Pattern Recognition Letters 15 (1994) 57–63
North-Holland

January 1994

PATREC 1118

A fast quadtree normalization algorithm

Chuan-Heng Ang

Department of Information Systems and Computer Science, National University of Singapore, Singapore

Hanan Samet

Computer Science Department and Institute of Advanced Computer Studies and Center for Automation Research, University of Maryland, College Park, MD 20742, USA

Received 20 February 1992

Abstract

Ang, C.-H. and H. Samet, A fast quadtree normalization algorithm, Pattern Recognition Letters 15 (1994) 57–63.

A region quadtree representation of an image can be normalized thereby yielding a quadtree that contains the least number of nodes in $O(s^2 \log_2 s)$ time where s is the length of the grid. A new algorithm is proposed whose asymptotic time bound is the same but whose first part only takes $O(s^2)$. It is shown empirically to be able to produce the normalized quadtrees for a number of images in real applications.

Keywords. Quadtree normalization, region representation, image processing.

1. Introduction

The region quadtree [4,8,9] is a simple data structure which can be used to store an image. It can be stored in main memory (usually with pointers) or on disk (via a B-tree containing the leaf nodes). In either case, it is easy to manipulate. Many operations such as finding a neighboring pixel or block [7] or computing some geometric properties [11] can be easily carried out on a region quadtree (termed *quadtree* from now on). Therefore, quadtrees and octrees (the three-dimensional extension of a quadtree) find use in image processing [6], computer graphics [1], cartography, geographic information systems [10], and other related applications.

Given an image I, its corresponding quadtree can be constructed as follows. If its color is homogeneous (i.e., either white or black), then it is represented as a single node of that color. Otherwise, the image is represented by a gray node which has four son nodes. Each son node corresponds to one component of the image after it has been decomposed into four parts, namely NW, NE, SW, and SE quadrants with respect to its center. The decomposition process is repeated until either the portion of the image under consideration is homogeneous or the grid resolution is reached. This top-down process used to construct the quadtree is called *regular decomposition*. The origin of the grid is defined to be at the top leftmost corner with the x-axis pointing to the right (east) and the y-axis pointing downward (south). The grid with length s

Correspondence to: H. Samet, Center for Automation Research, University of Maryland, College Park, MD 20742, USA.
This work was supported in part by the National Science Foundation under Grant IRI-9017393.

partitions the image plane into s^2 grid cells, or pixels, which is also the result of applying regular decomposition to a checkerboard image. In the following discussion we do not distinguish between a quadtree and the image, nor between a quadtree node and the part of the image that it represents.

One shortcoming of the quadtree representation is the sensitivity of its storage requirements to its position. By placing a given black square at different positions in the image plane, we may be able to store the image in as little as just one quadtree node or as many as $O(p+n)$ nodes, where p is the length of the perimeter of the image and n is such that 2^n is the width of the grid [2,3]. Since the difference in the storage requirements is so significant, at times it may be deemed worthwhile to minimize the space requirements of the images stored in quadtrees, especially when the number of images is large. Given an image, the quadtree that contains the least number of leaf nodes among all the quadtrees that represent the same image is called the *normalized quadtree* of the image. If there is more than one quadtree with the minimum number of leaf nodes, then we choose the one which has the leftmost and the uppermost black pixel in order for the *normalized quadtree* to be unique.

In [5], an $O(s^2 \log_2 s)$ algorithm is given to find the amount of translation required by an image in order to minimize its space requirements when it is stored in a quadtree. We describe this algorithm in Section 2, as well as point out an error in its original formulation. Although the original algorithm appears to be quite efficient, it still can be improved upon, and an optimal algorithm is yet to be determined. We propose an improvement in Section 3. The new algorithm is analyzed in Section 4 together with some empirical results. Section 5 contains some concluding remarks.

2. Quadtree normalization algorithm

Let the resolution of the grid be $n+1$ and its length be $s=2^{n+1}$. A two-dimensional array of size $2^{n+1}\times 2^{n+1}$ is needed to record the color of each pixel. Initially, the given image is assumed to be enclosed completely in the NW quadrant of the grid, with some of its black pixels touching the x- and y-axes. This array can also be viewed as the 4^{n+1} leaf nodes of pixel size in the complete quadtree before merging of nodes takes place. Merging is just the reverse process of regular decomposition. Whenever four son leaf nodes of a gray node are found to be of the same color, they are deleted and their parent node is changed to that color. Effectively, they are being merged into a bigger node representing a block of size 2×2 or 4 times the size of a son leaf node. Each time such a merging takes place, the number of leaf nodes is reduced by 3.

There are four ways to move the image zero or one pixel in the x- and y-directions. After the translation, the number of leaf nodes that are of the size of a pixel can be found in each of these four situations. These quantities will never change when the image is translated further by 2^k pixels, $k>0$, in either direction. In other words, these leaf nodes will never merge into bigger nodes no matter how the image is being translated subsequently with the amount of translation being at least twice the size of a leaf node. These leaf nodes appear in quadtree blocks of size 2×2 that are marked as gray whereas the others are merged and marked with appropriate colors. In fact, this is what we get when we reduce the length of the grid by half, or equivalently its resolution by 1, in each direction. The size of the array required to store the information is now reduced to 4^n, which is one quarter of the size of the array used before merging. At this point, each entry of the array records the color of the corresponding 2×2 block. Each one pixel transition in this array is equivalent to a two-pixel transition in the array used before merging.

What has been described is just one step of merging. This merging process can be repeated recursively until either only one 2×2 block is left or further processing will not produce any savings when all the elements of the resulting array are gray. The number of leaf nodes L and the translations in the x- and y-directions are recorded if L is found to be the smallest. The sizes of the arrays used in the recursion are 4^{n+1}, 4^n, ..., 4. For each of these arrays, there are four ways to translate the image before one step of merging is carried out. Therefore the merging step is invoked $\Sigma_{i=1}^{n} 4^i = O(4^n)$ times. Since the size of the array in each step of recursion has been reduced by 3/4, the cost of this processing can be shown to be within $O(4^n n)$ time and $O(4^n)$ space.

Figures 1 and 2 are an image and its purported nor-

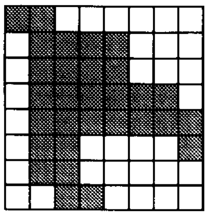
Figure 1. Sample image.

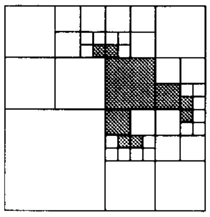
Figure 2. The normalized quadtree for Figure 1 using the algorithm in [5].

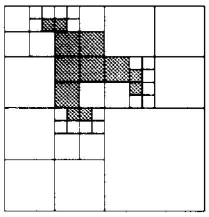
Figure 3. Normalized quadtree after translating Figure 1 by 3 pixels in the x-direction and 1 pixel in the y-direction.

malized quadtree labeled as Figures 5a and 5b in [5]. Figure 2 contains 46 leaf nodes of which one node is of size 4×4 and 2 nodes are of size 2×2. Without referring to our definition of a normalized quadtree, we are misled into accepting Figure 2 as the normalized quadtree of Figure 1 due to its simplicity. However, there are other translations with the same number of leaf nodes. For example, Figure 3 shows the result of translating the image in Figure 1 by 3 pixels in the x-direction and by 1 pixel in the y-direction. It also has 46 leaf nodes. Since the black pixels in Figure 3 are nearer to the y-axis than those of Figure 2, Figure 3 should be chosen as the normalized quadtree for the image in Figure 1.

3. An improved quadtree normalization algorithm

The algorithm of Li et al. (termed *Li's algorithm*) is essentially a depth-first search through a space of configurations. Each configuration is completely described by the resulting array after the merge, the amount of translation made, and its leaf node count. Initially, the size of the array is $2^{n+1}\times2^{n+1}$ with 4^{n+1} leaf nodes and it is represented as the root node. Since there are four ways to perform one step of translation and merging, there are four son nodes corresponding to the four resulting configurations. Each step of recursion will produce four more nodes at the next level of the tree describing the space to be searched. It is easy to see that the related configurations can be linked together to form a quadtree. The leaf nodes of this search tree are either arrays of size 2×2 or arrays with only gray elements. Li's algorithm simply searches exhaustively through the whole space, looking for the one with the minimum leaf node count. Its performance can be improved if the search space can be pruned.

Let us look at one example for illustration. A $2^2\times2^2$ image and its normalized quadtree are shown in Figures 4a and 4b, respectively. Figure 5 shows the search space of the configurations used by Li's algorithm in determining the normalized quadtree given in Figure 4b. The four sons of any configuration can be arranged from left to right to correspond to the resulting configurations obtained by merging and translating the array with the values (0, 0), (1, 0), (0, 1), and (1, 1) in the directions of the x- and y-axes. The configurations chosen to produce the optimal solution are those terminating in the node labeled with 10. According to Li's algorithm, this final configuration can be further reduced by removing the three white 2×2 blocks that surround the original image.

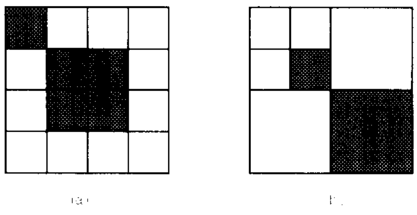

Figure 4. (a) An example image and (b) its normalized quadtree.

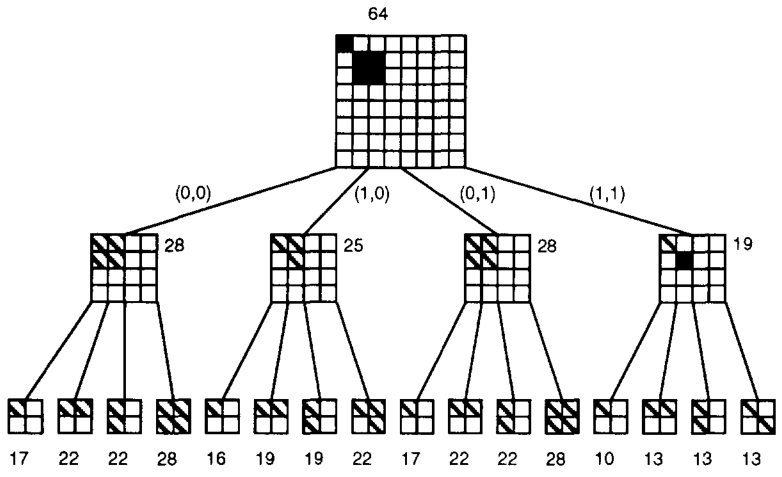

Figure 5. The search space of the configurations used by Li's algorithm in determining the normalized quadtree given in Figure 4b. Each node is labeled with the number of leaf nodes in its configuration. Each edge is labeled with its translation value. Gray nodes are represented by squares with diagonal lines.

Therefore, the normalized quadtree contains only 7 leaf nodes.

Li's algorithm is quite efficient in comparison to the brute force approach in the sense that the amount of work to be performed in each step of the recursion has been significantly reduced. Unfortunately, it does not make use of the partial result computed so far to cut down further the number of recursions that need to be performed. To improve its performance, the number of recursive calls has to be reduced as each call will incur some overhead cost. The sooner a particular path of recursion is pruned, the greater is the saving. One way to terminate the recursion earlier is described below.

The basic idea is to make use of C, the candidate normalized quadtree obtained so far in the computation. Let the number of leaf nodes in C be C_{LEAVES}. Let the number of leaf nodes that cannot be merged in the current recursion before the translation be N_{LEAVES}. After the translation and one step of merging, the number of white and black nodes of size 2×2 are denoted by N_{WHITE} and N_{BLACK}, respectively, and N_{LEAVES} will be updated by adding those leaf nodes that cannot be merged, i.e., of size 1×1. Initially, N_{LEAVES} is zero. After one step of translation and merging, N_{LEAVES} is the number of pixel-sized leaf nodes and N_{WHITE} (N_{BLACK}) is the number of white (black) nodes of size 2×2. They are not necessarily the leaf nodes in the quadtree resulting from possible merging in the subsequent steps. Therefore, they require special treatment in the following calculation.

It is obvious that if $N_{\text{LEAVES}} > C_{\text{LEAVES}}$, then the resulting quadtree cannot be optimal, and we can stop the recursion. Otherwise, calculate I_{WHITE} which is the minimum number of white nodes that result after the N_{WHITE} nodes are merged. There will be $N_{\text{W1}} = \lfloor(N_{\text{WHITE}}/4\rfloor$ white nodes after one step of merging if all of the N_{WHITE} nodes are adjacent to each other so that only the minimum number of white nodes will be required to represent them. This will leave $N_{\text{WHITE}} - 4 \cdot N_{\text{W1}}$ nodes that cannot be merged. Similarly, we can find the number of white nodes left over after two, three, and in general, k steps of merging. The sum of the numbers of all the white nodes left over after the merging is I_{WHITE}.

I_{WHITE} is the minimum number of leaf nodes required to represent N_{WHITE} white nodes at this stage of merging. Therefore, if $N_{\text{LEAVES}} + I_{\text{WHITE}} > C_{\text{LEAVES}}$, then we can also stop the recursion. Otherwise, we perform the analogous calculation for the set of black nodes to obtain I_{BLACK}.

If $N_{\text{LEAVES}} + I_{\text{WHITE}} + I_{\text{BLACK}} > C_{\text{LEAVES}}$, then stop the recursion. Otherwise, recur and, if necessary, update C_{LEAVES} when a new candidate is found.

The sequence of tests that we have described above is just a minor improvement to Li's algorithm. A major improvement can be achieved if a good candidate can be found as early as possible in the search process. It is obvious that a better candidate will be one with a smaller value of C_{LEAVES}. In Li's algorithm, a candidate whose leaf node count is to be used to establish or update the bound can only be found in the order dictated by the depth-first search. This may not be efficient especially when the best translation made in each step of the recursion happens to be the last of the four possible moves considered. When this situation arises, a candidate that has just been obtained may be replaced very frequently in the subsequent recursion steps, and many steps of recursion are still required.

If we can find a heuristic that will lead us to some leaf nodes of the configuration search tree with leaf node counts close to that of the optimal solution, then the search space can be pruned more effectively. A better method to get a good candidate quickly is as follows. Start with 4^{n+1} leaf nodes. In each iteration, four translations will be attempted with the translations along the x- and y-axes being 0 or 1 pixel long. As a result, four different arrays will be produced in each iteration. After one step of translation and merging, the N_{LEAVES} value of each array is computed. Further recursive calls using these arrays will be made according to the increasing order of their N_{LEAVES} values. That is, the search space remains the same but the order in which it is searched has been changed to favor those translations that are more likely to lead us to the optimal solution. After n iterations in which $4n$ different translations have been attempted, we obtain the first four quadtrees that have been translated, as well as the required translations of the ones with the least N_{LEAVES} value at each level of iteration. The one with the least N_{LEAVES} value will be the first candidate used to establish the bound. If more than one translation results in the least number of leaf nodes, then we choose the one with the minimum translation in the y- and x-directions.

Figure 6 shows the search tree for the same image given in Figure 4. In order to make it easy for us to compare the search space explored by the new algorithm with that used by Li's algorithm, the arrangement of the configurations remains the same with the understanding that the order of evaluation is now governed by the ascending order of their leaf node counts. Whichever has the least number of leaf nodes

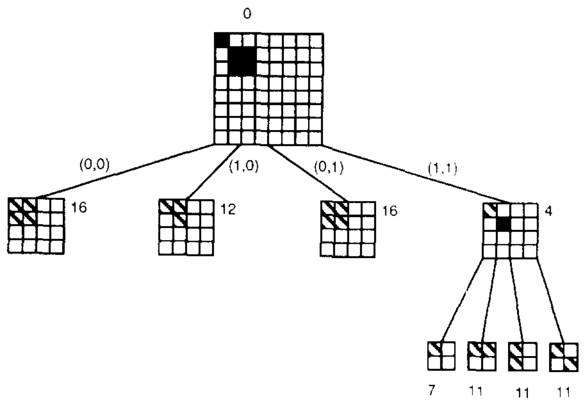

Figure 6. The search space of the configurations used by the new algorithm in determining the normalized quadtree given in Figure 4b. Each node is labeled with the number of leaf nodes in its configuration. Each edge is labeled with its translation value. Gray nodes are represented by squares with diagonal lines.

will be evaluated first. A node is labeled with its N_{LEAVES} value. Therefore, among the four sons of the root node, the one labeled with 4 will be evaluated first – that is, before those labeled with 12, 16, and 16 on the same level. After 2 iterations, which produce 8 configurations, we have found the first candidate which is the configuration labeled with 7. This happens to be the optimal solution.

For ease of reference, when this new algorithm is implemented in a program, we refer to the portion of the program that executes up to the point at which the first candidate is obtained as *step 1 of the new algorithm*. After this step, the bound established will be used in the search through the configuration search tree.

4. Analysis and empirical results

It is easy to see that step 1 of the new algorithm requires $O(4^n)$ or $O(s^2)$ time, whereas the execution time of the entire algorithm is still $O(s^2 \log s)$ time – that is, the same complexity as the original algorithm. In practice, the improvement is significant as can be seen from the results obtained in the following experiments using the QUILT geographic information system [10].

Three programs were written and run on a SUN 3/50 workstation. The first program is used to implement Li's algorithm. The second program is just the first step of the new algorithm. The third program implements the complete new algorithm. No attempt is made to optimize these programs. The execution times of the programs also include the time needed to read in the image file from the disk which takes about 2 to 3 seconds. Thus the execution times that we give do not reflect fully the actual time taken by the algorithms. Nevertheless, the improvements are quite clear.

In Table 1, we list the number of translations made to normalize three image files called floodplain, topography, and landuse. The grid size is 512×512. Their normalized quadtrees contain 5182, 24859 and 28237 leaf nodes respectively. It can be seen that the number of translations made has been reduced drastically from 87381 to about 300. In other words, almost all the work necessary to evaluate those arrays of small size is eliminated.

Table 2 lists the execution times of the three programs using the above three image files. Since step 1 of the new algorithm is implemented as a separate program, its execution time also includes the time required to read in an image file. Despite using such a crude method of measuring the execution times, the time taken by step 1 of the new algorithm is only 1/7 of the old algorithm. The complete new algorithm also reduces the total execution time of the old algorithm by more than half. This reduction is significant. It also demonstrates that evaluating the big arrays in the first four iterations in step 1 of the new algorithm is very time consuming (i.e., four arrays of size $O(2^{n+1} \times 2^{n+1})$).

Although we have shown empirically that the new algorithm outperforms the old one, further improvement is still possible. Notice that the image is stored in the NW quadrant of a big array with other quadrants of the array being used to store white pixels. By not storing these white pixels explicitly, we can reduce the space required by 3/4. With this reduction in the required space and some minor changes in the way that the pixels are being examined, the processing of the array can also be speeded up.

One striking discovery from our experiments is that the candidate chosen by step 1 of the new algorithm is always the normalized quadtree. This is a very interesting and important result. If it could be proved that step 1 always produces the normalized quadtree instead of just a candidate, then step 1 will be an optimal quadtree normalization algorithm with time bound $O(s^2)$

Unfortunately, we can demonstrate that step 1 alone cannot produce a normalized quadtree by using the image in Figure 7a. Without translation, Figure 7a will have 32 leaf nodes of pixel size and 8 black or white nodes of size 2×2. Translating the image by one pixel to the right and one pixel down results in an image that also has 32 leaf nodes of pixel size and the same number of black or white nodes of size 2×2, as shown in Figure 7b. Since the N_{LEAVES} values of both Figure 7a and Figure 7b are equal after one step of translation and merging (i.e., 8), Figure 7a will be evaluated first as it is nearer to the y-axis. Therefore, Figure 7a is chosen as the first candidate by step 1 of the new algorithm and it is used in the search for the normalized quadtree which is Figure 7b.

Table 1
Number of translations made

Image file	Old algorithm	New algorithm
floodplain	87381	120
topography	87381	228
landuse	87381	304

Table 2
Timings of the three algorithms (seconds)

Image file	Old algorithm	New algorithm	Step 1 of new algorithm
floodplain	278.0	80.5	42.0
topography	274.4	120.1	41.8
landuse	275.3	109.2	41.9

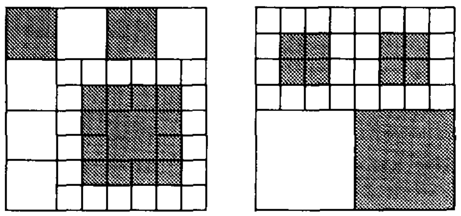
Figure 7. Sample image showing that step 1 is not sufficient by itself to produce the normalized quadtree; (a) is chosen by step 1 while (b) is the normalized quadtree.

5. Conclusion

An improvement to a well known quadtree normalization algorithm [5] has been described and analyzed. Conceptually, the new algorithm consists of 2 steps. Although step 1 of the new algorithm has been found to be sufficient by itself to produce the normalized quadtree very quickly for the three image files that were tested, we have shown that it fails to guarantee an optimal result in general. Nevertheless, it is still of interest to users who wish to save some disk space used by images stored in quadtree files without paying the high computing cost normally associated with quadtree normalization.

References

[1] Foley, J.D., A. van Dam, S.K. Feiner and J.F. Hughes (1990). *Computer Graphics: Principles and Practice.* Addison-Wesley, Reading, MA, second edition.

[2] Hunter, G.M. (1978). Efficient Computation and Data Structures for Graphics. PhD thesis, Department of Electrical Engineering and Computer Science, Princeton University, Princeton, NJ.

[3] Hunter, G.M. and K. Steiglitz (1979). Operations on images using quad trees. *IEEE Trans. Pattern Anal. Machine Intell.*, 1 (2), 145–153.

[4] Klinger, A. (1971). Patterns and search statistics. In: J.S. Rustagi, Ed. *Optimizing Methods in Statistics.* Academic Press, New York, 303–337.

[5] Li, M., W.I. Grosky and R. Jain (1982). Normalized quadtrees with respect to translations. *Computer Graphics and Image Processing* 20 (1), 72–81.

[6] Rosenfeld, A. and A.C. Kak (1982). *Digital Picture Processing.* Academic Press, New York, second edition.

[7] Samet, H. (1982). Neighbor finding techniques for images represented by quadtrees. *Computer Graphics and Image Processing* 18 (1), 37–57.

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

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

[10] Shaffer, C.A., H. Samet and R.C. Nelson (1990) QUILT: a geographic information system based on quadtrees. *Internat. J. Geographical Information Systems* 4 (2), 103–131.

[11] Shneier, M. (1981). Calculations of geometric properties using quadtrees. *Computer Graphics and Image Processing* 16 (3), 296–302.