis for the periodic fitness check will be performed. The rest of the sections are devoted to indexing (and retrieval via *PQ*) performance of *HCT* in real multimedia databases. In order to present the experimental conditions, Section B briefly introduces MUVIS and particularly *MBrowser* application under which *HCT Browsing* and *PQ* over *HCT* retrieval schemes are primarily developed and tested. Afterwards, we begin the comparative evaluation of *HCT* versus M-Tree indexing policies, particularly focusing on the amount of cell corruption with the increasing database size. Finally, Section C is devoted to experimental results obtained from *PQ* over *HCT* operations and their evaluation with respect to Sequential *PQ* and *NQ*.

## A. *HCT Clustering Performance in Synthetic Databases*

*HCT* in the most basic terms can function as a clustering algorithm, which groups items with respect to their proximity in multidimensional (feature) space. In order to test its clustering performance, we create several synthetic databases, which provide straightforward (clean) clusters for the human eye in 2-D for illustration purposes. Four databases are depicted in Figure 7 (left) with various numbers of items, which are represented by white pixels distributed in a 2-D space according to some formations (clusters). The performance evaluation includes both computational complexity measurements and clustering accuracy with and without the use of the optional (periodic) Fitness Check operation in order to examine its effect on the overall performance.

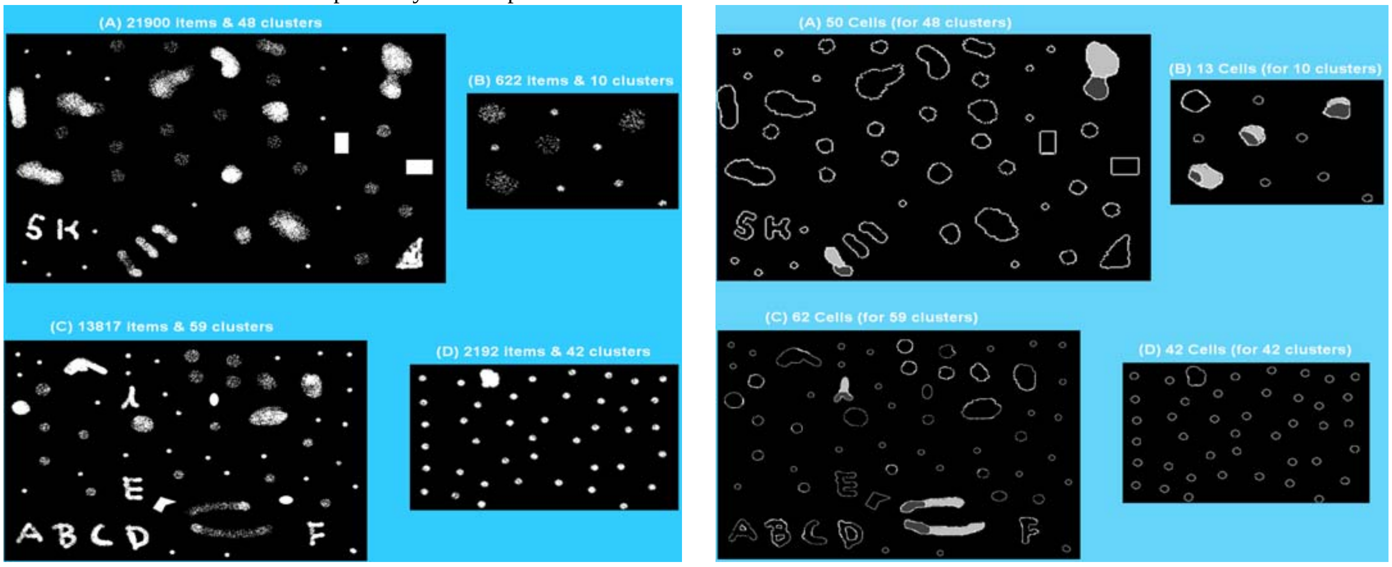

**Figure 7: 4 synthetic databases with different scales and dimensions (left) and the cluster boundaries obtained via *HCT* (right)**

1) *Clustering Accuracy: HCT* naturally forms the clusters on the ground level (level 0) where each cell corresponds to a unique cluster. In order to test the clustering accuracy of HCT, the optional HCT operation, periodic Fitness Check is enabled to see whether or not HCT can converge to the (natural) clusters present; otherwise, early mitosis operations may cause irreversible clustering errors. Another important factor in the evaluation is to examine HCT performance against potential variations in database size and cluster properties. The examples shown in Figure 7 are selected particularly to provide significant variations in the shape and size of the clusters, cluster density and inter-cluster distances. Moreover, in order to simulate dynamic construction of such a database, each synthetic example is formed by different numbers of items and clusters into which the items (white dots) are inserted one by one (i.e. incremental HCT formation) in a random order to examine its robustness against such random arrivals. The same HCT instance is used (with the same HCT parameters) to perform clustering all of the examples and the results are shown in Figure 7 (right). Each contour shown on the right of Figure 7.represents a cell formed at the end of HCT formation process, and if more than one cell is formed for a cluster, then those cells are indicated with dark and light shading. Due to random insertions, we also observed that the clustering scheme can be slightly different, i.e. say one or a few changes may occur, so each example is clustered 10 times and a typical (the most frequent) clustering scheme is shown. It was also noticed that the number of cells is always equal to or larger than the number of clusters, i.e. a slight over-segmentation may happen, but no under-segmentation. In order to show the loose parameter dependency of HCT, for each experiment we used random values of $N_M$, $k_0$ and $N_M^T$ within the following values:

$6 \le N_M \le 12,\ 0.25 \le k_0, \le 0.7, 12 \le N_M^T \le 30$. The following regularization function is used for clustering.

$$CF_C = f(\mu_C, \sigma_C, r_C, \max(w_C), N_C) = K(\mu_C + \sigma_C)\, r_C \max(w_C)\sqrt{N_C} \quad (4)$$

where *K*, is a scaling coefficient; $\mu_C$ and $\sigma_C$ are the mean and standard deviation of the MST branch weights, $w_C$, of cell *C*; $r_C$ is the covering radius; and $N_C > N_M$ is the number of items in cell *C*. With an increasing number of items in the cell ($N_C$) and in order to keep the cell compact (i.e. $CF_C \le CThr_L$), MST branch statistics such as $\mu_C$, $\sigma_C$, max($w_C$) and $r_C$ should all remain small in order to yield a more focused cell. Otherwise, the cell undergoes a mitosis operation, which eventually reduces $N_C$ and max($w_C$) and separates the irrelevant item or group of items from the cell.

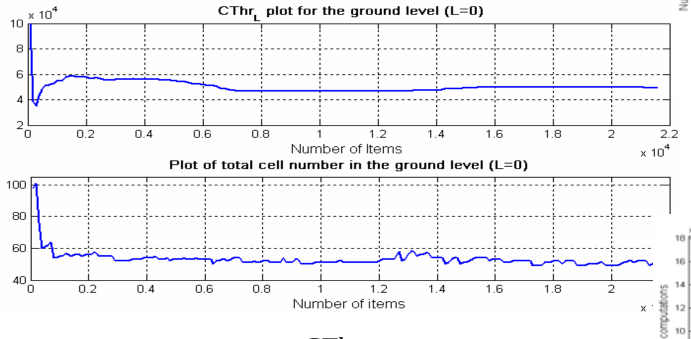

**Figure 8: Ground level $CThr_L$ (top) and cell number (bottom) plots for the example (A) in Figure 7.**

During the clustering experiments performed, *HCT* with periodic Fitness Check operation achieves a high clustering accuracy and also robustness against the random arrivals of the items. Moreover, when a cluster is split into multiple cells, in most cases (>95%) this happens to the same clusters, and these are the ones with loose item density and/or big and long shapes. This is an expected result since the regularization function for compactness feature penalizes such cases with parameters such as $\mu_C$, $\sigma_C$ and $r_C$. This can be easily seen in examples A and C (the longest clusters) and B (the biggest/loose clusters) in Figure 7. Furthermore, despite the significant variations in inter-cluster distances, number of items per cluster, shape/density of each cluster and the total number of clusters/items in each example, *HCT* accurately extracts the true clusters. In this aspect, one can conclude that the *Median* operator (with a trend factor, i.e. $0.2 < k_0 < 0.8$) for the estimation of $CThr_L$ value for a particular level *L* works effectively to allow the cells to grow all the way to the "true" boundaries of each cluster but surely avoiding to merge multiple clusters (separated with a certain inter-cluster distance) into one cell. In the experiments performed, $CThr_L$ shows a smooth convergence towards a steady value (after some initial transient) since the cells become more compact (denser) due to ever-increasing amount of items in the cells. As a typical example, the plots shown in Figure 8 illustrate the dynamic $CThr_L$ setting (top) and the number of cells (bottom) converging to the close vicinity of true number of the clusters (with incoming items) during the *HCT* formation for the clustering example (A) in Figure 7.

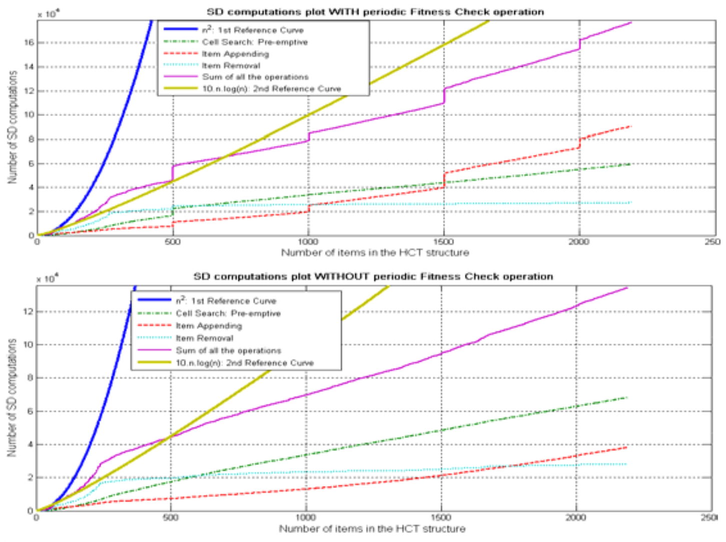

**Figure 9: Plots showing SD Computations with (top) and without (bottom) Fitness Check during *HCT* formation of the example (D) in Figure 7.**

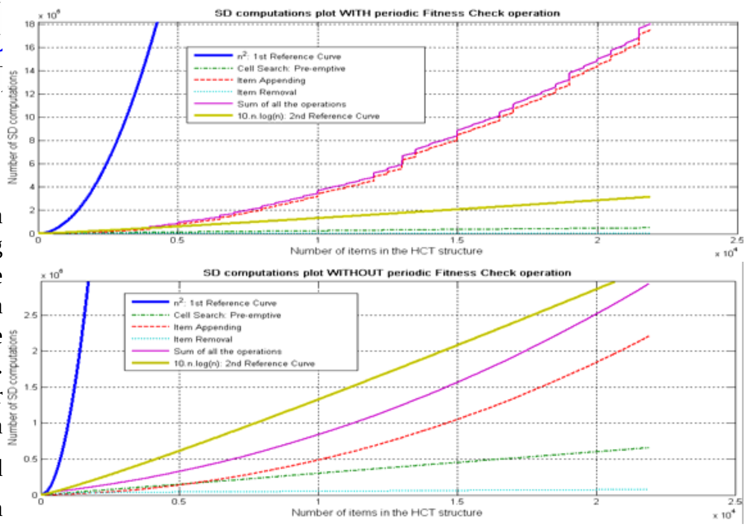

**Figure 10: Plots showing SD Computations (top) with and (bottom) without Fitness Check during *HCT* formation for example (A) in Figure 7.**

*2) Computational Complexity Analysis:* The computational complexity analysis is based on the amount of (dis-) similarity (proximity) distance computations performed during the incremental formation of the *HCT* body. These computations are performed for three individual *HCT* operations:

- (*Pre-emptive*) cell search: the computations performed to find out a host cell on a certain level.
- Item insertion into a host cell: the computations performed to insert a new item into a cell MST.

- Item(s) removal from a cell: the computations performed to rebuild the cell MST after item (or items) removal.

So the total number of computations is the sum of the ones from the individual operations as listed above and it can be measured with and without performing periodic Fitness Check operation to see its effect on the computational complexity. Two plots representing the *HCT* formation of the clustering examples (A) and (D) in Figure 7 are shown in Figure 10 and Figure 9 respectively.

The Fitness Check operation usually increases the computations for the item insertions and reduces the ones for cell search since its basic outcome is the increase in the cell populations and thus reduction in the *HCT* height (total number of levels). However, the total number of computations is increased due to the fact that the (*Pre-emptive*) cell search requires $O(n\log(n))$; whereas, dynamic MST formation (the operation for item insertion) requires $O(n^2)$ operations. Especially for the highly populated examples, where one or more clusters host a massive number of items, the item insertion operations will eventually dominate the other two and therefore, becomes the major part of overall computations. A typical example is given in the example (A) in Figure 7 (6 biggest clusters have more than 2000 items each) and its performance plot shown in Figure 10 (left). In such a case the order of computations will be between $O(n^2)$ and $O(n\log(n))$, see Figure 10 (left). On the other hand, when the cluster sizes are limited within a reasonable upper bound, *HCT* formation can still be a $O(n\log(n))$ operation even with the presence of periodic Fitness Check operation, as one typical example is given in the example (D) in Figure 7 and its plot is shown in Figure 9 (left)

3) *M-Tree vs. HCT:* Two major properties of *HCT*, flexible cell size and Pre-emptive cell search are evaluated against M-Tree policies (i.e. fixed cell size: M, and MS-Nucleus cell search) in terms of clustering accuracy and computational cost. In fact it is obvious to see that no matter how M is chosen, M-Tree is bound to fail to extract the natural clusters showing significant variations in size, shape and density. Setting M too big will cause erroneous merging of small clusters to their close neighbors (under clustering) or setting M too small will fraction several (big) clusters into a large number of cells (over clustering) and hence the overall indexing body will be too crowded and not so useful for any indexing or clustering purposes. Table I presents the number of cells and SD computations for the examples in Figure 7 for three different *HCT* construction scheme. The first and second rows present regular *HCT* constructions with and without applying periodic Fitness Check (FC), the third row presents clustering with M-tree policies (with M=12 and using MS-Nucleus cell search).

**Table I: Number of cells and SD computations for the synthetic examples shown in Figure 7.**

| | Number of Cells (Clusters) | | | | Number of SD Computations (x1000) | | | |
|---|---|---|---|---|---|---|---|---|
| | **(A: 48)** | **(B: 10)** | **(C: 59)** | **(D: 42)** | **(A)** | **(B)** | **(C)** | **(D)** |
| ***HCT* (with *FC*)** | 50 | 13 | 62 | 42 | 17945.4 | 55.6 | 8993.96 | 192.27 |
| ***HCT* (no *FC*)** | 153 | 55 | 179 | 146 | 3189.5 | 23.52 | 1437.03 | 136.03 |
| **M-Tree (*M*=12)** | 3744 | 108 | 2330 | 377 | 2038.13 | 25.94 | 1104.74 | 123.38 |

As it can be clearly seen from this table, M-Tree policies cannot cope with any clustering scheme and usually result in an extremely over-crowded clustering whereas even without the presence of periodic *Fitness Check*, *HCT* policies, especially the flexible cell size property can mostly avoid such a degraded scheme and achieves a reasonable clustering performance with slight increase in the computational cost. Of course, the best clustering performance is obtained with the use of the periodic *Fitness Check*; however, the computational cost is drastically increased especially when there are cells carrying massive number of items (e.g. A and C in Figure 7) due to the aforementioned reason.

## B. *HCT Multimedia Indexing within MUVIS*

MUVIS framework is developed to bring a unified and global approach to indexing, browsing and querying of various multimedia types such as audio/video clips and still images. One of its major applications is *DbsEditor*, which performs offline feature extraction and indexing operations along with some basic database management tasks such as creation and editing. *MBrowser* is the primary media browser and retrieval application into which *PQ* technique is integrated as the primary retrieval (QBE) scheme. A sequential scan based *Normal Query* (*NQ)* is the alternative scheme within *MBrowser*. Both *PQ* modes (sequential and over *HCT*) and *NQ* can be used for retrieval of multimedia primitives with respect to their similarity to a queried media item (an audio/video clip, a video frame or an image). Similarity distances will be calculated by the particular functions, each of which is implemented in the corresponding visual/aural feature extraction (*FeX* or *AFeX*) modules. More detailed information about MUVIS can be found in [18], [19] and [24].

In the experiments performed in this section, we used 6 sample (multimedia) databases:

1) **Open Video** Database: This database contains 1130 video clips, each of which is downloaded from “The Open Video Project” web site [26]. The clips are quite old (from 1960s) but contain color video with sound. The total duration of the database is around 20 hours.

2) **Corel_1K** Image Database: There are 1000 medium resolution (384x256 pixels) images from diverse contents such as wild life, city, buses, horses, mountains, beach, food, African natives, etc.

3) **Corel_10K** Image Database: There are 10000 low resolution images (in thumbnail size) from similar contents with *Corel_1K.*

4) **Corel_60K** Image Database: The entire Corel database with 60000 medium resolution images.

5) **Shape** Image Database: There are 1500 black and white (binary) images that mainly represent the shapes of different objects such as animals, cars, accessories, geometric objects, etc.

6) **Texture** Image Database: There are 1760 texture images representing the pure textures from several materials and products.

**Table II: The databases and their features.**

| Feature Type | Texture | Color | Shape | Audio |
|---|---|---|---|---|
| **Features Used** | Gabor [23] GLCM [27] | HSV Hist. YUV Hist. | Edge Direction Hist. | MFCC [30] |
| **Database Number** | 1,2,3,4 and 6 | 1,2,3 and 4 | 1,2,3,4 and 5 | 1 |

Table II presents what features are used in the sample databases. All experiments are carried out on a P5 1.8 GHz computer with 1024 MB memory. In order to have unbiased experimental evaluations, each query experiment are performed using the same queried multimedia item with the same instance of *MBrowser* application. The evaluations of the retrieval results by *PQ* are performed subjectively using ground-truth method, i.e. a group of people evaluates the query results of a certain set of retrieval experiments, upon which all the group members totally agreed about the query retrieval performance. Among these a certain set of examples were chosen and presented in this article for visual inspection and verification.

1) *HCT vs. M-Tree Indexing*: In this section we will particularly make the comparative performance evaluations based on the cell search algorithms and cell size policies of *HCT* and M-Tree. The sample MUVIS databases are indexed using both (*HCT* and M-Tree) policies. We used typical settings ($N_M = 6$, $N_M^T = 24$) for all the experiments with the same regularization function given in Eq. (4). For the numerical comparison, the ground level statistics are used to measure the average cell compactness and the total amount of computations performed during the entire indexing process. The cell compactness is a measure of how focused the cell items are and it is therefore proportional with the number of items, $N_C$, in a cell $C$ and inversely proportional with the covering radius, $r_C$. In this way it can be defined for any cell (mature or not) containing multiple items (i.e. $N_C > 1$). So the following expression, which is nothing but the ratio of the average cell size to average covering radius can be used to calculate the average cell compactness for a level, *l*, in *HCT*.

$$\mu_{CC}^{l} = \left. \frac{\sum_{C \in L(l)} N_C}{\sum_{C \in L(l)} r_C} \right|_{N_C > 1} \qquad (5)$$

where *L(l)* is the set of cells on level *l*. presents the following statistics obtained from the sample databases by using both M-Tree (with *M*=12) and *HCT* policies: the average cell compactness for ground level ($\mu_{CC}^{0}$), the total number of cells and the percentage of mature cells along with the number of SD computations.

**Table III: Statistics obtained from the ground level of HCT indexing of the sample MUVIS databases.**

| Statistics (*l* = 0) | Construction Policy | Open Video (Visual) | Open Video (Aural) | Corel_1K | Corel_10K | Corel_60K | Shape | Texture |
|---|---|---|---|---|---|---|---|---|
| $\mu_{CC}^{0}$ | M-Tree | 5.971 | 3.469 | 5.007 | 5.084 | 8.094 | 31.231 | 57.861 |
| | *HCT* | *7.418* | *5.136* | *6.307* | *17.356* | *26.142* | *39.688* | *72.599* |
| | *HCT* (with *FC*) | *8.838* | *6.825* | *8.065* | *21.522* | *34.004* | *43.117* | *85.62* |
| **Cell Number** | M-Tree | 227 | 240 | 248 | 2116 | 7864 | 260 | 306 |
| | *HCT* | *203* | *164* | *223* | *584* | *2304* | *227* | *288* |
| | *HCT* (with *FC*) | *156* | *85* | *149* | *367* | *1386* | *168* | *220* |
| **SD Comp. Number (x1000)** | M-Tree | 70.343 | 67.556 | 72.57 | 1306.23 | 12604.76 | 82.55 | 95.629 |
| | *HCT* | *244.814* | *162.271* | *191.14* | *4439.03* | *73782.01* | *251.21* | *240.802* |
| | *HCT* (with *FC*) | *301.962* | *159.511* | *251.945* | *4649.23* | *103427.14* | *288.74* | *277.382* |

The numerical results given in Table III approve that two key *HCT* policies, namely *Pre-emptive* cell search algorithm and flexible cell size property, achieve a major compactness improvement with respect to what M-Tree can establish. One of the main reasons is that M-Tree policies usually produce excessive (more than necessary) number of cells, as we named as "the crowd effect" or in other words an over-crowded scheme, which is mainly due to fixed cell size property and this fact can be clearly seen by the cell number data in Table III. Therefore, the group of media items having the same content is fractioned into numerous cells, which in turn makes the indexing body over-crowded. Such a crowded indexing body further makes the *MS-Nucleus* cell search algorithm less accurate, inducing more and more corruption proportional with the database size due to the reasoning explained earlier. Once the corruption evolves into a certain level, it further causes more corruption in a positive feedback mechanism since any statistical measures from the over-crowded and corrupted cells will be less reliable. So the speed and accuracy of cell search will further be degraded. As a result all the M-tree levels, particularly the ground level where all the database items are present, will contain smaller and corrupted (loose) cells (e.g. see Figure 11). This can be verified by comparing the respective values of $\mu_{CC}^{0}$ and cell number data obtained from both approaches on **Corel_1K, Corel_10K** and **Corel_60K** databases.

Apart from the database size, the reliability (discrimination power) of the feature(s) is also an important factor. With the improved discrimination factors of the features, more robust similarity distance measures can be obtained and hence even more focused cells can be formed using *Pre-emptive* cell search algorithm. Using ground-truth methodology over several QBE oriented retrieval experiments, the most reliable features are proven to be the texture features (GLCM and Gabor) extracted particularly for **Texture** database. Hence, a relatively high difference in terms of average cell compactness can be seen in **Texture** database which has a rather small size (e.g. only 1760 images). As some visual examples, Figure 11 shows four ground level cells obtained from both indexing policies over this database. It is obvious from the figure that the cells from the proposed *HCT* policies show a high compactness (textural similarity) level; whereas, M-tree cells (with *M*=12) show signs of corruption (dis-similarity) among its items. It can be thought that with a smaller *M* (i.e. *M*=6), such irrelevant images can be (forcefully via split mechanism) removed from the host cell so as to yield a focused cell. However, this will cause a massive "crowd effect" for the cells at any level, henceforth causing more corruption (due to its sub-optimum cell search, *MS-Nucleus*) since we know that there are several groups in this database having a large number of images (i.e. >60) with the same texture category. In short no matter what value is set for *M*, as long as the cell size is kept fixed and *MS-Nucleus* cell search is used, M-Tree is bound to induce an indefinite level of corruption into any multimedia database.

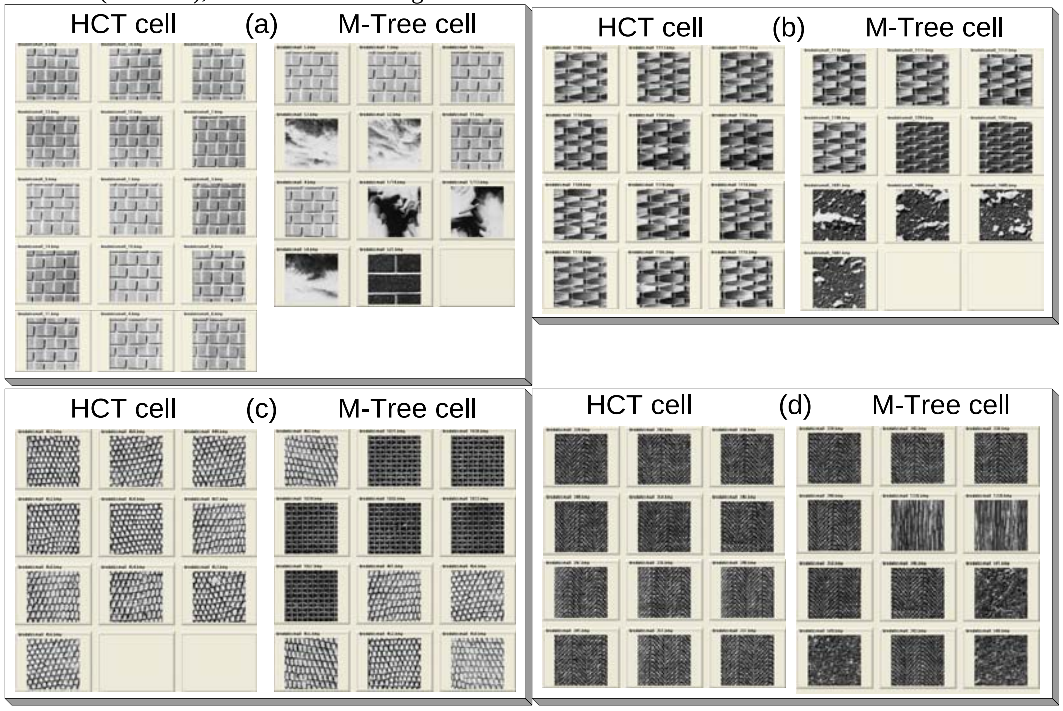

**Figure 11: 4 ground level cells (a, b, c and d) in Texture database indexed by *HCT* (left) and M-Tree (right) policies.**

The primary cost for using *HCT* policies is the increased computational complexity for the construction of the indexing structure. However, since indexing is an off-line process that is performed only once during the creation of the database, this cost can be compensated by the accuracy and time gains during query and browsing, both of which are real-time processes that are subject to be performed several times during the lifetime of any multimedia database. Moreover M-tree indexing over a large multimedia database might cause such a corruption level that makes the indexing nearly useless for any content-based querying and browsing purposes.

## C. *PQ over HCT*

Two tests are performed to evaluate the performance of *PQ* operations over *HCT* indexing structure. First the relevancy of the *Query Path (QP)* where *PQ* will proceed can be examined from a typical *QP* (similarity distance) plot. Such a plot can indicate whether or not the order of the items within *QP* is formed in accordance with the similarity of the query item so that the most similar items can be retrieved earliest. In Figure 12 the query image comes from a group of 97 similar images among 1000 images in **Corel_1K** database. It can be seen from the figure that *HCT* tracer successfully captures all relevant items in the earliest possible order, i.e. the beginning of *QP*. Therefore, *PQ* operation will be ranking and presenting them (first) to the user immediately after the query operation is initiated. Another important remark should be made about the "up-hill trend" of the *QP* plot, that is, it traces along with the increasing order of SD (dissimilarity) as intended.

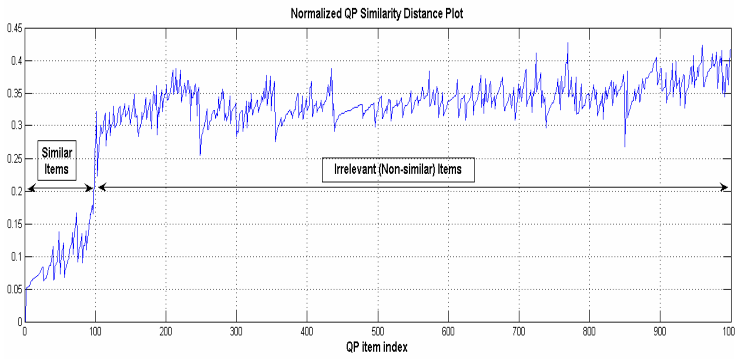

**Figure 12: *QP* plot of a sample image query in *Corel_1K* database.**

The second performance evaluation is about the speed (or timing) of *PQ* over *HCT* operation compared with the Sequential *PQ* and *NQ*. In an earlier work [17] where the initial version of *HCT* was first proposed, a promising gain in speed was observed for small multimedia databases. In the current *HCT*, particularly designed for large databases, we perform several retrieval experiments in the form of QBE on large databases, such as Corel_60K database, where 100 *PQ*s and *NQ*s are performed with

100 query images bearing a pure content. We used $t_p = 1\sec$ and thus measured the query time to retrieve relevant images (a maximum of one miss was allowed) among the first (highest ranked) 12 results. The query histograms are drawn according to the measurements and shown in Figure 13.

As expected *PQ* over *HCT* achieves the earliest retrieval times where almost half of the retrievals are achieved within one second and only in 7 (out of 100) *PQ* over *HCT* experiments resulted in retrieval times exceeding 4 seconds. As a traditional query mechanism, *NQ* in general provides the slowest retrieval speed, almost all in 18 seconds, only after the full-scan search is completed over the entire database. Sequential *PQ*, on the other hand, provides a significantly varying scheme since it is designed for the databases with no similarity indexing structure (hence *HCT* is not used at all) and the majority of the query results provides the required amount of relevant items after 11 or more seconds.

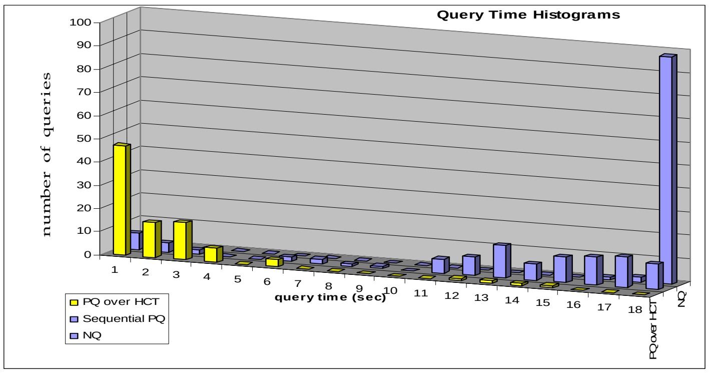

**Figure 13: The query time histograms obtained from 100 *PQ*s (using both modes) and *NQ*s in *Corel_60K* database.**

## VII. Conclusions

In this paper, we proposed *HCT* indexing structure designed for multimedia databases to achieve the following innovative properties:

- *HCT* is a dynamic, parameter independent and flexible cell (node) sized indexing technique, which is optimized to achieve as focused cells as possible.
- *HCT* is particularly designed for indexing multimedia databases, which are created incrementally and subject to random item insertions and removals. Furthermore, their visual and aural features are noisy descriptors and usually have a limited discriminative power.
- By means of the flexible cell size property, one or the least number of cell(s) are created to host the group of similar items, which in effect reduces the performance degradations caused by "crowd effect" that is a natural deficiency for M-Tree due to its fixed cell size policy.
- During their life-time cells are put under a close surveillance of their levels in order to enhance the compactness using mitosis operations whenever necessary to get rid of dissimilar item(s). Furthermore, for an item insertion, a *pre-emptive* cell search technique is used to find out the most suitable (target) cell on a host level. In this way another major source of corruption due to sub-optimum M-Tree cell search technique (*MS-Nucleus*) is also avoided.
- *HCT* has a dynamic reaction capability in such a way that the cell and level primitives are updated whenever the need arose. For example a cell nucleus item is changed whenever a better candidate is available and once a new nucleus item is assigned, its owner cell in the upper level is determined after a new cell search instead of using the old one's owner cell. Such instantaneous reactions keep the *HCT* body intact by doing the required updates after any *HCT* operation.
- By means of a dynamic MST formation within each cell, the optimum nucleus item can be assigned whenever necessary and with no extra cost. Furthermore the optimum split management can be done when the mitosis operation is performed (again with no cost). Most important of all, MST provides a reliable compactness measure via "cell similarity" for any item instead relying on only to a single (nucleus) item. By this way a better judgment can be done whether or not a particular item is suitable for a mature cell.
- *HCT* is mainly designed to work with *PQ* in order to provide the earliest possible retrievals of the most relevant items. Furthermore, *HCT* indexing body can be used for efficient browsing and navigation among database items. The user is guided at each level by nucleus items and several hierarchic levels help the user to have a "mental picture" about the entire database.

Experimental results show that *HCT* achieves all the abovementioned properties and capabilities in an automatic way with no or loose parameter dependency. It further achieves significant improvements in cell compactness and shows no sign of corruption when the database size is getting larger. The experiments performed over several multimedia databases suggest that *HCT* usually yields a better clustering performance when the discrimination power of the features is significant and henceforth the cells can provide better item relevancy for the semantic point of view.

Current and planned future studies include: the design of alternative models for enhanced level compactness threshold setting and a better cell compactness regularization function or a template based model and the implementation of a generic "relevance feedback" option during an *HCT Browsing* operation so that the user can manually edit and update any cell structure.

## References

[1] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: An efficient and robust access method for points and rectangles", *In Proc. of ACM SIGMOD Int. Conf. on Management of Data*, Atlantic City, US, pp. 322-331. 1990.

[2] J. L. Bentley, "Multidimensional binary search trees used for associative searching", *In Proc. of Communications of the ACM*, v.18 n.9, pp.509-517, September 1975.

[3] S. Berchtold, C. Bohm, H. V. Jagadish, H.-P. Kriegel, J. Sander, 'Independent Quantization: An Index Compression Technique for High-Dimensional Data Spaces', *In Proc. of the 16th Int. Conf. on Data Engineering*, San Diego, USA, pp.577-588, Feb. 2000.

[4] S. Berchtold , C. Böhm , H.-P. Kriegal, "The pyramid-technique: towards breaking the curse of dimensionality", *In Proc. of the 1998 ACM SIGMOD International conference on Management of data*, pp.142-153, Seattle, Washington, US, June 01-04, 1998.

[5] S. Berchtold, D. A. Keim, and H.-P.Kriegel, "The X-tree: An index structure for high-dimensional data*", In Proc. of the 22th International Conference on Very Large Databases (VLDB) Conference*, 1996.

[6] T. Bozkaya, Z. M. Ozsoyoglu, "Distance-Based Indexing for High-Dimensional Metric Spaces", *In Proc. of ACM-SIGMOD*, pp.357-368, 1997.

[7] S. Brin, “Near Neighbor Search in Metric Spaces”, *In Proc. of International Conference on Very Large Databases (VLDB)*, pp. 574-584, 1995.

[8] K. Chakrabarti and S. Mehrotra. “The hybrid tree: An index structure for high dimensional feature spaces”, *In Proc. of Int. Conf. on Data Engineering*, pp. 440-447, February 1999.

[9] S.F. Chang, W. Chen, J. Meng, H. Sundaram and D. Zhong, “VideoQ: An Automated Content Based Video Search System Using Visual Cues”, *In Proc. of ACM Multimedia*, Seattle, US, 1997.

[10] P. Ciaccia, M. Patella, and P. Zezula, “M-tree: an efficient access method for similarity search in metric spaces”, *In Proc. of International Conference on Very Large Databases (VLDB)*, pp. 426-435, Athens, Greece, August 1997.

[11] M. J. Fonseca, J. A. Jorge, “Indexing High-Dimensional Data for Content-Based Retrieval in Large Databases”, *In Proc. of Eighth International Conference on Database Systems for Advanced Applications (DASFAA '03)*, pp. 267-274, Kyoto-Japan, March 26 – 28, 2003.

[12] A. Guttman, “R-trees: a dynamic index structure for spatial searching”, *In Proc. of ACM SIGMOD*, pp. 47-57, 1984.

[13] D. B. Johnson, P. T. Metaxas, “Optimal Algorithms for the Single and Multiple Vertex Updating Problems of a Minimum Spanning Tree”, *Algorithmica* 16(6): pp. 633-648, 1996.

[14] N. Katayama , S. Satoh, “The SR-tree: an index structure for high-dimensional nearest neighbor queries”, *In Proc. of the 1997 ACM SIGMOD international conference on Management of data*, pp.369-380, Tucson, Arizona, US, May 11-15, 1997.

[15] S. Kiranyaz, M. Gabbouj, “A Novel Multimedia Retrieval Technique: Progressive Query (Why Wait?)”, *IEE Proceedings Vision, Image and Signal Processing*, vol. 152, pp. 356-366, May 2005.

[16] S. Kiranyaz, M. Gabbouj, “Hierarchical Cellular Tree: An Efficient Indexing Method for Browsing and Navigation in Multimedia Databases”, *In Proc. of European Signal Processing Conference*, *Eusipco 2005*, Paper ID: 1063, Antalya, Turkey, September, 2005.

[17] S. Kiranyaz, M. Gabbouj, “A Dynamic Content-based Indexing Method for Multimedia Databases: Hierarchical Cellular Tree”, *In Proc. of IEEE Int. Conference on Image Processing, ICIP 2005*, Paper ID: 2896, Genova, Italy, September, 2005.

[18] S. Kiranyaz, K. Caglar, O. Guldogan, and E. Karaoglu, “MUVIS: A Multimedia Browsing, Indexing and Retrieval Framework”, *Proc. of Third International Workshop on Content Based Multimedia Indexing, CBMI 2003*, Rennes, France, 22-24 September 2003.

[19] S. Kiranyaz, K. Caglar, E. Guldogan, O. Guldogan, and M. Gabbouj, “MUVIS: a content-based multimedia indexing and retrieval framework”, *Proc. of the Seventh Int. Symposium on Signal Proc. and its Applications, ISSPA 2003*, Paris, France, pp. 1-8, 1-4 July 2003.

[20] P. Koikkalainen and E. Oja. ”Self-organizing hierarchical feature maps”, *In Proc. of the International Joint Conference on Neural Networks*, San Diego, CA, 1990.

[21] J. R. Kruskal, “On the shortest spanning subtree of a graph and the traveling salesman problem”, *Proc. of AMS, 71, 1956.*

[22] J. T. Laaksonen, J. M. Koskela, S. P. Laakso, and E. Oja, ”PicSOM - content-based image retrieval with self-organizing maps”, *Pattern Recognition Letters*, 21(13-14), pp. 1199-1207, December 2000.

[23] W. Y. Ma, B. S. Manjunath, ”A Comparison of Wavelet Transform Features for Texture Image Annotation”, *Proc. IEEE International Conf. On Image Processing*, 1995.

[24] MUVIS. http://muvis.cs.tut.fi/

[25] K. Lin, H.V. Jagadish, and C. Faloutsos. “The TV-tree: an index for high dimensional data”, *Very Large Databases (VLDB) Journal*, 3(4), pp. 517-543, 1994.

[26] Open Video Project. http://www.open-video.org/

[27] M. Partio, B. Cramariuc, M. Gabbouj, A. Visa, “Rock Texture Retrieval Using Gray Level Co-occurrence Matrix”, *Proc. of 5th Nordic Signal Processing Symposium*, Oct. 2002.

[28] A. Pentland, R.W. Picard, S. Sclaroff, “Photobook: tools for content based manipulation of image databases”, *In Proc. of SPIE (Storage and Retrieval for Image and Video Databases II)*, 2185, pp. 34-37, 1994.

[29] R. C. Prim, “Shortest Connection Matrix Network and Some Generalizations”, *Bell System Technical Journal*, vol. 36, pp. 1389-1401, November, 1957.

[30] L. R. Rabiner and B. H. Juang, Fundamental of Speech Recognition, Prentice hall, 1993.

[31] Y. Sakurai , M.Yoshikawa , S. Uemura , H. Kojima, “The A-tree: An Index Structure for High-Dimensional Spaces Using Relative Approximation”, *In Proc. of the 26th International Conference on Very Large Data Bases*, pp. 516-526, September 10-14, 2000.

[32] T. K. Sellis , N. Roussopoulos , C. Faloutsos, “The R+-Tree: A Dynamic Index for Multi-Dimensional Objects”, *In Proc. of the 13th International Conference on Very Large Data Bases*, pp.507-518, September 01-04, 1987.

[33] I. K. Sethi, I. Coman, “Image retrieval using hierarchical self-organizing feature map”, *Pattern Recognition Letters*, 20:1337–1345, 1999.

[34] J.R. Smith and Chang, “VisualSEEk: a fully automated content-based image query system”, *In Proc. of ACM Multimedia*, Boston, November 1996.

[35] C. Traina Jr., A. J. M. Traina, B. Seeger, and C. Faloutsos, “Slim-trees: High performance metric trees minimizing overlap between nodes”, *In Proc. of EDBT 2000*, pp. 51-65, Konstanz, Germany, March 2000.

[36] H. Wang, C.-S. Perng, “The $S^2$-Tree: An Index Structure for Subsequence Matching of Spatial Objects”, *In Proc. of 5th Pacific-Asic Conf. On Knowledge Discovery and Data Mining (PAKDD)*, Hong Kong, 2001.

[37] R. Weber , H.-J. Schek , S. Blott, “A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces”, *In Proc. of the 24rd International Conference on Very Large Databases*, pp.194-205, August 24-27, 1998.

[38] D. White and R. Jain, “Similarity Indexing with the SS-tree”, *In Proc. of the 12th IEEE Int. Conf. On Data Engineering*, pp. 516-523, 1996.

[39] Virage. www.virage.com

[40] P. N. Yianilos, “Data structures and algorithms for nearest neighbor search in general metric spaces”, *In Proc. of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, pp.311-321, Austin, Texas, US, January 25-27, 1993.

[41] H. Zhang and D. Zhong, “A scheme for visual feature based image indexing”, *In Proc. of SPIE/IS&T Conf. On Storage and Retrieval for Image and Video Databases III*, vol. 2420, pp. 36-46, (San Jose, CA), February 9-10, 1995.

[42] X. Zhou , G. Wang , J. X. Yu , G. Yu, “M+-tree: a new dynamical multidimensional index for metric spaces”, *In Proc. of the Fourteenth Australasian database conference on Database technologies 2003*, pp.161-168, Adelaide, Australia, February 2003.