Acta Informatica 30, 425–439 (1993)

Decomposing a window into maximal quadtree blocks*

Walid G. Aref and Hanan Samet

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

Received October 1, 1991/February 15, 1993

Abstract. Window operations serve as the basis of a number of queries that can be posed in a spatial database. Examples of window-based queries include the exist query (i.e., determining whether or not a spatial feature exists inside a window), the report query (i.e., report the identity of all the features that exist inside a window), and the select query (i.e., determine the locations covered by a given feature inside a window). Often spatial databases make use of a quadtree decomposition, which yields a set of maximal blocks, to enable the features to be accessed quickly without having to search the entire database. One way to perform a window query is to decompose the window into its maximal quadtree blocks. An algorithm is described for decomposing a two-dimensional window into its maximal quadtree blocks 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).

1 Introduction

A central operation in spatial databases is the window operation. A window is the region specified by the cross-product of two closed intervals over the integers. This operation serves as a building block for a number of queries and as a component of other queries. Usually, spatial features span a wide feature space. However, users are more interested in viewing or querying only portions of the feature space instead of the whole space. Extracting parts of the space to work with in subsequent operations is done by windowing. Given a window w, some examples of window-based queries are: report all features inside w, intersect feature f with feature b only inside w, determine if feature f exists in w, etc. (e.g., [1]).

There are numerous data structuring techniques in use for representing spatial data. Of interest here is the quadtree data structure which is based on the principle of *recursive regular decomposition* of space into a maximal set

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

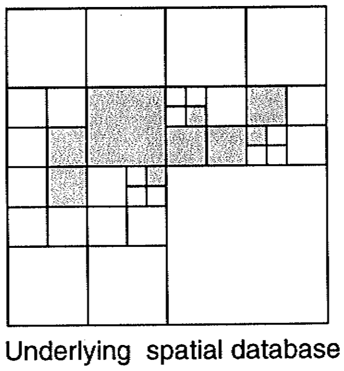


Fig. 1. An example region quadtree

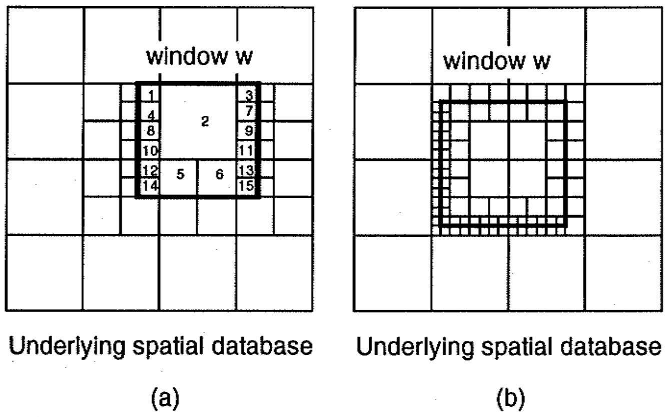


Fig. 2a, b. Possible decompositions of **a** a 12 × 12 window, and **b** a 13 × 13 window into maximal quadtree blocks

of blocks whose sides are of size power of two and are placed at predetermined positions (for a comprehensive discussion of quadtrees and other hierarchical data structures, see [4, 5]). For example, the *region quadtree* data structure [3] is based on the successive subdivision of a bounded image array into four equal-sized quadrants. If the array does not consist entirely of 1 s or entirely of 0 s (i.e., the region does not cover the entire array), then it is subdivided into quadrants, subquadrants, and so on, until blocks are obtained that consist entirely of 1 s or entirely of 0 s; that is, each block is entirely contained in the region or entirely disjoint from it. Figure 1 shows an example quadtree decomposition of space for a given region.

Our strategy in answering window queries is to decompose the window operation into sub-operations over smaller window partitions. These sub-operations should be easier to perform on the smaller partitions if the partitions are chosen carefully. Since we are interested here in quadtree representations of spatial features, one good window decomposition to consider is to partition the window into quadtree blocks. For example, Fig. 2 shows possible quadtree

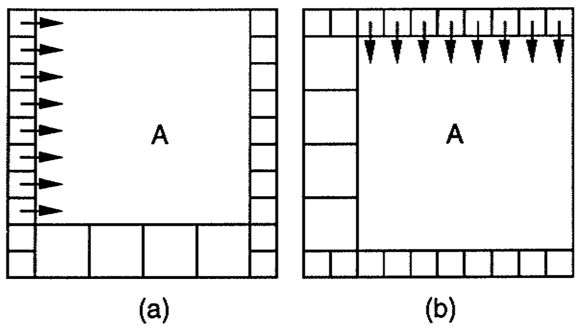


Fig. 3a, b. In **a** and **b**, maximal block A is accessed more than once by the smaller blocks to its left and top, respectively

decompositions of two windows. Notice that the decomposition varies depending on the location and size of the window.

In this paper, we concentrate on the efficient decomposition of a window into its maximal blocks. This is useful in implementing the window operation for use with a feature space that is represented as a region quadtree. More specifically, given a window, say *w*, we describe a bottom-up algorithm for decomposing *w*'s internal region into maximal quadtree blocks. Any window operation (e.g., retrieval, clipping, etc.) can be implemented on top of this window decomposer which generates the maximal quadtree blocks inside the window.

2 The algorithm

A window is stored implicity in a region quadtree [3, 5]. It can be conceptualized as having black nodes in its interior and white nodes outside it. Note that we do not need to store the window explicitly in a quadtree structure since we can generate all the maximal quadtree blocks inside the window without building the quadtree. The generation process only requires that the location and size of the window be specified. It works for an arbitrary rectangular window (i.e., it need not be square although in this paper, for simplicity, we assume it is square). This process is controlled by procedure `WINDOW_GEN` (given in the Appendix). It represents each block by a record of type `block` containing the coordinate values of its upper-left corner and its length.

Achieving a low execution time bound for the window generation process is non-trivial. In particular, we want to visit and process each maximal quadtree block only once. Also, we want to avoid generating non-maximal blocks (or at least generate a bounded number of them as in the case of our algorithm), since they introduce redundant work that can be avoided by processing only the maximal blocks that contain them.[1] Also, it is preferable to process each maximal block only once regardless of its size. Figure 3 illustrates some of the aforementioned cases.

Procedure `WINDOW_GEN` scans the window row-by-row (in the block domain rather than in the pixel domain), and visits the blocks within it that have not

[1] Note that there are $O(n^2)$ non-maximal quadtree blocks inside an $n \times n$ window.

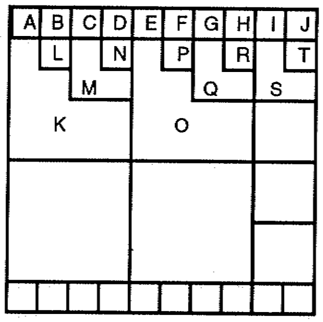


Fig. 4. The neighboring blocks to the south of blocks A-J in a 10×10 window. Blocks L, M, N, P, Q, R, and T are non-maximal, while blocks K, O, and S are maximal

been visited in previous scans. During this process, for each block that is visited, say B in row r, procedures GEN_SOUTHERN_MAXIMAL and MAX_BLOCK (given in the Appendix) generate B's maximal southern neighboring blocks. WINDOW_GEN also ensures that any of the remaining columns of row r that lie within B are skipped. For example, consider the 12×12 window in Fig. 2a, where six scans are needed to cover it with maximal quadtree blocks. The blocks are labeled with numbers that correspond to the order in which they have been visited. The first scan visits blocks 1, 2, and 3; the second scan visits blocks 4, 5, 6, and 7; the remaining scans visit blocks 8 and 9; 10 and 11; 12 and 13; and 14 and 15. Notice that once blocks 5 and 6 have been visited, their columns (i.e., 2–5) have been completely processed.

Observe that we are always generating maximal neighboring blocks, or at least a bounded number of non-maximal neighboring blocks. An example of this situation arises when processing blocks A-J in the first row of the 10×10 window in Fig. 4. Each of blocks B, C, D, F, G, H, and J can generate at most one non-maximal neighboring block. Even though these non-maximal blocks are generated, procedure WINDOW_GEN makes sure that they are skipped by the next scan since they are subsumed (i.e., contained) in the previously processed maximal block in the scan. For example, when scanning block K in Fig. 4, blocks L, M, and N are skipped since they are contained in it. This is easy to detect because for each block we know the coordinate values of its upper-left corner and its size.

Procedure WINDOW_GEN uses variable CurrentList to keep track of the blocks comprising the row currently being processed. It also uses variable NextList to keep track of the list of the blocks to be processed on the next scan (i.e., the maximal southern neighbors) that it is constructing. Once a row has been completely processed, CurrentList is set to NextList and the next row is processed. WINDOW_GEN terminates whenever processing a row does not result in any new entries in NextList (i.e., it is empty).

When WINDOW_GEN processes block B in CurrentList, it checks if B is contained in the most recently encountered maximal block, say M. If B is contained in M, then B is non-maximal and is skipped. If B is not contained in M, then it is maximal (see Lemma 6). In this case, B's southern neighbors are generated by GEN_SOUTHERN_MAXIMAL, and B replaces M as the last maximal block encountered. Notice that the first block in CurrentList is always maximal (see Lemma 5).

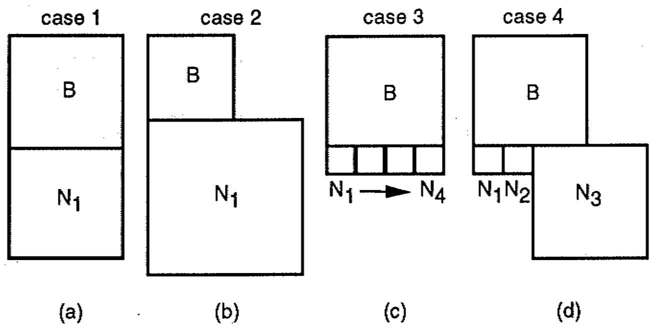


Fig. 5. a, **b**, and **c** are possible block/southern-neighbor pairs; **d** cannot occur in a quadtree decomposition

Procedure `GEN_SOUTHERN_MAXIMAL` generates the southern neighbors (maximal blocks) N_1 through N_m for each maximal block B in `CurrentList` that is not contained in a previous maximal block. A number of cases are possible as illustrated in Fig. 5. If $m=1$, then N_1 is greater than or equal to B. Otherwise, the total width of blocks N_1 through N_m is equal to that of B. It is impossible for the total length to exceed that of B unless there is only one neighbor (see Fig. 5b). Procedure `MAX_BLOCK` takes as its input a window w and the values of the x and y coordinates of a pixel, say (col,row), and returns the maximal block in w with (col,row) as its upper-leftmost corner. The resulting block has width 2^s where s is the maximum value of i ($0 \leqq i \leqq \log T$, where $T \times T$ is the size of the image space) such that $row \bmod 2^i = col \bmod 2^i = 0$ and the point $(row+2^i, col+2^i)$ lies inside w. Notice that all coordinate values assume that the origin is in the upper-left corner and that x and y increase towards the right and downwards, respectively.

3 Correctness

By the definition of maximal quadtree blocks inside the window and by the quadtree decomposition rules, we know that maximal blocks inside a window do not overlap. However, proving that the algorithm is correct involves showing that the execution of the algorithm generates a list of maximal blocks that lie entirely inside the window and that cover each point inside the window. In other words, each point p inside window w is covered by one maximal block that is generated through the execution of the algorithm. This is facilitated by using the concept of a solid boundary. A *solid boundary* at point p is defined to be a vertical or horizontal line segment passing through point p which is part of a northern or western boundary of the window, or an eastern or southern boundary of a maximal block. For example, consider Fig. 6a which shows two solid boundaries passing through point p. Line segment b_1 is the northern boundary of window w, while line segment b_2 is the eastern boundary of maximal block B. Figure 6b shows point p cornered between two solid boundaries that belong to maximal blocks B and C.

Lemma 1 *If two perpendicular solid boundaries pass through point p inside window w, then applying a maximal block computation at p generates a maximal block of w.*

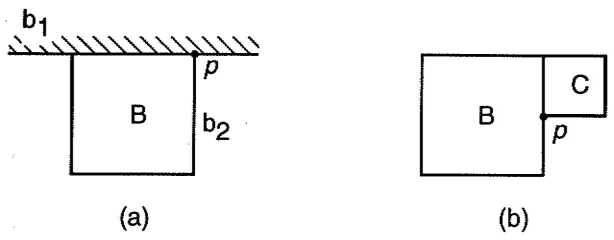


Fig. 6. a point p is the intersection of the window boundary and the side of a maximal block; **b** point p is at the corner of the sides of two maximal blocks

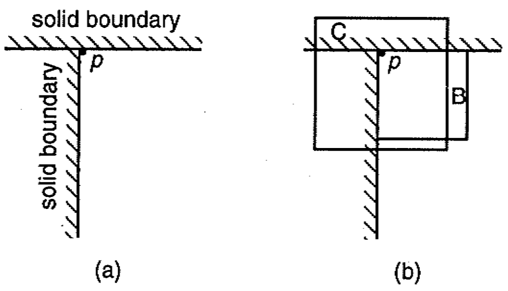


Fig. 7. a point p is the intersection of two solid boundaries; **b** cannot occur in a quadtree decomposition

Proof. Consider point p inside w such that there are solid boundaries to the north and west of p as shown in Fig. 7a. Applying a maximal block computation at point p (i.e., executing procedure `MAX_BLOCK`) generates a block, say B, inside w. B is the largest block that can cover p. Otherwise, if p is covered by another maximal block C of the same or bigger size than B (and C is not the same as B), then C should interleave with the solid boundaries of p as shown in Fig. 7b (otherwise, if C and B have the same upper-left corner but C is bigger than B, then C should have been generated by the maximal block computation initiated at p). The case of C interleaving with the solid boundaries of p implies that either C lies partly outside the window (if one of the interleaved solid boundaries is a window boundary) or that C overlaps with another maximal block (if one of the interleaved solid boundaries is a maximal block boundary). However, both of the possibilities contradict the definition of a maximal block (i.e., that it has to lie inside the window and that maximal blocks do not overlap), and thus C and B cannot be different. Since B is generated by a maximal block computation, B must be the largest block inside w that can cover point p. Therefore, B is maximal. □

Lemma 2 *If two perpendicular solid boundaries pass through any point p inside window w, then p is visited by the algorithm.*

Proof. Since maximal blocks do not overlap, point p inside w can be bounded by solid boundaries in one of five cases as illustrated in Fig. 8a–e. It is easy to see that in each case point p is visited by the algorithm, and that a maximal

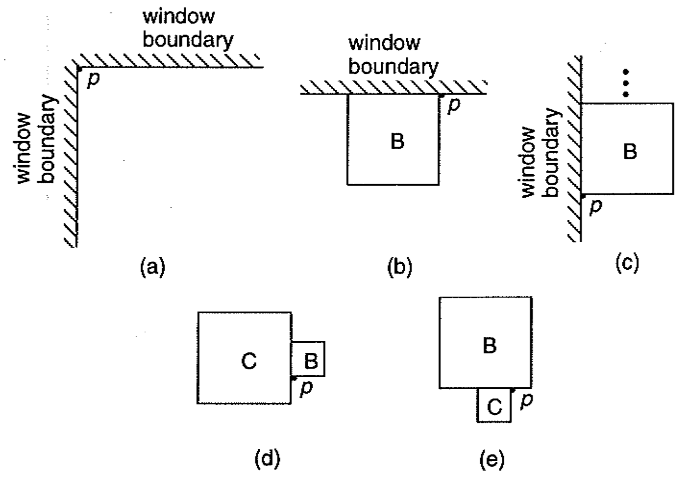


Fig. 8. **a** Two window boundaries border point p; **b**, **c** one window boundary and a side of a maximal block border p; **d**, and **e** sides of two maximal blocks border p

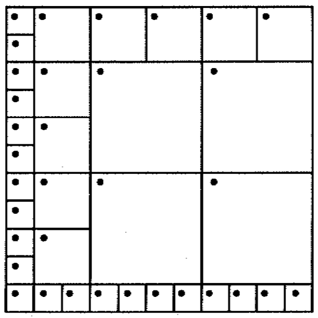
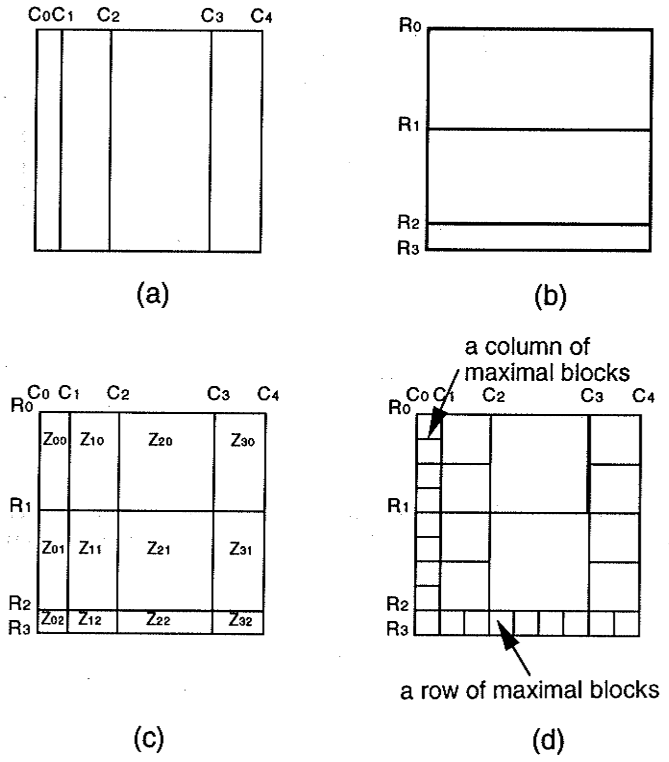


Fig. 9. Marked points are bounded by two solid boundaries and are visited by the algorithm. Each point corresponds to a maximal block

block computation is applied in each case (thereby generating a maximal block cornered at p as shown in Lemma 1). In cases a and b, p is visited in the first iteration of the main loop of WINDOW_GEN (i.e., while processing the first row of the window where we generate the southern neighbors of the fictitious block having a width equal to that of the window). In cases c–e, p is visited when generating the southern neighbors of the maximal block B (in cases c and d, p is visited when generating the first southern neighbor of B, while in case e p is visited when generating the block to the right of block C to the south of B). □

Theorem 1 *Each point inside a window is covered by one and only one maximal block generated by the algorithm.*

Proof. Given a window w and its decomposition into maximal blocks (e.g., Fig. 9), then all the points at the upper-left corner of the maximal blocks inside w correspond to points with two perpendicular solid boundaries. We know



C_0C_1 C_2 C_3 C_4
R_0 R_1 R_2 R_3
(a)
(b)
Z_{00} Z_{10} Z_{20} Z_{30}
Z_{01} Z_{11} Z_{21} Z_{31}
Z_{02} Z_{12} Z_{22} Z_{32}
a column of maximal blocks
a row of maximal blocks
(c)
(d)

Fig. 10. **a** The window is divided into vertical strips; **b** horizontal strips; **c** maximal zones; **d** the relation between maximal blocks and maximal zones

from Lemma 2 that all these points are visited by the algorithm. Moreover, from Lemma 1 we know that maximal blocks are generated at each of these points. Therefore, all the maximal blocks inside the window are visited and generated by the algorithm. Now, making use of the fact that maximal blocks cover the whole window and that they do not overlap, we have that every point inside the window is covered by one and only one maximal block. □

Theorem 1 is not sufficient to prove the algorithm's correctness. It remains to show that non-maximal blocks are handled properly by the algorithm. In particular, we need to show that first block in each scan of the algorithm is a maximal block and that non-maximal blocks are processed by the algorithm immediately after the maximal blocks that contain them and nowhere else. In this case, the way the algorithm detects and suppresses non-maximal blocks would be correct.

In order to carry out the proofs, we first introduce the following concepts and definitions. Assume a window having (c, r) as the coordinate values of its upper-left corner with height w_h (i.e., in the y direction) and width w_w (i.e., in the x direction). First, let us look at the x direction. Processing along the width w_w, we subdivide the window into p vertical strips with (c_i, r) $(0 \leqq i \leqq p)$ as coordinate values of their upper-left corner where $c_0 = c$, and $c_i = c_{i-1} + 2^j$ such that $c_{i-1} \bmod 2^j = 0$ and $c_{i-1} \bmod 2^{j+1} \neq 0$ and $c_{i-1} + 2^j \leqq c + w_w$. p is defined so that $c_p = c + w_w$. An example of such a decomposition into vertical strips is shown in Fig. 10a.

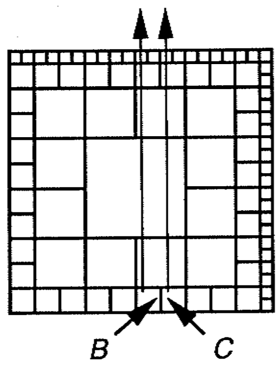


Fig. 11. Blocks B and C are in the same row of a maximal zone. There are five blocks between each of B and C and the northern border of the window

We now subdivide the window into horizontal strips in the same way. In particular, we have q horizontal strips with (c, r_i), $(0 \leqq i \leqq q)$ as the coordinate values of their upper-left corner where $r_0 = r$ and $r_i = r_{i-1} + 2^j$ such that $r_{i-1} \bmod 2^j = 0$ and $r_{i-1} \bmod 2^{j+1} \neq 0$ and $r_{i-1} + 2^j \leqq r + w_h$. q is defined so that $r_q = r + w_h$. An example of such a decomposition into horizontal strips is shown in Fig. 10b.

Now we define the term *maximal zones* as follows. A maximal zone, say Z_{ij}, is the region between the vertical strips having c_i and c_{i+1} as the x-coordinate values of their upper-left corner and the horizontal strips having r_j and r_{j+1} as the y-coordinate values of their upper-left corner where $0 \leqq i < p$ and $0 \leqq j < q$. The result is a decomposition of the window into a two-dimensional grid of maximal zones. An example of a decomposition into maximal zones is shown in Fig. 10c.

In the interest of brevity, we give below some properties of maximal zones without proving them. They are illustrated in Fig. 10d.

Property 1 *Each maximal block inside the window is entirely contained in one and only one maximal zone.*

Property 2 *All the maximal blocks inside a maximal zone are of the same size.*

Property 3 *A maximal zone contains either one maximal block, or one row of maximal blocks, or one column of maximal blocks.*

Property 4 *All the southern neighbors of a block lie in one maximal zone.*

Lemma 4 *All the maximal blocks arranged in a row inside a maximal zone are processed in the same iteration of the main loop of procedure* `WINDOW_GEN`.

Proof. To prove this we need to show that any two maximal blocks, say B and C, lying in the same row inside a maximal zone, say z, have the same number of neighboring northern maximal blocks if we start counting from the northern boundary of the window and descend in the southern direction until we reach B or C. For example, consider Fig. 11 where blocks B and C lie in the same row inside a maximal zone and there are five northern maximal blocks between each of B and C and the northern boundary of the window. Assume that there are l maximal zones between the northern boundary of the

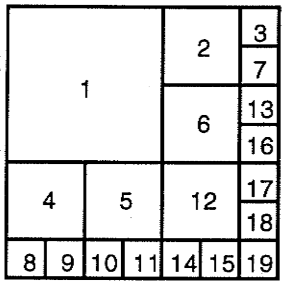


Fig. 12. Block 14 is the first in the list in the fourth traversal, while block 17 is the first in the list in the fifth traversal

window and maximal zone z. By Property 4, the southern neighbors of any of the blocks inside the l zones can only be inside one of these l zones. Also, recall from Property 3 that a maximal zone may contain either one maximal block, or one row of maximal blocks, or one column of maximal blocks. Each one of the l zones to the north of z falls into any one of these three cases. It is easy to see that no matter how each maximal zone of the l zones is filled with maximal blocks, blocks B and C will always have the same number of neighboring northern maximal blocks when we start counting from the northern boundary of the window. Now, since blocks B and C have the same number of northern blocks above them, each execution of the main loop of procedure `WINDOW_GEN` will process one level of these northern neighbors of B and C and will generate the southern blocks in the following level until blocks B and C are reached. Therefore, blocks B and C will also be processed by the same iteration of the main loop of procedure `WINDOW_GEN`, and thus they will be processed in one scan. □

Lemma 5 *The first element in the current scan (i.e., the first block in* `CurrentList`*) is always a maximal block.*

Proof. When processing the first row, both boundaries of the first block, say B, are window boundaries and hence are solid boundaries. Therefore, B is maximal. The first southern neighbor of B that is generated (and hence the first element in `CurrentList` of the next scan), say C, is also maximal since it is cornered by two perpendicular solid boundaries (i.e., the western boundary of the window and southern boundary B which is maximal). This is true for all southern neighboring blocks that are generated and that have the western boundary of the window as one of their solid boundaries. One more case that has to be considered arises when the first several vertical strips are completely processed (e.g., Fig. 12). From Lemma 4 we know that a row of maximal blocks inside a maximal zone is always processed at the same time. Therefore, the first element in `CurrentList` is always the left-most block in a maximal zone (which is always maximal since its upper and left boundaries are solid). □

Lemma 6 *Non-maximal blocks appear in* `CurrentList` *immediately following the maximal blocks that contain them and nowhere else.*

Proof. Maximal zones mean that non-maximal blocks can be generated only when a zone contains a row of maximal blocks, say $B_1 \dots B_l$, and each of them

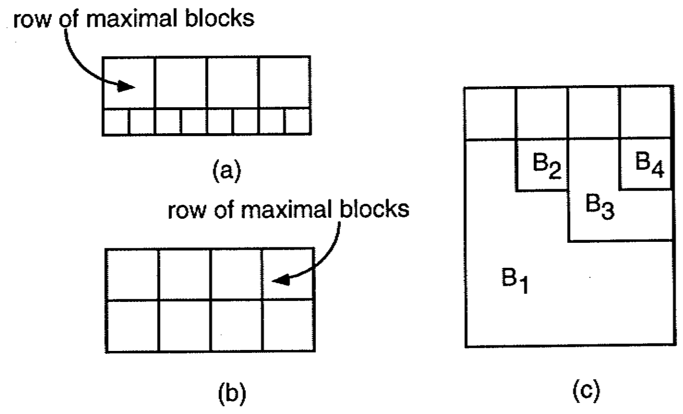


Fig. 13. a All southern neighbors are of smaller size (all are maximal); **b** southern neighbors are of the same size (all are maximal); **c** B_2, B_3, and B_4 are non-maximal. The southern neighbor B_1 is maximal and contains B_2, B_3, and B_4

generates its southern neighbors. Some of these southern neighbors can be non-maximal. In this case, all blocks except B_1 can generate non-maximal blocks since from Lemma 1 we have that B_1's southern neighbor(s) is(are) always maximal. From Lemma 4 we know that blocks B_1 through B_l are added to `CurrentList` at the same time and in the same relative order (since scanning and processing is from left to right). For each block $B_i (1 < i \leqq l)$ one of the three cases illustrated in Fig. 13 can occur. In each case, the positions of the non-maximal blocks in `CurrentList` immediately follow the maximal blocks that contain them. □

Lemma 6 is important because it means that non-maximal blocks appear immediately adjacent to the maximal blocks. We can detect their presence, and eliminate them without wasting work in generating their southern neighbors. Also, since they are adjacent, we only need to store one maximal block (the current one) and compare it with the blocks next to it in the list, possibly the non-maximal ones, until another maximal block is encountered. This enables us to detect non-maximality in constant time.

From the above lemmas we have the following theorem.

Theorem 2 *The window decomposition algorithm* (`WINDOW_GEN`) *generates all the maximal blocks inside the window, and only maximal blocks, and hence is correct.* □

4 Complexity analysis of the window decomposition algorithm

In this section we derive the worst case execution time complexity of the window decomposition algorithm. Our analysis assumes that the window is square.

It is known that in the worst case, the number of maximal quadtree blocks inside a square window of size $n \times n$ is $N = 3(2n - \log n) - 5$ [2]. What remains to be done is to compute the cost of determining the maximal blocks comprising the window. This consists of the work, say T_m, to generate a maximal block, say B, and the work that is wasted, say T_w, in generating southern neighboring blocks of B that are non-maximal. Therefore, the total execution time of the window decomposition algorithm is $N \cdot (T_m + T_w)$.

Given a point (x, y) in a $T \times T$ space, there can be at most $\log T + 1$ different blocks of size 2^i $(0 \leqq i \leqq \log T)$ with (x, y) as their upper-left corner. We use binary search through this set of blocks to determine the maximal block inside the window. Thus T_m is $O(\log \log T)$.

To compute T_w, we need to show that each maximal block inside the window is generated once, and that only a limited number of non-maximal blocks are generated. We say that the work required to generate blocks that are not maximal with respect to a particular window is *wasted*. Such blocks are ignored (i.e., bypassed) in subsequent processing. For example, the work in generating the southern neighbors of blocks B, C, D, F, G, H, and J (i.e., L, M, N, P, Q, R, and T, respectively) in Fig. 4 is wasted. This is formulated and proved in the following two Lemmas.

Lemma 7 *Each maximal block inside window w is generated only once.*

Proof. In Theorem 1, we proved that every maximal block inside window w is generated by the algorithm. To show that it is generated only once we observe that each block processed by the algorithm generates only its southern neighbors. The facts that non-maximal blocks are bypassed by the algorithm, and that maximal blocks do not overlap, mean that each maximal block is generated as the southern neigbor of only one other maximal block. □

Lemma 8 *Each block visited by the algorithm can waste at most* $O(\log \log T)$ *work.*

Proof. Assume that the visited block B generates wasted work. We show that this work takes $O(\log \log T)$ time. B can generate neighboring southern maximal blocks that are either smaller or larger. When the size of the neighboring block is greater than or equal to the size of B, then the algorithm takes $O(\log \log T)$ time regardless of whether or not it is wasted and the Lemma holds. When more than one southern block is generated (this number can be of the same order as the size of B), we need to show that all the generated southern blocks are maximal, and cannot be bypassed, i.e., they are not wasted work. We shall prove this by contradiction. Assume that B generates more than one southern block and that all of them are bypassed (not visited) in subsequent processing. It should be clear that due to the nature of the quadtree decomposition of space, either all of them are visited, or all are bypassed. Our assumption means that there exists a block C whose width is greater than the total width of B's southern neighbors. Let (B_x, B_y) and (C_x, C_y) be the locations of the upper-leftmost pixels of blocks B and C, respectively. Also, let B_s and C_s be the widths of blocks B and C, respectively. It is easy to see that the fact that B and C are maximal blocks that are southern neighbors of other visited maximal blocks means that $C_y = B_y + B_s$. The fact that $C_s > B_s$ means that the lower-rightmost pixel of C is at $(C_x + C_s - 1, C_y + C_s - 1)$ which is in the window. Therefore, $(B_x + B_s - 1, B_y + B_s - 1)$ which is the lower-rightmost pixel of B's southern neighbor of equal size, say D, is also in the window. This means that D is B's neighboring

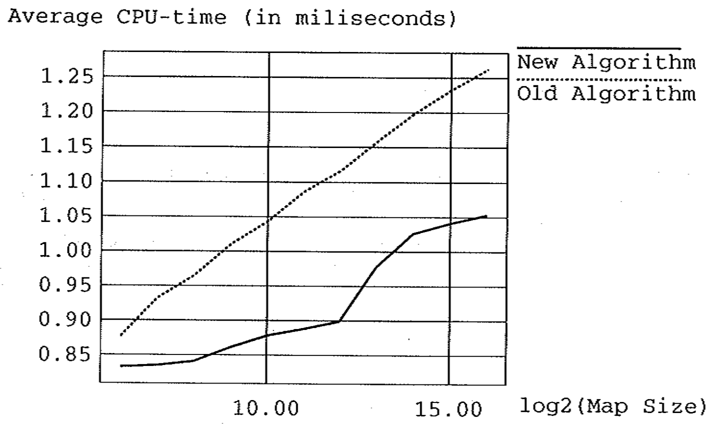


Fig. 14. Comparison of our bottom-up algorithm and a top-down window decomposition algorithm [6] using 10000 random rectangles

southern maximal block. However, this contradicts the existence of more than one such block. Thus the assumption that all of the southern neighboring blocks of B are bypassed is invalid. Therefore, no work is wasted in generating B's southern neighbors in this case, and the Lemma holds. $\square$

Because there are at most $O(n)$ visited blocks, the total time wasted is $T_w = O(n \log\log T)$. Combining the results for T_m and T_w means that we have proved the following theorem.

Theorem 3 *Given an $n \times n$ window in a $T \times T$ image, the worst case execution time for the window decomposition algorithm is $O(n \log\log T)$.* $\square$

5 Empirical results

The algorithm has been implemented on a Sun workstation (Sparc I). We compared it experimentally with an alternative top-down approach [6]. We randomly generated 10000 rectangles of a given window area and computed the average CPU time to decompose each rectangle using the two approaches. Figure 14 shows the result of the comparison. Our new algorithm proved to be practical and faster than the approach in [6].

6 Conclusions

A bottom-up algorithm for decomposing a window into maximal quadtree blocks has been presented. The algorithm serves as the underlying mechanism on top of which query answering algorithms can be built. We plan to investigate the algorithm's usage for answering window queries in a disk-based spatial database environment where the algorithm's I/O behavior is of greater concern.

7 Appendix

Pseudo code for the window decomposition routine

```
pointer list procedure WINDOW_GEN(W);
/* Generate and return the maximal blocks that comprise window W. The window is represented by
   a record of type window with four fields ROW, COL, WIDTH, and HEIGHT corresponding to the y
   coordinate value of its upper-leftmost pixel, the x coordinate value of its upper-leftmost
   pixel, its width, and its height, respectively. The blocks are added by repeatedly finding
   southern neighbors and keeping them in a linked list whose first and last elements are
   pointed at by NextList and EndNextList, respectively. The block currently being processed
   is pointed at by CurrentList. CurrentList is initialized to contain one block of length
   WIDTH with an upper-leftmost pixel at (COL,ROW-WIDTH). */
begin
  value pointer window W; /* declaration of arguments to WINDOW_GEN */
  pointer list MaxBlockList,EndMaxBlockList; /* contains the resulting blocks */
  pointer list CurrentList,NextList,EndNextList; /* local variable definitions */
  pointer block CurrentBlock;
  NextList:=NIL;
  CurrentBlock:=create(block);
  ROW(CurrentBlock):=ROW(W)-WIDTH(W);
  COL(CurrentBlock):=COL(W);
  LEN(CurrentBlock):=WIDTH(W);
  GEN_SOUTHERN_MAXIMAL(NextList,CurrentBlock,W,EndNextList);
  if null(NextList) then return(NIL)
  else
    begin
      MaxBlockList:=EndMaxBlockList:=create(list);
      DATA(MaxBlockList):=DATA(NextList);
      NEXT(MaxBlockList):=NIL;
      NextList:=NEXT(NextList);
      do
        begin
          CurrentList:=NextList;
          NextList:=EndNextList:=NIL;
          while not(null(CurrentList)) do
            begin
              CurrentBlock:=DATA(CurrentList);
              NEXT(EndMaxBlockList):=create(list);
              EndMaxBlockList:=NEXT(EndMaxBlockList);
              DATA(EndMaxBlockList):=CurrentBlock;
              CurrentList:=NEXT(CurrentList);
              while not(null(CurrentList)) and CONTAINED(DATA(CurrentList),CurrentBlock) do
                CurrentList:=NEXT(CurrentList); /* CONTAINED is not given here. */
              GEN_SOUTHERN_MAXIMAL(NextList,CurrentBlock,W,EndNextList);
            end;
        end
      until null(NextList);
      return(MaxBlockList);
    end;
end;

procedure GEN_SOUTHERN_MAXIMAL(NextList,B,W,EndNextList);
/* Find the maximal blocks to the south of block B in window W and add them to the end
   of the list having fields DATA and NEXT which starts at NextList and ends at EndNextList.
   If NextList is NIL, then set it to the first block that is added. All blocks are represented
   by records of type block with three fields ROW, COL, LEN corresponding to the y coordinate
   value of its upper-leftmost pixel, the x coordinate value of its upper-leftmost pixel,
   and its length, respectively. */
begin
  reference pointer list NextList,EndNextList; /* declaration of arguments */
  value pointer block B;
  value pointer window W;
  pointer block T; /* local variable */
  integer LEFT,RIGHT;
  T:=MAX_BLOCK(ROW(B)+LEN(B),COL(B),W); /* Allocate first block. */
  if null(T) then return
  else
    begin /* Allocate first block and initialize start of NextList. */
      if null(NextList) then NextList:=EndNextList:=create(list)
      else EndNextList:=NEXT(EndNextList):=create(list);
      DATA(EndNextList):=T;
      LEFT:=COL(B)+LEN(T);
      RIGHT:=COL(B)+LEN(B);
      while LEFT < RIGHT do /* Generate rest of blocks. */
        begin
          EndNextList:=NEXT(EndNextList):=create(list);
          DATA(EndNextList):=MAX_BLOCK(ROW(B)+LEN(B),LEFT,W);
          LEFT:=LEFT+LEN(DATA(EndNextList));
        end;
      NEXT(EndNextList):=NIL; /* Set pointer at the end of the list to NIL. */
    end;
end;

pointer block procedure MAX_BLOCK(ROW,COL,W);
/* Find the largest square block inside window W for which (ROW,COL) is the first
   (upper-leftmost) pixel. The length of the side of the block is a power of 2. */
begin
  value integer ROW,COL;
  value pointer window W;
  integer I;
  pointer block b;
  I:=0;
  while IN_WINDOW(ROW+2**I-1,COL+2**I-1,W) and ((ROW mod 2**I)=0) and ((COL mod 2**I)=0)
    do I:=I+1; /* IN_WINDOW is not given here. */
  if I=0 then return(NIL) /* No maximal block exists. */
  else
    begin
      B:=create(block);
      ROW(B):=ROW;
      COL(B):=COL;
      LEN(B):=2**(I-1);
      return(B);
    end;
end;
```

References

1. Aref, W.G., Samet, H.: 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 (PODS), pp. 265–272, Nashville, TN, April 1990
2. Dyer, C.R.: The space efficiency of quadtrees. Comput. Graph. Image Process. **19**(4), 335–348 (1982)
3. Klinger, A.: Patterns and search statistics. In: Rustagi, J.S. (ed.) Optimizing methods in statistics, pp. 303–337. New York: Academic Press 1971
4. Samet, H.: Applications of spatial data structures: computer graphics, image processing, and GIS. Reading, MA: Addison-Wesley 1990
5. Samet, H.: The design and analysis of spatial data structures. Reading, MA: Addison-Wesley 1990
6. Samet, H., Rosenfeld, A., Shaffer, C., Nelson, R., Huang, Y., Fujimura, K.: Application of hierarchical data structures to geographical information systems: Phase IV. Technical Report CS-1578, University of Maryland, College Park, MD, December 1985