Time and Space Efficient Content Queries for Video Databases

C. Makris, K. Perdikuri, S. Sioutas, A. Tsakalidis, K. Tsichlas
Department of Computer Engineering and Informatics,
University of Patras, 26500 Patras, Greece
and
Computer Technology Institute P.O. BOX 1192, 26110 Patras, Greece

Abstract:- Indexing video content is one of the most important problems in video databases. In this paper we present a simple optimal algorithm for this problem that answers certain content queries invoking video functions in linear time and space in terms of the number of the objects appearing in the video. To accomplish this, we make a straightforward reduction of this problem to the intersection problem in Computational Geometry. Our result is an improvement over the one of V. S. Subrahmanian [10] by a logarithmic factor in storage and is achieved by using different basic data structures. This logarithmic save is of great importance in video databases because vast space is needed to store videos and metadata for each video. Finally, we present two time-efficient approaches. We also compare the CPU times of our algorithms by presenting experimental results.

Keywords: video databases, data structures, computational geometry.

1 Introduction

A video database consists of a collection of videos and a mechanism, which permits the user to perform certain tasks. A ubiquitous task in such databases is that of indexing video content. Informally, we define video content to be a collection of objects or activities, which appear in the specific video at certain frame ranges. An object may be an item, a person or generally something tangible. An activity is an action or a relation between certain objects. There is no difference in the way we handle objects and activities and so we will refer only to objects. This information is stored as metadata and is not extracted from the video in real time. Thus, we must seek an efficient organization of the metadata of each video.

We would like our video database to be able to answer efficiently queries of the form: *Find and report all objects that appear in a given range of frames*. In order to accomplish this we must store the range sequences, where each object appears, in a data structure, associate each of these frames to its respective object and query this structure. The query segment is used to extract all the frame sequences intersecting it. Having found these segments it is straightforward to find the objects appearing in this query frame segment. Thus, our query is a simple intersection query of segments on the line, where the query is also a segment. The intersection problem defined above is static because in a video there is a predefined set of objects and respective frame segments.

The previous solution was based on a well-known data structure used extensively in the domain of Computational Geometry, the *segment tree* [4,5,6,7]. This is a very simple and elegant data structure but exhibits certain deficiencies in specific geometric problems. If we assume that each object is appearing in a sequence of frame segments, then we store each of these segments in $O(\log n)$ nodes of the segment tree, where n is the number of distinct endpoints of the frame segments stored in the segment tree. This is a sheer waste of space if we imagine that each video may have many objects associated to many frame segments. In addition, the solution described in Subrahmanian may report each of these ranges $O(\log n)$ times, which in many cases may be undesirable.

To remedy these problems we resort to a more suitable data structure, the *interval tree* [5,6,7]. The interval tree uses linear space because it stores each range only once, as we show in the following sections.

In section 2 we are going to give a thorough review of the previous solution while in section 3 certain preliminary data structures are going to be described synoptically. In section 4 the data structure is presented and the result is given respectively. In section 5 a reduction to another geometric problem is considered while in section 6, fusion tree methods are applied. In section 7 we compare the CPU times of some of our algorithms presenting experimental results. Finally, in section 8 conclusions are made.

2 The previous solution

Assume that initially we are given a table of n objects o_i and the associated frame segments of a given video v with total number of frames equal to $framenum(v)$. We want to organize this metadata in order to answer efficiently video content queries.

Assume that $[s_1,e_1),\ldots,[s_w,e_w)$ are all the intervals in this table. Let $q_1,\ldots,q_n$ be an enumeration, in ascending order, of all members of $\{s_i,e_i \mid 1\le i\le w\}$, with duplicates eliminated. If n is not an exponent of 2, then do as follows: let r be the smallest integer such that $2^r > n$ and $2^r > framenum(v)$. Add a number of new elements $q_{n+1},\ldots,q_{2^r}$ such that $q_{2^r} = framenum(v)+1$ and $q_{n+j} = q_n + j$ (for $j>0$ such that $n+j<2^r$). Now we may proceed under the assumption that n is an exponent of 2. The next step is to construct the indexing data structure that is called the frame segment tree. This is a full binary tree.

Each node in the frame segment tree represents a frame sequence $[x,y)$, starting at frame x and including all frames up to, but not including, frame y. All leaves are at level r, where obviously $r = O(\log n)$. The leftmost leaf denotes the interval $[q_1,q_2)$, the 2nd from the left represents the interval $[q_2,q_3)$ and so on. If u is a node with two children representing the intervals $[p_1,p_2),[p_2,p_3)$, then u represents the interval $[p_1,p_3)$. Thus, the root of the frame segment tree represents the interval $[q_1,q_n)$ if q_n is an exponent of 2, otherwise it represents the interval $[q_1,+\infty)$.

Without proof we give below some elementary results on segment trees.

1. The segment tree uses $O(n\log n)$ space
2. Each segment is stored in $O(\log n)$ nodes.

The second property of the segment tree is a cause of problems because of two reasons:

1. There is a sheer waste of space.
2. In the reporting procedure one particular segment may be reported $O(\log n)$ times.

The reporting procedure given by Subrahmanian is given below. By R we represent the subtree rooted by the current node. In the first call of the procedure, R is the whole tree. The parameters s and e define the endpoints of the query segment. The variable S is the output of the procedure. In each node of the frame segment tree we store a linked list of the objects. These objects have frame segments associated to the frame sequence of the specific node. In this way, if a node's frame sequence intersects the query frame segment then the linked list of this node is appended to variable S. The frame sequence of a node v is represented by $[v.LB, v.RB)$. The right and left child of the node v is represented by $v.RLINK$ and $v.LLINK$ respectively.

```
FindOInV(R, s, e);
{
  S = NIL;
  if (R = NIL) then {Return(S); Halt};
  else
  {
    if ([R.LB, R.UB) ⊆ [s, e))
      then S = append(S, preorder(R))
    else
    {
      if (([R.LB, R.UB) ∩ [s, e)) ≠ ∅) then
      {
        S = append(S, R.obj);
        S = append(S, FindOInV(R.LLINK, s, e));
        S = append(S, FindOInV(R.RLINK, s, e));
      }
    }
  }
  Return(S); end;
}
```

The running time of the algorithm is proportional to the total number of nodes visited, which may be at most $O(n)$.

3 Preliminary data structures

This section is devoted to the interval tree. It allows us to store a set of *n* intervals in linear space such that intersection queries can be answered in logarithmic time.

Let $S = [x_i, y_i] 1 \le i \le n$ be a set of *n* closed intervals on the real line. Let $Q = q_1, \ldots, q_n$ be an enumeration, in ascending order, of all members of $\{x_i, y_i \mid 1 \le i \le n\}$, with duplicates eliminated. An interval tree *T* for *S* is a leaf-oriented search tree for *Q* where each node of the tree is augmented by additional information.

We define $xrange(v)$, where v is a node of the interval tree, as the interval $\lfloor q_{left}, q_{right} \rfloor$ such that: q_{left} is the leftmost leaf of the subtree rooted at *v* while q_{right} is the rightmost one.

The *node list* $NL(v)$ of node *v* is the set of intervals in *S* containing the split value of *v* but of no ancestor of *v:*

$$NL(v) = \{[x, y] \in S; split(v) \in [x, y] \subseteq xrange(v)\}$$

We store the node list of node *v* as two sorted sequences: the ordered list of left endpoints and the ordered list of right endpoints. Both sequences are stored in balanced search trees; furthermore, we provide pointers to the maximal (minimal) element of the sequence of right (left) endpoints.

The main power of interval trees stems from the node lists. The following lemma shows that interval trees use linear space, can be constructed efficiently, support insertions and deletions of intervals and answers intersection queries efficiently.

Lemma 1: Let *S* be a set of *n* intervals.

a) An interval tree for *S* uses space $O(n)$.

b) An interval tree for *S* of depth $O(\log n)$ can be constructed in time $O(n \log n)$.

c) Intervals can be inserted into an interval tree of depth $O(\log n)$ in time $O(\log n)$. The same holds for deletion.

d) Let *S* be a set of intervals and let $I = [x_0, y_0]$ be a query interval. Let $t = \{[x, y] \in S; [x, y] \cap [x_0, y_0] \neq \varnothing\}$ be the set of intervals in *S* intersecting *I*. Then, given an interval tree of height $O(\log n)$ for *S*, one can compute the intersection query *t* in time $O(\log n + t)$.

Proof. See [6].

4 The new algorithm

We store each frame segment associated to an object in the interval tree. We must note that it is not imperative to extend the tree to a full binary one since this leads to waste of space.

Each frame segment is inserted in the interval tree. It is stored only in one node and thus we obtain a logarithmic save in space. This frame segment is associated with an object. Namely, the object appears in the frame sequence defined by this frame segment. In this way, each segment is related to only one object and is stored only once.

In each node of the interval tree there may be many segments stored to its node list. These segments are stored sorted in each of these node lists. Attached to each such segment is the object to which the specific segment belongs. The attachment is expressed by a pointer to a list of objects corresponding to the specific node. In this way, when we find a segment, which intersects the query segment *I*, we immediately report the respective object.

The search for all the segments intersecting the query segment is described below. We start from the root and search for the two endpoints of the query segment. The two paths from the root to the two leaves of the interval tree comprise the set *P*. These two paths coincide until they reach a node, called *split node*. The leftmost and rightmost leaves of its subtree are the endpoints of the query segment *I*. Let P_{Left} and P_{right} be the two paths from split node to the left and right endpoints of *I* respectively. Similarly, the set *C* is defined as the set of nodes *v* where $xrange(v) \subseteq I$.

Intuitively, the nodes that belong to the set C are defined by the right (left) subtrees of the nodes in set P_{Left} (P_{Right}). Thus, when we are reporting the answer corresponding to a query segment we may just append the list of objects associated to a node, belonging to set *C,* to the answer. In nodes of set *P* we are obliged to search inside the node lists. These node lists, as mentioned in the preceding section, are organised as binary trees whose leaves are connected in a linked list and whose root has pointers to the smallest and to the largest element (endpoint). In each node list of a node

in a set P some segments may intersect I and some others may not. Because of the fact that the segments in the node lists are stored sorted we are able to report the answer in these in constant time per object.

The largest or smallest element is accessed in constant time through the respective root pointers. We then traverse the linked list of leaves until we reach a segment that does not intersect the query segment. If the node under consideration lies on the path to the left endpoint we begin the traversal of the linked list from the largest element (rightmost leaf). If the node lies on the right we begin from the smallest element (leftmost leaf). If the node lies on both of them, that is, on the path from root to the split node of the two paths, we compare the endpoints of the query segment to the largest and smallest elements and we proceed analogously.

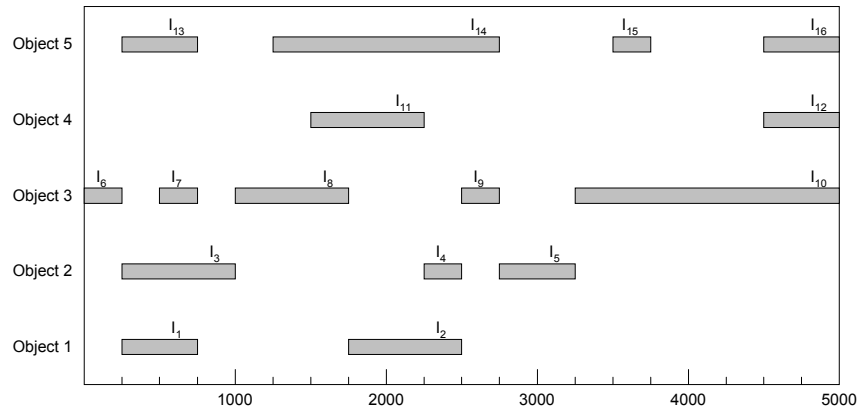

Figure 1. Example of the contents of a video

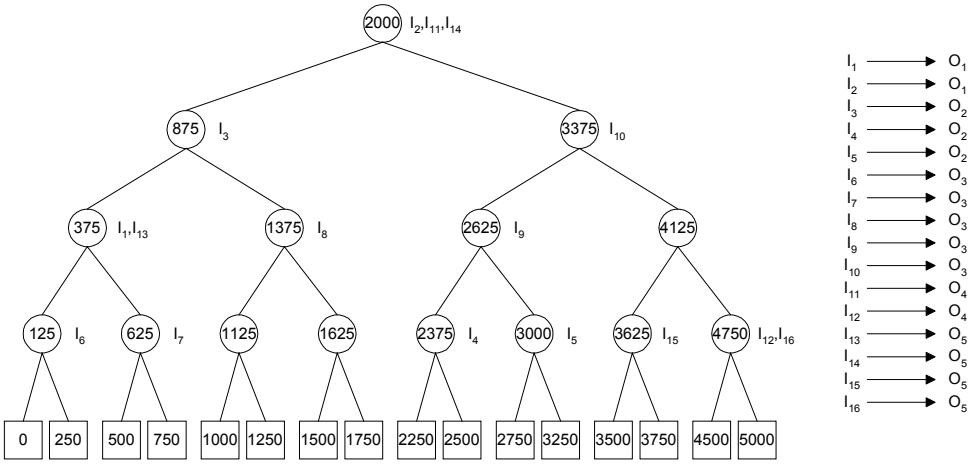

Figure 2. The frame interval tree with intervals associated with nodes and each interval associated with an object

For better comprehension of the above method we give a simple example. Assume that we have a video and we are interested for only 5 objects. Each object appears in the video in a sequence of series of frames defining frame segments. Figure 1 depicts objects and their location in a video consisting of 5000 frames. Figure 2 depicts the frame interval tree.

5 Reduction to Dominance

In the preceding sections we associated the problem of querying video content with that of finding segments that intersect a specific segment. In this section we outline a different approach to this problem. Specifically, we show how to reduce the problem of querying video content to the dominance problem. The dominance problem is a pure geometrical problem.

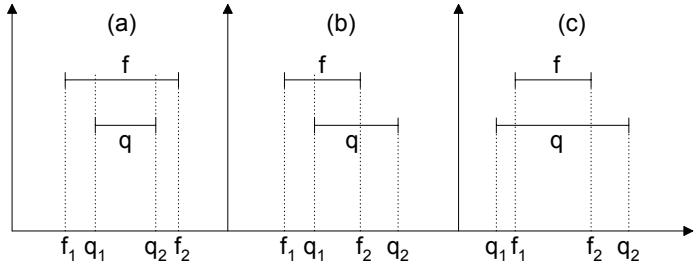

Figure 3. Query segment q compared to a frame segment f, (a) q is contained in f, (b) q's left endpoint is contained in f and (c) q contains f.

The d-dimensional dominance problem is defined as follows: given a set Q of d-dimensional points and a query point p, report all points $p' \in Q$ such that p' is dominated by p. A point $p' = (p'_1, p'_2, \ldots, p'_d)$ is dominated by $p = (p_1, p_2, \ldots, p_d)$ if and only if $p'_i \le p_i, \forall i$. Here we consider the 2-dimensional dominance problem.

Let $f = [f_1, f_2]$ be a given frame segment. A query segment $q = [q_1, q_2]$ intersects f if and only if one of the following conditions hold:

(i) f contains q, that is $f_1 \le q_1 \le q_2 \le f_2$ (see figure 3(a)).

(ii) f partially intersects q, that is $f_1 \le q_1 \le f_2$ (f intersects q on the left) or $f_1 \le q_2 \le f_2$ (f intersects q on the right) (see figure 3(b) for the first case).

(iii) f is contained in q that is $q_1 \le f_1 \le f_2 \le q_2$ (see figure 3(c)).

From the above cases it is trivial to see that it suffices to store the frame segments in a two-dimensional dominance-searching problem. This structure stores each frame segment $f = [f_1, f_2]$ as a two-dimensional point (f_1, f_2). Then, given a query segment $q = [q_1, q_2]$ we find all segments intersecting q by querying the structure with $(q_1, -q_2)$ (case (i)), $(q_1, -q_1)$ (case (ii) left partial intersection), $(q_2, -q_2)$ (case (ii) right partial intersection) and $(-q_1, q_2)$ (case (iii)).

The two-dimensional dominance-searching problem has an optimal RAM dynamic solution using linear space, exhibiting query time $O(\log n / \log\log n + k)$ and update time $O(\log n / \log\log n)$, where k is the size of the output (number of reported objects in our case) ([11]).

The static counterpart of this problem has an optimal RAM solution of linear space, and $O((\log n / \log\log n)^{1/2} + k)$ query time (simple combination of persistence and the search structure of [3]).

Finally, in secondary memory an optimal dynamic structure has been proposed recently ([2]) that occupies $O(n/B)$ disk pages (B is the size of a page), supports insertions and deletions in $O(\log_B n)$ I/Os and answers queries in $O(\log_B n + k/B)$ I/O's.

6 The fusion tree solution

First, we will briefly review the data structures used in this solution.

6.1 The fusion tree

Let S be an ordered set of n w-bit keys. The *fusion tree* [11] is a dynamic data structure that supports $O(\log n / \log\log n)$ amortized time queries in linear space. This structure is a two-level data structure where the upper level consists of an ordinary B-tree while the lower level consists of weighted-balanced trees. The amortized cost of searches and updates is $O(\log n / \log b + \log b)$ for any $b = O(w^{1/6})$. The first term corresponds to the number of B-tree levels and the second to the height of the weighted-balanced trees.

The main advantage of the fusion technique is that we can decide in constant time in which subtree to continue the search by compressing the b-keys of every B-tree node using w-bit words.

6.2 The exponential search tree

The *exponential search tree* [1] answers queries in one-dimensional space. It is a multi-way tree where the degree of the internal nodes decrease

exponentially as we traverse the levels of the tree starting from the root. Auxiliary information is stored in each node to support efficient search queries. The exponential search tree has the following properties:

1. Its root has degree $\Theta(n^{1/5})$.
2. The keys of the root are stored in a local data structure. During a search procedure, the local data structure is used to determine in which subtree of a node the search is to be continued.
3. The subtrees are exponential search trees of size $\Theta(n^{4/5})$.
4. The local data structure of each node of the tree is a combination of van Emde Boas trees and perfect hashing. As a result we achieve $O(\log w \log\log n)$ worst-case time cost for a search query.

Anderson, by using an exponential search tree in the place of *B*-trees in the fusion tree structure, avoids the need for weight-balanced trees at the bottom while at the same time improves the complexity for large word sizes. This structure is a significant improvement on linear space deterministic sorting and searching. On a unit-cost RAM with word size *w*, an ordered set of *n* *w*-bit keys (viewed as binary strings or integers) can be maintained in $O\left(\min\left\{\sqrt{\log n}, \log n/\log w + \log\log n, \log w \log\log n\right\}\right)$ time per operation, including *insert*, *delete*, *member search* and *neighbour search*. The cost for searching is worst–case while the cost of updates is amortized. For range queries there is an additional cost of reporting the found keys. As an application, *n* keys can be sorted in linear space at a worst-case time cost of $O(n\sqrt{\log n})$. The best previous method for deterministic sorting and searching in linear space has been the fusion tree, which supports search queries in $O(\log n/\log\log n)$ amortized time and sorting in $O(n\log n/\log\log n)$ worst–case time.

6.3 The fusion interval tree

Let *T* be a *B*-ary tree, that is a tree for which each node has *B* sons. We set the branching factor $B = O(\sqrt{\log n})$. Each node *v* in ordinary interval trees such that $v = nca(x_1, x_2)$ (nca=nearest common ancestor), stores the value $range(v) = [x_1, x_2]$. Thus, in each node *v* such that $v = nca(x_1, x_2)$ we store the following *B* slabs:

$sl_1 = (x_1, k_1], sl_2 = (k_1, k_2], \ldots, sl_B = (k_{B-1}, x_2]$

We define the structure of each node list $NL(v)$ as follows: $NL(v) = \{s \in S;$ *s* spans a slab of *v* and *s* is included in a slab of *parent(v)*$\}$.

If $s \in NL(v)$, then *s* spans a continuous set of slabs $sl_i, \ldots, sl_{i+k}$ and *s* cuts the two bound-slabs sl_{i-1}, sl_{i+k+1}. In the slabs $sl_i, \ldots, sl_{i+k}$, *s* is stored in an unordered list but in sl_{i-1}, sl_{i+k+1} *s* is stored in an ordered list (see figure 4) of endpoints that is organized as an exponential search tree [1]. Thus, the total required space for the node lists is $O(nB)$.

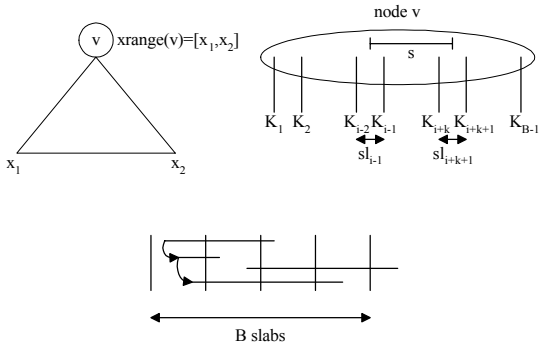

Figure 4: Depiction of the *xrange* of a node, of the slabs and their order inside ordered lists.

Lemma 2: Let $U \subseteq \Re$ be an ordered finite universe and let *S* be a set of *n* intervals with left endpoints in *U*. To simplify our applications we assume that $|S| = |U| = n$.

a) The fusion interval tree *T* for *S* requires $O(nB)$ space.

b) The fusion interval tree *T* for *S* can be constructed in time $O(nB)$.

c) Intervals (with left endpoint in *U*) can be inserted into *T* in time $O(B)$. Deletions are completely symmetric.

d) The intersection query *t* can be solved in $O(\log n/\log\log n + t)$ time.

Proof:

a) A *B*-tree for *U* clearly uses linear space $O(|U|) = O(n)$. Furthermore, the total space required for the node lists is $O(nB)$ since every interval in *S* is stored in the *B* slabs. The space complexity follows.

b) A *B*-tree can be built in time $O(|U|) = O(n)$ and has depth:

$O(\log_B |U|) = O(\log n / \log B) = O(\log n / \log\log n)$

It remains to construct the node lists. We show how to insert an interval $s = [x_1, x_2]$ in $O(B)$ time. Let *v* be the nearest common ancestor of the endpoints of *s*. This node can be computed in constant time [9]. It remains to insert the interval *s* in the *B* slabs of node *v*. Assuming that *s* spans a continuous set of slabs $sl_i, \ldots, sl_{i+k}$ and cuts the two bound-slabs sl_{i-1}, sl_{i+k+1} we can insert the interval *s* in the slabs $sl_i, \ldots, sl_{i+k}$, in an unordered list with $O(B)$ cost. However, we have to insert interval *s* in the ordered lists of sl_{i-1}, sl_{i+k+1}. This incurs $O(\sqrt{\log n})$ cost since the ordered list of each slab is organized as an exponential search tree [1]. Thus, the sequence of insertions of *n* intervals require $O(nB)$ time and as a result the total construction time is $O(n + nB) = O(nB)$.

c) It is obvious from the construction above, that we can insert an interval $s = [x_1, x_2]$ in $O(B)$ time. The same holds for deletions.

d) We define sets *P* and *C* in the same way as in section 4. *P* consists of the nodes on the search paths to *x* and *y* and *C* is the set of nodes between these paths. Let *t* the output of intersection query *I*. Since $[x, y] \in NL(v)$ and $[x, y] \cap I \neq \varnothing$ implies $xrange(v) \cap I \neq \varnothing$, *t* is defined as follows:

$$t = \sum_{v \in C} NL(v) \cup \sum_{v \in P} \{(x, y) \in NL(v), [x, y] \cap I \neq \varnothing\}$$

In addition, the fact that $v \in C$ clearly implies that $NL(v) \subseteq t$. Now consider nodes *v* such that $v \in P$. Recall that we organized $NL(v)$ as two ordered lists, the list of left endpoints and the list of right endpoints. Let $x_1 \leq x_2 \leq \ldots \leq x_k$ be the former list and let $y_1 \leq y_2 \leq \ldots \leq y_k$ be the latter list. We have to discuss three cases, two of which are symmetric. Suppose first that $xrange(v) \subseteq I$. Then $NL(v) \subseteq A$, since we know that $xrange(v) \subseteq [x, y]$ for all $[x, y] \in NL(v)$.

Suppose that $xrange(v) \subseteq [x_1, x_2] \not\subset I$ and $x_1 \leq x_0$ (the other case is symmetric). Then, interval $[x_i, y_i] \in NL(v)$ intersects *I* if $x_0 \leq y_j$. We can thus find all such intervals by inspecting $y_k, y_{k-1}, \ldots$ in turn as long as they are at least as large as x_0. Hence we can determine $NL(v) \cap t$ in time proportional to $|NL(v) \cap t|$.

Thus, the time required for the computation of *t* is $O(|P| + |C| + t)$. For the case of a "small" universe *U*, which means that *U* contains only endpoints of intervals in *S*, according to [6], $|C| = O(t)$ since all leaves in *C* are endpoints of intervals in *t*. Since $|P| \leq 2h$, where $h = O(\log n / \log\log n)$, the time bound follows.

7 Experimental Results

We implemented the solution described in the second approach (let T2 the CPU time) and the one of V. S. Subrahmanian [10] (let T1 the CPU time) in Visual C++ 6.0 and we executed the two programs in Pentium based PC with the following hardware and software characteristics.

- 200MHz Pentium
- 48 MB of RAM
- 1,2 Gbyte hard disk space
- Windows 98 Operating System

We ran several types of $FindOInV(R, s, e)$ queries and the time improvement seems to be significant as we show below.

In order to count the execution time of our algorithms we used the *clock*( ) function included in <*time.h*> header. This function counts the number of executed cycles for the time period that the computation took place for the answer of a given query. We finally divided that number of cycles by CLOCKS_PER_SEC of system in order to evaluate the CPU TIME in seconds (or msec). In order to evaluate a non-zero number of cycles we included the procedure above within a while–loop of 10*CLOCKS_PER_SEC dividing finally the cycle-number we found by the loop's number we computed.

Table 1 and figure 5 below show the results of the execution of nine $FindOInV(R, s, e)$ queries, in a set of various number of multimedia objects (each multimedia object is appeared a constant number of times):

Table 1: CPU times of the two algorithms

n (number of multimedia objects)	T1 (msec)	T2 (msec)
50	16,93157	6,78163
100	19,93157	7,29554
150	21,68646	7,599257
200	22,93157	7,815003
250	23,89735	7,982237
300	24,68646	8,118714
350	25,35363	8,23395
400	25,93157	8,333636
450	26,44134	8,421451

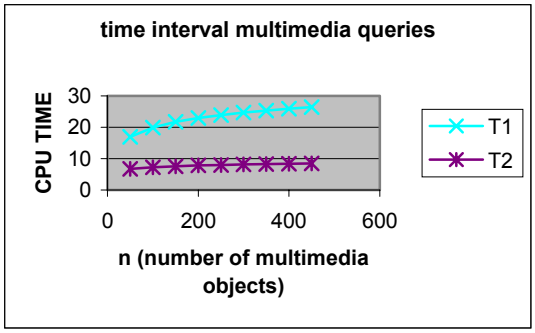

Figure 5: A comparison of the CPU times of two algorithms

8 Conclusion

We have discussed in this paper a way to store video metadata in order to perform efficiently content queries. Metadata specifies either objects or activities in the video. Because of the vast space needed for a video database it is essential to save as much space as possible.

The solutions we suggest are space optimal and perform efficiently video content queries. All the proposed structures are currently being implemented and the results so far are promising. Our next step for future continuation of this work is an extensive experimental evaluation of various structures for several classes of selection queries.

References

[1]Anderson, A. Faster deterministic sorting and searching in linear space. *In Proc. of 37th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, 1996.

[2] L. Arge, V. Samoladas, J.S. Vitter. Two dimensional indexability and optimal range search indexing. *In Proceedings of the ACM Symposium Principles of Database Systems (PODS)*, 1999.

[3] Paul Beam and Faith Fich. Optimal Bounds for the Predecessor Problem. *In Proceedings of the Thirty First Annual ACM Symposium on Theory of Computing (STOC)*, Atlanta, GA, May 1999.

[4] Bentley J.L. *Solution to Klee's Rectangle Problem.* Carnegie-Mellon Univ., Dept. of Computer Science, unpublished notes, 1977.

[5] Mark de Berg, Marc van Kreveld, Mark Overmars, Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications.* Springer-Verlag, Heidelberg, 1997.

[6] K. Mehlhorn. *Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry.* EATCS Monographs on Theoritical Computer Science, Springer-Verlag, 1984.

[7] F.P. Preparata, M.I. Shamos. *Computational Geometry: An introduction,*. Springer-Verlag, New York, 1985.

[8] H. Samet. *The Design and Analysis of Spatial Data Stuctures.* MA: Addison-Wesley, 1989.

[9]B. Schieber, U. Vishkin. On finding lowest common ancestors: simplifications and parallelization. *SIAM J. of Comput.*, 17:1253-62, 1988.

[10] V.S. Subrahmanian. *Principles of Multimedia Database Systems.* Morgan Kaufmann Publishers Inc., pp. 179-213, 1998.

[11] Dan E. Willard. Applications of the fusion tree method for Computational Geometry and searching. *In Proc. 3rd Symposium on Discrete Algorithms (SODA)*, pp. 286-296, 1992.