S*–Tree: An Improved S+-Tree for Coloured Images*

Enrico Nardelli[1,2] and Guido Proietti[1]

[1] Dipartimento di Matematica Pura ed Applicata, Università di L'Aquila,
Via Vetoio, 67010 L'Aquila, Italy
{nardelli,proietti}@univaq.it
[2] Ist. di Analisi dei Sistemi e Informatica,
CNR, V.le Manzoni 30, 00185 Roma, Italy

Abstract. In this paper we propose and analyze a new spatial access method, namely the S*-tree, for the efficient secondary memory encoding and manipulation of images containing multiple non-overlapping features (i.e., coloured images). We show that the S*-tree is more space efficient than its precursor, namely the S+-tree, which was explicitly designed for binary images, and whose straightforward extension to coloured images can lead to large space wastage. Moreover, we tested time efficiency of the S*-tree in answering classical window queries, comparing it against a previous efficient access method, namely the HL-quadtree [7]. Our experiments show that the S*-tree can reach up to a 30% of time saving.

Keywords: Spatial databases, bintree, quadtree, window queries.

1 Introduction

In this work we focus on secondary memory representations of images containing multiple non-overlapping spatial features, like for instance agricultural maps, thematic maps, satellite views and many others. This is a very hot research topic, especially with the increasing interest of the database community towards the development of efficient management systems for geographical data (GIS). Therefore, we are implicitly assuming that the underlying images have all the peculiar aspects of images containing *region data*, and specifically the most prominent one, that is the *aggregation* of pixels of a given colour into patches. This induces a couple of observations: first, the number of features (i.e., colours) in the representing picture is limited (generally, from 8 to 32), second, and perhaps more important, it makes sense to apply hierarchical methods of representation of the image to save space and time.

One of the most successful hierarchical strategy for representing images containing region data is based on the decomposition of the image space containing the data into recursively nested subimages, until an homogeneous pattern is

* This research was partially supported by the CHOROCHRONOS TMR Program of the European Community.

Eder et al. (Eds.): ADBIS'99, LNCS 1691, pp. 156–168, 1999.

obtained. The most popular decomposition techniques are the *binary decomposition* (which splits the image into two equal parts alternating an horizontal and a vertical subdivision) and the *quaternary decomposition* (which splits the image into four equal quadrants). The corresponding main memory representations of such split policies are the *bintree* [12] and the *region quadtree* [9]. Both data structures are easy to implement in main memory. On the other hand, when a secondary memory representation is needed (which is usually the case, given the large amount of data to be stored), things become more complicated. The problem is that of mapping a 2-dimensional set onto a 1-dimensional universe, while attempting to preserve as much as possible spatial proximity properties.

For images containing multiple non-overlapping features (for the sake of brevity, *coloured images* in the following, even though this term could be misleading, since it does not convey the concept that the underlying image is representative of region data and therefore well-suited to be managed by hierarchical spatial data structures), a number of different secondary memory implementations have been proposed. These can be subdivide into two categories: *leafcode representations*, obtained as a collection of the leaf nodes in the tree (such as, for example, the *linear quadtree* [5]), and *treecode representations*, obtained by a preorder tree traversal of the nodes in the quadtree (also called *DF-expressions* [6]). The latter approach is asymptotically more compact than the former one, but it has suffered for a long time the lacking of a paged version able to support the access to a given element without being forced to scan, in the worst case, the entire database. This difficulty have been overcome by de Jonge et al. [3], who developed the *S+-tree*, a spatial access method combining the advantages of treecode and leafcode representations, essentially by indexing through locational codes the space-compact DF-expression. However, as we shall see in the rest of the paper, the S+-tree is tailored on binary images, and a straightforward extension of it to coloured images has a severe space utilization drawback, which affects in its turn the time efficiency in solving classical operations that can be posed on the stored data.

In this paper we present a new spatial access method, that we named *S**-tree, which extends capabilities of the S+-tree to coloured images. We first show that for practical cases the S*-tree saves up to 25% of space with respect to the S+-tree. We then prove that the S*-tree performs asymptotically the same number of disk accesses of the S+-tree to retrieve any given subset of the represented image. Finally, to assess the practical usefulness of our method, we perform experiments over an important class of queries on coloured images, namely the *window queries*. Window queries have a primary importance since they are the basis of a number of operations that can be executed in a spatial database. Window-based queries we analyze are the following:

- *exist*(*f*,*w*): Determine whether or not the feature f exists inside window w;
- *report*(*w*): Report the identity of all the features that exist inside window w;
- *select*(*f*,*w*): Report the locations of all occurrences of the feature f inside window w.

We compare our structure with the *HL-quadtree* [7], another hybrid access method combining advantages of leafcode and treecode representations, by using locational codes to represent all the nodes of a region quadtree. The HL-quadtree has been shown to be very time efficient in solving window queries. Obtained results are extremely encouraging, showing a superiority of our method both in terms of space occupancy and time performances. Since we are comparing time performances of secondary memory oriented data structures, when we state that our new data structure outperforms previous approaches for answering window queries, we refer to the fact that it reduces the number of accesses to the buckets storing the data. This is the classical *I/O complexity*.

The paper proceeds as follows. In Section 2 we briefly recall the various pixel tree (binary and quaternary) structures that have been proposed in the past for managing coloured images. In Section 3 we present our new spatial access method, namely the S*-tree. In Section 4 we give experimental results assessing the space and time efficiency of our approach, and finally, in Section 5 we present considerations for further work and concluding remarks.

2 Survey

2.1 The Bintree and the Quadtree

The *region quadtree* is a progressive refinement of an image that saves storage being based on regularity of the feature distribution. Assume we are given an image space of size $T \times T$ (e.g., pixel elements), where T is such that $T = 2^m$, containing k non-overlapping features. We proceed in the following way: at level 0 there is the whole image, of side length T. The decomposition process carried out by the quadtree recursively splits a quadrant into four equal size quadrants, until each quadrant is covered by only one feature. In the worst case, the decomposition can go on up to the pixel level, with squares of side length $\frac{T}{2^m} = 1$. The decomposition can be represented as a tree of outdegree 4, with the root (at level 0) corresponding to the whole image and each node (at level d) corresponding to a square (or *block*) of side length $\frac{T}{2^d}$. The sons of a node are, in preorder, labelled NW, NE, SW and SE. For a given image, nodes are then *homogeneous* (leaf nodes) or *heterogeneous* (non-leaf nodes). Correspondingly, we speak of homogeneous and heterogeneous blocks. Note that there exist several extensions of the region quadtree, even for representing set of overlapping images [14].

The *bintree* is the binary version of the region quadtree: the image is progressively refined alternating horizontal and vertical splits, until an homogeneous pattern is reached. Notice that in this case such a pattern is not necessarily a square. Figure 1 shows an example of an image containing 4 non-overlapping features (note that the white background is treated as a feature), along with its representing quadtree and bintree.

The bintree and the quadtree can be implemented as a tree (pointer-based representation) or as a list (pointerless representation). In the former, direct access to specific image elements is privileged, while the latter makes sequential

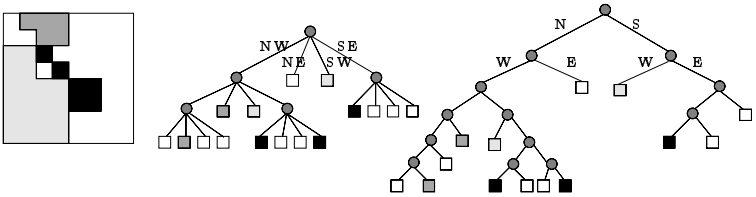

Fig. 1. Multiple non-overlapping features and their quadtree (left) and bintree (right).

access easier and simplifies disk-based representations, absolutely needed for large amounts of spatial data [10,11,13].

2.2 Secondary Memory Implementations

It should be clear from the definition that bintrees and quadtrees share a lot of properties; therefore, a secondary memory implementation of a bintree can be easily adapted to a quadtree, and vice versa. Henceforth, in the following we will freely interexchange them.

There exist substantially two categories of list-based representation of a pixel tree: the collection of the leaf nodes and the linear list resulting from a preorder traversal of the tree. One of the most attractive approaches in the first category is the *FL linear quadtree* [5] (simply *linear quadtree* in the following), introduced by Gargantini with reference to a binary image. The extension to multiple non-overlapping features is straightforward. In fact, also in this case the collection of leaf nodes can be stored as a sorted linear list, but each node now contains two fields: the *locational key*, whose digits resemble the path in the tree from the root to the node, and the *value string*, that contains the index of the feature associated with the node. Representing a pixel tree as an ordered list of the homogeneous nodes is efficient since space occupancy is reduced and performances of sequential operations are improved.

Concerning representations in the form of a linear list resulting from a preorder traversal of the pixel tree, the DF-expression [6] is surely one of the most used techniques. The DF-expression for multiple non-overlapping features can be viewed, treating the background as a feature, as a string containing two symbols: 'N', denoting non-leaf (internal) nodes, and 'L', denoting leaf nodes, followed by the index of the contained feature for leaf nodes. The representing tree is visited in preorder, and an 'N' is emitted whenever an internal node is encountered, while an 'L' followed by the index of the contained feature is emitted whenever a leaf node is encountered. As an example, suppose that the four features in Figure 1 have index 0 for the white, 1 for the light gray, 2 for the dark gray and 3 for the black. The following string is the DF-expression for the bintree in Figure 1:

$$\mathrm{NNNNNNL_0L_2L_0L_2NL_1NN\ L_3L_0NL_0L_3L_0NL_1NNL_3L_0L_0}.$$

Representing a pixel tree as a DF-expression is space efficient because of the data compression, but accessing specific blocks is time-consuming, since indexing is not provided. and this is a serious handicap for window queries processing. Therefore, an implementation based on B^+-trees for a linear quadtree representation is straightforward [1], while it is not possible for a DF-expression.

A first step towards the integration of leafcode and treecode representations has been done by de Jonge et al. [3], who defined a secondary memory representation of binary images named *S^+-tree*. The S^+-tree is obtained in the following way: preliminary, we visit in preorder the pixel tree, emitting a '0' ('1') when an internal (leaf) node is encountered. The bitstring thus produced is called a *linear bintree*. An additional bitstring, called *colour table*, records the colours of the leaves in preorder, by letting a '0' ('1') represent a white (black) leaf. The two bitstrings thus obtained are named *S-tree*. The S^+-tree is then built by indexing with a B^+-tree a list of data pages containing a segmented and augmented S-tree representation of the image. Each data page constitutes a self-contained local S-tree that can be searched independently. A data page consists of a bitstring merging the linear bintree (which grows from the beginning of the page) and the colour table (which grows from the tail of the page) of the local S-tree. Between them there is some unused space (actually, this unused space is negligible for binary images but it is not for coloured images, as we shall see in the next section). Moreover, at the very beginning of the page there is a *linear prefix* which can be regarded as the summary of all the data pages preceding the actual one. This linear prefix is determined in the following way: when a data page becomes full during the building process, a new page is created and a *separator* between the pages is stored in the index. Such a separator is built by encoding the path from the root of the bintree to the node which caused the filling of the page, emitting a '0' when moving towards left, a '1' otherwise. Since it is imposed that the last node stored in a page must be a leaf (we will analyze this constraint more in detail in the next section), it follows from the preorder visit properties that the last bit of a separator is always a 1[1]. Consequently, the linear prefix is built by encoding with a '0' a 0 in the separator, and with a '01' a 1 in the separator. The 0 added before the 1 actually represents a *dummy leaf*, staying for a left subtree (stored in a previous page) along the path to the node which caused the filling. The linear prefix therefore provides the information needed to retrieve a node in a page, since it resembles the whole bintree preceding the nodes in such a page by condensing all the left subtrees in leaves.

3 The S*-Tree

In the previous section, we mentioned that a tight constraint during the process of building the S^+-tree is that the last node stored in a page must be a leaf. There are several convincing reasons to do that for binary images:

[1] In fact, if the last stored node is a left leaf, then the node which caused the filling is a right leaf (its sibling), while if the last stored node is a right leaf, then the node which caused the filling is some right son of an ancestor of the right leaf

1. Since the last node is a leaf, by preorder visit properties it follows that the first node on the next page is a right son, and therefore the separator between the pages will end with a 1. This is important, since it allows to store the separators using only $2m$ bits, where m is the resolution of the image, without encoding the depth of the node the separator refers to which.
2. Since for binary images no information is associated to internal nodes (they are simply gray), we have at most $2m-1$ unused bits per page. Considering that a page is generally 512 bytes in size and that a reasonable upper bound on m is 16, it follows that we waste in the worst case less than 1% of space.

However, the latter observation does not hold any more for coloured images. In such a case an internal node has an amount of information associated with it: more precisely, to each internal node we have to associate a *colour string* of k bits (where k is the number of features contained in the image), in which the ith bit is 1 iff the node contains the ith feature. In fact, associating a colour string to internal nodes greatly improves the performances in executing several spatial operations [7]. Thus, if $m = 16$ and $k = 32$, the wasted space could be as big as 124 bytes, i.e., about a 25% of the page size! Therefore, it is clear that for coloured images we have to abandon the constraint that the last node stored in a page must be a leaf node. The question is: can this be done without modifying the separators, i.e., without augmenting the space used for the index? The answer is yes, on condition that a small overhead is paid in terms of the time spent when a search to a given node is performed. In fact, a problem arises letting the last node stored inside a page to be internal, that is, it fails the statement that the last bit of a separator is always a 1. This is because the node which caused the filling could be a left son, and iteratively its parent could be a left son and so on. Therefore, in the separator, after the rightmost 1, there could be some meaningful 0s (actually, as many as $2m-1$), i.e., 0s that effectively lead to the node which caused the filling. Does this affect the search of a given node through the structure? Only to a small extent, as the following theorem states:

Theorem 1. *Let $\ell = 2m$ be the length of the index keys in the B^+-tree storing the S^*-tree, and let $\pi(x) = \{0,1\}^t$ with $t \leq \ell$, be the path from the root to a node x to be retrieved in the S^*-tree. Then, as soon as each page in the B^+-tree contains at least ℓ nodes of the bintree, it follows that at most two contiguous pages in the B^+-tree must be visited to retrieve x.*

Proof. We start by noting that the assumption that each node in the B+-tree contains at least ℓ nodes of the bintree is not restrictive in applicative cases: for example, for $m = 16$ and $k = 32$, it suffices to fix the page size of the B+-tree to 128 bytes.

Let $\pi_i \in \{0,1\}$, $i \leq \ell$ be the ith bit of $\pi(x)$ and let π_r be the rightmost 1 of $\pi(x)$. We can therefore write $\pi(x) = \pi_1 \dots \pi_r \pi_{r+1} \dots \pi_t$, with $\pi_{r+1} = \dots = \pi_t = 0$. To retrieve x, we will search in the B+-tree for the key $k_x = \pi_1 \dots\dots \pi_r \pi_{r+1} \dots \pi_\ell$, with $\pi_{r+1} = \dots = \pi_\ell = 0$. Let k_a be the key in the B+-tree reached by searching k_x and let P_1, P_2 be the two pages separated by k_a. Without loss of generality, let us assume that $k_a \leq k_x$. We will show that x

must be either in P_1 or in P_2. Notice that k_a represents a separator, i.e. a node in the associated bintree, say a, having a path $\pi(a)$ from the root. Of course, $\pi(a) \leq k_a$. Two cases are possible: $k_a < k_x$ or $k_a = k_x$.

The former case is trivial. In fact, if $k_a < k_x$, then in a preorder visit, a must precede x, i.e., $a \prec x$, from which it follows that x must be in P_2.

Let us now analyze the latter case, i.e., $k_a = k_x$. Remember that $\pi(a)$ is the path to the first node stored in P_2. To establish the thesis, we have to prove that x cannot be stored in any page preceding P_1. We start by noting that k_x does not only represent the sequence $\pi(x)$, but also all the sequences of the following set:

$$S = \{\sigma \in \{0,1\}^s | \sigma = \pi_1 \dots \pi_r \pi_{r+1} \dots \pi_s, \pi_r = 1, \pi_{r+1} = \dots = \pi_s = 0, r \leq s \leq \ell\}.$$

Notice that $|S| = \ell - r \leq \ell$ and that $\pi(a), \pi(x) \in S$. If x is stored in a page preceding P_1, then for any node y stored in P_1, it will be $x \prec y \prec a$, from which it follows that $\pi(y) \in S$. This means, all the nodes in P_1 have a path belonging to S. But this is a contradiction, since P_1 contains at least ℓ nodes and $|S \setminus x| \leq \ell - 1$. □

The above result guarantees that the only delicate case to be managed is when the returned key from the searching in the B$^+$-tree equals the key we are looking for. In this case, we will load in main memory both the pages pointed by such a key, thus performing an extra access on secondary memory. This scenario is quite unlike to happen, and therefore we conclude that our approach works well for all practical purposes.

We finally remark that we choose in our design of the S*-tree to eliminate the linear prefix from the pages, since it can easily be recomputed from the separators in the B$^+$-tree. This will add a small overhead in terms of CPU time, but, on the other hand, we will reduce the space occupancy and simplify the standard B$^+$-tree merging operation: in fact, when two pages of the B$^+$-tree are merged together as a consequence of an underflow, the separator in the B$^+$-tree must be changed, and so for the linear prefix inside the page. This can produce a time expensive shifting of all the bits inside the page. Eliminating the linear prefix will eliminate this problem. The actual layout of a page of the S*-tree is given in Figure 2. Note that the free space will be at most k bits. The *tree pointer* points to the next available position in the linear tree stack, the *colour pointer* points to the next available position in the colour string stack while *next* is a pointer to the next page in the sequence set [3]. The field *length* stores the length of the separator.

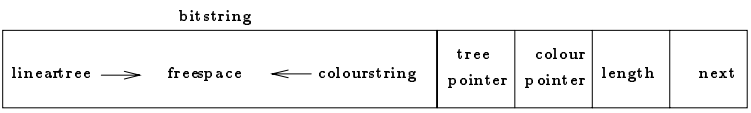

Fig. 2. Layout of a page of the S*-tree.

Figure 3 provides the complete B+-tree containing the bintree of Figure 1. Note that we set the size of the bitstring to 36 bits. With any external node we have associated a colour key: we encoded the white feature with '00', the light gray with '01', the dark grey with '10' and the black with '11'. For internal nodes, the associated colour string has 4 bits associated, from left to right, to white, light gray, dark gray and black. A bit in the colour string is set to 1 iff the associated feature is contained in the subimage individuated by the node. The two separators of the resulting three pages are 00001 and 001110, respectively. Thus, the second separator will be ambiguous, since its last digit is a 0. For example, looking for the node 00111 will retrieve the key 001110 from the B+-tree. As proved above, in this case we will not visit only the page following the retrieved key; instead, we will preliminary visit the page preceding the key: we compute the linear prefix by using the key 000010 and the length 5 stored in the page (thus the separator will be 00001 and the linear prefix will be 000001, since we codify a '0' with a '0' and a '1' with a '01'). Using the linear prefix, we are then able to retrieve the node 00111 as the last one of the second page (see [3] for details).

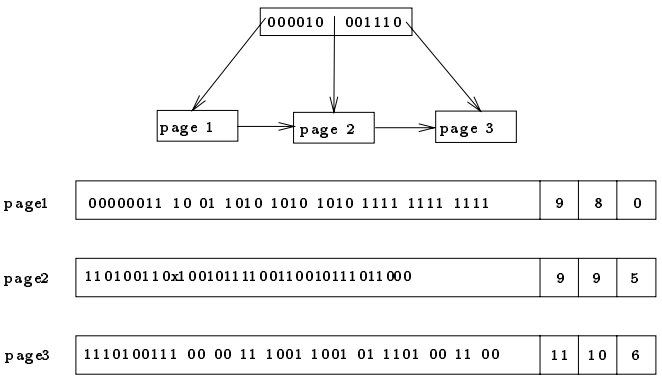

Fig. 3. The resulting B+-tree storing the S*-tree.

4 Experimental Results

In this section we present detailed experiments comparing the S*-tree with the HL-quadtree, since this latter spatial access method for coloured images has been shown to be very efficient with respect to other linear quadtree implementations [7]. We implemented both methods in C language and run the experiments on a SUN SPARC workstation with UNIX operating system. We executed the window queries on a set of images of size $2^{10} \times 2^{10}$ containing multiple non-overlapping features, ranging from satellite views to landuse maps. For the sake of brevity, we here present results for a set of 10 images containing 32 features and consisting of meteorological satellite views of North America. Figure 4 shows a sample image.

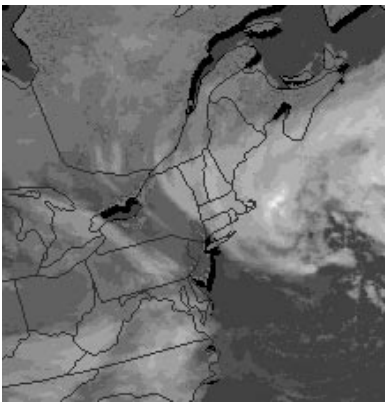

Fig. 4. A sample image (North America) containing 32 features.

We considered the following window queries, of primary importance for multiple non-overlapping features [2]:

- *exist*(*f,w*): Determine whether or not the feature f exists inside window w;
- *report*(w): Report the identity of all the features that exist inside window w;
- *select*(*f,w*): Report the locations of all occurrences of the feature f inside window w. This means to output all blocks homogeneous for feature f.

The basic approach to process all window queries was to decompose the query over a window into a sequence of smaller queries onto the maximal blocks contained inside the window [2]. Given a square window query of side n, these maximal blocks are $O(n)$ [4] and can be determined in linear time [8]. A detailed description of the algorithms used to process the queries can be found in [7]. For our purposes, it suffices to recall here that the *exist*(f, w) and the *report*(w) queries can be answered in $O(n \log_r T)$ I/O time, where r is the order of the B$^+$-tree, while the *select*(f, w) query can be answered in $O(n \log_r T + n^2/r)$ I/O time [7]. In our experiments, the page size of the storing B$^+$-tree was fixed to 512 bytes, while the order r has been fixed to 16.

4.1 Space Usage

A first comparison has been made on the space used by the HL-quadtree and the S*-tree. To this aim, we let the resolution of the images change from $2^8 \times 2^8$ to $2^{10} \times 2^{10}$. Figure 5 shows the results. From the drawing, it emerges that the S*-tree uses about 1/4 of the space used by the HL-quadtree. Therefore, the improving is substantial. This positively influences time performances, as we shall see in the next section.

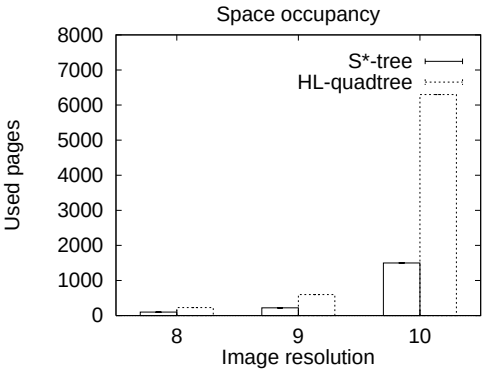

Fig. 5. Space occupancy comparison between the two methods.

4.2 Time Performances

To analyze the time performances of the HL-quadtree and the S*-tree, we used the classical measure of I/O complexity, that is, the number of disk accesses on secondary memory. The CPU time is indeed negligible with respect to the time spent in retrieving a page on secondary memory. In the following, we make the standard assumption that each secondary memory access transmits one page of data (a *bucket*), and we count this as one operation. We tested the two methods for each of the window queries and for each of the considered images. We randomly generated the anchor of 100 square windows of side $n = 50 \cdot i, i = 1, \ldots, 8$ (i.e., a total of 800 query windows). The windows have been "wrapped around" the image space whenever they extended beyond the borders of the image. Then, we computed the number of disk accesses for solving the queries using the two approaches, for each of the images, and we computed the arithmetic mean.

Concerning the *exist*(f, w) query, we focused on two different densities of feature distribution, namely features covering about 5% and 30% of the image space, respectively. We found such features for all the images, since the images were statistically similar. Figure 6 shows the results. From the drawings, we derive that in both cases there is approximately a 20% of saving in the number of accesses using the S*-tree. It is worth noting that as soon as the feature density increases, the I/O complexity decreases dramatically, and even though the theoretical I/O complexity depends on the window side, the query is solved in much less time. In fact, in case of high density, the probability of finding the feature after few accesses is very high.

Concerning the *report*(w) query, Figure 7(a) shows the results. Here, the I/O complexity of the two methods increases as soon as does the window side, since all the maximal blocks of the window need to be examined. In general, the S*-tree answers the query by doing about 20% less of the I/O accesses done by the HL-quadtree.

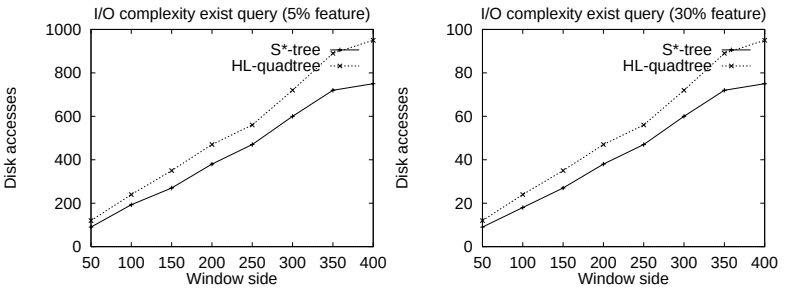

Fig. 6. Time performances of the two methods for the *exist*(f, w) query: (a) feature covering 5% of the image space; (b) feature covering 30% of the image space.

Finally, concerning the *select*(f, w) query, we focused on the feature covering about 30% of the image space. As for the *report*(w) query, the I/O complexity of the two methods increases as soon as does the window side, since also in this case all the maximal blocks of the window need to be examined. Furthermore, we here have a larger number of disk accesses, since to return all the occurrences of a feature we need to examine all the descendants of a node corresponding to a given maximal block. Figure 7(b) presents the results, showing that the S*-tree answers the query using about 70% of the I/O accesses used by the HL-quadtree. Therefore, we have up to 30% of time saving.

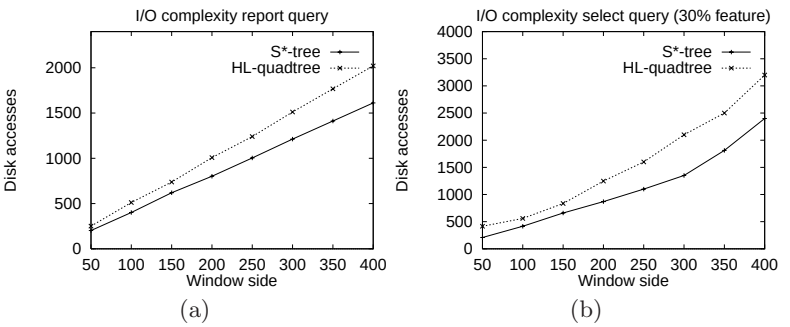

Fig. 7. Time performances of the two methods: (a) *report*(w) query; (b) *select*(f, w) query.

5 Conclusions

In this paper we have proposed and analyzed the S*-tree, a new time and space efficient disk-based representation of images containing multiple non-overlapping features. We used as time performance measure the number of secondary storage accesses for solving the classical window queries, and our experiments showed that the new approach outperforms a previous efficient spatial access method proposed in literature, namely the HL-quadtree [7]. More precisely, saving in time and space is about 20%.

Future work will be in the direction of an extension of this new encoding technique to the more general case of images containing multiple overlapping features. We also plan to test the S*-tree in performing other spatial operations.

References

1. W.G. Aref and H. Samet. A B+-tree structure for large quadtrees. *Computer Vision, Graphics and Image Processing,* 27(1):19–31, July 1984. 160
2. W.G. Aref and H. Samet. Efficient processing of window queries in the pyramid data In *Proc. of the 9th ACM-SIGMOD Symposium on Principles of Database Systems,* pages 265–272, Nashville, TN, 1990. 164, 164
3. W. de Jonge, P. Scheuermann, and A. Schijf. S+–trees: an efficient structure for the representation of large pictures. *Computer Vision, Graphics and Image Processing: Image Understanding,* 59(3):265–280, May 1994. 157, 160, 162, 163
4. C. Faloutsos, H.V. Jagadish, and Y. Manolopoulos. Analysis of the n-dimensional quadtree decomposition for arbitrary hyperectangles. *IEEE Transactions on Knowledge and Data Engineering,* 9(3):373–383, 1997. 164
5. I. Gargantini. An effective way to represent quadtrees. *IEEE Trans. on Pattern Analysis and Machine Intelligence,* 25(12):905–910, 1982. 157, 159
6. E. Kawaguchi, T. Endo, and M. Yokota. Depth-first expression viewed from digital picture processing. *IEEE Trans. on Pattern Analysis and Machine Intelligence,* pages 373–384, July 1983. 157, 159
7. E. Nardelli and G. Proietti. Time and space efficient secondary memory representation of quadtrees. *Information Systems,* 22(1):25–37, 1997. 156, 158, 161, 163, 164, 164, 167
8. G. Proietti. An optimal algorithm for decomposing a window into maximal quadtree blocks. *Acta Informatica, 1999.* To appear. 164
9. H. Samet. The quadtree and related hierarchical data structures. *Computing Surveys,* 16(2):187–260, June 1984. 157
10. H. Samet and R.E. Webber. A comparison of the space requirements of multidimensional quadtree-based file structures. *Visual Computer,* 5(6):349–359, December 1989. 159
11. C.A. Shaffer and P.R. Brown. A paging scheme for pointer-based quadtrees. In D. Abel and B.C. Ooi, editors, *Advances in Spatial Databases*, pages 89–104. Lecture Notes in Computer Science 692, Springer Verlag, 1993. 159
12. M. Tamminen. Encoding pixel trees. *Computer Vision, Graphics and Image Processing,* 2:174–196, 1984. 157
13. M. Vassilakopoulos and Y. Manolopoulos. Analytical comparison of two spatial data structures. *Information Systems,* 19(7):269–282, 1994. 159

14. M. Vassilakopoulos, Y. Manolopoulos, and K. Economou. Overlapping quadtrees for the representation of similar images. *Image and Vision Computing,* 11(5):257–262, 1993. 158