Window query processing in Fast Dynamic Linear Quadtrees

Mirela Catrina, Sorana Catrina

June 19, 2022

Abstract

A quadtree is a tree data structure used in storing geographic information about objects in images. It is build by successively decomposing regions in quadrants and storing said splits in a tree structure. The "leaves" of this structure differ in meaning according to its usage, so we will refer to it as "a unit of interesting spatial information". This building method results in a structure that grows its size exponentially. In consequence a number of solutions have been proposed. This paper focuses on an efficient way to build, store and search information in the quadtree, particularly showing its efficiency in window query processing. The efficiency in building and storing the quadtree is yield by using FDLQE encoding and saving the information extracted in a B+ tree. Thus searching for information contained in a window also is done efficiently.

I. Introduction

Hierarchical data structures are important ways of storing data. Alongside storing, these structures must deal with specific operations in efficient ways. In consequence, a large number of such structures have been developed for dealing with certain operation.

A quadtree is a hierarchical data structure. Each level coresponds to a further refinement of the space under consideration. In order to achieve this, recursive decomposition of the space is applied, until one is satisfied with the region obtained and stops the decomposition. "Satisfaction" can be measured by different methods, such as measuring the similarity of colors in pixels contained in the region or just stopping when all pixels reach a certain colour. The satisfactory limit usually depends on the number of details that should be stored or the size of the analyzed image.

Quadtrees play an important role in various domains related to image processing and consequently variants have been developed to support different operations and analysis. Some notable variants are: region quadtrees (used for image union and intersection, connected component labelling), point and point-region quadtrees (built for solving spatial queries efficiently), edge and polygonal map quadtrees (with applications in location queries), quadtrees built for mesh generation and labelling. [2]

In higher dimensions siblings of quadtrees can be built, such as octtrees (for 3D) and hyperoctrees.

Moerover, due to the exponentially growing size of quadtrees and its variants, methods of compressing them have been an important topic.

A proposed solution are *linear quadtrees*. Essentially, they store only the leaves of the quadtree. Each leaf receives a *code* built for saving properties necessary for different operations.

This paper follows the usage of a region quad-tree in window query processing. A linear quad-tree is built on a given image and stored using a B+ tree. In consequence, it will be able to answer effectively to questions: *"Given a region determined by a set of (x, y) coordinates, which objects in given image are contained in it?"*.

II. Related work

The usage of linear quad-trees is highly dependent on the way of encoding data contained in a "unit of interesting spatial information".

This paper relies heavily on the encoding method put forward by Sheng Yehua, Tang Hong, Zhao Xiaohu [4] and the storing and window query processing method proposed by Ashraf Aboulnaga and Walid G. Aref [1].

III. The linear quad-tree

A linear quadtree, as previously stated, stores only the leaves of the quadtree for a given image. The quadtree, alongside its linear quadtree can be visualized as follows:

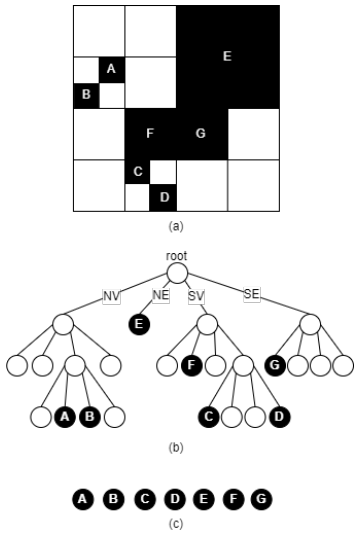

Figure 1: *(a) represents the image given, (b) the quadtree, (c) the linear quadtree of the image*

As seen in Figure 1, the images we will use are black and white. The white will represent the "background" and the black represents the objects that hold the spatial information we are looking for.

Analysing (b) from the figure above, we observe a large number of nodes that do not hold units of interesting information. The unnecessary nodes that hold pointers to their children without holding any data occupy 90% of the entire memory used for the tree [5]. In consequence, (c) would be a much better option. In order for it to be viable, we need to hold enough data in the leaves (usually under the form of encoding) to be able to extract relationships and locations within the image. These aspects rely on choosing the right encoding for the operations that will be executed on the tree.

i. Morton space filling curve

The Morton space filling curve, also called Z-order curve is a curve that can be seen as formed by a depth-first traversal of a quadtree.

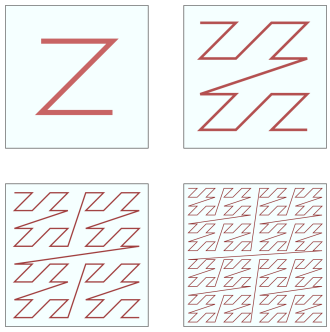

Figure 2: *Four level z-curve*

Therefore, a z-code can be assigned to each node of the quadtree. The z-code of a point situated on line x, column y can be calculated as follows:

- Get the representation in base 2 of x and y
- Interleave the binary values

Decoding can be done quickly by extracting the bits in positions corresponding to x and, respectively, y.

ii. Encoding overview

As stated above, encodings may differ according to the usage of the linear quadtree.

A largely used method to encode data is by using quaternary codes [3]. Following a convention regarding the numbers given to quadrants by place (for example, the NE child

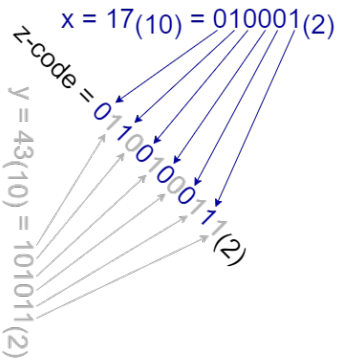

Figure 3: *Creating the z-code by position*

might receive a code of 0, the NW a code of 1, SE - code 2, SW - code 3), relationships between the leaves could be extracted by following these codes (eg: adjiacency).

Another method would be encoding by using the Morton space filling curve shown in the section above or Figure 3. This method is highly used due to the easy way of constructing and decoding the assigned code, the low complexity of the operations being done on bits and its property of simulating a depth-first traversal of the linear quadtree. The assigned code is also called Decimal Morton Code (abbreviated as MD).

In this paper, we will follow the MD encoding mechanism, enriching it with other fields and properties in order to combine succesfully and eficiently the object extraction from the images and the window query processing based on the B+ tree.

iii. The Fast Dynamic Linear Quadtree Encoding Model

The Fast Dynamic Linear Quadtree Encoding Model refers to the solution of retrieving a linear quadtree from an image proposed by Zhao Xiaohu, Shen Yehua, Tang Hong [4].

The most used method is "split and merge". This method proposes recursively decomposing a region into four quadrants, which we will refer to as children: the NW child, the NE child, the SW child, the SE child. Each region that would be split becomes a "parent" to the 4 sub-regions, consequently storing pointers to these children. The splitting is done until reaching an region-size limit. This is followed by a "merging" phase, which consists in a bottom-up traversal of the obtained tree and "merge" of all 4 children of the same parent that qualify as "units of interesting information".

This process could also be modified so that "merging" will not be necessary; thus the splitting will be stopped once the region contains only "units of interesting information" (in our case, black pixels). We will refer to this method as top-to-down partition.

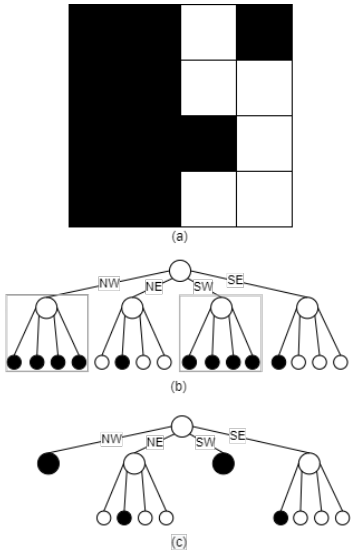

Figure 4: *(a) represents the image given, (b) result of splitting, (c) result of merging*

The latter method can be implemented and used for forming linear quadtrees instead of memorizing the quadtree as it forms.

The FDLQE model proposes traversing the image in a way that guarantees visiting blocks part of the "same region" in a given order. More specifically, we traverse the image in increasing order of the Morton Decimal Code, saving this codes in a stack. If the stack contains more than 4 elements, we try to "compress" it if the last 4 blocks are all black (meaning we can now save only the bigger region, this is similar to the merging process presented above). Therefore, there are 2 major steps in the algorithm:

1. Extracting values from the image in order of the Morton Code and adding them to the stack if they are black

2. Try to condense the stack

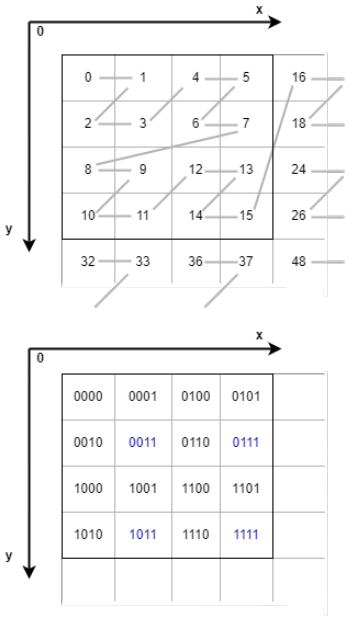

Figure 5: *Z-order codes, in base 10 and in base 2*

The images above depict the order of traversing the image, taking into consideration the Morton Codes.

The most important part of this process then is the condensation of the stack. The top MD code in the stack is used in order to asses how many times the stack can be condensed [4]. For example, if blocks 0,1,2,3 ... 14 are all black, once we reach 15, the number of "merging steps" we must take is two, since we need to condense the last 4 blocks (12-15) and then condense again to obtain the bigger block 0-15. The process is illustrated in the image below:

Intuitively, we can see that if the number can be written as $4 * k - 1$, a condensation needs to be applied, and the more factors of 4 are present in the number, the more condensation steps. Therefore, the number of steps of condensation (T) can be calculated with formula given by Sheng Yehua, Tang Hong and Zhao Xiaohu [4]:

$$mod[\frac{MD+1}{4^{k+1}}] = 0; T = max(K) \quad (1)$$

The formula is intuitively explained above. For example if the top of the stack is 3, the maximum power of 4 that gives us mod 0 is 1, so we can apply at most one condensation (meaning we try to condense the block containing 0-1-2-3). However, if the last element is 15, the biggest power in 16 of 4 is 2, meaning two steps of condensation. However, there is an easier way to compute T. We just need to know how many zeros trail MD+1 in binary representation, and take the log in base 2 of that number divided by two. Therefore, we can define the following functions, where ^ is bitwise XOR and & is bitwise AND (also, for T we only get the quotient):

$$\begin{aligned} zeros(x) &= (x\hat{}(x-1))\&x \\ T(x) &= log2(zeros(x))/2 \end{aligned} \quad (2)$$

In order to see how these functions work, we provide the following table containing some examples.

x	x-1	x^(x-1)	zeros(x)	T(x)
4=100	011	111	100	1
16=10000	01111	11111	10000	2

Table 1: *Formula exemplification*

By using this formulas and traversing the image in the right order we can construct the linear quadtree that will be used in the window query. The algorithm is presented in Algorithm 1 in pseudocode.

However, due to the "merging" process, we can't store only the starting point of a block, and need therefore more information, such as length. For ease of use, we chose to also save the "depth" of the block, which is equivalent to how many times we split the image in order to get the block of that size. For example, a code-block encoding the whole image has depth 0, its children have depth 1, and so forth.

To encode this information into one index, by which we'll construct the B+ tree we developed the following encoding, inspired by the

encoding in [1] (however, we will still use decimal encoding). The most significant bits of the base two representation will contain the Morton code, followed by the bits representing the depth. A more in-depth example is presented in section IV.

Algorithm 1 Building the FDLQ

Input: image
Output: vector containing morton blocks

```
v ← null
stack ← null
code ← 0
size ← image.size
while code ≤ size do
    lin ← getLine(code)
    col ← getColumn(code)                ▷ Decoding
    if img[lin][col] is black then
        stack ← stack ∪ img[lin][col]
    end if
    if stack.size ≥ 4 then               ▷ Try condensing
        numsteps ← T(code + 1)
        for l = 1 to numsteps do
            if can condense then
                stack ← stack − last4Blocks
                stack ← stack ∪ newBlock
            end if
        end for
    end if
    code ← code + 1
end while
```

IV. Storing: B+ tree

A B+ tree is a m-ary tree. Usually, each node has a large number of children. It is similar to a B-tree, yet a major difference can be observed: in B-trees all nodes (internal or leaves) can hold data, whereas in a B+ trees only the leaves hold data (and keys), the internal nodes being used only for the keys they store. In consequence, a left to right traversal of all leaves would yield a sorted vector consisting of all the information that was inserted in the B+ tree, keys and data.

A B+ trees allows us to hold the information extracted in the linear quadtree alongside efficient algorithms for searching the stored data. Since a large number of window querries will be applied to the analyzed image, storing and providing efficiency in solving such querries is crucial.

Insertion in the B+ tree uses as keys the b-codes, explained in the section above.

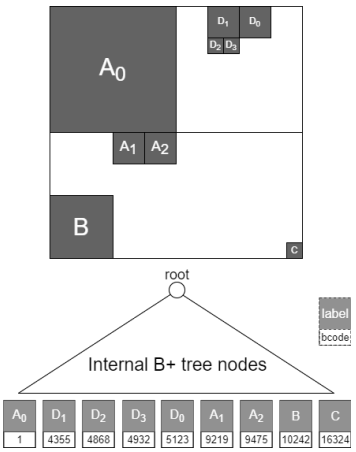

Figure 6: *Example of B+ tree built on image*

Label	$bcode_{(10)}$	$bcode_{(2)}$
A0	1	0000000000 0001
D1	4355	0100010000 0011
D2	4868	0100110000 0100
D3	4932	0100110100 0100
...	...	...

Table 2: *Meaning of bcodes based on example*

In the table above, in the $bcode_{(2)}$ highlighted in blue we find the binary representation of x, in gray the binary representation of y and in orange the depth.

V. Window query processing

Window query processing refers, in our case, to retreiving quadtree blocks (leaves) that we have identified to be contained in a certain area given as input.

In order to write the algorithm we use a stack to simulate the recursion. Therefore, as long as we still have quadrants to explore, we continue splitting the space.

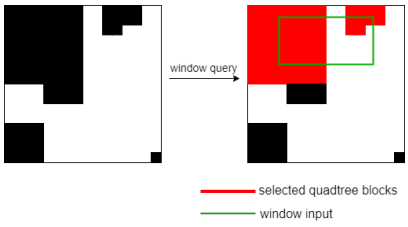

Figure 7: *Example of output from window query*

The algorithm works by recursively splitting the space into quadrants and searching for the corresponding Morton Blocks into the B+ tree. We start with the initial window as the whole image, and can distinguish between two cases considering a window b, that is given to intersect the window.

If the current window (b) is enclosed within the query window (w), this means that all the Morton Blocks enclosed in b will be enclosed in w as well, and therefore need to be added to the solution. Since we generated the codes in order by the bindex, we only have to search for the position our window b would occupy in the B+ tree, and the maximum bindex of the Morton Block that would be encapsulated in b as well (we called this Mbmax) - we calculate this by considering the lower right corner of our b window with maximum depth. This corresponds with the first if in the while. An example of this could be observed in Figure 1, considering the image being 32x32, the query window to be the bottom half of the image and the current window as the SW corner (the one that contains B, A1, A2). In this case, b would be totally enclosed in w. Therefore, we would consider Mbmax. Since the lower right corner would be of coordinates (31, 15) and of depth 5, the Mbmax code would be

$$10111111110101_2 = 12277$$

We then search the B+ tree for our current bindex (8225) and return all codes between these two values, which are the Morton Blocks B, A1, A2.

If b only partially intersects the window, we can only be sure that b is part of the solution (it is not guaranteed that all blocks enclosed in b will be enclosed in w as well), therefore we only add b to the solution, if we can find it in the B+ tree. To find the other Morton Blocks that intersect with our window w, we decompose the window into its four children. If the child intersects the query window, it is added to the stack. The children are added to the stack in reverse order of their bindex (the child with the smallest bindex will be on top, in order to be the first to be analysed).

Algorithm 2 Window query processing

Input: window as set of coordinates
Output: vector of morton blocks retreived

```
stack ← stack ∪ 0                          ▷ MDC of image
sol ← null
while stack not empty do
    b ← stack.top
    stack ← stack.pop
    if b is enclosed in window then
        pos ← position of b
        lcode ← b's window SE corner MD
        Mbmax ← lcode's B − code
        for code = b to Mbmax in B + tree do
            sol ← sol ∪ code's block
        end for
    else
        if b is in B + tree then
            sol ← sol ∪ b's block
            continue
        end if
        for SE, SW, NE, NW children of b do
            if child ∩ window ≠ ∅ then
                stack ← stack ∪ child.bcode
            end if
        end for
    end if
end while
return sol
```

VI. Future Work

Future improvements could be added in order to increase the size of the image that can be analysed (this is currently determined by

how we encode the pixels into the Morton Decimal Code). Therefore, we could start using hex codes in order to have also a more straightforward representation of the bits ordering, but to also decrease the size of the bindex and let us use bigger images. Minor changes have to be done in order for the algorithm to work on databases with overlapping objects as well.

References

[1] Walid G. Aref Ashraf Aboulnaga. "Window Query Processing in Linear Quadtrees". In: *Distributed and Parallel Databases* 10 (2001), pp. 111–126. DOI: 10.1023/A:1019256828670. URL: https://doi.org/10.1023/A:1019256828670.

[2] Anthony D'Angelo. "A Brief Introduction to Quadtrees and their Applications". In: *Canadian Conference on Computational Geometry* 28 (2016).

[3] Irene Gargantini. "An effective way to represent quadtrees". In: 25.12 (Dec. 1982), pp. 905–910. DOI: 10.1145/358728.358741. URL: https://doi.org/10.1145/358728.358741.

[4] Zhao Xiaohu Sheng Yehua Tang Hong. "Linear Quadtree Encodings of Raster Data and Its Spatial Analysis Approaches". In: (2012).

[5] I. P. Stewart. "Quadtrees: Storage and Scan Conversion". In: *The Computer Journal* 29.1 (Jan. 1986), pp. 60–75. ISSN: 0010-4620. DOI: 10.1093/comjnl/29.1.60. eprint: https://academic.oup.com/comjnl/article-pdf/29/1/60/1042434/290060.pdf. URL: https://doi.org/10.1093/comjnl/29.1.60.