Programming Assignment 1: A Data Structure For VLSI Applications[1]

Abstract

In this assignment you are required to implement an information management system for handling data similar to that used in VLSI (very large scale integration) applications. In such an environment the primary entities are small rectangles and the problem in which we are interested is how to manage a large collection of them. In the following we trace the development of a variant of the quadtree data structure that has been found to be useful for such a problem. Your task is to implement this data structure in such a way that a number of operations can be efficiently handled. An example JAVA applet for the data structure can be found on the home page of the class.

This assignment is divided into four parts. PASCAL is the preferred programming language although you may use C or C++. For the first two parts, you must read the attached description of the problem and data structure. A detailed explanation of the assignment including the specification of the operations which you are to implement is found at the end of the description. After you have done this, you are to turn in a proposed implementation of the data structure using PASCAL's (or C or C++) record (structure) definition facility. One week later you must turn in a PASCAL (or C or C++) program for the command decoder (i.e., scanner for the commands corresponding to the operations which are to be performed on the data structure). For the third part, you are to write a PASCAL (or C or C++) program to implement the data structure and operations (1)-(8). For the fourth part, you are to implement operations (9)-(13). Operations (14)-(16) are optional and you will get extra credit if you turn them in with part four.

[1]Copyright ©2003 by Hanan Samet. No part of this document may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the express prior permission of the author.

1 Region-Based Quadtrees

The quadtree is a member of a class of hierarchical data structures that are based on the principle of recursive decomposition. As an example, consider the point quadtree of Finkel and Bentley [1] which should be familiar to you as it is simply a multidimensional generalization of a binary search tree. In two dimensions each node has four subtrees corresponding to the directions `NW`, `NE`, `SW`, and `SE`. Each subtree is commonly referred to as a quadrant or subquadrant. For example, see Figure 1.6[2] where a point quadtree of 8 nodes is presented. In our presentation we shall only discuss two-dimensional quadtrees although it should be clear that what we say can be easily generalized to more than two dimensions. For the point quadtree the points of decomposition are the data points themselves (i.e., in Figure 1.6, Chicago at location (35,40) subdivides the two dimensional space into four rectangular regions). Requiring the regions to be of equal size leads to the region quadtree of Klinger [5,6,7]. This data structure was developed for representing homogeneous spatial data and is used in computer graphics, image processing, geographical information systems, pattern recognition, and other applications. For a history and review of the quadtree representation, see pp. 1–16 in [6].

As an example of the region quadtree, consider the region shown in Figure 1.1a which is represented by a 2^3 by 2^3 binary array in Figure 1.1b. Observe that 1's correspond to picture elements (termed pixels) which are in the region and 0's correspond to picture elements that are outside the region. The region quadtree representation is based on the successive subdivision of the array into four equal-size quadrants. If the array does not consist entirely of 1's or 0's (i.e., the region does not cover the entire array), then we subdivide it into quadrants, subquadrants, ... until we obtain blocks (possibly single pixels) that consist entirely of 1's or entirely of 0's. For example, the resulting blocks for the region of Figure 1.1b are shown in Figure 1.1c. This process is represented by a quadtree in which the root node corresponds to the entire array, the four sons of the root node represent the quadrants, and the leaf nodes correspond to those blocks for which no further subdivision is necessary. Leaf nodes are said to be BLACK or WHITE depending on whether their corresponding blocks are entirely within or outside of the region respectively. All non-leaf nodes are said to be GRAY. The region quadtree for Figure 1.1c is shown in Figure 1.1d.

2 MX Quadtrees

There are a number of ways of adapting the region quadtree to represent point data. If the domain of data points is discrete, then we can treat data points as if they were BLACK pixels in a region quadtree. An alternative characterization is to treat the data points as non-zero elements in a square matrix. We shall use this characterization in the subsequent discussion. To avoid confusion with the point and region quadtrees, we call the resulting data structure an *MX quadtree* (MX for matrix).

The MX quadtree is organized in a similar way to the region quadtree. The difference is that leaf nodes are BLACK or empty (i.e., WHITE) corresponding to the presence or absence, respectively, of a data point in the appropriate position in the matrix. For example, Figure 2.30 is the $2^3 \times 2^3$ MX quadtree corresponding to the data of Figure 1.6. It is obtained by applying the mapping f such that $f(z) = z \div 12.5$ to both x and y coordinates. The result of the mapping is reflected in the coordinate values in the figure.

Each data point in an MX quadtree corresponds to a 1×1 square. For ease of notation and operation using modulo and integer division operations, the data point is associated with the lower left

[2] All numbered figures and page numbers refer to [6].

corner of the square. This adheres to the general convention followed throughout this presentation that the lower and left boundaries of each block are closed while the upper and right boundaries of each block are open. We also assume that the lower left corner of the matrix is located at (0,0). Note that, unlike the region quadtree, when a non-leaf node has four BLACK sons, they are not merged. This is natural since a merger of such nodes would lead to a loss of the identifying information about the data points, as each data point is different. On the other hand, the empty leaf nodes have the absence of information as their common property; so, four WHITE sons of a non-leaf node can be safely merged.

Quadtrees are especially attractive in applications that involve search. A typical query is one that requests the determination of all nodes within a specified distance of a given data point - e.g., all cities within 50 miles of Washington, D.C. The efficiency of quadtree-like data structures lies in their role as a pruning device on the amount of search that is required. Thus, many records will not need to be examined.

As an example we use the point quadtree of Figure 1.6 although the extension to an MX quadtree is straightforward. Suppose that we wish to find all cities within eight units of a data point with coordinate values (83,10). In such a case, there is no need to search the `NW`, `NE`, and `SW` quadrants of the root (i.e., Chicago with coordinate values (35,40)). Thus, we can restrict our search to the `SE` quadrant of the tree rooted at Chicago. Similarly, there is no need to search the `NW` and `SW` quadrants of the tree rooted at Mobile (i.e., coordinate values (50,10)).

As a further clarification of the amount of pruning of the search space that is achievable by use of the point quadtree, we make use of Figure 2.17. In particular, given the problem of finding all nodes within radius r of point A, use of the figure indicates which quadrants need not be examined when the root of the search space, say R, is in one of the numbered regions. For example, if R is in region 9, then all but its `NW` quadrants must be searched. If R is in region 7, then the search can be restricted to the `NW` and `NE` quadrants of R. For more details on MX quadtrees, see pp. 86–92.

3 CIF Quadtrees

The *MX-CIF quadtree* is a quadtree-liked data structure devised by Kedem [4] (who used the term *quad-CIF tree*) for representing a large set of very small rectangles for application in VLSI design rule checking. The goal is to rapidly locate a collection of objects that intersect a given rectangle. An equivalent problem is to insert a rectangle into the data structure under the restriction that it does not intersect existing rectangles.

The MX-CIF quadtree is organized in a similar way to the region quadtree. A region is repeatedly subdivided into four equal-size quadrants until we obtain blocks which do not contain rectangles. As the subdivision takes place, we associate with each subdivision point a set containing all of the rectangles that intersect the lines passing through it. For example, Figure 3.20 contains a set of rectangles and its corresponding MX-CIF quadtree. Once a rectangle is associated with a quadtree node, say *P*, it is not considered to be a member of any of the sons of *P*. For example, in Figure 3.20 rectangle 11 overlaps the space spanned by both nodes D and F but is only associated with node D, while rectangle 12 is associated with node F.

At this point, it is also appropriate to comment on the relationship between the MX-CIF quadtree and the MX quadtree. The similarity is that the MX quadtree is defined for a domain of points with corresponding nodes that are the smallest blocks in which they are contained. Similarly, the domain of the MX-CIF quadtree consists of rectangles with corresponding nodes that are the smallest blocks in which they are contained. In both cases, there is a predetermined limit on the level of decomposi-

tion. One major difference is that in the MX-CIF quadtree, unlike the MX quadtree, all nodes are of the same type. Thus, data is associated with both leaf and non-leaf nodes of the MX-CIF quadtree. Empty nodes in the MX-CIF quadtree are analogous to WHITE nodes in the MX quadtree. An empty node is like an empty son and is represented by a NIL pointer in the direction of a quadrant that contains no rectangles.

The set of rectangles that intersect the lines passing through a subdivision point is subdivided into two sets. For example, consider subdivision point P centered at (CX,CY) which partitions a $2 \cdot LX \times 2 \cdot LY$ area. All input rectangles that intersect the line $x = CX$ form one set and all input rectangles that intersect the line $y = CY$ form the other set. Equivalently, these sets correspond to the rectangles intersecting the y and x axes, respectively, passing through (CX,CY). If a rectangle intersects both axes (i.e., it contains the subdivision point P), then we adopt the convention that it is stored with the set associated with the y-axis.

These subsets are implemented as binary trees, which in actuality are one-dimensional analogs of the MX-CIF quadtree. For example, Figure 3.21 illustrates the binary tree associated with the x and y axes passing through A, the root of the MX-CIF quadtree of Figure 3.20. The subdivision points of the axis lines are shown by the tick marks in Figure 3.20.

Insertion and deletion of rectangles in an MX-CIF quadtree are described on pp. 202–209. The most common search query is one that seeks to determine if a given rectangle overlaps (i.e., intersects) any of the existing rectangles. This operation is a prerequisite to the successful insertion of a rectangle. Range queries can also be performed. However, they are more usefully cast in terms of finding all the rectangles in a given area (i.e., a window query). Another popular query is one that seeks to determine if one collection of rectangles can be overlaid on another collection without any of the component rectangles intersecting one another.

These two operations can be implemented by using variants of algorithms developed for handling set operations (i.e., union and intersection) in region-based quadtrees [3,8]. The range query is answered by intersecting the query rectangle with the MX-CIF quadtree. The overlay query is answered by a two-step process. First, intersect the two MX-CIF quadtrees. If the result is empty, then they can be safely overlaid and we merely need to perform a union of the two MX-CIF quadtrees. It should be clear that Boolean queries can be easily handled. An example JAVA applet for the MX-CIF quadtree data structure can be found on the home page of the class.

4 Assignment

This assignment has four parts. It is to be programmed in PASCAL or C or C++. The first part is concerned with data structure selection. The second part requires the construction of a command decoder. The third and fourth parts require that you implement a given set of operations.

The first part is to be turned in one week after this assignment has been distributed to you. It is worth 10 points. The second part is also worth 10 points. It is to be turned in two weeks after this assignment has been distributed to you. There will be NO late submissions accepted for these two parts of the assignment. While doing parts one and two you are also to start thinking and coding the program necessary to implement the operations. This should be done in such a way that the data structure is a BLACK BOX. Thus you need to specify your primitives in such a way that they are independent of the data structure finally chosen. You are strongly advised to begin implementing some of the operations. For example, you should implement an output routine so that you can see whether your program is working properly.

For the third and fourth parts of the assignment, you are to write a PASCAL (or C or C++) program to implement the data structure and the specified operations. Together they are worth 60 points. Part three consists of operations (1)-(8) given below. They are worth 30 points. Part four consists of operations (9)-(13) given below. They are worth 30 points. Operations (14)-(16) are for extra credit and are to be turned in with part four. They are worth up to 7 points apiece.

In order to facilitate grading and your task, you are to use the data structure implementation that will be given to you in class on the first meeting date after you turn in the first two parts of the assignment. For any operation that is not implemented, say `OP`, your command decoder must output a message of the form ``` ``COMMAND OP IS NOT IMPLEMENTED'' ```.

In order to facilitate your program as well as lend some realism to your task you are to implement the MX-CIF quadtree in a raster-based graphics environment. This means that you are dealing with a world of pixels. The size of the world can be varied, and is a $2^w \times 2^w$ array of pixels. As a default, you should assume $w = 7$, i.e., a size of 128×128. The pixel at the lower left corner has coordinate values (0,0) and the pixel at the upper right corner has coordinate values $(2^w - 1, 2^w - 1)$. Each pixel serves as the center of a square of size 1×1. This is the smallest unit into which our quadtrees will decompose the world. In order to simplify the project and for optional operation (16) (i.e., `LABEL` for connected component labeling) to be meaningful, we stipulate that the centroids and the distances from the centroids to the borders of the rectangles are integers. All rectangles are of size $i \times j$, where $3 \leq i \leq 2^w$ and $3 \leq j \leq 2^w$. In other words, the smallest rectangle is of size 3 by 3 and the largest is $2^w \times 2^w$. One class meeting date before the due date of each part of the project you will be informed of the availability of and name of the test data file which you are to use in exercising your program for grading purposes. You should also prepare your own test data. A sample file for this purpose will also be provided.

4.1 Data Structure Selection

You are to select a data structure to implement the MX-CIF quadtree. Turn in a definition in the form of a set of PASCAL (or C or C++) records (structures). In doing this part of the assignment you should bear in mind the type of data that is being represented and the type of operations that will be performed on it. In order to ease your task, remember that the primitive entity is the rectangle. We specify a rectangle by giving the x and y coordinate values of its centroid, and the horizontal and vertical distances from the centroid to its borders. The rest of your task is to build on this entity adding any other information that is necessary. The nature of the operations is described in Sections 4.3–4.5.

From the description of the operations you will see that a name is associated with each rectangle. At times, the operations are specified in terms of these names. Thus you will also need a mechanism to efficiently keep track of these names. It should be integrated with the rest of your data structures.

4.2 Command Decoder

You are to turn in a working command decoder written in PASCAL (or C or C++) for all the commands (including the optional ones) given in Sections 4.3–4.5. You are not expected to do error recovery and can assume that the commands are syntactically correct. All commands will fit on one line. Lengths of names are restricted to 6 characters or less and can be any combination of letters or digits (e.g., `A`, `1`, `2A`, `B33`, etc.). However, for your own safety you may wish to incorporate some primitive error handling. Test data for this part of the assignment will be found in a file specified by

the Teaching Assistant.

The output for the command decoder consists of the number of the operation (e.g., "1" for command `INIT_QUADTREE`) and the actual values of the parameters if the command has any parameters (e.g., the value of `WIDTH` for the `INIT_QUADTREE` command).

4.3 Part Three: Basic Operations

(1) Initialize the quadtree. The command `INIT_QUADTREE(WIDTH)` is always the first command in the input stream. `WIDTH` determines the length of each side of the square are covered by the quadtree. Each side has the length $2^{\texttt{WIDTH}}$. It also has the effect of starting with a fresh data set.

(2) Generate a display of a $2^{\texttt{WIDTH}} \times 2^{\texttt{WIDTH}}$ square from the MX-CIF quadtree. It is invoked by the command `DISPLAY()`. To draw the MX-CIF quadtree, you are to use the drawing routines provided. An appendix to the project description covers their use, and the utilities `SHOWQUAD` and `PRINTQUAD`, that can be used to render the output of your programs on a screen or a printer. A dashed (broken) line should be used to draw quadrant lines, but the rectangles should be solid (i.e., not dashed). Rectangle names should be output somewhere near the rectangle or within the rectangle. When this convention causes the output of a quadrant line to coincide with the output of the boundary of a rectangle, then the output of the rectangle takes precedence and the coincident part of the quadrant line is not output.

(3) List all the rectangles in the data base in alphanumerical order. This means that letters come before digits in the collating sequence. Similarly, shorter identifiers precede longer ones. For example, a sorted list is `A`, `AB`, `A3D`, `3DA`, `5`. It is invoked by the command `LIST_RECTANGLES`() and yields for each rectangle its name, the x and y coordinate values of its centroid, and the horizontal and vertical distances to its borders from the centroid. This is of use in interpreting the display since sometimes it is not possible to distinguish the boundaries of the rectangles from the display. You should list all of the rectangles in the database whether or not they have been deleted.

(4) Create a rectangle by specifying the coordinate values of its centroid and the distances from the centroid to its borders, and assign it a name for subsequent use. It is invoked by the command `CREATE_RECTANGLE(N,CX,CY,LX,LY)` where `N` is the name to be associated with the rectangle, `CX` and `CY` are the x and y coordinate values, respectively, of its centroid, and `LX` and `LY` are the horizontal and vertical distances, respectively, to its borders from the centroid. `CX`, `CY`, `LX`, and `LY` may be real or integer numbers. However, in the case of this assignment, in order for the optional operation (16) (i.e., `LABEL` for connected component labeling) to be meaningful, recall from the introduction to Section 4 that we stipulate that the centroids and the distances from the centroids to the borders of the rectangles are integers. Output an appropriate message indicating that the rectangle has been created as well as its name and endpoints. Note that any rectangle can be created — even if it is outside the space spanned by the MX-CIF quadtree.

(5) Determine whether a query rectangle intersects (i.e., overlaps) any of the existing rectangles. This operation is a prerequisite to the successful insertion of a rectangle in the MX-CIF quadtree. It is invoked by the command `RECTANGLE_SEARCH(N)` where `N` is a name of a rectangle. If the rectangle does not intersect an existing rectangle, then `RECTANGLE_SEARCH` returns a value of false and outputs an appropriate message such as ``` ``N DOES NOT INTERSECT AN EXISTING RECTANGLE'' ```. Otherwise, it returns the value true and uses the name associated with one of the intersecting rectangles (i.e., if it intersects more than one rectangle) to output one of the following two messages: ``` ``N INTERSECTS RECTANGLE [NAME OF RECTANGLE]'' ```. Note that if an endpoint of the query rectangle touches the endpoint of an existing rectangle, then `RECTANGLE_SEARCH` returns

false. You are only to check against the rectangles that are in the MX-CIF quadtree of existing rectangles, and not the rectangles that existed at some time in the past and have been deleted by the time this command is executed.

(6) Insert a rectangle in the MX-CIF quadtree. If the rectangle intersects an existing rectangle, then do not make the insertion and report this fact by returning the name of the intersecting rectangle. Also, if any part of the rectangle is outside the space spanned by the MX-CIF quadtree, then do not make the insertion and report this fact by a suitable message such as `INSERTION OF RECTANGLE N FAILED AS N LIES PARTIALLY OUTSIDE SPACE SPANNED BY MX-CIF QUADTREE.` Otherwise, return the name of the rectangle that is being inserted as well as output a message indicating that this has been done. It is invoked by the command `INSERT(N)` where `N` is the name of a rectangle. It should be clear that the MX-CIF quadtree is built by a sequence of `CREATE` and `INSERT` operations.

(7) Given a point, return the name of the rectangle that contains it. It is invoked by the command `SEARCH_POINT(PX,PY)` where `PX` and `PY` are the *x* and *y* coordinate values, respectively, of the point. If no such rectangle exists, then output a message indicating that the point is not contained in any of the rectangles.

(8) Delete a rectangle or a set of rectangles from the MX-CIF quadtree. This operation has two variants, `DELETE_RECTANGLE` and `DELETE_POINT`. The command `DELETE_RECTANGLE(N)` deletes the rectangle named `N`. It returns `N` if it was successful in deleting the specified rectangle and outputs a message indicating it. Otherwise, it outputs an appropriate message. The command `DELETE_POINT(PX,PY)` has as its argument a point within the rectangle to be deleted whose *x* and *y* coordinate values are given by `PX` and `PY`, respectively. `DELETE_POINT` returns as its value the name of the rectangle that has been deleted and prints an appropriate message indicating its name. If the point is not in any rectangle, then an appropriate message indicating this is output. The code for `DELETE_POINT` should make use of `SEARCH_POINT`. Note that rectangle `N` is only deleted from the MX-CIF quadtree and not from the database of rectangles.

4.4 Part Four: Advanced Operations

(9) Determine if a given rectangle touches (i.e., is adjacent along a side or a corner) an existing rectangle in the MX-CIF quadtree. It is invoked by the command `TOUCH(N)` where `N` is the name of a rectangle. The command returns the names of all the touched rectangles in conjunction with the following messages ``` ``N SHARES ENDPOINT [X AND Y COORDINATE VALUES OF ENDPOINT] WITH THE RECTANGLES [NAME OF RECTANGLES]'' ```. Otherwise the command returns `NIL`. Rectangle `N` need not necessarily be in the MX-CIF quadtree. For each rectangle *r* that touches `N`, display (i.e., highlight) the point in *r* for which the *x* and *y* coordinate values are minimum (i.e., the lower-leftmost corner).

(10) Determine if there exists a rectangle in the MX-CIF quadtree within a given distance of a given rectangle. This is the so-called 'lambda' problem. Given a distance *d*, it is invoked by the command `WITHIN(N,D)` where `N` is the name of the query rectangle, and `D` is a number (integer here although it could also be a real number in the more general case) corresponding to the distance. In essence, this operation constructs a query rectangle `Q` with the same centroid as `N` and distances `LX+D` and `LY+D` to the border. Now, the query returns the identity of all rectangles whose intersection with the region formed by the difference of `Q` and `N` is not empty (i.e,, any rectangle *r* that has at least one point in common with `Q-N`). In other words, we have a shell of width `D` around `N` and we want all the rectangles that have a point in common with this shell. Rectangle `N` need not necessarily be in the MX-CIF quadtree. Note that for this operation you must recursively traverse the tree to

find the rectangles that overlap the query region. You will NOT be given credit for a solution that uses neighbor finding, such as one (but not limited to) that starts at the centroid of `N` and finds its neighbors in increasing order of distance. This is the basis of another operation.

(11) Find the nearest neighboring rectangle in the horizontal and vertical directions, respectively, to a given rectangle. To locate a horizontal neighbor, use the command `HORIZ_NEIGHBOR(N)` where `N` is the name of the query rectangle. `VERT_NEIGHBOR(N)` locates a vertical neighbor. By "nearest" horizontal (vertical) neighboring rectangle, it is meant the rectangle whose vertical (horizontal) side, or extension, is closest to a vertical (horizontal) side of the query rectangle. If the vertical (horizontal) extension of a side of rectangle *r* causes the extended side of *r* to intersect the query rectangle, then *r* is deemed to be at distance 0 and is thus not a candidate neighbor. In other words, the distance has to be greater than zero. The commands return as their value the name of the neighboring rectangle if one exists and `NIL` otherwise as well as an appropriate message. Rectangle `N` need not necessarily be in the MX-CIF quadtree. If more than one rectangle is at the same distance, then return the name of just one of them.

(12) Given a point, return the name of the nearest rectangle. By "nearest," it is meant the rectangle whose side or corner is closest to the point. Note that this rectangle could also be a rectangle that contains the point. In this case, the distance is zero, It is invoked by the command `NEAREST_RECTANGLE(PX,PY)` where `PX` and `PY` are the *x* and *y* coordinate values, respectively, of the point. If no such rectangle exists (e.g., when the tree is empty), then output an appropriate message (i.e., that the tree is empty). If more than one rectangle is at the same distance, then return the name of just one of them.

(13) Find all rectangles in a rectangular window anchored at a given point. It is invoked by the command `WINDOW(LLX,LLY,LX,LY)` where `LLX` and `LLY` are the *x* and *y* coordinate values, respectively, of the lower left corner of the window and `LX` and `LY` are the horizontal and vertical distances, respectively, to its borders from the corner. Your output is a list of the names of the rectangles that are completely inside the window, and a display of the MX-CIF quadtree that only shows the rectangles that are in the window. This is similar to a clipping operation. Draw the boundary of the window using a dashed rectangle. Do not show quadrant lines within the window. All arguments to `WINDOW` are integers (i.e., `LX`, `LY` `LLX`, and `LLY`). Note that for this operation you must recursively traverse the tree to find the rectangles that overlap the query region. You will NOT be given credit for a solution that uses neighbor finding, such as one (but not limited to) that starts at the centroid of the window and finds its neighbors in increasing order of distance. This is the basis of another operation.

4.5 Optional Operations

(14) Find the nearest neighbor in all directions to the boundary of a given rectangle. It is invoked by the command `NEAREST_NEIGHBOR(N)` where `N` is the name of a rectangle. By "nearest," it is meant the rectangle *C* with a point on its side, say *P*, such that the distance from *P* to a side of the query rectangle is a minimum. `NEAREST_NEIGHBOR` returns as its value the name of the neighboring rectangle if one exists and `NIL` otherwise as well as an appropriate message. Rectangle `N` need not necessarily be in the MX-CIF quadtree. If more than one rectangle is at the same distance, then return the name of just one of them. Note that rectangles that are inside `N` are not considered by this query. In order to facilitate grading of this operation, please provide a trace output of the execution of the operation which lists the nodes (both leaf and nonleaf) that have been visited as well as the nearest neighbor. In order for the output to be concise, you are to represent each node that has been visited by a unique number which is formed as follows. The root of the quadtree is assigned

the number 0. Given a node with number N, its `NW`, `NE`, `SW` ,and `SE` children are labeled $4 \cdot N + 1$, $4 \cdot N + 2$, $4 \cdot N + 3$, and $4 \cdot N + 4$, respectively. For example, starting at the root the, the `NE` child is numbered 2, while the `SE` child of the `NW` child of the root is numbered 4*(4*0+1)+4=8.

(15) Given a rectangle, find its nearest neighbor with a name that is lexicographically greater. It is invoked by the command `LEXICALLY_GREATER_NEAREST_NEIGHBOR(N)` where `N` is the name of a rectangle. By "lexically greater nearest" it is meant the rectangle C whose name is lexically greater than that of `N` with a point on `C`'s side, say P, such that the distance from P to a side of the query rectangle is a minimum. `LEXICALLY_GREATER_NEAREST_NEIGHBOR` returns as its value the name of the neighboring rectangle if one exists and `NIL` otherwise as well as an appropriate message. Rectangle `N` need not necessarily be in the MX-CIF quadtree. If more than one rectangle is at the same distance, then return the name of just one of them. Note that rectangles that are inside `N` are not considered by this query. This operation should not examine more than the minimum number of rectangles that are necessary to determine the lexically greater nearest neighbor. Thus you should use an incremental nearest neighbor algorithm (e.g., [2]). As in the `NEAREST_NEIGHBOR` operation in (14), in order to facilitate grading of this operation, please provide a trace output of the execution of the operation which lists the nodes (both leaf and nonleaf) that have been visited as well as the lexically greater nearest neighbor. Use the same numbering convention as in (14).

(16) Perform connected component labeling on the MX-CIF quadtree. This means that all touching rectangles are assigned the same label. By "touching," it is meant that the rectangles are adjacent along a side or a corner. This is accomplished by the command `LABEL()`. The result of the operation is a display of the MX-CIF quadtree where all touching rectangles are shown with the same label. Use integer labels.

References

[1] R. A. Finkel and J. L. Bentley, Quad trees: a data structure for retrieval on composite keys, *Acta Informatica 4*, 1(1974), 1–9.

[2] G. R. Hjaltason and H. Samet, Distance browsing in spatial databases, *ACM Transactions on Database Systems*, 24(2):265–318, June 1999. Also Computer Science TR-3919, University of Maryland, College Park, MD, and *Advances in Spatial Databases — 4th International Symposium, SSD'95*, M. J. Egenhofer and J. R. Herring, eds., pages 83–95, Portland, ME, August 1995, and Springer-Verlag Lecture Notes in Computer Science 951.

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

[4] G. Kedem, The quad-CIF tree: a data structure for hierarchical on-line algorithms, *Proceedings of the Nineteenth Design Automation Conference*, Las Vegas, June 1982, 352–357.

[5] A. Klinger, Patterns and Search Statistics, in *Optimizing Methods in Statistics*, J. S. Rustagi, Ed., Academic Press, New York, 1971, 303–337.

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

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

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