Loading Spatial Features into the Incomplete Pyramid Data Structure

Walid G. Aref
Hanan Samet

Spatial data usually is stored in images where spatial features are classified using image processing techniques. However, for efficient processing, spatial features must be loaded into appropriate in-core spatial data structures. The pyramid data structure is of particular interest because it can handle both overlapping and non-overlapping features. The incomplete pyramid, a variant of the pyramid, has several advantages over the pyramid in the context of spatial processing and answering queries on spatial features. Algorithms are presented for loading overlapping and non-overlapping features into the incomplete pyramid from a number of disk-based pointerless quadtree feature representations. The running time for these algorithms is linearly proportional to the perimeter of the spatial feature being loaded and is independent of the pyramid depth.

1 Introduction

The large volume of spatial data imposes the need to store spatial features in disk files. Many disk-based data representations exist for storing spatial data. However, for efficient processing and query answering spatial data has to be, partially or if possible totally, loaded into suitable in-core spatial data structures.

The pyramid is proposed as one in-core representation for spatial features because its hierarchical and multi-resolution nature facilitates the performance of a number of useful operations. Nevertheless, spatial data is often represented using non-pyramidal methods. One reason is that most images (a common form of spatial data) are traditionally represented by use of methods such as binary arrays, runlength codes, boundary codes, or polygons (i.e., vectors). Other reasons include the nature of the application (e.g., runlength codes are particularly useful for raster-like display devices). Also, from an implementation perspective, some of these representations are pointer-based and hence are more suitable as main memory structures while other representations are pointerless and hence can be useful as disk-based representations. Therefore, techniques that can switch efficiently between these representations are needed. This paper discusses a number of such techniques and presents algorithms for some of them.

Our presentation is primarily in the context of loading variants of pointerless quadtree representations (a disk-based representation) into the pyramid data structure (a main memory representation). The quadtree is a refinement of the array representation of an image that attempts to save storage by taking advantage of some regularity in the image by decomposing the array into homogeneous disjoint two-dimensional blocks centered at predetermined positions. Image elements are now two-dimensional blocks instead of pixels. The quadtree can be implemented as a list or as a tree. The list representation is a pointerless representation that facilitates sequential access but is inefficient for random access to specific image elements. It is more suitable for disk-based storage. The tree representation is an attempt to balance the costs of sequential and random access.

In this paper, we provide a family of efficient algorithms for loading spatial features into a variant of the pyramid data structure, termed the *incomplete* pyramid, from several pointerless quadtree feature representations. In practice, pointerless quadtree representations are widely used for storing and manipulating spatial data (e.g., [1, 6, 12, 15]). The pointerless quadtree representation (in the form of lists) can further be grouped into two categories. The first treats the image as a collection of leaf nodes (e.g., the linear quadtree [3]). The second represents the image in the form of a traversal of the nodes of its quadtree (e.g., DF-expressions [4]). By virtue of being pointerless, these representations are suitable for disk-based storage of spatial data. In addition, they are relatively compact methods of aggregating regions with the same features or attributes. This is especially true when they are compared with the array representation. Finally, mapping feature raster data into one of these representations is well-known [7, 9, 11].

In previous work [2], we introduced the incomplete pyramid data structure as an in-core candidate for representing spatial features. The incomplete pyramid is a multiresolution data structure, in contrast to the region quadtree which is a variable resolution data structure. The incomplete pyramid is of particular interest because it can handle both overlapping and non-overlapping features. It also permits direct access of nodes at all levels. We make extensive use of this property. The incomplete pyramid is an efficient query answering tool [2]. We summarize these results in this paper as well.

The rest of the paper is organized as follows. In Section 2 we classify spatial features into two categories according to how they interfere in space (i.e., whether they overlap or not). In Section 3 we give a brief description of the incomplete pyramid data structure along with some of its important characteristics that we utilize in this paper. In Section 4 we review two appropriate pointerless quadtree representations for storing spatial data. The first is a DF-expression [4] (described in Section 4.1), and the second is a linear quadtree [3] (described in Section 4.2). Sections 5 and 6 present and analyze, respectively, algorithms for loading the incomplete pyramid by spatial features represented using these representations. The novel aspect of this work is that both overlapping and non-overlapping features are permitted. In addition, these algorithms make use of the fact that the incomplete pyramid is an in-core structure and hence parts of it (the pyramid nodes) can be randomly accessed. Thereby avoiding, whenever possible, the descent of the pyramid nodes in tree order.

2 Overlapping vs. Non-overlapping Features

We classify spatial features into two groups: overlapping and non-overlapping. Examples of overlapping features are cities, roads, lakes, and counties that comprise a given region. Examples of non-overlapping features are landuse maps where every land region has only one usage and is assigned a feature name to distinguish it from regions with different usages. Figures 1 and 2 show a collection of non-overlapping and overlapping features in a 16×16 feature space.

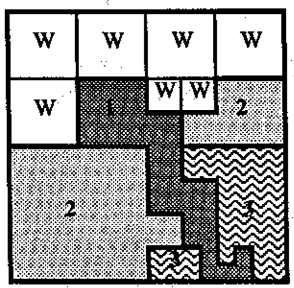

Figure 1: Three non-overlapping spatial features 1, 2, and 3 in a 16×16 feature space.

3 The Incomplete Pyramid Data Structure

The incomplete pyramid data structure was first introduced in [2]. It is an adaptation of the pyramid data structure [13]. This section is organized as follows. We first review the pyramid and briefly cover its use for spatial query processing. Next we show how to modify the pyramid to form the incomplete pyramid variant.

The pyramid is a multiresolution data structure. This is in contrast to variable-resolution hierarchical data structures such as the region quadtree [5, 8]. Also, the pyramid is used to store a collection of related layers each representing one overlay of the feature space. In this sense, the pyramid can be thought of as an aggregation of complete quadtrees where each quadtree represents one overlay. Features in one quadtree do not overlap in space. This is due to the definition of the region quadtree and its decomposition rules. On the other hand, features in the pyramid from the different spatial overlays may or may not overlap. We use Figures 3 and Figure 4 for illustration.

Figure 3 shows the region quadtree representations of two overlays. Each overlay contains one of the features given in Figures 2b and 2d. This is for illustration only. However, an overlay may contain more than one non-overlapping feature. Nodes in a region quadtree are either leaf or gray. Leaf nodes correspond to those blocks that are either entirely inside the feature region in the image (labeled *black*) or are entirely outside (labeled *white*). Each non-leaf node is said to be *gray* (i.e., its corresponding block is neither entirely inside nor entirely outside the feature region). The parent of a leaf node is a gray node (i.e., leaf nodes are maximal).

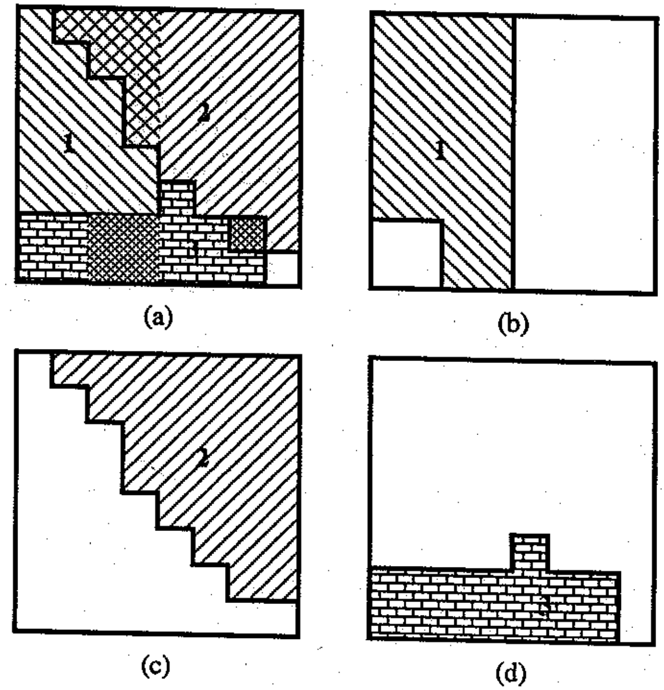

Figure 2: (a) the result of overlaying the three overlapping spatial features 1, 2, and 3 shown in (b), (c), and (d). The features lie in a 16×16 feature space. W denotes an empty region (i.e., white).

Figure 4 shows the pyramidal representation of the two overlays of the same region. Notice that, for example, feature 1 of overlay 1 (Figure 2b) and feature 3 of overlay 3 (Figure 2d) overlap and both are loaded into the pyramid. More formally, given a $2^d \times 2^d$ image, the pyramid is defined as a tree where the root node represents the whole image space, and then a recursive decomposition into quadrants is performed just as in quadtree construction, except that we keep subdividing until we reach the individual pixels. The leaf nodes of the resulting tree represent the pixels while the nodes immediately above the leaf nodes correspond to image blocks of size $2^{d-1} \times 2^{d-1}$. The non-leaf nodes are assigned a value that is a function of the nodes immediately below them (i.e., their four sons) such as the average gray level. For our purposes, this function is the Boolean OR set operation. In other words, we say that a certain feature is present in the parent node when this feature is present in at least one of its four sons.

Nodes in the pyramid have the following internal structure: each node has a fixed number m of feature bit-slots and summarizing variables that contain information about the descendants of the node. For now, we are interested in the feature bits.

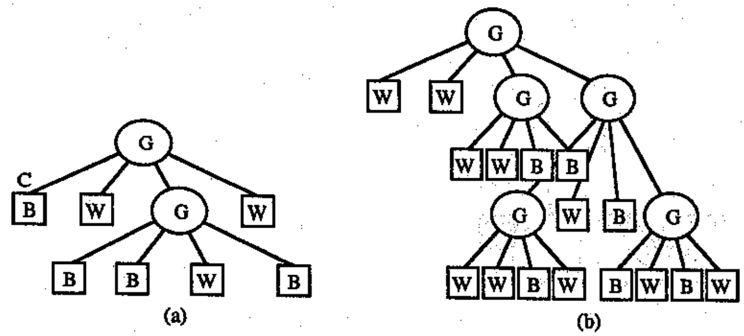

Figure 3: The region quadtree representation of two overlays: (a) the overlay containing the feature given in Figure 2b, (b) the overlay containing the feature given in Figure 2d.

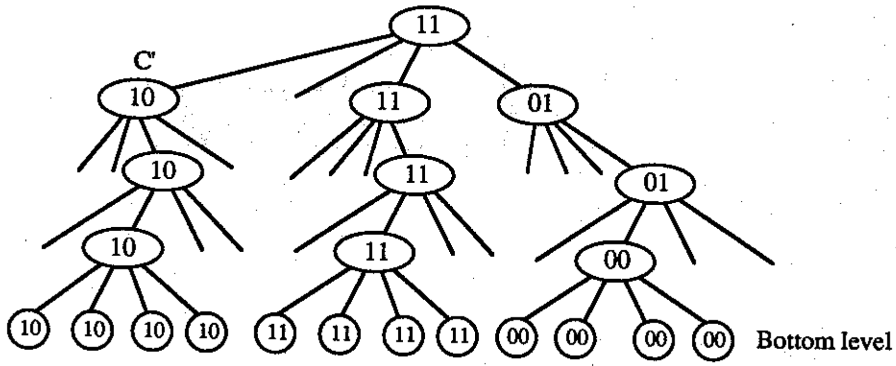

Figure 4: The pyramid representation of spatial features.

Each bit-slot corresponds to a feature stored in the pyramid. Each feature is encoded into its bit-slot through the whole pyramid independent of the features in the other bit-slots. In the following, we show how one feature is encoded into the pyramid. The same technique is applied for the remaining features. In this discussion, we use the term *node* to denote a one bit-slot.

As can be seen from Figures 3 and 4, the way spatial features are encoded into the nodes of the pyramid hides the information with respect to which nodes correspond to maximal blocks in the recursive decomposition. As an example, consider block C of Figure 3. C is a maximal quadtree block. Its feature is represented in the pyramid by the first bit-slot of pyramid node C' of Figure 4. In the pyramid, there is no specific information to indicate whether C' is a maximal block or not. This information, although not required, can speed-up processing time significantly. In many algorithms, when operating on the feature corresponding to the first bit-slot, processing can stop at node C' without a need to access the nodes of higher resolution below C'.

In an attempt to solve this problem, Shaffer and Samet [10] modify slightly the way spatial features are coded into the pyramid so that the information about maximal quadtree blocks of a given overlay is preserved inside the pyramid. Their suggested method is called the GL coding method. Figure 5 illustrates this coding method where the overlay of Figure 3a is represented in the pyramid data structure using the GL coding method. The GL method works as follows. If an intermediate

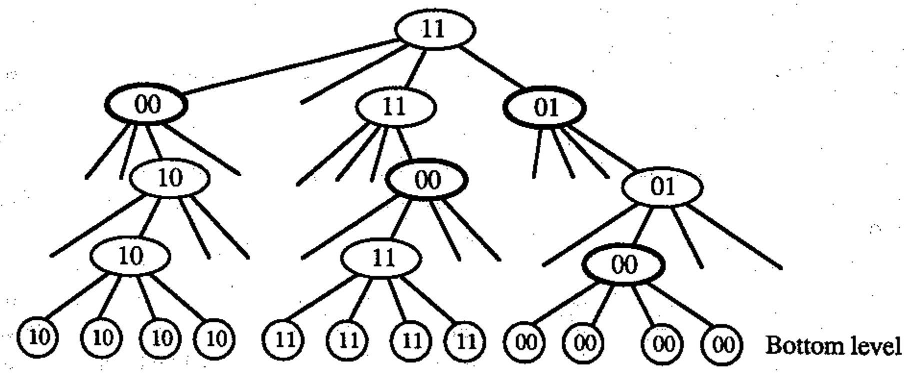

Figure 5: Example of GL coding. Nodes with thick border are leaf nodes for either the first feature or the second feature or both.

node is gray, then its bit slot value is 1. If an intermediate node is a leaf node, then its bit slot value is 0. Since it is necessary to determine the actual value of a leaf node (i.e., black or white), this value is stored in all of the descendants of the leaf node down to the pixel level (i.e., 1 and 0, respectively). If a bottom-level node is a leaf node, then its value is stored in the node itself since it has no descendants. Clearly, a gray node cannot appear in the bottom level.

Although it succeeds in locating maximal quadtree blocks inside the pyramid, the GL encoding method has two deficiencies. First, if pyramid nodes were to be accessed directly without descending from the root of the pyramid, it would be a difficult task (logarithmic) to know whether a given node is gray or black since both are coded with the value 1. Notice that some algorithms may make use of this capability if it exists (e.g., the selection operation presented in [2]). In order to distinguish between gray and black nodes using the GL coding method, one has to either descend or ascend the pyramid nodes and test whether a value 0 is encountered or not. The number of nodes to be accessed in this process of descent or ascent to distinguish between gray and black nodes is of the order of the depth of the pyramid. In addition to the fact that this is dependent on the pyramid depth and hence a log factor is to be introduced as a computational overhead, it also violates the essence of direct accessing the pyramid nodes and gaining the desired information.

The second deficiency in the GL method is that when loading one overlay containing some spatial features (e.g., lakes) into the pyramid data structure, eventually, all the pyramid nodes have to be accessed. This slows down the loading of spatial features significantly. This is especially true if we consider the fact that the pyramid contains m overlays (one overlay per bit-slot in a pyramid node) and each one has to be loaded independently. Therefore, saving the loading time of overlays is significant.

One way to overcome the two deficiencies of the GL method is by using the modified GL coding method suggested by Aref and Samet in [2]. The difference between the normal GL coding scheme and the modified scheme is that in the modified scheme when a leaf node (a 0 value) appears at an intermediate level, then all its descendants have a 0 value except for those nodes at the bottom level which are assigned the values 1 or 0 depending on whether they are black or white, respectively. If a leaf node appears at the bottom level then it is assigned the value 1 or 0 depending on whether it is black or white, respectively. Notice that using this coding scheme, we are able to overcome the deficiency that arises with normal GL coding. In particular, when nodes are accessed directly, we can distinguish between gray and leaf nodes. As an example, consider Figure 6. Assume that we want to access the second bit-slot of the pyramid only. When node A is accessed directly, we know it is a gray node since its value is 1, while node B is a leaf node since its value is 0. To determine if node B is black or white, we access node C directly at the bottom level of the pyramid which indicates that B's value is black (since C has a value of 1). In addition, in contrast to the GL method, in the process of loading

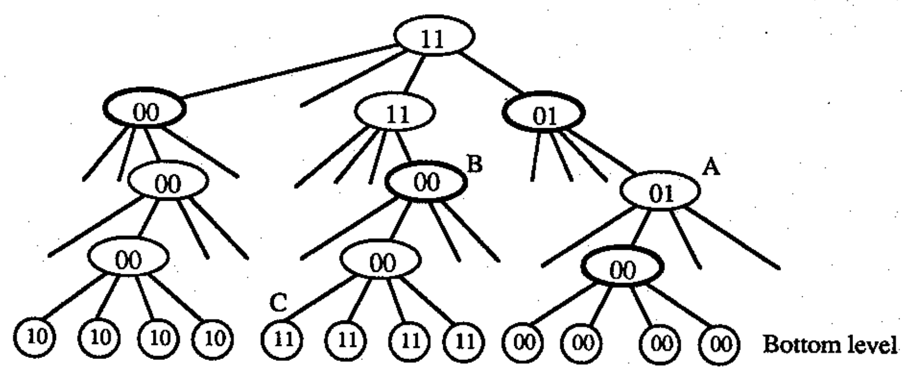

Figure 6: Example of modified GL coding. Nodes with thick border are leaf nodes for either the first feature or the second feature or both.

spatial overlays using the modified GL method, descending down the pyramid nodes can stop at maximal quadtree blocks without accessing the nodes below them, i.e., all the other nodes are left unchanged (reset to 0) except for the bottom level. The bottom level has to be loaded for both the GL and the modified GL methods. The bottom level of the pyramid contains the whole overlay at full resolution and can be loaded as a single entity by a method other than pyramid traversal. This is possible since, as we will see below, the pyramid is implemented as a stack of arrays and hence the bottom level of the pyramid has contiguous storage.

Combining the modified GL encoding of spatial features with the pyramid data structure yields what we term the *incomplete* pyramid. Since the encoding of a feature starts at the root of the pyramid and stops at a leaf node which can be at any level, the pattern of ones inside the pyramid looks like a partially filled pyramid as shown in Figure 7. Another illustration of the difference between the regular

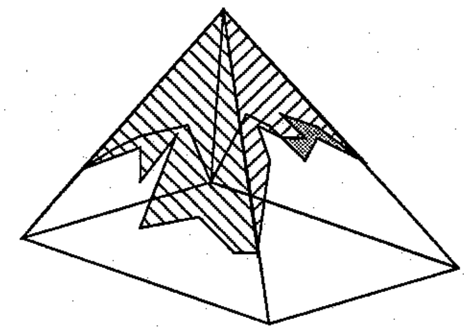

Figure 7: The incomplete pyramid where the shaded areas correspond to features represented by patterns of ones descending from the root down to leaf nodes.

pyramid and the incomplete pyramid is that the regular pyramid is an aggregation of complete quadtrees each representing a feature in the image space, while the incomplete pyramid is an aggregation of region quadtrees with some enhancements that improve the run-time performance.

The pyramid is implemented by a linear array in a heap-like fashion [16]. Each pyramid node is represented by an array entry. Given a parent node p with relative index j at level i, we can access the k^{th} son of p (where $k = 0, 1, 2,$ or 3) by the following formula:

$$son(i,j,k) = 4^i + 4 \times j + k$$

The number of array elements at levels 0 (the root level) through $i-1$ is $(4^i-1)/3$.

Also, there is a correspondence between the pyramid nodes and the spatial database blocks. Assume a coordinate system such that the origin is at the upper-left corner, x increases to the right, and y increases downwards. A block of size $2^i \times 2^i$ $(0 \leq i \leq d)$, having (r, c) as the coordinates of its upper-left corner $(0 \leq r, c \leq T-1)$ such that $r \bmod 2^i = c \bmod 2^i = 0$ corresponds to the pyramid node at level $d-i$ whose relative index within this level is equal to

$$(bit\_interleave(r,c)) \gg (2 \times i),$$

where $\gg$ is the shift-right operation. Thus $y \gg x$ means that y is shifted to the right by x bit positions.

The pyramid representation has an overhead of one third extra storage because of the need to store intermediate nodes. In other words, for a $T \times T$ image where $T = 2^d$, the entire pyramid requires $4 \times (T \times T - 1)/3$ nodes. This is the same amount of storage necessary for a complete quadtree [8] where every node of the quadtree has been expanded down to the pixel level.

Even though it is a space-redundant data structure, the incomplete pyramid has major benefits in its multiresolution nature as well as the run-time improvements that it permits. Many queries can be answered with the same accuracy at lower levels of resolution. Examples of these queries are window-based queries that include the exist query (i.e., determining whether or not a spatial feature exists inside a window), and the report query (i.e., reporting the identity of all the features that exist inside a window). In [2], we showed that these queries can be answered in $O(n \log\log T)$ time for a window of size $n \times n$ in a feature space (e.g., an image) of size $T \times T$ (e.g., pixel elements). The significance of this result is that even though the window contains n^2 pixel elements, the worst-case time complexity of the algorithms is almost linearly proportional (and not quadratic) to the window diameter, and does not depend on other factors. The techniques used to answer the report and exist queries can be used to answer other queries as well (e.g., the containment query).

Some Properties of the Incomplete-Pyramid

- **Property 1:** Direct Access
  The physical location of any node in the incomplete pyramid can be directly accessed in constant time [14]. The incomplete pyramid is implemented as a linear array with the root at location 0, nodes at depth 1 in locations 1 – 4, followed by the nodes at depths 2 through d (the maximum level). Given the values of the x and y coordinates of the upper-left corner of a block in the image and its size (i.e., $2^i \times 2^i$), we can compute the pyramid address of the 'corresponding' node, say Q, in constant time as follows:

$$pyr\_address(i,x,y) \quad = \quad (4^i - 1)/3 + (bit\_interleave(x,y) \gg 2i)$$

  The same formula applies for computing the incomplete pyramid address of a given node in constant time.

- **Property 2:** Parent and Son Addresses
  Given the address (or the index inside the incomplete pyramid array) of a node, we can generate the address of the node's parent, or any of its sons, in constant time.

- **Property 3:** Contents of a Node
  Given the address (or the index inside the incomplete pyramid array) of a node, we can retrieve the features contained or overlapping with this node's block in at most two direct accesses to the incomplete pyramid.

  Procedure *features*, given in Figure 8, shows how features stored in Q can be retrieved from the incomplete pyramid structure in constant time. Q can be at an intermediate or bottom level in the pyramid.


```
set_of_features procedure features(i, x, y);
begin
  value integer i, x, y;
  set_of_features s;
  global integer d;
  if(i = d) then /* bottom level*/
    s ← contents(pyr_address(d, x, y))
  else   /* features in the node or in one of its sons */
    s ← contents(pyr_address(i, x, y))
        ∪ contents(pyr_address(d, x, y));
  return (s);
end;
```

Figure 8: Retrieving the features stored in a node of the incomplete pyramid.

These properties along with other interesting properties are explained further in [2].

4 Pointerless Quadtree Representations

In this section, we review two pointerless quadtree feature representations: the DF-expression and the linear quadtree. The former is a linear list resulting from a preorder traversal of the active nodes in the incomplete pyramid data structure, while the latter is a collection of leaf nodes of a quadtree representing the spatial features. Other pointerless quadtree representations exist, but they are of less practical use. The reader is referred to [7] for further details.

4.1 DF-expressions

A DF-expression [4] is a pointerless quadtree representation of a feature. It can also be used with an incomplete pyramid data structure. It is in the form of a preorder tree traversal of the bit-slot(s) corresponding to the feature(s) in the active nodes of the incomplete pyramid. The result is a string consisting of the symbols 'G', 'B', 'W', corresponding to gray, black, and white, respectively. Other symbols are used when *multiple* features are stored in one DF-expression, as presented below. In the following, we show how a DF-expression is used to store multiple features regardless of whether or not they overlap in space. We first describe the simple one-feature case. More details about operations based on DF-expressions can be found in [7].

One Feature

Using the symbols ('G', 'B', and 'W') and applying a preorder traversal to the bit-slots of the active nodes in the incomplete pyramid corresponding to some spatial feature, we get the DF-expression string representing this feature. For example, the DF-expression of the feature of Figure 9 is given by:

GGWWWB
GWWGWWBWW
W
GGBWBBWGWBWGWBWBGWWGWBBBW.

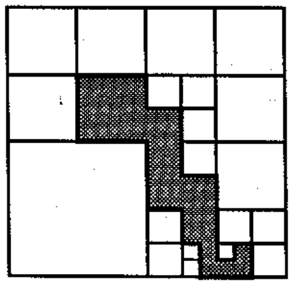

Figure 9: A spatial feature in a 16 × 16 feature space.

Multiple Features
Although multiple features can be represented by multiple DF-expressions (i.e., one DF-expression for each feature), there are several advantages to storing multiple features in a single DF-expression. It is economical from the standpoint of saving execution time when the same operation must be applied to all the features. An example of such an important operation is down loading and storing features into and out of the on-line incomplete pyramid data structure. In Section 5 we show that by storing multiple features together in one DF-expression, one traversal of the the incomplete pyramid is sufficient. This is in contrast to traversing the incomplete pyramid several times, once for each feature.

To support multiple features, we add more symbols to the DF-expression alphabet. In particular, in addition to 'G', 'B', and 'W', we also use i, $\{i_1, i_2, ..., i_n\}$, to correspond to an index for feature i, and a set of feature indexes, respectively.

Non-Overlapping Features
Representing multiple non-overlapping features by a single DF-expression is straightforward. It is the same as for the one-feature case with one slight modification. Whenever a black symbol 'B' appears in the DF-expression string, the index of the corresponding feature is concatenated to it (for example, 'B3' refers to a black block covered by feature 3). This is the same as coding colored images by a DF-expression. Notice that since features do not overlap in space, only one feature index is stored with each black symbol. As an example, the following string is the DF-expression for the image in Figure 1 which consists of 3 different features.

GGWWWB1
GWWGWWB1B2B2
B2
GGB1B3B1B1B3GB2B1B3GB3B1B3B1GB3B3GB3B1B1B1B3.

Overlapping Features
The representation of overlapping features requires the attachment of more information to gray and black symbols. The problem is that (1) a node may be gray for some features and black for several other features; and (2) a black node may correspond to several overlapping features. For example, for the three overlapping features in Figure 2, we have the following DF-expression:

GG1GWB2WWB2WGWB2WB2
B2
GB1B1B3B1,3
GGB2B2B3B2B2B3GB2,3B2B3W.

Note the use of commas to separate multiple overlapping features.

4.2 Linear Quadtrees

A linear quadtree only supports non-overlapping features. It represents a quadtree as a collection of the leaf nodes comprising it. This collection of leaf nodes is stored as a sorted linear list. Each node consists of three fields:

1. The address field: contains the two-dimensional locational code of the node's corresponding block. The locational code is composed by interleaving the bits of the binary representations of the value of the x and y coordinates of the block's upper left corner.

2. The depth field: contains the size of each block (i.e., the depth of the corresponding node in the quadtree).

3. The value field: contains the index of the feature associated with this node.

For example, the first twelve nodes of the linear quadtree for the non-overlapping features shown in Figure 1 are

(0000,2,W),(0100,2,W),(0200,2,W),(0300,2,1),(1000,2,W),(1100,2,W),(1200,3,W),(1210,3,W),(1220,3,1),(1230,3,2),(1300,2,2),(2000,1,2),(3000,3,1),(3010,3,3), ...

where (*address*, *depth*, *value*) correspond to the address field (coded in base 4), the depth field, and the value fields of a node, respectively.

5 Algorithms

We present several algorithms for loading non-overlapping and overlapping features into the incomplete pyramid data structure. For non-overlapping features we consider two algorithms based on how the input features are represented, either as a DF-expression or as a linear quadtree. For overlapping features we consider only the case where features are represented as a DF-expression. Recall that a linear quadtree only supports non-overlapping features.

One major benefit of the incomplete pyramid is that the subtrees descending from an active leaf node (recall that this is also the leaf node of the corresponding linear quadtree) are never accessed while loading the features (except for the bottom part which contains a raster representation of the feature space). This saves time when loading the features without sacrificing the spatial processing power of the regular pyramid as shown in [2]. Our three algorithms take advantage of this property.

Our presentation makes use of the following predefined functions:

- all_sons_reported(p) : returns *true* if all sons of p reported to it, and returns *false* otherwise. This routine is described in greater detail in Section 5.1.2.
- bit_slot(f) : returns the bit-slot in a node of the incomplete pyramid corresponding to the feature index f.
- data(l) : reads the current data element from the list l.
- depth(q) : returns the value stored in the depth field of node q of the linear quadtree.
- eol(l) : returns *true* when there are no more data elements in l; otherwise, it returns *false*.
- father(p) : computes the address of p's father in the incomplete pyramid. p is an address of a node in the incomplete pyramid.
- incr_intermed(p) : increments the 2-bit report field associated with node p in the incomplete pyramid. This 2-bit field is described in greater detail in Section 5.1.2.
- load_pyr_base(x, y, s, v) : fills a block in the bottom level of the incomplete pyramid with the value v. v is usually a feature index. This block is specified by its size s and the location of its upper-left corner (x and y).
- next(l) : advances to the next data element in the list l.
- number(s) : returns *true* when the data element s contains a number; otherwise, it returns *false*.
- or_bits(p, q) : is equivalent to contents(p) $\leftarrow$ contents(p) OR contents(q).
- pyr_node_address(d, x, y) : returns the address of the node at depth d in the incomplete pyramid that corresponds to a block whose upper-left corner coordinate values are x and y.
- rc_to_pi(r, c) : bit-interleaves r and c to form an index into the incomplete pyramid or a quadtree address field.

- set_node(i, p) : sets the ith bit-slot of the node in the incomplete pyramid pointed at by p to 1 and sets the other bit-slots to 0.
- set_pixel(i, v) : sets the node in the incomplete pyramid corresponding to a bottom-level pixel whose index is i to the value v.
- size(q) : returns the size of the block corresponding to the quadtree node q.
- son(p, i) : computes the address of the i^{th} son of the node pointed at by p in the incomplete pyramid.
- son_x(x, i) : computes the x-coordinate value of the upper left corner of the block corresponding to the i^{th} son given that the upper left corner of his parent's block has an x-coordinate value of x.
- son_y(y, i) : computes the y-coordinate value of the upper left corner of the block corresponding to the i^{th} son given that the upper left corner of his parent's block has a y-coordinate value of y.
- store_features(l, p) : stores the set of feature indexes currently at the front of the list l into the node of the incomplete pyramid pointed at by p and, as a side effect, advances to the next data element in the list l. This operation is used for loading multiple overlapping features in the same node p. Refer to Section 4.1 for the exact format for storing multiple features in DF-expressions.
- value(q) : returns the value stored in the value field of node q of the linear quadtree.
- x(q) : returns the value of the x-coordinate stored in the address field of the quadtree node q. This involves accessing the depth field as well so that we can determine the number of significant bits in the address field.
- y(q) : returns the value of the y-coordinate stored in the address field of the quadtree node q.

5.1 Non-Overlapping Features

We can speed-up the loading process by taking advantage of the fact that features do not overlap in space. We scan the image space once, one block at a time, and load the intersecting feature with this block (if any). Since features are non-overlapping, we know that at most one feature intersects this block.

We give two new algorithms for loading the incomplete pyramid with non overlapping features. The first algorithm assumes that the input is represented using a DF-expression while the second assumes that it is represented using a linear region quadtree. Since the intermediate nodes of the incomplete pyramid include summary information about the existence of features in the subtrees below them, both algorithms access each intermediate node a constant number of times independent of the number of features being loaded (which is non-trivial to achieve).

5.1.1 The DF-Expression Algorithm

In this section, the input is assumed to be a DF-expression. The basic idea behind the algorithm is that the DF-expression is scanned at the same time that the incomplete pyramid is traversed in postorder. Actions in the incomplete pyramid are performed according to the current DF-expression symbol.

```
1  Boolean recursive procedure load_nonoverlap(l, p, x, y, size)
   begin
     reference pointer list l;
     value pointer pyr_node p;
     value integer x, y, size;
     quadrant i;
2    if data(l) = 'G' then
       begin /* Gray node */
3        for i in {'NW', 'NE', 'SW', 'SE'} do
           begin
4            l ← next(l);
5            load_nonoverlap(l, son(p, i), son_x(x, i), son_y(y, i), size/2);
6            or_bits(p, son(p, i)); /* Store features in the parent. */
           end;
       end
7    else if data(l) = 'B' then
         begin /* Black node */
8          l ← next(l);
9          which_slot ← bit_slot(data(l));
10         set_node(which_slot, p);
11         load_pyr_base(x, y, size, which_slot);
12         l ← next(l);
         end;
     /* No action is necessary for a white node. */
   end;
```

Figure 10: Loading multiple non-overlapping features represented by a single DF-expression into the incomplete pyramid.

The algorithm performs a depth-first search in the incomplete pyramid guided by the symbols in the DF-expression. The code for the algorithm is given in Figure 10. The 'G' symbol instructs the algorithm to descend the incomplete pyramid by one level (line 5). Once a 'B' symbol is encountered, the corresponding node in the incomplete pyramid is set to include this feature in it (lines 8 and 9). The 'B' symbol is followed by the feature index which is used to set the corresponding bit slot in the leaf node. After processing a gray node's son, the features it represents are propagated into the gray node by the OR Boolean operation (line 6). Note that each non-leaf (i.e., gray) node has to report to its parent as soon as all its sons report back to it. This way each node reports to its parent only once. In other words, each non-leaf node is accessed a constant number of times. The code for loading blocks in the bottom level of the incomplete pyramid is given in Figure 11. Procedure


```
Boolean procedure load_pyr_base(x, y, size, val)
begin
  value integer x, y, size, val;
  integer size_sqr, base_xy, index;
  size_sqr ← size ↑ 2;
  if(size_sqr > 1) then
    begin /* The corresponding node is above the pixel level. */
      base_xy ← rc_to_pi(x, y);
      for index ← base_xy to base_xy + size_sqr - 1 do
        set_pixel(index, which_slot);
    end;
end;
```

Figure 11: Loading the base of the incomplete pyramid with a given feature index.

load_nonoverlap is initially invoked by the call load_nonoverlap(df_list, pyr_root, 0, 0, pyr_base_size).

5.1.2 The Linear Quadtree Algorithm

In this section, the input is assumed to be a linear list of the leaf nodes. Each node contains the following information:

- The locational code of the node's corresponding block.
- The block's size (or equivalently, the node's depth in the quadtree).
- The value of this node: white (i.e., the node is empty) or black (i.e., a feature covers the node's corresponding block). In the latter case, the value of the node is a feature index identifying the feature covering the block.

The algorithm performs the following four steps for each leaf node q:

1. Directly access the corresponding node in the incomplete pyramid, say p.
2. Mark p with the feature stored in the value field of q,
3. Report p's features to p's father, say P. We assume that each intermediate node of the incomplete pyramid has an extra 2-bit field to count the number of direct sons that report to it. Node p is said to *report* to its father P by adding its features to P, and incrementing P's counter. Once all of P's sons have reported to P, P also has to report to its father, say G. The same may apply for G and so on up to the root node. The root node does not report to any other node.

4. If q is BLACK (i.e., the feature covers q), then fill the base-level nodes in the incomplete pyramid descending from p with the same feature index. The base-level nodes below p do not report to their father. Note that if the bottom level of the incomplete pyramid is initialized with the image raster representation, then this step is superfluous. In other words, it is important to note that performing the filling step for all input leaf nodes of the linear quadtree is equivalent to loading the raster representation of the image in the bottom level of the incomplete pyramid.

```
Boolean procedure HelpLoadMap(quad_node)
begin
  value quadtree_node quad_node;
  integer quad_depth, quad_size; /* Quadtree node depth, size */
  integer quad_x, quad_y; /* The (x,y) coordinates of the upper left corner */
  integer which_slot; /* Bit-slot index */
  pyr_node p_addr, f_addr; /* Temporary incomplete pyramid pointers */
  quad_depth ← depth(quad_node);
  quad_size ← size(quad_node);
  quad_x ← x(quad_node);
  quad_y ← y(quad_node);
  p_addr ← pyr_node_address(quad_depth, quad_x, quad_y);
  which_slot ← bit_slot(value(quad_node));
  set_node(which_slot, p_addr);
  load_pyr_base(quad_x, quad_y, quad_size, which_slot);
  do
    begin
      f_addr ← father(p_addr);
      or_bits(f_addr, p_addr);
      incr_intermed(f_addr);
      p_addr ← f_addr;
    end
  until not (all_sons_reported(p_addr));
end;
```

Figure 12: Loading a linear region quadtree into the incomplete pyramid.

The code for the algorithm is given in Figure 12. Note that it is applied to each leaf node (whether black or white) in the linear quadtree. Also, it involves use of procedure load_pyr_base given in Section 5.1.1.

5.2 Overlapping Features

Overlapping features are stored in one DF-expression in the format described in Section 4.1. Our algorithm is similar to the one for loading non-overlapping features

presented in Section 5.1.1. The difference is that the algorithm of this section takes into account the case that (1) a node can be gray for some features as well as black for other features, and (2) that a black node can correspond to several overlapping features.

The algorithm performs a depth-first search guided by the symbols in the DF-expression. The code for the algorithm is given in Figure 13. For each 'G' symbol the algorithm descends the incomplete pyramid by one level (line 8). In such a case, the algorithm checks if there are features to be stored along with this gray node (line 4). If so, their feature indexes are read from the DF-expression and the corresponding bit-slots are set in the node of the incomplete pyramid (lines 5 and 6). The node will not report to its father until all its sons report to it (lines 8 and 9). Once a 'B' symbol is encountered, the corresponding node in the incomplete pyramid is set to include the feature(s) in it (lines 11 and 12). The 'B' symbol is followed by the set of feature indexes to be used to set the corresponding bit slots in the leaf node. After processing a gray node's son, the features gathered in this son are propagated into the node by the OR Boolean operation (line 9). Procedure load_overlap is initially involved by the call load_overlap(df_list, pyr_root, 0, 0, pyr_base_size).

6 Analysis

In this section, we analyze the three algorithms for loading features into the incomplete pyramid data structure. Our discussion is simplified by the following notation:

- NDF: algorithm to load non-overlapping features from a DF-expression.
- NLQ: algorithm to load non-overlapping features from a linear quadtree.
- ODF: algorithm to load overlapping features from a DF-expression.

The algorithms are compared on the basis of the length of the input (as dictated by the specific coding method), the space complexity of each algorithm, and the worst-case execution time.

6.1 Length of the Input

Assume that m features are loaded into the incomplete pyramid. Let $T \times T$ be the size of the feature space, L be the sum of the number of their non-white leaf nodes (which is linearly proportional to the total perimeter of the features), S_f be the number of bits necessary to encode the feature index, d be the depth of the incomplete pyramid (and the linear quadtree as well), W be the number of white nodes in the linear quadtree encoding of the features. Notice that $T = 2^d$, and that G, the number of gray nodes, is $(L + W - 1)/3$.

For the NLQ algorithm, the length of each node is the sum of the lengths of the three fields that comprise it. The address field (locational code) takes $2 \cdot d$ bits.

```
1 Boolean recursive procedure load_overlap(l, p, x, y, size)
  begin
    reference pointer list l;
    value pointer pyr_node p;
    value integer x, y, size;
    quadrant i;
2   if data(l) = 'G' then
      begin /* Gray node */
3       l ← next(l);
4       if number(data(l)) then
          begin
5           store_features(l, p); /* Load features into the gray node. */
6           load_pyr_base(x, y, size, contents(p));
          end;
7       for i in {'NW', 'NE', 'SW', 'SE'} do
          begin
8           load_overlap(l, son(p, i), son_x(x, i), son_y(y, i), size/2);
9           or_bits(p, son(p, i)); /* Store features in the parent. */
          end;
      end
    else
10    if data(l) = 'B' then
        begin /* Black node */
11        l ← next(l);
12        store_features(l, p);
13        load_pyr_base(x, y, size, contents(p));
        end;
    /* No action is necessary for a white node. */
  end;
```

Figure 13: Loading multiple overlapping features represented as a single DF-expression into the incomplete pyramid.

The depth field takes $\log d$ bits. And the value field takes S_f bits. Finally, the tota number of nodes in a linear quadtree is $L + W$.

Interestingly, NDF and ODF have the same input length (when L is the same fo both of them). The extra commas in ODF (in the worst-case) are equal in numbe to the extra 'B' symbols in NDF.

Therefore, the lengths of the inputs are:

- NDF: $(L + W - 1)/3 + L + W + L \cdot S_f$
- NLQ: $(2 \cdot d + \log d + S_f) \cdot (L + W)$
- ODF: $(L + W - 1)/3 + L + W + L \cdot S_f$

6.2 Space Complexity of the Loading Algorithms

The NLQ algorithm accesses a leaf node directly, and then moves upwards (i.e. from the addressed leaf node towards the root). The NLQ algorithm uses some extra space (2 bits for each intermediate node) to guarantee that each non-leaf node waits until all its children report their features to it before reporting once to its parent.

The NDF and the ODF algorithms need no explicit extra space to keep track of how many nodes report to their parents since this information is inherent in the postorder processing of the nodes. Recall that both algorithms perform either a postorder traversal or a combination of a preorder and a postorder traversal in order to propagate feature indexes through the incomplete pyramid. Therefore, the only storage required is the stack area to maintain the recursive calls and the actual argument values at each call. These sum up to 3 integers and 2 pointers per call, and the maximum number of calls is the same as the maximum depth (d) of the incomplete pyramid.

Thus the space complexity (in addition to the space occupied by the incomplete pyramid) of the three algorithms is summarized as follows:

- NDF: $d \cdot (3 \cdot int\_size + 2 \cdot pointer\_size)$
- NLQ: $2 \cdot (4^{d-1} - 1)/3) \cong T^2/6$
- ODF: $d \cdot (3 \cdot int\_size + 2 \cdot pointer\_size)$

6.3 Worst-case Execution Time Complexity

First, let us examine the case that the features do not overlap. Here we consider only the NDF and NLQ algorithms. Our analysis is based on the cost of calculating the address of a node's son and parent, or of accessing a leaf node directly (as in the case of the NLQ algorithm). Recall from Section 3 that explicit pointers to sons or parents are not stored in the incomplete pyramid.

The NDF algorithm descends through the incomplete pyramid via a combination of a preorder and a postorder traversal. This implies that for each non-leaf (gray) node we must perform four son-address calculations while descending, one per child.

For each leaf node the NLQ algorithm performs an address calculation to directly access the corresponding leaf node in the incomplete pyramid. On the other hand, each non-leaf node is accessed four times. Each child node has to compute its parent's address once the child wants to report to it. Hence four parent-address calculations are performed per non-leaf node, once from each of its children.

When features are permitted to overlap, we use the ODF algorithm. It has the same run-time complexity as the NDF algorithm. The only difference is that more input symbols may have to be read when encountering a gray or a leaf node. This is because a gray or leaf node may contain more than one overlapping feature with its corresponding block. However, this is compensated by the fact that less 'B' symbols are input (in contrast to the NDF algorithm). Note that for comparison purposes, L is the same for each of the three representations.

The following formulas summarize the execution times for all three algorithms. A_S is the time required to compute the address in the incomplete pyramid of a node's son, given the node's address in the incomplete pyramid, and to access the son's location. A_P is the time required to compute the address in the incomplete pyramid of a node's parent given the node's address in the incomplete pyramid and to access the parent's location. A_L is the time required to compute and access a leaf node's address in the incomplete pyramid given its corresponding block's parameters (size and the location of its upper left corner). Notice that the definition of the incomplete pyramid means that $A_S, A_P,$ and A_L are constants (i.e., they have constant execution times).

- NDF: $4 \cdot A_S \cdot G$
- NLQ: $4 \cdot A_P \cdot G + A_L \cdot L$
- ODF: $4 \cdot A_S \cdot G$

Recall that G, the number of gray nodes, is $(L + W - 1)/3$.

7 Conclusion

Algorithms for loading spatial features into the incomplete pyramid data structure from two pointerless quadtree representations have been presented. The algorithms differ on the basis of the representation of the input feature(s). Two of these algorithms only work for non-overlapping features, while the third algorithm only works for overlapping features. The execution time for all three algorithms is linear in the length of the perimeter of the features loaded, and is independent of the depth of the incomplete pyramid.

Analysis of the space and execution time requirements of the algorithms revealed that the DF-expression is superior to the linear quadtree. This is not surprising

because the principal drawback of the DF-expression is its sequential nature (i.e., it does not support random access) which the linear quadtree does. However, the loading algorithms only require that the whole tree be processed in sequence and hence the DF-expression is adequate.

At the moment, the conversion algorithm has only been implemented for the linear quadtree. It takes approximately 9 seconds on a VAX 11/785 to load a 512×512 image. We are presently implementing the algorithms for overlapping and non-overlapping DF-expressions.

Our work shows the usefulness of the incomplete pyramid data structure. The algorithms for loading features into the incomplete pyramid are much faster than that for the regular complete pyramid. This speed-up arises from the fact that for the incomplete pyramid there is no need to fill all the intermediate nodes descending from a leaf node.

8 Acknowledgements

The support of the National Science Foundation under Grant IRI-9017393 is gratefully acknowledged.

References

[1] D. J. Abel. SIRO-DBMS: a database tool-kit for geographical information systems. *International Journal of Geographical Information Systems*, 3(2):103–116, April–June 1989.

[2] W. G. Aref and H. Samet. Efficient processing of window queries in the pyramid data structure. In *Proceedings of the 9th ACM-SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 265–272, Nashville, TN, April 1990.

[3] I. Gargantini. An effective way to represent quadtrees. *Communications of the ACM*, 25(12):905–910, December 1982.

[4] E. Kawaguchi, T. Endo, and M. Yokota. Depth-first expression viewed from digital picture processing. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 5(4):373–384, July 1983.

[5] A. Klinger. Patterns and search statistics. In J. S. Rustagi, editor, *Optimizing Methods in Statistics*, pages 303–337. Academic Press, New York, 1971.

[6] J. A. Orenstein. Spatial query processing in an object-oriented database system. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, pages 326–336, Washington, DC, May 1986.

[7] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, Reading, MA, 1990.

[8] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, Reading, MA, 1990.

[9] C. A. Shaffer. *Application of Alternative Quadtree Representations.* PhD thesis, University of Maryland, College Park, MD, June 1986. Technical Report Computer Science TR 1672.

[10] C. A. Shaffer and H. Samet. An in-core hierarchical data structure organization for a geographic database. Technical Report Computer Science TR 1886, University of Maryland, College Park, MD, July 1987.

[11] C. A. Shaffer and H. Samet. Optimal quadtree conversion algorithms. *Computer Graphics, Vision, and Image Processing*, 37(3):402–419, March 1987.

[12] C. A. Shaffer, H. Samet, and R. C. Nelson. QUILT: A geographic information system based on quadtrees. *International Journal of Geographical Information Systems*, 4(2):103–131, April–June 1990.

[13] S. Tanimoto and T. Pavlidis. A hierarchical data structure for picture processing. *Computer Graphics and Image Processing*, 4(2):104–119, June 1975.

[14] L. Tucker. *Computer Vision Using Quadtree Refinement.* PhD thesis, Polytechnic Institute of New York, Brooklyn, May 1984.

[15] T. C. Waugh and R. G. Healey. The GEOVIEW design: A relational data base approach to geographical data handling. *International Journal of Geographical Information Systems*, 1(2):101–118, April–June 1987.

[16] J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, June 1964.