R-trees: A Programmer's Introduction

A Presentation for the University of Calgary's Problem Solving Club.

Prepared by Kent Williams-King, kawillia@ucalgary.ca.

1 Basic properties

Leaf nodes contain entries of the form $(I, object)$, where I is an n-dimensional rectangle which is the spatial bounding box of the data tuple. Non-leaf nodes contain entries of the form $(I, child)$, where I is an n-dimensional rectangle which is the spatial bounding box that covers all of $child$'s entries. M represents a value such that it is the highest number of entries any node can reference, and m a value such that it is the lowest number of the same.

1) Every leaf node contains between m and M nodes, unless it is the root node.
2) For each index record $(I, object)$ in a leaf node, I is the smallest rectangle that spatially contains the n-dimensional data object.
3) Every non-leaf node contains between m and M child nodes, unless it is the root node.
4) For each index record $(I, child)$ in a non-leaf node, I is the smallest rectangle that spatially contains the rectangles in the child node.
5) Unless the root node is a leaf, it contains at least two children.
6) All leaf nodes appear at the same level of the tree.

If these properties are met, the height of an R-tree with n nodes is at most $\lceil \log_m n \rceil - 1$ and at least $\lceil \log_M n \rceil - 1$.

2 Basic algorithms

Several algorithms are required for the usual elementary operations on an R-tree. These can be classified into three categories: those related to searching, to insertion, and finally to deletion.

For an index entry E, the bounding rectangle is denoted by $E.I$, child node by $E.N$, and value by $E.V$.

2.1 Searching

The search algorithm for an R-tree is remarkably similar to any other tree that has a variable number of children.

Algorithm *Search*. Given an R-tree with a root node T, find all index records whose rectangles overlap a search rectangle S.

S1 [Search subtrees] If T is not a leaf node, invoke *Search* on the nodes $E.N$ such that $E.I$ overlaps S.

S2 [Search leaf nodes] If T is a leaf node, check all entries against the search rectangle. If $E.I$ overlaps S, then E is a result of the search.

2.2 Inserting

Inserting entries into an R-tree requires a small amount of finesse to ensure all tree properties are met. The algorithm *SplitNode* is detailed on the reverse side.

Algorithm *Insert*. Insert a new entry E into a given R-tree with a root node T.

I1 [Find position for new record] Invoke *ChooseLeaf* to select a leaf node L to place E into.

I2 [Add record to leaf node] If L is not full, add E to L. Otherwise, invoke *SplitNode* with L and E to create the node LL.[1]

I3 [Upwards change propagation] Invoke *AdjustTree* on L, also passing along LL if a split was performed in *I2*.

I4 [Tree growth] If the propagation results in a root split, create a new root node with the children of the resulting nodes.

Algorithm *ChooseLeaf*. Selects a leaf node to place an entry E_0 into, given an R-tree with root node T.

CL1 [Initialization] Set $N = T$.

CL2 [Leaf check] If N is a leaf node, return N.

CL3 [Subtree choosing] Consider all entry bounding rectangles $E.I$. Let F be the element such that $F.I$ is the rectangle that needs the least enlargement to cover $F.I$ and $E_0.I$. Ties should be resolved by the creation of the smallest possible rectangles.

CL4 [Decension] Set N to be $F.N$, and repeat from *CL2*.

Algorithm *AdjustTree*. Ascends from a leaf node L to the root, adjusting bounding rectangles and splitting nodes as required.

AT1 [Initialization] Set $N = L$. If L was previously split, set NN to be the resulting second node.

AT2 [Completion check] If N is the root node, stop.

AT3 [Bound adjustment] Let P be the parent node of N, and E_N be the entry in P such that $E_N.N = N$. Adjust $E_N.I$ so that it tightly covers all the bounds of all the elements of N.

AT3 [Node propagation] If N has a partner node NN resulting from an earlier split, create a new entry E_{NN} with $E_{NN}.N$ referencing NN and $E_{NN}.I$ enclosing all rectangles in NN. If there is space, add E_{NN} to P. Otherwise, invoke *SplitNode* to create the node PP, such that P and PP together contain all entries of the original P, plus E_{NN}.

AT3 [Ascension] Set $N = P$, and $NN = PP$ if a split occurred. Repeat from *AT2*.

[1] The elements of the old L, plus E, will be spread out between the new L and LL.

2.3 Deleting

Algorithm *Delete*. Removes an entry E from a given tree.

D1 [Find leaf node] Locate the leaf node L that contains E. (Algorithm similar to *ChooseLeaf*.)

D2 [Remove entry] Remove the entry E from L.

D3 [Propagate upwards] Invoke *CondenseTree* on L.

D4 [Shorten tree] If the root node has only one child, make the child the new root.

Algorithm *CondenseTree*. Given a leaf node L from which an entry has been removed, ensure that all affected elements of the tree have at least m nodes.

CT1 [Initialize] Set $N = L$, and let Q be an empty set.

CT2 [Find parent entry] If N is the root, go to *CT6*. Otherwise, let P be the parent node of N, and E_P be the entry in P such that $E_P.N = N$.

CT3 [Underfull node removal] If N has less than m entries, remove E_N from P and add N to Q.

CT4 [Adjust bounding rectangle] If N has not been removed, adjust $E_N.I$ to cover all entries in N.

CT5 [Ascend tree] Set $N = P$ and repeat from *CT2*.

CT6 [Re-insertion] For each entry in each member of Q, re-insert into the proper position in the tree such that the leaf depth property is satisfied.

2.4 Node splitting

Algorithm *LinearSplit*. Split $M + 1$ entries into two groups such that the bound of both groups is minimized.

LS1 [Find first elements for both groups] Invoke *LinearPickSeeds* to choose two entries suitable for the "seed" elements of the two groups.

LS2 [Check for completion] If all entries have been assigned, then stop. If one group has few enough entries such that it will require all the remaining entries to meet the minimum number of nodes, m, assign them and stop.

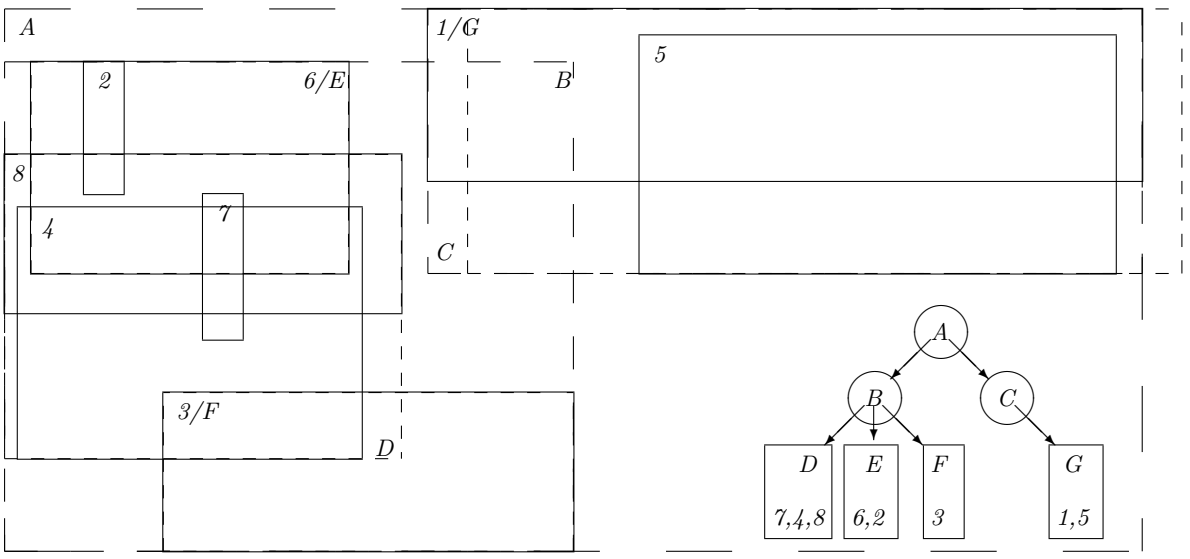
LS3 [Add entry to group] Select one unassigned entry and add it to the group such that the bounding rectangle increases by the smallest amount. Ties can be resolved by selecting the group with the smallest area, then smallest entry count. Repeat from *LS2* as required.

Algorithm *LinearPickSeeds*. Given a list of $M + 1$ entries, choose two such that they have the greatest separation.

LPS1 [Find extreme rectangles] Along each dimension, find the entry whose bound has the highest low side and lowest high side. Record the separation.

LPS2 [Normalize separations] Normalize the separations along each dimension by dividing by the size of the dimension ($|coord_{highest} - coord_{lowest}|$).

LPS3 [Select extreme pair] Choose the pair with the greatest normalized separation.



Numbers are data rectangles, letters are ranges of tree nodes.

Figure 1: Example R-Tree with 8 random elements ($M = 3$, $m = 1$).