Pergamon

0306-4379(94)E0004-9

Information Systems Vol. 19, No. 4, pp. 33–54, 1994
Copyright© 1994 Elsevier Science Ltd
Printed in Great Britain. All rights reserved
0306-4379/94 $7.00 + 0.00

AN EFFICIENT SPATIAL ACCESS METHOD FOR SPATIAL IMAGES CONTAINING MULTIPLE NON-OVERLAPPING FEATURES

ENRICO NARDELLI[1] and GUIDO PROIETTI[1]

Dipartimento di Matematica Pura ed Applicata, Università di L'Aquila, Via Vetoio, 67010 L'Aquila, Italy and
Istituto di Analisi dei Sistemi e Informatica, CNR, Viale Manzoni 30, 00185 Roma, Italy.

(Received 18 June 1993; in final revised form 1 November 1993)

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). The S*-tree is based on a non-straightforward and space efficient extension to coloured images of its precursor, namely the S$^+$-tree, which was explicitly designed for binary images. To assess experimentally the qualities of the S*-tree, we test it against the HL-quadtree, a previous spatial access method for coloured images, which is known to be space and time efficient. Our experiments show that the S*-tree reaches up to a 75% of space saving, and performs constantly less I/O accesses than the HL-quadtree in solving classical window queries.

Key words: Spatial Data, Spatial Access Method, Bintree, Quadtree, Window Query.

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 spatial database management systems. 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 64), 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 into recursively nested subimages, until a homogeneous pattern is obtained. The most popular decomposition techniques are the *binary decomposition* (which splits the image into two equal parts alternating a 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* [13] and the *region quadtree* [10]. 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 subdivided 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 tree (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 leafcode and treecode 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 to 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 in a non-trivial way the capabilities of the S^+-tree to handle coloured images. We first show that for practical cases, the S^*-tree allows to save up to 25% of space with respect to a trivial extension of the S^+-tree, while performing asymptotically the same number of disk accesses to retrieve any given subset of the represented image. Furthermore, to assess the practical usefulness of our method, we compare it against the *HL-quadtree* [8], a space and time efficient spatial access method for coloured images, which combines advantages of leafcode and treecode representations by using locational codes to represent all the nodes of a region quadtree. Obtained results are extremely encouraging, showing a superiority of our method both in terms of space occupancy and time performances. More precisely, concerning the space occupancy, we show that the S^*-tree enjoys a 75% of space saving with respect to the HL-quadtree. Regarding the time complexity, we performed experiments over an important class of queries, namely the *window queries*, which constitute the basis of a number of operations that can be executed on coloured images. Since we are comparing time performances of secondary memory oriented data structures, we will use as efficiency measure the classical *I/O complexity*, by counting the number of accesses to the buckets storing the data. We will show that the S^*-tree performs constantly less I/O accesses than the HL-quadtree in solving the queries, saving up to 80% of time.

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, along with a description of the S^+-tree. In Section 3 we firstly present a straightforward extension of the S^+-tree to coloured images, and we then 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

In this section we present a survey of the various pixel tree (binary and quaternary) structures that have been proposed in the past for managing coloured images, along with a description of the S^+-tree. Table 1 contains main symbols used throughout the paper.

Symbol	Definition
T	Image space side
m	Image space resolution
k	Number of features
w	Query window
n	Query window side
r	B^+-tree order
f_i	i-th feature

Table 1: Symbol table

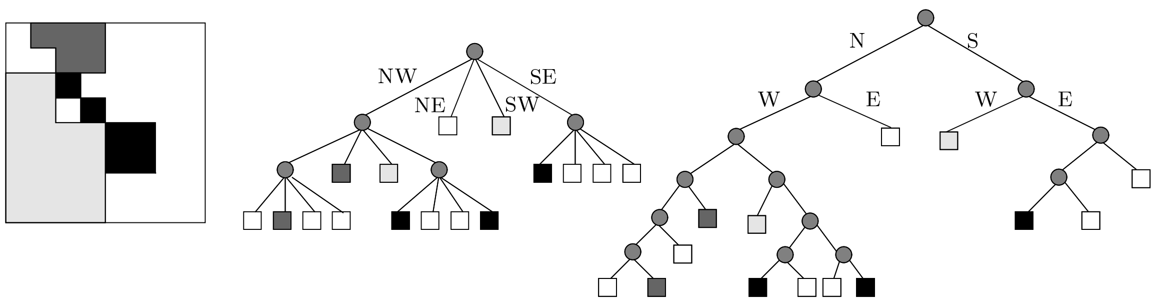

Fig. 1: Multiple non-overlapping features and their quadtree (left) and bintree (right).

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 $T \times T$ 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 extreme, the decomposition can go on up to the pixel level, with squares of side length 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 $T/2^d$. 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 [15].

The *bintree* is the binary version of the region quadtree: the image is progressively refined alternating horizontal and vertical splits, until a 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 either as a tree or as a list. In the former, direct access to specific image elements is privileged, while the latter makes sequential access easier and simplifies disk-based representations, absolutely needed for large amounts of spatial data [11, 12, 14].

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 defined for a bintree, can be easily adapted to handle a quadtree, and vice versa. There exist substantially two categories of secondary memory representation of a pixel tree: the collection of the leaf nodes (*leafcode representation*), and the linear list resulting from a preorder traversal of the tree (*treecode representation*).

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. A linear quadtree contains the collection of black leaves in the corresponding quadtree, encoded by means of a *locational key* (whose digits resemble the path in the tree from the root to the leaf) and indexed through a B$^+$-tree [1]. The locational key $\lambda(x)$ for a node x of level d in the quadtree is recursively defined as follows: Let the locational key for the root be an all-zero string of length m, and let x' be the parent of x in the quadtree. We have that $\lambda(x) = \lambda(x') + s \cdot 5^{m-d}$, where $s = 1$, 2, 3 or 4 if x is the NW, NE, SW or SE child of x', respectively. Then, the locational key is a base

5 code of length m, and requires $3m$ bits to be stored[†].

The extension to multiple non-overlapping features of a linear quadtree 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 consists of two fields: the locational key and the *feature value*, storing in $\lceil \log k \rceil$ bits 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 treecode representations, 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_i', $1 \leq i \leq k$, denoting a leaf nodes containing the i-th feature. The representing tree is visited in preorder, and an 'N' is emitted whenever an internal node is encountered, while an 'L_i' is emitted whenever a leaf node containing the i-th feature is encountered. As an example, suppose that the four features in Figure 1 have index 1 for the white, 2 for the light gray, 3 for the dark gray and 4 for the black. The following string is the DF-expression for the bintree in Figure 1:

$$\mathrm{NNNNNNL_1L_3L_1L_3NL_2NN\ L_4L_1NL_1L_4L_1NL_2NNL_4L_1L_1}.$$

Representing a pixel tree as a DF-expression is space efficient with respect to a leafcode representation, but accessing specific blocks is time-consuming, since indexing is not provided, and this is a serious drawback for window queries processing. Therefore, while an implementation based on B$^+$-trees for a linear quadtree representation is straightforward, this is not the case for a DF-expression.

2.3. The S$^+$-tree

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 implementation of binary images named *S$^+$-tree*. This was originally described by using the leafcodes generated by a bintree, though a quadtree could similarly be used.

The S$^+$-tree is obtained in two phases. In the first phase, we apply a preorder traversal on the bintree, emitting a '0' ('1') when an internal (leaf) node is encountered. The outcome will be a bitstring, named *linear (bin)tree*. Concurrently, during this traversal we store the colours of the leaves in an additional bitstring, called *colour table*, where a '0' ('1') represents a white (black) leaf. The two bitstrings thus obtained are named *S-tree*. In the second phase, the S$^+$-tree is built by storing the original tree into a list of data pages containing a segmented and augmented S-tree representation of the image. These data pages will be indexed by a B$^+$-tree[†]. This way, each data page constitutes a self-contained local S-tree that can be searched independently.

More specifically, a data page consists of a portion of the linear tree (growing from the beginning of the page) along with the corresponding portion of colour table (which grows from the tail of the page). The two bitstrings fill the page as much as possible, under the constraint that the last node stored in a page must always be a leaf (we will see later why this restriction is introduced). Therefore, due to such constraint, some unused space might be left. 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 defined 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 first node stored in the next 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, it follows from preorder visit

[†]In a bintree, the root is an all-zero string of length $2m$, and $\lambda(x) = \lambda(x') + s \cdot 3^{2m-d}$, where $s = 1$ or 2 if x is the left or right child of x', respectively. Then, the locational key is a base 3 code of length $2m$, and requires $4m$ bits to be stored.

[†]Notice that in their original paper [3], the authors use a prefix B-tree to index the data pages, but a B$^+$-tree provides similar performances.

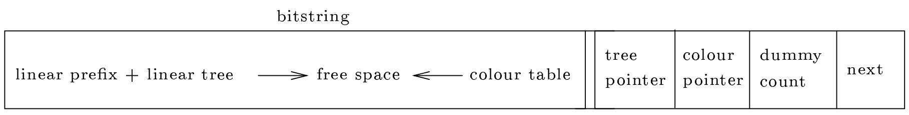

Fig. 2: The layout of a data page of the S^+-tree.

properties that the last bit of a separator is always a 1: In fact, if the last stored node is a left leaf, then the first node stored in the next page must be its right sibling, while if the last stored node is a right leaf, then the first node in the next page must be a right son of some of its ancestors. Such a property allows to store the separators using only $2m$ bits, without encoding the depth of the node the separator refers to which.

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. We should mention here that, as in all treecode representations, all nodes must be represented in the structure. The structure of an S^+-tree node can be seen in Figure 2. 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 table stack, *next* is a pointer to the next page in the sequence set, while *dummy count* indicates where the linear prefix ends and the linear tree starts.

Notice that building the S^+-tree by using a quadtree decomposition instead of a bintree, leads to a somewhat different creation of the separator. The path from the root of the quadtree to the node that caused the filling of the page is encoded by emitting a '0' when moving towards the first child (NW), and a '1', '2', or '3' when moving towards the second (NE), third (SW) or fourth (SE) child, respectively. Consequently, the linear prefix is built by encoding with a '0', a '01', a '011' and a '0111' a 0, 1, 2 and 3 in the separator, respectively.

This structure and the characteristics of the S^+-tree, in particular the property that each data page constitutes a self-contained local S-tree that can be searched independently, is its great advantage when used for window queries. As we have already mentioned, it provides for a very compact representation of the data and the index, while, concurrently, it behaves like B^+-trees and permits easy sequential and random access. As noted in [3], using the binary array representing the image as input, we can construct the corresponding S^+-tree in such a way that the pages of the sequence set are generated from left to right, which allows for almost 100% storage utilization of these data pages. Subsequent insertions and deletions will degrade the storage utilization, but only down to 69%, which is the typical storage utilization of B^+-trees.

3. THE S*-TREE

In this section, we firstly present a straightforward extension of the S^+-tree to coloured images, and we then present our new spatial access method, namely the S^*-tree.

3.1. A straightforward extension of the S^+-tree

The natural extension of the S^+-tree to coloured images is the following: concerning leaf nodes, the bit color of the colour table is replaced by a feature value of $\lceil \log k \rceil$ bits, as for the corresponding extension of the linear quadtree, while for internal nodes, we have to augment the colour table by associating with each internal node a *features string* of k bits, one bit for each feature, in which the i-th bit is set to 1 if and only if the node contains the i-th feature. In fact, associating a features string with internal nodes greatly improves the performances in executing several spatial operations [8].

However, such a straighforward extension has a severe drawback in terms of space utilization. In fact, as we described in the previous section, 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. 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 property is important, since it allows to store the separators using only $2m$ bits, without encoding the depth of the node the separator refers to which.

2. Since for binary images no information is associated with internal nodes (they are simply gray), we have at most $2m-1$ unused bits per page. Considering that a page size is generally 1 Kbyte, 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. Therefore, a large amount of information associated with internal nodes, that could potentially be stored in a page, might be shifted to the next one as a consequence of the above constraint, thus determining a large wasting in space. For instance, if the page size is 1 Kbyte, $m = 16$ and $k = 64$, the wasted space could be as large as $\frac{k}{8} \cdot (2m - 2) = 240$ bytes (this is the case when the next leaf that should be stored lies at the end of a path in the associated bintree of $2m - 2$ internal nodes[†] that have not yet been visited), i.e., about a 25% of the page size!

To make things concrete, Figure 3 provides what we should obtain from the image in Figure 1 by representing it using the trivial extension of the S$^+$-tree just described. For the sake of simplicity, we set to 36 bits the size of the bitstring; moreover, to improve readability, the linear tree has been underlined, and unused bits have been depicted with an 'x'. The length of a separator (i.e., a key in the B$^+$-tree index) is exactly $2m = 6$ bits. The features string of an internal node consists of $k = 4$ bits, corresponding, from left to right, to white, light gray, dark gray and black colour, respectively. On the other hand, with any external node, a feature value of $\lceil \log k \rceil = 2$ bits is associated: we encoded the white feature with '00', the light gray with '01', the dark grey with '10' and the black with '11'. Notice that the third page had enough space to store an additional internal node (i.e., the internal node corresponding to the rightmost nephew of the root), but due to the above constraint, we have to shift it to the next page, thus wasting 5 bits.

3.2. A space efficient extension of the S$^+$-tree: the S*-tree

From the above discussion, 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: 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 $2m - 2$), 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.*

[†]Remember that if the image resolution is m, then the height of the associated bintree is $2m$, where it is assumed that a single node is a tree of height 1.

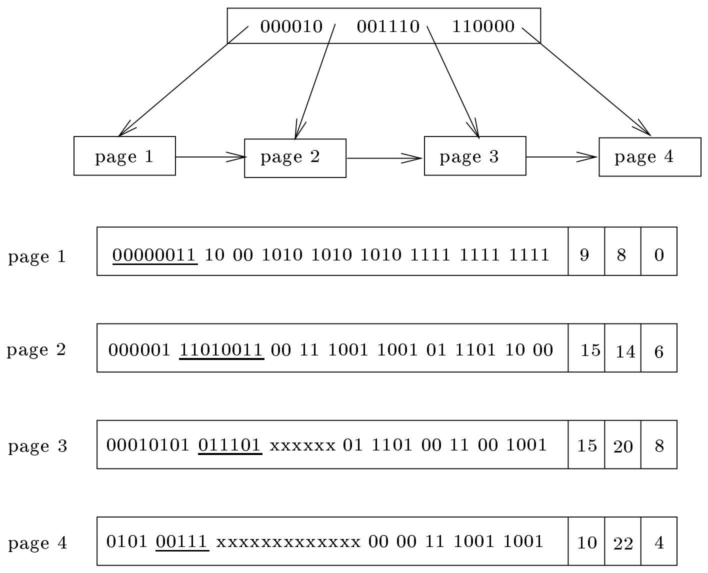
000010
001110
110000
page 1
page 2
page 3
page 4
page 1 | 00000011 10 00 1010 1010 1010 1111 1111 1111 | 9 | 8 | 0
page 2 | 000001 11010011 00 11 1001 1001 01 1101 10 00 | 15 | 14 | 6
page 3 | 00010101 011101 xxxxxx 01 1101 00 11 00 1001 | 15 | 20 | 8
page 4 | 0101 00111 xxxxxxxxxxxxx 00 00 11 1001 1001 | 10 | 22 | 4

Fig. 3: The B$^+$-tree storing the image in Figure 1, as obtained by the trivial extension of the S$^+$-tree.

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 $k = 64$, it suffices to fix the page size of the B$^+$-tree to 256 bytes.

Let $\pi_i \in \{0,1\}$, $i \leq \ell$ be the i-th bit of $\pi(x)$ and let π_r be the rightmost 1 of $\pi(x)$. We can therefore write $\pi(x) = \pi_1 \ldots \pi_r \pi_{r+1} \ldots \pi_t$, with $\pi_{r+1} = \ldots = \pi_t = 0$. To retrieve x, we will search in the B$^+$-tree for the key $k_x = \pi_1 \ldots\ldots \pi_r \pi_{r+1} \ldots \pi_\ell$, with $\pi_{r+1} = \ldots = \pi_\ell = 0$. 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 precedes x (we write it as $a \prec x$), 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:

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

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 critical case to be managed is when the key returned 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 unlikely 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, will reduce the space occupancy

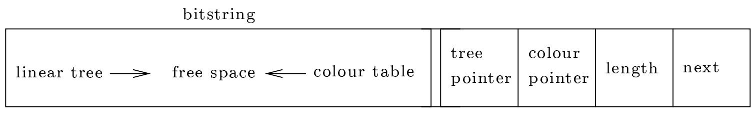

Fig. 4: Layout of a page of the S*-tree.

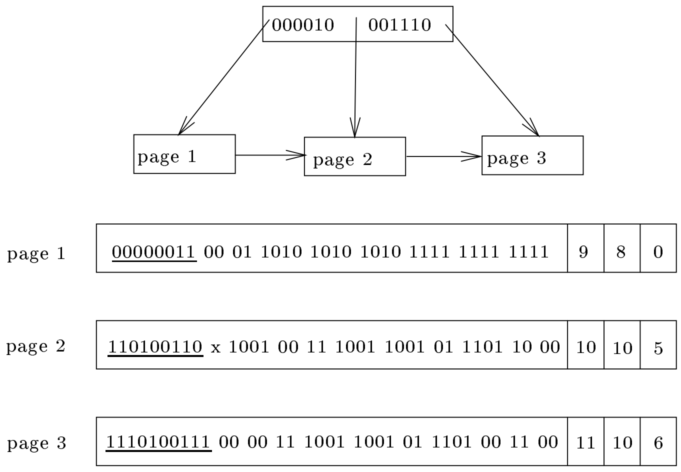
000010
001110
page 1
page 2
page 3
page 1 | 00000011 00 01 1010 1010 1010 1111 1111 1111 | 9 | 8 | 0
page 2 | 110100110 x 1001 00 11 1001 1001 01 1101 10 00 | 10 | 10 | 5
page 3 | 1110100111 00 00 11 1001 1001 01 1101 00 11 00 | 11 | 10 | 6

Fig. 5: The B$^+$-tree storing the S*-tree associated with the image in Figure 1.

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 4. Note that the free space will be at most k bits (i.e., the length of a features string). Notice that the space occupied by the field *dummy count* in the S$^+$-tree has been replaced by the field *length*, which stores the length of the separator associated with the page.

Figure 5 provides the S*-tree representing the image in Figure 1, by maintaining the same notation as for the trivial extension of the S$^+$-tree of Figure 3. It is worth noting that the second page is now completely filled, thanks to the fact that an internal node can be the last stored one (i.e., the internal node corresponding to the path '00111'). This allows us (along with the removal of the linear prefix) to store the image by using only 3 pages, instead of the 4 pages previously needed. Notice that 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 visit not only the page following the retrieved key; instead, we will preliminarily 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 on this latter operation).

4. EXPERIMENTAL RESULTS

In this section we present detailed experiments comparing the S*-tree with the *hybrid linear quadtree* (shortly, *HL-quadtree*), which has been shown to be very efficient with respect to other

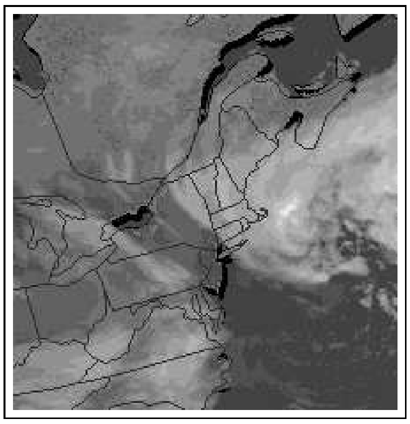

Fig. 6: A sample 512×512 meteorological image (North America) containing 64 features.

linear quadtree based representations of coloured images [8].

We recall that the main idea of the HL-quadtree is to represent both non-leaf and leaf nodes of the quadtree (like in the DF-expression), by coding them using a locational key (like in the linear quadtree). The result is a linear list containing all the nodes in the quadtree, which is then indexed through a B^+-tree. As for the S^*-tree, in the HL-quadtree we distinguish between records associated with non-leaf and leaf nodes. This is because a non-leaf node can contain more than one feature, and then it needs to store several features indexes. Then, we associate with such a record, along with the locational key defined as in the case of the linear quadtree, a features string of size k. Concerning records associated with leaf nodes, they have the same structure as for the linear quadtree, consisting of the two fields locational key and feature value. Finally, to distinguish between records associated with non-leaf and with leaf nodes, an additional bitfield called *leaf bit* is provided, whose value is 1 if and only if the associated node is a leaf.

We executed the window queries on a set of images containing multiple non-overlapping features, ranging from satellite views to landuse maps. More specifically we experimented with 3 groups of data. The first group of images, of size 256×256, was downloaded from the GRASS site, a public domain geographical information system[†], while the second and third group of images, of size 512×512 and 1024×1024, respectively, were meteorological satellite views of European, Asian and North American regions. Specifically, the second group was from the Meteosat Imagery site[‡], while the third was from the weather forecasts section of the CNN site[§]. Figure 6 shows a sample image.

Both the structures were implemented in C^{++} programming language under Windows NT, and the experiments run on a Pentium II workstation.

4.1. Window Queries

We considered the following window queries, of primary importance for multiple non-overlapping features [7]:

- *exist*$(w, f_{i_1}, f_{i_2}, \ldots, f_{i_h})$: check whether or not at least one of the features $f_{i_1}, f_{i_2}, \ldots, f_{i_h}$, $1 \le i_j \le k, j = 1, \ldots, h$, exists inside the window w.
- *report*(w): report all the features that are found inside the window w.
- *select*$(w, f_{i_1}, f_{i_2}, \ldots, f_{i_h})$: select all homogeneous blocks inside the window w containing the features $f_{i_1}, f_{i_2}, \ldots, f_{i_h}, 1 \le i_j \le k, j = 1, \ldots, h$.

[†]Available at `http://moon.cecer.army.mil`.
[‡]Available at `http://www.nottingham.ac.uk/~cczsteve/graphif.shtml`.
[§]Available at `http://cnn.com/WEATHER/images.html`.

The basic approach for answering the queries is that of decomposing the window query into a sequence of smaller queries, where each smaller query comprises a *maximal block* of the image space inside the window [2]. Without loss of generality, let us assume that the window w is a square of side n. We solve the query by initially decomposing in optimal time the window into its constituting maximal blocks [9], with respect to both a bintree and a quadtree decomposition process, and we associate with each maximal block x its respective node path $\pi(x)$ and locational key $\lambda(x)$, for the bintree and the quadtree decomposition, respectively. The list of node paths (locational keys) thus obtained is then used for searching in the S*-tree (HL-quadtree) to solve the queries. In the following, it is explained how these queries proceed according to the proposed methods.

Exist Query:

Consider a query over a specified window, where a search for existence of features $f_{i_1}, f_{i_2}, \ldots, f_{i_h}$ has to be performed. For each maximal block x in w, corresponding to a node path $\pi(x)$ (locational key $\lambda(x)$) in the representing S*-tree (HL-quadtree), searching starts from the root of the associated B$^+$-trees, and stops only when the leaf level is reached.

Concerning the HL-quadtree, it is certain that either x or a homogeneous ancestor of it will be located, since all quadrants are stored. Hence, we can process the maximal block by simply looking to the content of the corresponding feature field, with at most an additional access on the previous page to locate the ancestor, if needed. Regarding the S*-tree, the situation is similar. In this case, reaching the leaf level means that we reached one of the S*-tree pages, namely we reached part of the corresponding bintree. Three situations might arise for the searched maximal block:

1. it is a leaf in the corresponding bintree, and therefore we can immediately find its colour from the colour table;
2. it is contained in a leaf, and therefore we can find its colour by looking to its ancestor, with at most an additional access on the previous page;
3. it is an internal (i.e., non-homogeneous) node, and therefore we can immediately find all the contained features from the colour table.

Notice that in both the cases, the query ends either as soon as one of the queried features is found in w, in which case the answer is positive, or when all the maximal blocks in w have been examined and none of the queried features appeared, in which case the answer is negative. Since the number of maximal blocks inside w is $O(n)$ [4], it follows that by applying the above procedures, the exist query can be answered, both for the S*-tree and the HL-quadtree, in $O(n \log_r T)$ I/O time, where r is the order of the B$^+$-tree [8].

Report Query:

In a report query, the user asks for all the features comprised by the queried window. The query is answered similarly to the exist query in both the methods, but now the query ends only after all the maximal blocks in w have been examined, and the answer is a (possibly empty) set of features. Therefore, the report query can be answered in $O(n \log_r T)$ I/O time as well [8], both for the S*-tree and the HL-quadtree.

Select Query:

The last window query is the select query, where the user asks for the blocks of the map inside the queried window which are homogeneous with respect to the queried features. As in the case of the exist query, for each maximal block, searching starts by examining the entries at the B$^+$-tree root, and proceeds similarly in the S*-tree and in the HL-quadtree. Once the leaf level is reached, we search for the current maximal block. As described for the exist query, if this searching is not successful, then we try to see if a homogeneous ancestor exists in the B$^+$-tree (in such a case we output the searched maximal block if the ancestor is homogeneous with respect to one of the queried features). On the contrary, if the search is successful, two cases are possible:

1. the maximal block is a leaf: in this case, if it is homogeneous with respect to one of the queried features, we return it;
2. the maximal block is an internal node: in this case we look to all its descendants, returning those that are homogeneous with respect to one of the queried features.

Notice that the query ends only after all the maximal blocks in w have been examined. It can be shown that by applying the above procedure, the select query can be answered in $O(n \log_r T + n^2/r)$ I/O time [8], both for the S*-tree and the HL-quadtree.

4.2. Space occupancy

In the first set of experiments we measured the space usage that was involved in the two methods. More precisely, for each class of images (i.e., for each image size), we averaged the number of pages used. Figure 7 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 will positively influence time performances for solving the queries, as we shall see in the next section.

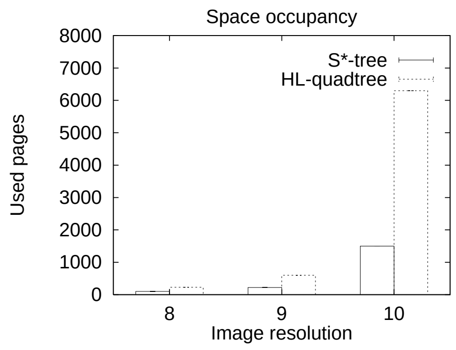

Fig. 7: Space occupancy comparison between the two methods.

4.3. 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. The window queries were performed on images of size 256×256, 512×512 and 1024×1024, containing 8, 16 and 64 features. The query windows sides were 1, 5, 10 and 25% of the image width. We randomly generated the anchor of the query windows, "wrapping around" the image space whenever a window extended beyond the borders of the image. The page size used was 1K for smaller images and 2K for larger ones, leading to a fanout of 84 and 169 entries, respectively. For each image, 50 queries were performed for the four different window sizes and the results were averaged. To eliminate the repeated traversal of B$^+$-tree nodes, we kept in main memory the root of the B$^+$-tree and we made use of buffering techniques. Due to space limitations, we only show the results for the 1024×1024 images containing 64 features, since the results are similar for all cases.

The selection of the queried features for the exist and the select query was based on their frequencies. Suppose that h features are to be selected out of k ones. First, we sort the features according to decreasing frequency and, then, we select the first, the $\lfloor \frac{k}{h} \rfloor$-th, the $\lfloor \frac{2k}{h} \rfloor$-th, ..., and the $\left\lfloor \frac{(h-1)k}{h} \right\rfloor$-th feature. For instance, if h=4 and k=64, then we select the first, the 16th, the 32nd and the 48th feature.

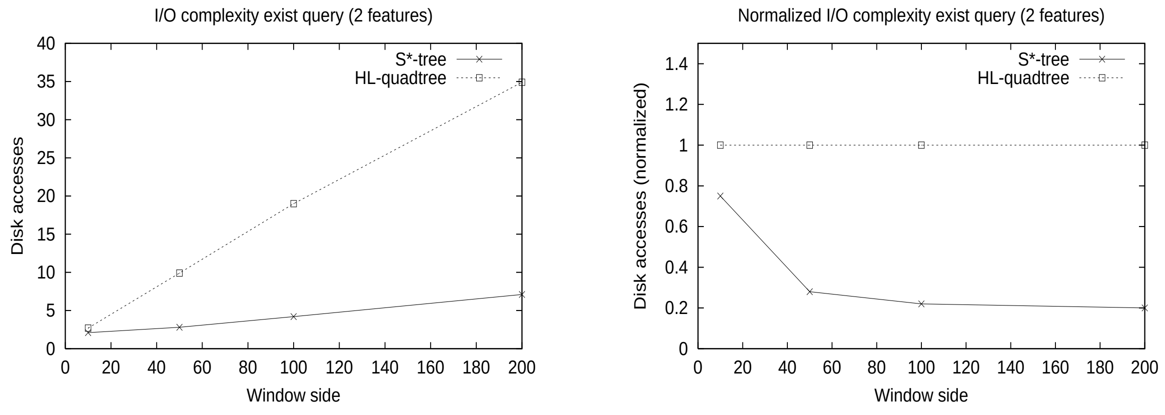

Fig. 8: Exist query where 2 features were queried, image size 1024×1024, 64 features: (left) averaged results, (right) normalized results.

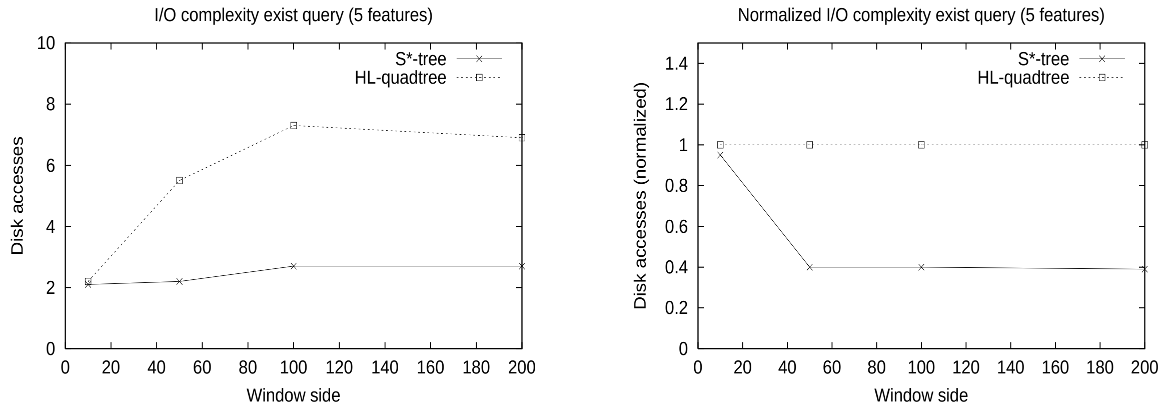

Fig. 9: Exist query where 5 features were queried, image size 1024×1024, 64 features: (left) averaged results, (right) normalized results.

4.3.1. Exist Query

In the experiments performed, we searched for the existence of a varying number of features. More specifically, we initially queried for 2, 5 and 10 features, and results can be seen in Figures 8, 9 and 10. The left side of each figure provides the obtained values, while the right side depicts the normalized results with respect to the worst method. From these figures it is easy to realize that the S*-tree outperforms the HL-quadtree, showing almost a constant behavior independent of the number of features searched, while on the contrary the HL-quadtree degrades as soon as this number decreases. Our interpretation of these results is that the S*-tree, apart from possibly creating a shallower B$^+$-tree, also takes advantage from the buffering techniques we have used, since each page contains much more blocks than a page of the HL-quadtree, and then it can be used several times during the query processing, without additional accesses on secondary memory. Notice that, despite of the worst case theoretical analysis, both methods do not suffer of the window enlargement, since the response to the query is generally positive, and the searched features are found rapidly in the window.

Afterwards, we experimented by fixing the window side (i.e., 100), while increasing the number of queried features. The results depicted in Figure 11 show that in this case the methods exhibit roughly the same performances, and both of them tend to answer the query in a single descent of the B$^+$-tree, as soon as the number of queried features increases.

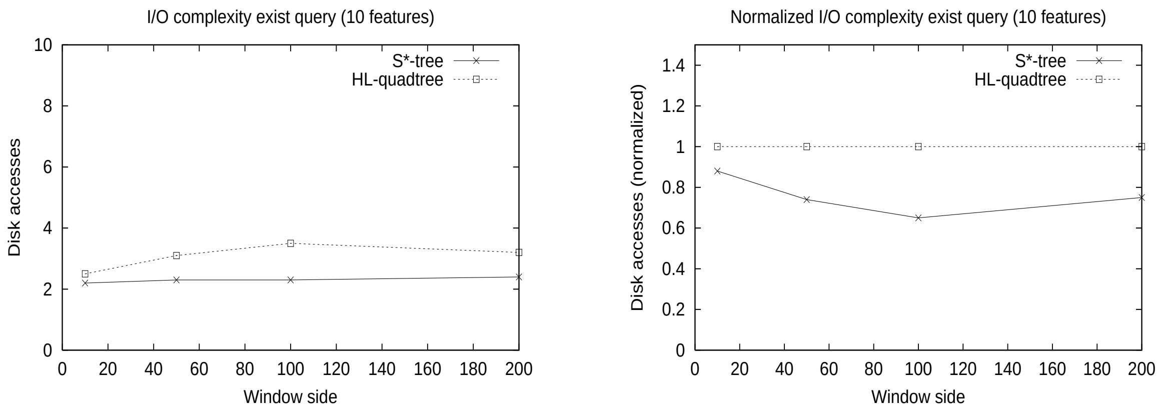

Fig. 10: Exist query where 10 features were queried, image size 1024×1024, 64 features: (left) averaged results, (right) normalized results.

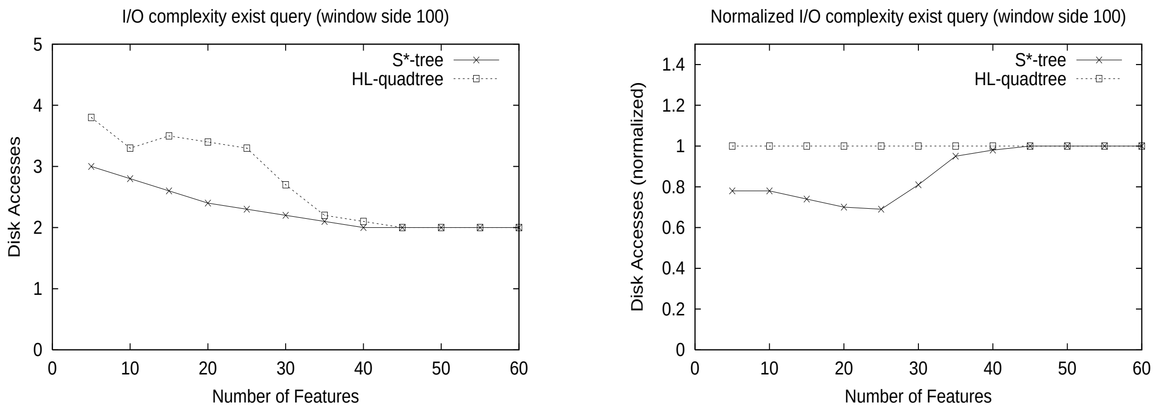

Fig. 11: Exist query for a varying number of queried features, image size 1024×1024, 64 features, query window 100×100: (left) averaged results, (right) normalized results.

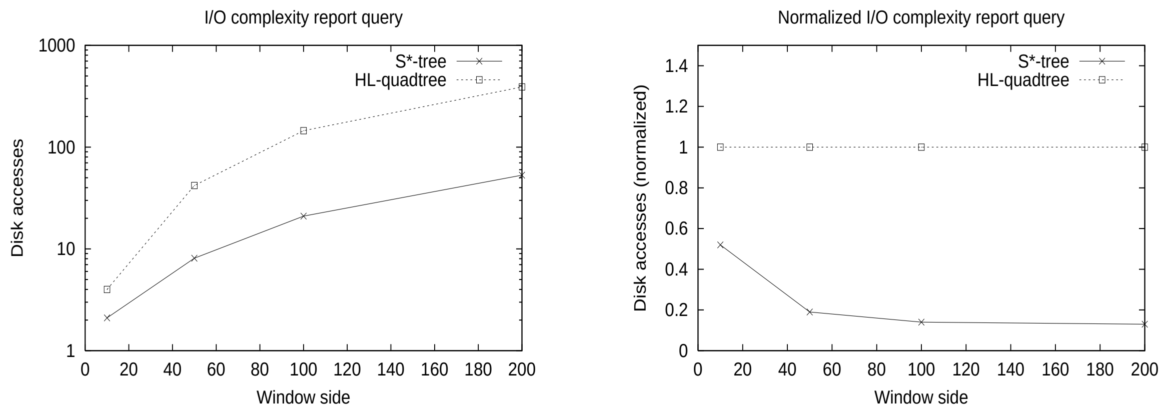

Fig. 12: Report query on images of 1024×1024 size containing 64 features: (left) averaged results, (right) normalized averaged results.

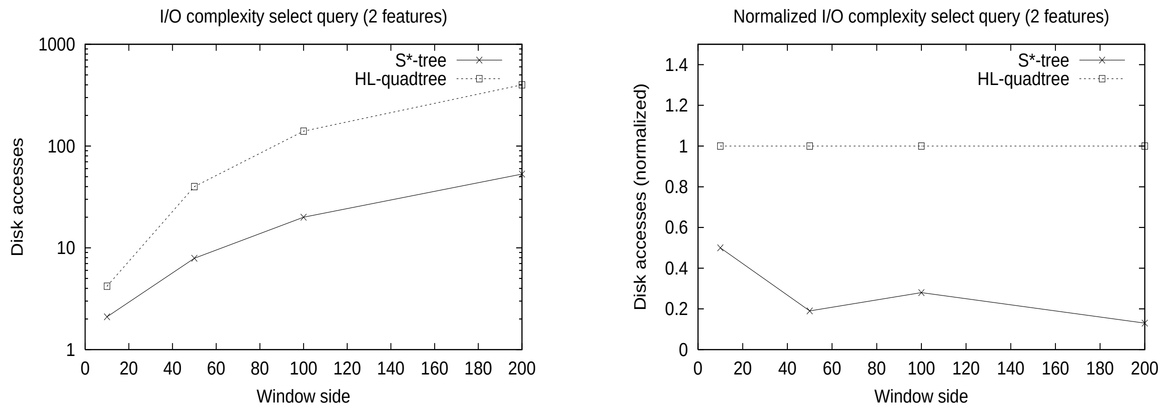

Fig. 13: Select query where 2 features were queried, image size 1024×1024, 64 features: (left) averaged results, (right) normalized results.

4.3.2. Report Query

Concerning the report query, results can be seen in Figure 12, where the left side reports in a log-linear diagram the obtained disk accesses for the two methods, while the right side contains results after normalization. From this figure it is easy to realize that the S*-tree outperforms the HL-quadtree. Notice that for both methods, the number of accesses is proportional to the window side, as expected from the theoretical analysis, but once again the S*-tree takes advantage of its space compactness.

4.3.3. Select Query

Regarding the select query, in the first set of experiments, as for the exist query, we queried with 2, 5 and 10 features. This time, however, the number of accesses almost does not change when the number of features increases: in fact, since all the blocks homogeneous with respect to the queried features must be returned, it follows that the overall number of accesses will be dominated by the number of accesses performed for selecting with respect to the most frequent feature. This phenomena can be observed in Figures 13, 14 and 15. Again, on the right side of the graphs the normalized results with respect to the worst method are depicted. Notice that once again, the S*-tree shows constantly the best behavior.

Finally, we experimented by fixing the window side (i.e., 100), while increasing the number of queried features. Notice that the number of accesses does not change when the number of

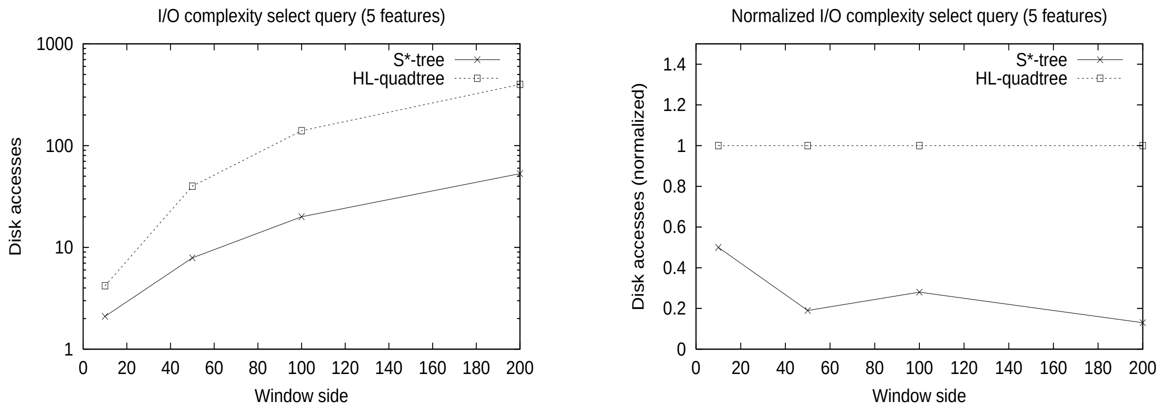

Fig. 14: Select query where 5 features were queried, image size 1024×1024, 64 features: (left) averaged results, (right) normalized results.

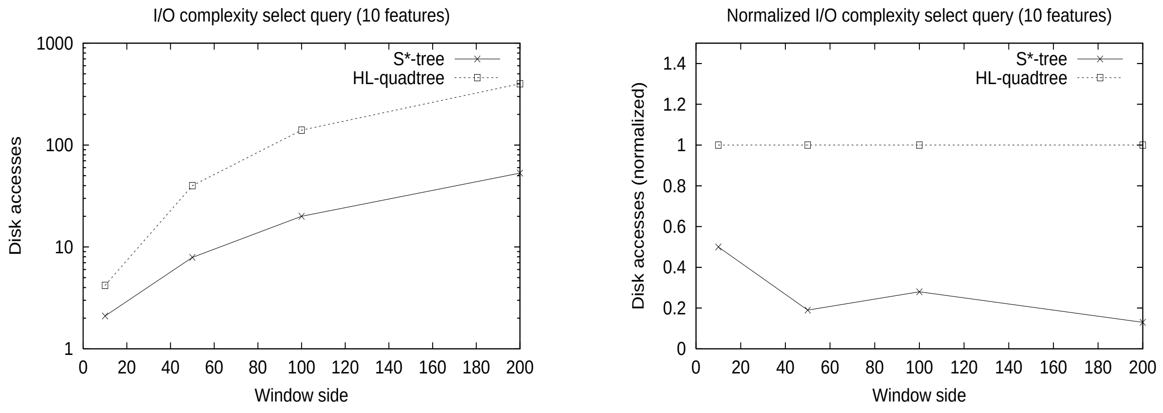

Fig. 15: Select query where 10 features were queried, image size 1024×1024, 64 features: (left) averaged results, (right) normalized results.

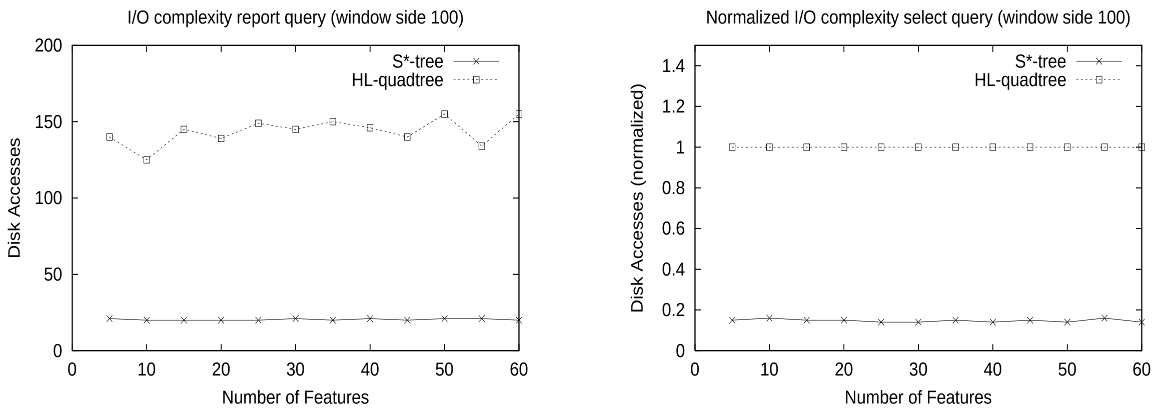

Fig. 16: Select query for a varying number of queried features, image size 1024×1024, window size 100×100: (left) averaged results, (right) normalized results.

queried features increases, as explained above. Moreover, the S*-tree performs roughly a fifth of the accesses made by the HL-quadtree. The results are shown in Figure 16.

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 [8]. More precisely, saving in time and space can reach up to 80%.

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, like for instance the classical spatial join.

Acknowledgements — The authors would like to thank Eleni Tousidou for her contribution in carrying out the experiments presented in the paper.

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 (1984).

[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*, pp. 265–272, Nashville, TN (1990).

[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 (1994).

[4] C.R. Dyer. The space efficiency of quadtrees. *Computer Graphics and Image Processing*, **19**(4):335–348 (1982).

[5] I. Gargantini. An effective way to represent quadtrees. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, **25**(12):905–910 (1982).

[6] E. Kawaguchi, T. Endo, and M. Yokota. Depth-first expression viewed from digital picture processing. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, pp. 373–384 (1983).

[7] Y. Manolopoulos, E. Nardelli, G. Proietti, and E. Tousidou. A generalized comparison of linear representations of thematic layers. Accepted for publication in *Data and Knowledge Engineering* (2000).

[8] E. Nardelli and G. Proietti. Time and space efficient secondary memory representation of quadtrees. *Information Systems*, **22**(1):25–37 (1997).

[9] G. Proietti. An optimal algorithm for decomposing a window into maximal quadtree blocks. *Acta Informatica*, **36**(4):257–266 (1999).

[10] H. Samet. The quadtree and related hierarchical data structures. *Computing Surveys*, **16**(2):187–260 (1984).

[11] H. Samet and R.E. Webber. A comparison of the space requirements of multi-dimensional quadtree-based file structures. *Visual Computer*, **5**(6):349–359 (1989).

[12] 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*, pp. 89–104. Lecture Notes in Computer Science 692, Springer Verlag (1993).

[13] M. Tamminen. Encoding pixel trees. *Computer Vision, Graphics and Image Processing*, **2**:174–196 (1984).

[14] M. Vassilakopoulos and Y. Manolopoulos. Analytical comparison of two spatial data structures. *Information Systems*, **19**(7):269–282 (1994).

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