An Improved MX-CIF Quadtree for Reducing Time Cost of Query

Yusi Wei [1, a] and Shojiro Tanaka [2, b]

[1] Information Systems Course, Interdisciplinary Graduate School of Science and Engineering
Shimane University, Japan

[2] Information Systems Division, Interdisciplinary Graduate School of Science and Engineering
Shimane University, Japan

[a]wayis@live.com, [b]tanaka@cis.shimane-u.ac.jp

Keywords: Spatial data structure, Moving-object, Performance evaluation

Abstract. When performing a search with MX-CIF quadtree, all the objects in the nodes which intersect a search-window will be taken as primary results, and then relations between the primary results and search-window are judged for an exact query. The amount of primary result is an important factor to affect time cost of query. This paper proposes an improved MX-CIF quadtree which reduces the time cost of query by decreasing primary results in number of objects. The time cost of tree-building and removal was slightly higher than the original MX-CIF quadtree. An update method which showed better performance than the original method is also proposed. We compared performance between MX-CIF quadtree and our structure in the ways of index, update, query, and memory usage by benchmark dataset. The result indicated that in average, our structure reduced 26.1% primary results, and the total time cost was reduced by 12.7%.

Introduction

MX-CIF quadtree is a data structure which applies to graphic [1] and spatial indexing [2]. It provides efficient insertion and deletion, and the time complexity of both of them is $O(d)$, where d represents the depth of tree. A query will be finished in $O(2^d)$ in the worst case, that is, when a search-window overlaps the area of root totally. For accelerating query, objects are indexed by the position of their Minimum Bounding Rectangles (MBRs). An MBR is created by the maximum and minimum x and y coordinates of an object. An object and its MBR will be inserted into the quadtree's node when its MBR overlaps the node's x or y axis. When launching a query, all the objects will be taken from the nodes which intersect a search-window as primary results to have a precise inspection, that is, to judge if those primary results overlap search-window exactly. The number of objects in primary result is an important factor to affect the time cost of query. In this paper, we propose an improved MX-CIF quadtree which reduces the time cost of query by decreasing primary results. The cost of tree-building and removal time was only slightly higher than the original MX-CIF quadtree.

MX-CIF quadtree

In an MX-CIF quadtree, the region of every node is one of the four quadrants of the parent node [3]. Consider the area of an MX-CIF quadtree as a two-dimensional square space, the space is being decomposed into four sub-squares with equal area recursively. Subdivision stops in two situations: 1. There are no more objects contained in a node. 2. Depth of the tree gets to a predetermined value.

Being different from other variants of quadtree, in an MX-CIF quadtree, more than one object can be associated with both of leaf node and non-leaf node. Moreover, an MBR should be associated with only one node. Once an MBR is associated with a node, term the node N, then the MBR will never be split to any sub-nodes of N. There are two ways to arrange the objects in a node [3]. One way is to create two binary trees for indexing the MBRs cross x and y axis respectively. The other way is to insert the objects into a list of this node directly. We use the secondary way to arrange objects in this paper. The planar partition and structure is given by Fig. 1. Four objects are indexed with MX-CIF quadtree, A and B belong to the root, C belongs to node 2, that is, the northwestern sub-node of the root, and D belongs to node 22. An object should be inserted to the first node which its *x* or *y* axis intersects the object's MBR. So although A overlaps both of the axes of the root and the root's northwestern sub-node, it is still associated with the root only. When making query to MX-CIF quadtree, the objects in the nodes which intersect the search-window will be taken as an approximate result and reported to a secondary query for an exact judgment. According to the usage of MX-CIF quadtree, assume D in Fig. 1 as a search-window to launch an overlap query, because of both of the root and its northwestern sub-node intersect D, the primary results should include object A, B and C. And then, A, B and C will be reported to have an exact comparison with the search-window.

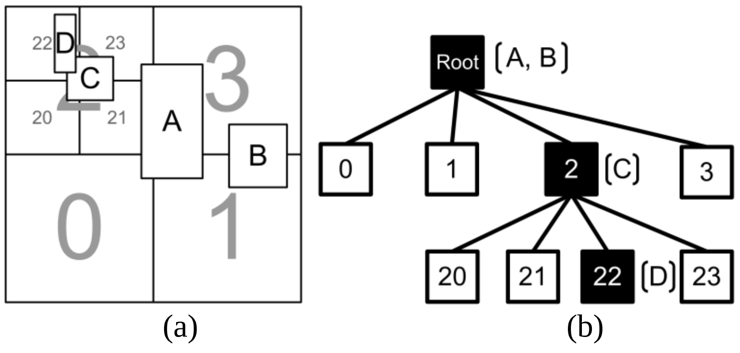

Fig. 1. The planar partition (a) and structure (b) of an MX-CIF quadtree.

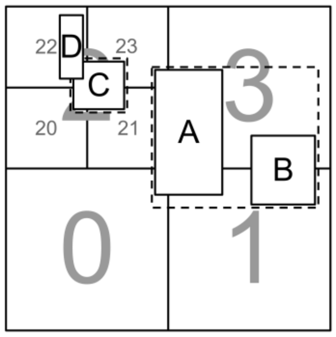

Fig. 2. The Region-MBR (dash line) of the root and node 2.

Improved MX-CIF quadtree

Region-MBR. In the example mentioned before, object A and B do not overlap the search-window, but they still be taken as a primary query result. This will lay a burden on the next exact query, because the more the objects reported to the exact query; the more the time will be used to finish it. For decreasing objects in the primary query result, we use a rectangle to present the minimum bounding of all the objects in the same node, termed Region-MBR. It is created by the maximum *x* and *y* coordinate, and the minimum *x* and *y* coordinate of all the objects in a node. Fig. 2 shows the Region-MBR of the root and node 2 of the example mentioned before. Because there is only one object in node 2, so the Region-MBR of node 2 is equal to the MBR of C. Comparing to the border of node, Region-MBR provides a more precise frame for judging relations between object and search-window. Usage of Region-MBR is described as follows, when making a query with a search-window, only if there are any node's Region-MBR interests a search-window, then all the objects of this node can be taken as primary results. Appling this rule to the example mentioned before, the primary result will be C only. Because D does not intersect the Region-MBR of root, so the objects in the root, that is, A and B, should not be included.

Insertion. The Region-MBR is added to a node as a new property besides others like level, border, center and so on. Based on insertion procedure of original MX-CIF quadtree, every time when insert an object to a node, the object's MBR should be compared with the node's Region-MBR, enlarging the Region-MBR to include the MBR completely if it does not. The insertion algorithm is listed as below.

```
Algorithm Insert (Node node, Object O)
if (O.MBR intersects x axis or y axis of node) then
    Insert O to node;
    Enlarge_RegionMBR (node, O);
else
    for each non-null sub-node of node do
        if (sub-node contains O.MBR) then Insert (node.subnode, O); end if;
    end for;
end if;
```

```
Algorithm Enlarge_RegionMBR (Node N, Object O)
Rectangle R = N.RegionMBR;
Rectangle M = O.MBR;
if M.minX < R.minX then R.minX = M.minX; end if;
if M.minY < R.minY then R.minY = M.minY; end if;
if M.maxX > R.maxX then R.maxX = M.maxX; end if;
if M.maxY > R.maxY then R.maxY = M.maxY; end if;
```

Query. To report the objects that probably intersect a search-window is achieved by traversing the improved MX-CIF quadtree, the objects of the nodes which their Region-MBRs intersect search-window should be taken as a primary result. The query algorithm is listed as follows,

```
Algorithm query (Node node, Rectangle searchwindow)
if (node.RegionMBR intersects searchwindow) then Take all objects in node as a primary result;
    for each not null sub-node of node do
        if(sub-node intersects searchwindow) then query (node.subnode, searchwindow);
        end if;
    end for;
end if;
```

Deletion. Like the processing of insertion, to delete an object, say *O*, first to locate *O*'s position by traversing the tree with the MBR of *O*. Once found the node which its *x* or *y* axis intersects the MBR of *O*, remove *O* from it. Different from original MX-CIF quadtree, if the MBR of the deleted object touches any border of the node's Region-MBR, for making the Region-MBR still represent the exact border of all objects in this node, the Region-MBR should be recalculated by traversing every object in the node.

```
Algorithm Delete (Node node, Object O)
if (O.MBR intersects x axis or y axis of node) then Delete O from node; Shrink (node, O.MBR);
else
    for each non-null sub-node of node do
        if (sub-node contains O.MBR) then Delete (node.subnode, O); end if;
    end for;
end if;
```

```
Algorithm Shrink (Node node, Rectangle deleted_object)
if (node.RegionMBR.maxX == deleted_object.maxX) or
 (node.RegionMBR.maxY == deleted_object.maxY) or
 (node.RegionMBR.minX == deleted_object.minX) or
 (node.RegionMBR.minY == deleted_object.minY) then
    for each object in node do Enlarge_RegionMBR(node, object); end for;
end if;
```

Update. Updating is used to renew the position of an indexed object, generally achieved by deleting the old object and then insert a new object. In the processing of deletion and insertion, sometimes there are some nodes are visited by both of the two processing. To avoid the nodes which are visited repeatedly will reduce the time cost of update. Here we propose a new update method that would reduce the time cost by avoiding visit nodes redundantly. When update an object, after deletion of the old object from a node, first to check if the node contains the object about to insert, that is, the node's border encloses the new object completely. If so, that means the node or one of its sub-nodes should contains the new object, traverse the tree from this node to its leaf nodes to find such a node. Otherwise, find the node contains the new object from root. If a node contains the object will be deleted, and one of its sub-nodes or the node itself contains the object will be inserted, in this situation, this method can avoid revisiting the nodes above the node repeatedly. The algorithm of update an object is shown below, it uses a special deletion method which will not shrink the Region-MBR immediately after removal of an object. Because of the objects that about to remove and insert are in the same node sometimes. For avoiding recalculation of the same Region-MBR, the processing of shrink Region-MBR is postponed until after insertion is finished. The algorithm of update is listed as follows,

```
Algorithm Update (Node root, Object remove, Object insert)
Node returnNode = DeleteWithoutShrink(root, remove);
if (returnNode == null) then Insert (root, insert);
else
    if (returnNode contains insert) then Insert (returnNode, insert);
    else Insert (root, insert);
    end if;
    Shrink (returnNode, remove.MBR);
end if;
```

```
Algorithm DeleteWithoutShrink (Node node, Object O)
if (O.MBR intersects x axis or y axis of node) then Delete O from node; return node;
else
    for each non-null sub-node of node do
        if (node.subnode intersects O.MBR) then DeleteWithoutShrink (node.subnode, O);
    end if;
    end for;
end if;
return null;
```

Related works

Expanded MX-CIF quadtree [4] is a variant of MX-CIF quadtree. For each object, say O, which crosses a node's x or y axis, term the node as N, split N once that makes N has four equal sized sub-nodes. This makes O contained by at least two of the sub-nodes of N. Check if all of the direct sub-nodes of N are the minimum nodes contain each part of O. Once there is any part of O, say O_i , is contained by a direct sub-node of N, say N_i, but not crossing the x or y axis of N_i, split N_i recursively till one of the sub-nodes of N_i is the minimum node which contains O_i.

Multiple storage quad tree [5] sets a capacity for every node, a node will keep on being split until the number of objects contained by this node reaches its capacity, it means an object could associate to not only one node. During query period, for avoiding the same object be reported several times, an object will be marked by the first time it is reported. All marked objects should not be reported again.

More recently, a new structure named RMX-quadtree is proposed by [6]. It provides efficient query through index different levels of resolution of spatial data. It is achieved by merging the trees which created for images have different resolutions into one tree. Moreover, to improve security and privacy, RMX-quadtree is able to associate different data with different authorization.

Different from variants mentioned above, an improved MX-CIF quadtree does not split any object to sub-nodes. The advantages are, 1. Avoid indexing redundancy. 2. Simple procedure of indexing and query processing. 3. Controlled depth of tree.

QR-tree is proposed in [7], it uses quadtree as main structure and creates an R-tree for each sub-node under the quadtree's root. That is, an object will be first located to a quadtree node, and then be inserted into an R-tree in the node. QR-tree does not split object to sub-nodes, but it needs to maintain two structures for each manipulation, so the time cost of insertion, deletion and query should be increased comparing to MX-CIF quadtree. Moreover, in a case that a search-window overlaps a node, but does not overlap the objects belongs to the node, it is wasting time to make query for the R-tree in this node.

Improved MX-CIF quadtree should takes less time on insertion and tree building than expanded MX-CIF quadtree, and multiple storage quad tree. In common, with the same dataset, the depth of our structure as same as the original MX-CIF quadtree should be less than expanded MX-CIF quadtree and the other variants, because they split object to sub-nodes, this will increase the depth of a tree. Another shortcoming of splitting objects is that when making a query, some objects may be reported as a result several times, it will reduce the efficiency of query. A solution is to marking the object the first time it is reported, and then does not report the marked object again. All marked objects have to be unmarked before the next query. Another solution proposed by [8] creates four lists in each node for storing objects, depending on the position of the object lies in a node. When making a query, decide the list the search-window belongs to, depending on the same rules of distributing objects. And then examine all objects in the same list. Comparing to query methods mentioned above, our structure uses at most only two judgments to finish a query in each node; most of the nodes only need one judgment. Only if the search-window covers a node, a secondary judgment between search-window and Region-MBR is necessary. Comparing to original MX-CIF quadtree, our structure excites only one more judgment during query period. Because of our structure reduces objects in primary result, the next exact query time is saved. Our structure takes less time cost on query than the original MX-CIF quadtree, which will be verified in Section Result.

Experiments

A suit of tests with both of moving-object and static-object datasets will be introduced in this section. We evaluated the performance of both of original MX-CIF quadtree and our structure based on JTS Topology Suit. It is a viable open-source API of 2D spatial predicates and functions which include a package for creating MX-CIF quadtree-based indexing. The package can be found: http://www.vividsolutions.com/jts/download.htm. These tests use real and synthetic datasets. Three datasets are included in the real and also static-object datasets part, "Administrative areas" (polygon), "Inland water" (line) and "Roads" (line) of United States. They can be found: http://www.diva-gis.org/gData. Synthetic and also moving-object datasets include 100,000 moving-points, 10,000 moving-rectangles and 27,146 moving-lines. Currently, each moving-object has 10 snapshots. Relations include "distance," "containment" and "intersection" will be queried with three types of object, point, line and polygon. The source code of this project including dataset and more detailed results can be found: http://wei-yusi.appspot.com. In each small test, we pick a different type of moving-object and a static-object dataset to index. Update every moving-object with its next position, and then find the moving-objects which are overlapped by any static-object after each update. So there are nine small tests are included in total.

Result

Among all the nine tests, comparing to original MX-CIF quadtree, in average, our structure causes indexing time and update time rises by 7% and 19% respectively, but instead of the increased time cost, query time reduces by 14%, and total time cost reduces by 12%. The comparison of indexing time, update time, query time, primary query result, and total time cost between MX-CIF quadtree and our structure are shown in Table 1. A more detailed result can be found: http://wei-yusi.appspot.com/resource/resultof9tests.xlsx. The results are in average values of 5-time calculation.

Table 1. Time cost performance of our sturcture (compare to original MX-CIF quadtree).

Test / Item	1	2	3	4	5	6	7	8	9
Index time	+6%	+17%	+16%	+5%	+7%	+5%	+3%	+1%	+5%
Update time	+15%	+18%	+18%	+29%	+45%	+43%	+8%	+7%	+7%
Query time	-30%	-9%	-1%	-37%	-2%	-10%	-8%	-25%	-5%
Primary results	-28%	-13%	-15%	-62%	-24%	-25%	-28%	-17%	-23%
Total time	-30%	-8%	-1%	-37%	-2%	-10%	-8%	-14%	-4%

CONCLUSION

This paper proposed a new structure which reduces the time cost of query by decreasing primary query results. It is achieved by adding one more element into a node structure, termed Region-MBR, which is used to present the minimum bounding of all the objects in the same node. Because that every time when inserting an object to a node, the node's Region-MBR should be calculated again if it is need to be enlarged to include the new object, the index time is increased. And when removing an object, in a very rare situation that the MBR of the removed object touches any of the four sides of the node's Region-MBR, this Region-MBR have to be recalculated by traversing every object in the node. So the time cost of updating an object is raised. We compared the performance between original MX-CIF quadtree and our structure. The result indicates that, comparing to original MX-CIF quadtree, our structure reduced by 26.1% objects from primary result in average, hence the time cost of queries based on primary result was greatly saved. At the same time, indexing and update time was increased by 7.2% and 19.1% respectively. But the total time cost was still reduced by 37% in the best case, and reduced by 1% in the worst case. On the other hand, the Region-MBR is created by 2 pairs of coordinates, that is, four double-precision variables. Each double-precision variable takes 8 bytes. So the memory usage will increase 8*4 = 32 byte for each node. A memory usage comparison can be found: http://wei-yusi.appspot.com. It is observed by "Task Manager" of windows. The MX-CIF quadtree is used in extension fields such as network routing [2], [9], [10] and skyline queries [11]. The improved MX-CIF quadtree will be able to manifest its ability in these fields, as well as traditional uses in the graphic and spatial.

References

[1] Samet, H., Sorting in Space: Multidimensional, Spatial, and Metric Data Structures for Computer Graphics Applications, ACM SIGGRAPH ASIA 2010 Courses (2010).

[2] Tanin, E., Harwood, A., Samet, H., Using a distributed quadtree index in peer-to-peer networks, The VLDB Journal — The International Journal on Very Large Data Bases, Vol.16, No.2 (2007) 165-178.

[3] Samet, H., Foundations of Multidimensional and Metric Data Structures., Morgan Kaufmann (2006).

[4] Able, D. J., Some elemental operations on linear quadtrees for geographic information systems, Computer Journal, Vol. 28, No. 1 (1985) 73-77.

[5] Brown, R. L., Multiple storage quad tree : a simpler faster alternative to bisector list quad trees, IEEE Trans. Computer Aided Design , Vol. CAD-5 (1986) 413-419.

[6] Vijayalakshmi, A., Qi, G., Heechang, S., Jaideep, V., A unified index structure for efficient enforcement of spatiotemporal authorisations, International Journal of Information and Computer Security, Vol. 4, No.2 (2010) 118-151.

[7] Fu, Y.-C., Hu, Z.-Y., Guo, W., Zhou, D.-R., QR-Tree: A Hybrid Spatial Index Structure. In: Proceedings of the Second International Conference on Machine Learning and Cybernetics, Xi'an, November 2-5 (2003).

[8] Weyten, L., Pauw, D., Quad list quad trees: A geometrical data structure with improved performance for large region queries. IEEE Transactions on Computer-Aided Design of Integrated Circuits, 8 (1989) 229-233.

[9] Zuo, H., Jing, N., Deng, Y., Chen, L., CAN-QTree: A Distributed Spatial Index for Peer-to-Peer Networks, Proceedings of the 2008 10th IEEE International Conference on High Performance Computing and Communications (2008) 250-257.

[10] Ranjan, R., Buyya, R., Decentralized overlay for federation of Enterprise Clouds, Handbook of Research of Scalable Computing Technologies, IGI Global USA (2009).

[11] Lin, X., Xu, J., Hu, H., Range-based Skyline Queries in MobileEnvironments, IEEE TKDE (2011).