

A Novel Improvement to the R*-tree Spatial Index using Gain/Loss Metrics

Donghui Zhang

College of Computer and Information Science
Northeastern University
Boston, MA 02115
donghui@ccs.neu.edu

Tian Xia

College of Computer and Information Science
Northeastern University
Boston, MA 02115
tianxia@ccs.neu.edu

ABSTRACT

The R*-tree is a state-of-the-art spatial index structure. It has already found its way into commercial systems. The most important improvement of the R*-tree over the original R-tree is that it utilizes forced reinsertion. That is, if a disk page overflows, some objects are removed from the page and reinserted into the index. The goals are: (a) to reduce the MBR area; and (b) to keep the shape of the MBR close to a square. However, no existing work consists of a unified metric which can be used to balance the two criteria. For example, if there are two methods to remove objects from a rectangle, and one results in a rectangle with smaller area, while the other results in a square with slightly larger area, which method shall we choose? The R*-tree algorithm selects objects whose distances to the center of the page's MBR are the largest. However, this is not optimal. In this paper, we formally define the *quality* of a rectangle and the *gain* to shrink a rectangle. Then we provide algorithms to shrink the MBRs with the goal to maximize the gain. The algorithms are experimentally compared with the R*-tree's reinsertion algorithm. Furthermore, as the opposite of *gain*, we define the *loss* of expanding a rectangle. While inserting an object into the R*-tree, we need to choose a sub-tree to put the object in. With the new metric, we can choose the sub-tree with the least loss. Finally, we integrate the new algorithms into the R*-tree.

Categories and Subject Descriptors

H.2.8 [Database Management]: Database Applications—*Spatial databases and GIS*

General Terms

Algorithms, Measurement, Performance

Keywords

R*-tree, quality, gain, reinsertion

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

GIS'04, November 12–13, 2004, Washington, DC, USA.
Copyright 2004 ACM 1-58113-979-9/04/0011 ...\$5.00.

1. INTRODUCTION

Over the past twenty-five years or so, there has been extensive work on indexing spatial data. Quite a few index structures have been proposed, as surveyed by [3, 9]. There is no consensus on the *best* spatial index structure. However, the R-tree [4] and its variant the R*-tree [2] are widely implemented and have found their ways into commercial systems [1, 6, 7]. The idea is to recursively cluster objects into *minimum bounding rectangles (MBR)* and organized them into a dynamic, disk-based, balanced tree structure. The root node corresponds to the largest MBR which contains the spatial extent of all objects. At one level down, the MBR of every child of the root contains the spatial extent of all objects in the sub-tree, and so on. The R-tree or the R*-tree can be used to speed up the range search: “find the objects in a given spatial range”. The tree is browsed in a top-down fashion. Any sub-tree where the root MBR does not intersect with the query range can be safely omitted.

The most important improvement of the R*-tree over the original R-tree is that it utilizes *forced reinsertion*. That is, if a disk page (node) overflows, some entries are removed from the page and reinserted into the index. The goals are: (a) to reduce the MBR area; and (b) to keep the shape of the MBR close to a square. The rationale is that if we do so, the range query is likely to touch a smaller number of disk pages. In the extreme, if all of the leaf MBRs are nearly as large as the whole space, every range query would need to examine the whole tree. However, no existing work consists of a unified metric which can be used to balance the two criteria. If there are two methods to remove objects from a rectangle, and one results in a rectangle with smaller area, while the other results in a square with slightly larger area, which method shall we choose? The R*-tree algorithm selects objects whose distances to the center of the page's MBR are the largest. However, this is not optimal.

Consider the example page layout of a R*-tree in Figure 1(a). Suppose it is overflowing, and suppose the number of objects to be re-inserted is two. The R*-tree will remove objects *a* and *b*, as their distances to the center of the MBR (the cross) are the largest. This will result in a slightly smaller MBR (Figure 1b). It is far from optimal. In this case, the optimal solution is to remove objects *c* and *d*, which results in a much smaller MBR as shown in Figure 1(c).

The major contributions of this paper are as follows.

- We mathematically define the *gain* of removing some objects from an MBR. This metric integrates both con-

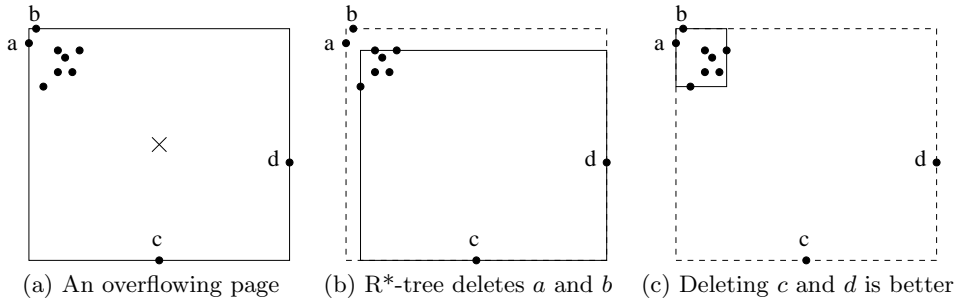


Figure 1: An example which illustrates that the R*-tree’s reinsertion algorithm may not be efficient.

cerns: the area of an MBR should be small and the shape of an MBR should be close to a square. Some related metrics are also defined: the *quality* of an MBR and the *loss* of expanding an MBR.

- We show how the defined metrics can help us improve the R*-tree. In particular, we define the *p-boundary* as the p objects which, if removed from an MBR, will result in the maximum gain (among all choices of removing p objects). Furthermore, while the R*-tree chooses a fixed number p of objects to re-insert, we argue that this is not the best strategy. For instance, if by removing $0.3p$ objects, we already can achieve a gain very close to that of removing the p -boundary, we should not insist to remove all the p objects (to re-insert an object back into the index costs time). In order to do so, we define the *minP-boundary*. This is the set of objects which should be removed from an MBR.
- We provide algorithms to find the p-boundary and the minP-boundary. A straightforward algorithm which finds optimal solution has time complexity $O(n \log p + p^4 \log p)$. We propose a greedy algorithm which has complexity $O(n \log p)$. Here n is the number of objects in a page.
- Experimental results are provided to show the degree of improvement resulting from the new approach.

The rest of the paper is organized in the following way. Section 2 defines the metrics and shows how they can be used to improve the R*-tree. Section 3 provides algorithms for computing the p-boundary and the minP-boundary. Section 4 shows the experimental results. Finally, Section 6 concludes the paper.

2. DEFINITION OF SOME METRICS

For simplicity we present our concepts mostly in 2-dimensional space. But our discussion applies to higher dimensions as well.

2.1 Quality and Gain/Loss

Definition 1. Given a rectangle r with width w and height h , the **quality** of the rectangle is defined as

$$Q(r) = \frac{1}{w * h} * \left(\frac{\min\{w, h\}}{\max\{w, h\}} \right)^\alpha$$

where $\alpha \in [0, 1]$ is a constant.

Intuitively, a rectangle whose area ($w * h$) is smaller and whose shape is closer to a square has larger quality. As an example, let $\alpha = 0.5$. Figure 2 compares the quality of three rectangles. Here Figure 2(a) has the worst (smallest) quality as it has the largest area. While Figure 2(b) and (c) have the same area, Figure 2(c) has the largest quality, since it is a square.

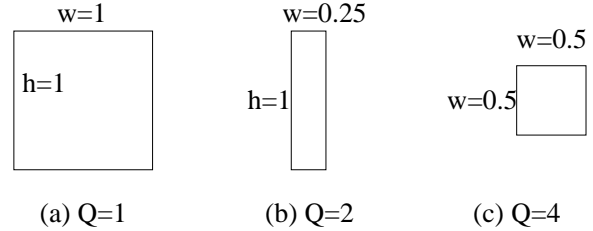


Figure 2: Comparison of the *quality* of three rectangles. Here we choose $\alpha = 0.5$.

In Definition 1, the constant α expresses how much emphasis we put on the shape. If $\alpha = 0$, the shape does not matter at all and we only consider the area. In this case, Figure 2(b) would have the same quality as Figure 2(c). To the other extreme, if $\alpha = 1$, the area does not matter and we only consider the length of the longer edge. In this case, Figure 2(b) would have the same quality as Figure 2(a).

From another perspective, let us examine the relationship between a rectangles area and shape. Without loss of generality, assume width $w \geq$ height h . Let *ratio* = w/h . When *ratio* = 1, the shape is a square. As *ratio* increases, if we want to maintain the same quality as that of the square, we have to decrease the area. It can be derived that $area = ratio^{-\alpha}/Q$. The relationship can be seen graphically from Figure 3.

A special case is when the rectangle becomes a line or a point. This occurs when the width and/or height of the rectangle is zero. The quality of such special rectangles typically is $+\infty$. Although theoretically this is fine, in practice it is not as it will cause number overflow. This special case can be handled gracefully by assigning a very small number *min* (e.g. 0.0001) as the minimum width or height a rectangle might have.

THEOREM 1. Let the minimum and maximum length of any rectangle edge be *min* and *max*, respectively. The quality of any rectangle is in the range $[\frac{1}{max^2}, \frac{1}{min^2}]$.

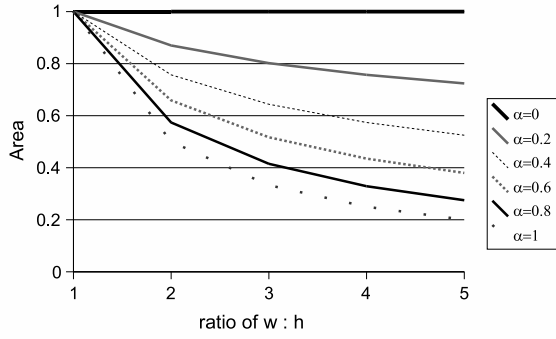


Figure 3: Illustration of the relationship between rectangle area and the $w : h$ ratio, where the quality of the rectangle is 1.

Note that Theorem 1 holds regardless of the choice of α .

The quality definition can naturally be extended to d -dimensional space. Consider a d -dimensional rectangle r , whose edges have length h_1, \dots, h_d . The quality of r is defined as:

$$\frac{1}{\prod_{i=1}^d h_i} \left(\frac{\min\{h_1, \dots, h_d\}}{\max\{h_1, \dots, h_d\}} \right)^\alpha$$

Definition 2. If a rectangle r_1 is shrunk to r_2 (r_1 spatially contains r_2), the **gain** is defined as

$$G(r_1, r_2) = 1 - \frac{Q(r_1)}{Q(r_2)}$$

Intuitively, the gain describes the percentage of the shrink. The more we shrink, the larger the percentage is. For example, if we shrink from Figure 2(a) to Figure 2(b), the gain is 50%. If we shrink from Figure 2(a) to Figure 2(c), the gain is 75%.

Symmetrically, the **loss** of expanding rectangle r_2 to r_1 is defined as the gain of shrinking r_1 to r_2 .

THEOREM 2. *If a rectangle r_1 contains rectangle r_2 , $Q(r_1) \leq Q(r_2)$.*

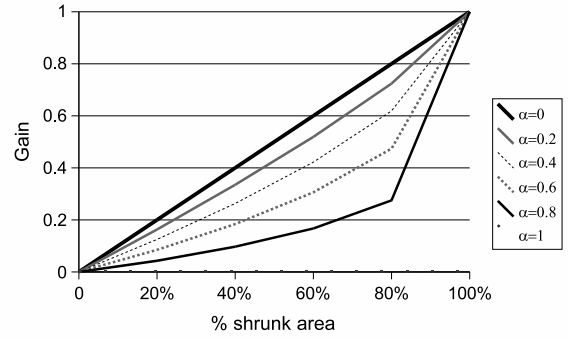
If a rectangle r_1 is shrunk to r_2 , although the area always decreases, the shape may be worse. So it is not straightforward to see that the quality of r_2 is always bigger or equal. However, theorem 2 guarantees that by shrinking a rectangle, the quality never decreases.

As a corollary, we have:

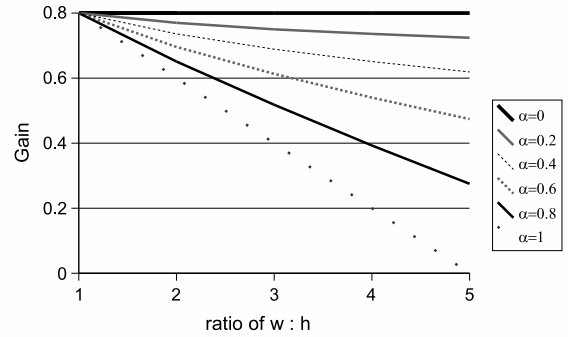
THEOREM 3. *The gain of shrinking a rectangle is in the range $[0, 1]$.*

If we do not shrink at all, the gain is 0. And the gain can never be more than 1. The maximum gain corresponds to shrinking from the biggest rectangle to the smallest rectangle, and the gain is $1 - \min^2 / \max^2$.

To get more insight understanding of the relationship between gain and shrunk area/shape/ α , we refer to Figure 4. Here Figure 4(a) measures the gain for shrinking a square only from one border. Without loss of generality, assume we shrink from the top border. If the shrunk area is 20%, the



(a) Shrink a square from one border, varying the shrunk area



(b) Shrink the area of a square by 80%, varying the $w : h$ ratio

Figure 4: Measurement of the gain for shrinking a square, among different values of α .

resulting rectangle has a height which is equal to 80% of the original height, while the width remains unchanged, and so on. If $\alpha = 0$, the gain increases linearly as the percentage of shrunk area increases. If $\alpha = 1$, the gain remains 0. The figure illustrates that a larger α typically considers the same shrinking as having less gain.

Figure 4(b) measures the gain for shrinking a square's area by 80%, while varying the shape of the resulting rectangle. The shape is described by the $w : h$ ratio. The larger the ratio is, the less the gain is.

The concepts of quality and gain can naturally be extended for a set of spatial objects, if we take the MBR as the rectangle. Given a set S of spatial objects, the *quality* of it, $Q(S)$, is the quality of $MBR(S)$. The gain of removing a subset P from S , denoted as $G(S, S - P)$, is the gain for shrinking $MBR(S)$ to $MBR(S - P)$.

2.2 p-boundary and minP-boundary

In the R^* -tree, if a page overflows, some entries (e.g. 30%) are removed from the page and re-inserted back into the index. However, the original method of picking objects to remove may not find the optimal set of entries, as revealed by Figure 1. Based on our definitions, we consider an algorithm as optimal if the removal maximizes the gain.

Definition 3. Given a set S of spatial objects and a number p , the **p-boundary** of S is a subset P consisting of p objects such that the gain of removing these objects, $G(S, S - P)$, is maximal among all cases of removing p objects.

The p -boundary is the optimal set of objects we want to remove from the overflowing page, if we want to remove exactly p objects.

However, reinserting an object into the index triggers additional insertion cost. Thus we only want to reinsert an object if necessary. For example, let $p = 20$. If by removing 5 objects, we can get 98% of the gain for removing the p -boundary, we probably shall stay with removing only 5 objects, not 20. Thus, while the original R*-tree always removes a fixed number p of objects, we allow the removal of $t \leq p$ objects. In particular, we have the following definition.

Definition 4. Given a set S of spatial objects, a constant $\beta \in [0, 1]$ and a number p , the **minP-boundary** of S is a subset T consisting of the smallest number of objects such that $G(S, S - T) \geq \beta \cdot G(S, S - P)$, where P is the p -boundary of S .

Intuitively, the minP-boundary is a subset of at most p objects which, if removed, will achieve a gain at least as large as β times the gain to remove the p -boundary. Furthermore, among all subsets that satisfy this condition, we need to pick the one with the smallest number of objects.

2.3 Thoughts on Improving the R*-tree

Using our metrics, we can improve the R*-tree in the following ways. First, and most importantly, the forced reinsertion algorithm should now pick the minP-boundary to re-insert. It will bring the maximal gain. Another modification for the reinsertion algorithm is that removing the minP-boundary shall not be enforced all the time. If the gain for removing the p -boundary is smaller than a given threshold δ (e.g. 0.001), it's best if we do not reinsert any object, but split the overflowing disk page directly.

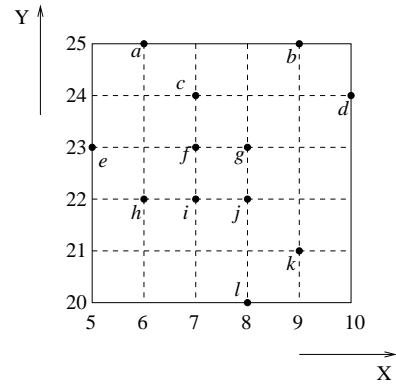
Second, to insert a new object into the R*-tree, if the object cannot be contained in the MBR of any of the sub-trees, the *ChooseSubtree* algorithm picks a sub-tree to insert the object in, and the MBR of the sub-tree is expanded. We hereby propose to choose the sub-tree whose expansion has the least loss.

Third, our idea can be used to improve the *promotion-based index* proposed by Kanth, et al. [5]. The promotion-based index stores objects with large extent at higher levels of the tree. By doing so, the lower-level MBRs are smaller. This idea can be extended to also promote objects which are far from the other objects, even though the objects may not have large extent. The minP-boundary acts as a good suggestion on which objects to promote.

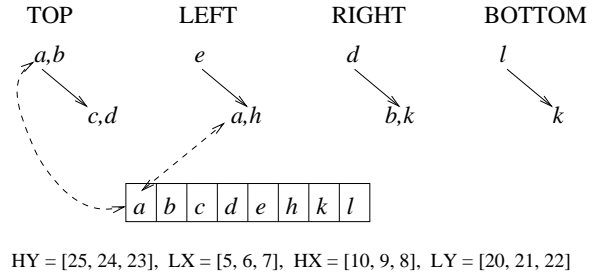
In the rest of the paper, we focus on designing algorithms to find the minP-boundary.

3. ALGORITHMS FOR FINDING THE MINP-BOUNDARY

In order to compute the minP-boundary, we have to compute the p -boundary first. Given a set of n spatial objects, straightforwardly, the p -boundary could be computed by enumerating all possible combinations of p objects and



(a) Object layout of an MBR



(b) The border structure

Figure 5: An example of the border structure, where $p = 4$.

choose the combination which achieves the largest gain. However, This method is not efficient in that the number of combinations to be examined is $\binom{n}{p}$, which can be exponential to n (e.g. when $p = n/2$, the complexity is $o(2^n)$).

We observe that the p -boundary should contain “outside” objects, since removing some “inside” objects without removing the outside objects does not shrink the MBR. Thus, if we take p objects which are the closest to the top border of the MBR, and if we similarly take p objects the closest each to the left, right, and bottom borders, we get a set *FourP* of at most $4p$ objects. It can be guaranteed that the p -boundary is a subset of *FourP*.

Based on the above observation, in Section 3.1 we present the *border structure* which maintains this set *FourP* (to be precise, as we will see the border structure only needs to maintain a subset of *FourP*). This structure is used by the algorithms to be presented in Sections 3.2 and 3.3.

3.1 The Border Structure

An MBR has four *borders*: top, left, right, and bottom borders. Let us focus on shrinking one of the borders, e.g. the left border. The left border of the shrunk MBR has limited choices. That is, it has to be the same as the X value of some object. In Figure 5(a), if e is removed, the left border is shrunk to $X = 6$. If both a and h are also removed, the left border is shrunk to $X = 7$. And so on. Notice that if we only remove one of a or h , the left border cannot be shrunk to $X = 7$. So if $p = 2$, we need to pick only one object e from the left border. This observation illustrates

that, in order to guarantee the picking of a superset of the p-boundary, the number of objects that need to be picked from each border may be less than p .

Definition 5. Given a border B of the MBR of a set of spatial objects S , a **level** is the set of objects in S that have the same distance to B . We denote the i^{th} level, $i \geq 0$, for the left border as $LEFT(i)$. Here $i = 0$ stands for the level of objects on the border. Similarly, $RIGHT(i)$, $TOP(i)$ and $BOTTOM(i)$ stands for the i^{th} level for the right, top and bottom borders, respectively.

For instance, $LEFT(0) = \{e\}$, $LEFT(1) = \{a, h\}$, $TOP(0) = \{a, b\}$. In order to guarantee that we pick a superset of the p-boundary, it is sufficient to pick for each border B all objects in levels $0, \dots, k-1$ such that $\sum_{i=0}^{k-1} |B(i)| \leq p < \sum_{i=0}^k |B(i)|$. Here the number k is called the *sufficient number* of levels for that border regarding to p . The union of the k levels is sometimes called the *sufficient levels*. Let t_p, l_p, r_p , and b_p denote the sufficient number of levels for the top, left, right, and bottom borders.

The *border structure* maintains the sufficient levels of objects for each border. To ensure efficient manipulation, the sufficient levels of objects for each border are maintained in a balanced binary search tree, ordered by the closeness to that border. Since an object may exist in multiple binary trees, we also keep an *object array* to store all objects which appear in any of the four trees. The array contains pointers which link with the objects in the trees.

For instance, the border structure corresponding to Figure 5(a) is shown in Figure 5(b). Here $p = 4$, and we only show the links for object a .

As illustrated in the figure, the border structure also keeps, for each border, a *coordinate array* which stores the X or Y coordinates of the sufficient number plus one level. For instance, for the left border, the sufficient number of levels is 2, and we keep 3 choices of low X . Starting from the low X of the left border, we have: $LX[0] = 5$, $LX[1] = 6$ and $LX[2] = 7$. Similarly, the coordinate arrays HX , LY and HY correspond to the right, bottom and top borders.

To sum up, the border structure is composed of: (a) four binary trees; (b) an object array; and (c) four coordinate arrays. Objects in the binary trees are linked with the corresponding objects in the object array.

Given a set S of spatial objects and a number p , to build the border structure, we scan through all objects in S , while building the four binary trees. In the process, we always make sure that each of the four trees contains no more than p objects. That is, if a tree already has p objects and a new object needs to be inserted in, a level of objects is removed from the tree. In order to link the object with the object array, we first need to determine if the object appears in some other tree. If it is not, the object is appended to the end of the object array. In any case, a double pointer is added to link the object in the tree to the object array. Finally, after all objects are scanned, we scan the four trees to construct the coordinate arrays.

One implementation issue is how to maintain a node in the binary trees, since the node may contain multiple objects. We should keep the objects in order, and the order depends on the particular border. For the left and right borders, the objects in a node have the same X coordinates, and thus they should be ordered by the Y coordinates. For the top and bottom borders, the objects are ordered by X . Since

Algorithm *pick-p*

Input: A set S of spatial objects and a value p .

Action: Output the p-boundary.

1. Build the border structure.
 2. $g_{max} = 0, k = 0$.
 3. **for** every combination of l, r, t and b , where $0 \leq l \leq l_p, 0 \leq r \leq r_p, 0 \leq t \leq t_p$ and $0 \leq b \leq b_p$
 - 3.1. Let R be the rectangle whose lower-left and upper-right corners are $(LX[l], LY[b])$ and $(HX[r], HY[t])$, respectively. Let T be the objects in S that are outside R .
 - 3.2. **if** R is a *valid* MBR and $|T| \leq p$
 - i. **if** $(g_{max} < G(S, S - T))$ or $(g_{max} = G(S, S - T))$ and $k > |T|$,
 - A. $ll = l, rr = r, tt = t, bb = b$.
 - B. $g_{max} = G(S, S - T), k = |T|$.
 - 3.3. **end if**
 - 3.4. **end if**
4. Let R be the rectangle whose lower-left and upper-right corners are $(LX[ll], LY[bb])$ and $(HX[rr], HY[tt])$, respectively. Return the objects in S that are outside R .
-

Figure 6: Algorithm *pick-p*.

dynamic search/insertion/deletion is needed, these objects (that correspond to one level) should be organized into a smaller binary search tree.

THEOREM 4. *Given a set of n spatial objects, the time complexity to build the border structures is $O(n \log p)$.*

3.2 Exhaustive Search to Find the Optimal Answer

To find the optimal p-boundary and minP-boundary, we need to probe every possible combinations of shrinking the borders. For those combinations that lead to a valid MBR which excludes no larger than p objects, we examine the gains. Figure 6 shows the algorithm *pick-p* which finds the optimal p-boundary.

The algorithm *pick-p* checks all valid combinations of left, right, top, and bottom borders of the shrunk rectangle, and chooses the one with the maximum gain. Here a combination is valid if (a) the MBR is valid; and (b) the number of removed objects is no more than p . An MBR is valid, if there exists objects on every border of the rectangle. The reason why we need to check the validity of an MBR is as follows. Given a combination of l, r, t , and b , we expect to shrink the rectangle to certain coordinate values. For instance, we expect that the left border of the shrunk rectangle is $X = LX[l]$. However, all the objects in S whose lower X coordinates are $LX[l]$ may have been removed via the shrinking of some other borders.

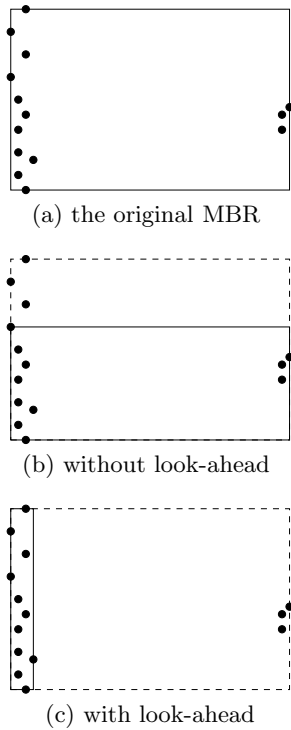


Figure 7: The greedy algorithm with look-ahead has better performance. Here $p = 3$.

An important optimization is: if we see a given combination l, r, b, t where the number of removed objects is no less than p , we should omit the examination of all combinations l', r', b', t' such that $l' \geq l, r' \geq r, t' \geq t$, and $b' \geq b$. This is because such combinations will lead to the removal of more than p objects and thus are not valid.

The algorithm to find the minP-boundary, which we call the *pick-minP*, is similar to *pick-p*. The difference is that *pick-minP* stores the intermediate results. When a valid combination of l, r, t, b is met, *pick-p* only keeps it if it corresponds to the best choice, as shown in step 3.2(i) of Figure 6. But *pick-minP* will maintain all combinations which correspond to distinct gains. For each distinct gain, only the combination which corresponds to the smallest number of removed objects is kept. At the end, *pick-minP* gets an array of shrinking choices, in increasing order of gain. No two maintained choices have the same gain. Note that this array is also in increasing order of the number of removed objects. At the end, *pick-minP* picks the combination whose gain is the smallest among those no less than $\beta \cdot G(S, p\text{-boundary})$.

THEOREM 5. *The space complexity of the algorithm pick-minP is $O(p^4)$.*

An optimization which requires less space is to dynamically remove the choices where the gain is less than β times the current max gain. Since $G(S, p\text{-boundary})$ is no less than the current max gain at any point, the removed choices cannot be the result of *pick-minP* anyways.

THEOREM 6. *The time complexity of algorithms pick-p and pick-minP is $O(n \log p + p^4 \log p)$.*

Algorithm *greedy-pick-p*

Input: A set S of spatial objects, and values p, m .

Action: Output an approximate p-boundary.

1. Build the border structure.
 2. Let M be the MBR of S . Let $P = \emptyset$.
 3. **loop**
 - 3.1. Set $g_{max} = 0, B_{max} = LEFT$ and $k_{max} = 0$.
 - 3.2. **for** every border B of M , **for** every $k \in [1..m]$ such that there exist k more unremoved levels for B and the number of objects in these k levels plus $|P|$ is no larger than p
 - i. Compute the average gain g per removed object, if we remove the k levels from border B .
 - ii. If $g > g_{max}$, set $g_{max} = g, B_{max} = B$ and $k_{max} = k$.
 - end for**
 - 3.3. If $g_{max} = 0$, **break**.
 - 3.4. Remove k_{max} levels from border B_{max} and add the removed objects to P . Let M be the new MBR.
 - end loop**
 4. Return P as the p-boundary.
-

Figure 8: Algorithm *greedy-pick-p*.

3.3 Greedy Algorithms with Better Performance

To get better space and time complexity, we propose to use the greedy algorithms, which shrink one border at a time. The basic idea is to always pick the border which, if we remove all objects on it, will result in the maximum gain per removed object. However, this may not result in a good gain, as shown in Figure 7. To shrink from Figure 7(a), the first step of the naive greedy algorithm is to shrink from the top border. Same with the next two steps. The result is shown in Figure 7(b). But as shown in Figure 7(c), shrinking from the right is much better.

We propose to use a greedy algorithm with look-ahead, which will shrink from the right border in our example. The idea is, to determine how good it is to shrink from a particular border, we examine multiple levels from that border. If we look ahead m levels, we compute the average gain per removed object for removing $1, \dots, m$ levels. Out of these, we pick the one with the largest average gain. The greedy algorithm, which looks ahead m levels, to find the p-boundary is shown in Figure 8.

Similar to how the algorithm *pick-p* is extended to *pick-minP*, we can extend the algorithm *greedy-pick-p* to *greedy-pick-minP*, which finds an approximate minP-boundary. The extension is to keep the intermediate removal results, and at the end choose the earliest intermediate result where the gain is no less than β times the largest gain.

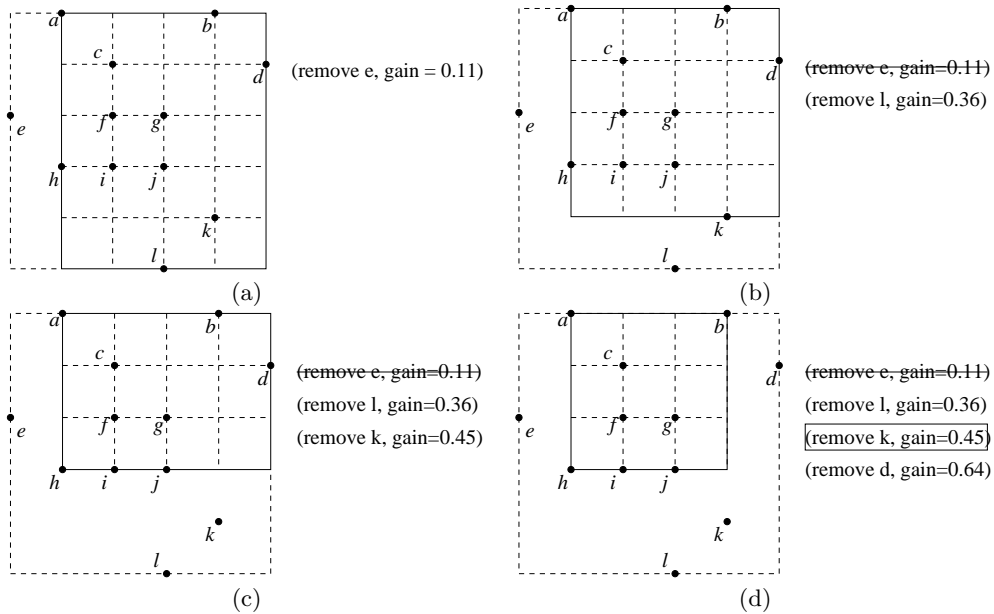


Figure 9: An running example of greedy-pick-minP. Here $p = 4$, $\alpha = 0.5$, $\beta = 60\%$, and $m = 1$.

Name	# of objects	Description
Postal dataset	123,593	postal locations in New York, Philadelphia and Boston
Street dataset	131,461	streets in Los Angeles

Table 1: Datasets used in the experiments.

Figure 9 shows a running example for finding the minP-boundary using the data in Figure 5(a). Here, for the first step of greedy-pick-minP, shrinking any of the left, bottom or right borders will have the same gain=0.11. Let us assume it picks the left border to shrink (Figure 9a). The next step has only one best choice, which is to shrink the bottom border. The gain from the original MBR is 0.36 (Figure 9b). At this point, since $0.36 * 60\% = 0.216 > 0.11$, there is no need to keep the record with gain 0.11 any more. It will not be the result for the minP-boundary. Next, the MBR is shrunk from the bottom again, and the gain is 0.45 (Figure 9c). Finally, the last step is to shrink from the right border. Overall, the gain to remove $p = 4$ objects is 0.64. The computed p-boundary is the set $\{e, l, k, d\}$. Since 0.45 is the smallest gain (in the maintained array) no less than $0.64 * 60\% = 0.384$, the minP-boundary is $\{e, l, k\}$.

The efficiency of the algorithm is guaranteed by the following theorems:

THEOREM 7. *The space complexity of the algorithm greedy-pick-minP is $O(p)$.*

THEOREM 8. *The time complexity of the algorithms greedy-pick-p and greedy-pick-minP is $O(n \log p + p \cdot m \log p)$.*

Discussion: If $\alpha = 1$, the greedy algorithm does not work for the case when an MBR is a square. The reason is that, when $\alpha = 1$, shrinking from only one edge may not bring any gain (according to the quality formula). To handle this special case, we can extend our algorithm so that we look ahead from two adjacent borders (e.g. TOP and BOTTOM) at the same time.

4. EXPERIMENTAL EVALUATION

4.1 Experimental Setup

In our experiments, the real datasets and the original R*-tree's implementation are taken from R-tree-Portal [8]. One dataset contains 123,593 point data (postal addresses), which represent three metropolitan areas (New York, Philadelphia and Boston). Another dataset contains line segments, which represent 131,461 streets of Los Angeles. We denote the point dataset as the *Postal dataset* and the line segments dataset as the *Street dataset*. The datasets are summarized in Table 1. The programs are coded in Java. All our experiments are performed on a Dell PC with a 2.66-GHz Pentium 4 processor.

Unless otherwise stated, each tree node has a capacity of 1KB which contains about 50 entries, and p is set to be 30% of the node capacity. The minimum fan-out of the R*-trees is 40% of the node capacity. We also utilize a LRU buffer which can hold 128 pages, and we assume that initially only the root node is in the buffer. In section 4.2, we use *pick-P* and *greedy-pick-P* in our experiments, and in section 4.4, we use *pick-minP* and *greedy-pick-minP*. Without ambiguity, we denote both *pick-P* and *pick-minP* as the **exhaustive** algorithms, and both *greedy-pick-P* and *greedy-pick-minP* as the **greedy** algorithms. In all exhaustive algorithms and greedy algorithms, we use $\alpha = 0.5$ and $\beta = 0.9$. Especially, our greedy algorithms look ahead 5 levels. The original algorithm to pick p objects in R*-tree is denoted as the **original** algorithm. In addition, to reduce the randomness, we report the average results of 100 runs.

This section is organized as follows. Section 4.2 presents the comparisons of *gain*, and section 4.3 compares the disk I/Os of building the R*-trees. Finally in section 4.4, we compare the performances of each algorithm in R*-tree using the range queries.

4.2 Gain Comparison

We first experiment with the gains achieved from removing $p = 30\%$ objects by our algorithms (*pick-P*, *greedy-pick-P*) and the original algorithm. The experiments in this section use the Postal dataset.

In this experiment, we randomly generate rectangles with size of 0.1% of the whole space, and among the Postal dataset objects contained in each rectangle, we randomly pick a fixed number of objects. Figure 10 shows the result of this experiment.

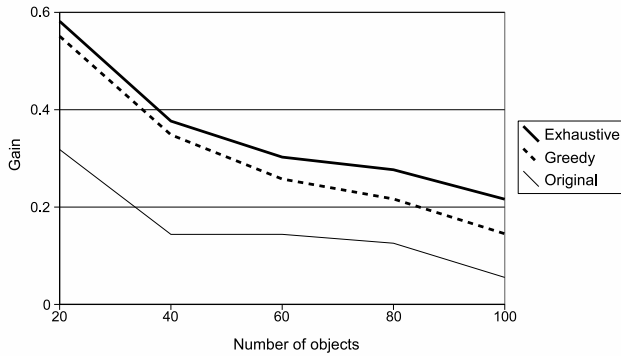


Figure 10: Comparison of gains under various number of objects.

In general, as the number of objects in a rectangle with fixed size increases, the gain which can be achieved decreases, as shown in Figure 10. It is because the number of objects remained in the rectangle increases. Also, since the number of objects increases, chances are that multiple objects crowd around borders, and the original algorithm will less possibly achieve the optimal gain, as we discussed in the previous sections. Both the greedy algorithm and exhaustive algorithm are always better than the original algorithm.

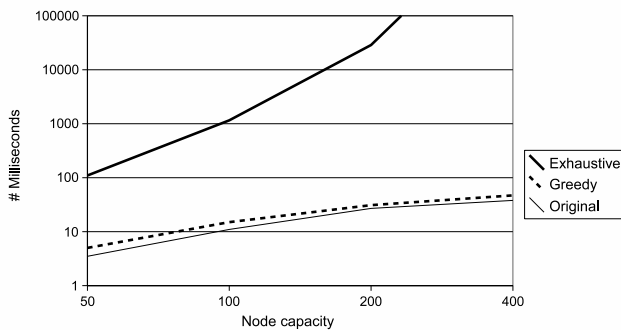


Figure 11: Comparison of the running time of various algorithm to pick 30% objects on the boundary.

Furthermore, we compare the actual running time of the three algorithms. The results are shown in Figure 11. Notice in Figure 11, we used logarithmic scale in the time axis. It shows that when the number of nodes increases, the exhaustive algorithm is not acceptable as it takes too much time. The cost of the exhaustive algorithm almost dominates the CPU cost of building an R*-tree. On the other hand, our greedy algorithm costs much less time than the exhaustive algorithm, close to that of the original algorithm.

4.3 Index Construction Time

We further compare the disk I/Os of building the R*-trees using the greedy algorithm (*greedy-pick-minP*) and the original algorithm, as shown in Figure 12. Notice that since the street data are stored in a special order such that objects close to each other are stored together, loading the street data costs much less I/Os than loading the postal data. In both cases, the improved version of the R*-tree has better index construction time. This is because our greedy algorithm allows the removal of less than p objects, which saves the construction time.

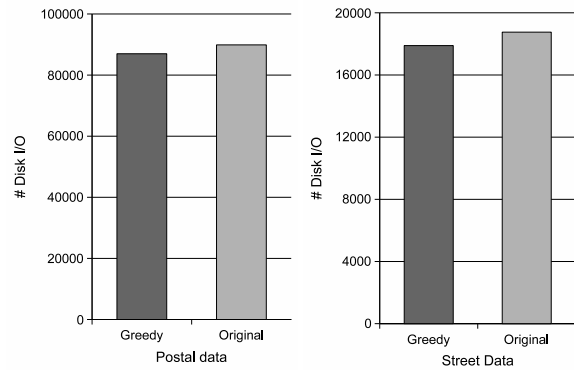
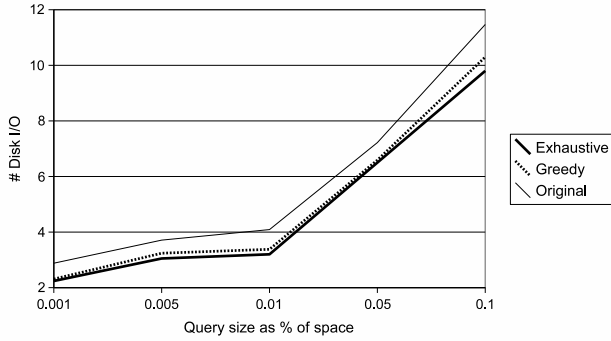


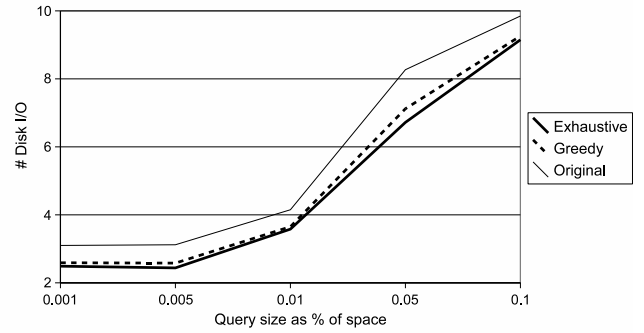
Figure 12: Comparison of the disk I/Os of the tree constructions.

4.4 Range Query Performance

In this section, we use both Postal dataset and Street dataset in all experiments. We first compare the range query performance of the original R*-tree (*Original*) and our improved versions by integrating *pick-minP* (*Exhaustive*) and *greedy-pick-minP* (*Greedy*) into the R*-tree. We perform these experiments by fixing the buffer sizes and page sizes of the trees and varying the query size. The heights of the trees are all 4. Figure 13(a) shows the query performance on the Postal dataset and Figure 13(b) shows the performance on the Street dataset. In both cases, our algorithms have better performance, up to 20% improvement. By improving the shapes and sizes of the MBRs of the nodes(pages) in the tree, queries of small sizes will have chances to touch less nodes(pages) in our improved R*-trees than the original R*-tree. As shown in Figure 13, the improvement increases with the decrease of the query size. Also, by comparing Figure 11 and Figure 13, we show that the query performance of the greedy algorithm is quite close to that of the exhaustive algorithm, while its running time is as good as that of the original algorithm.

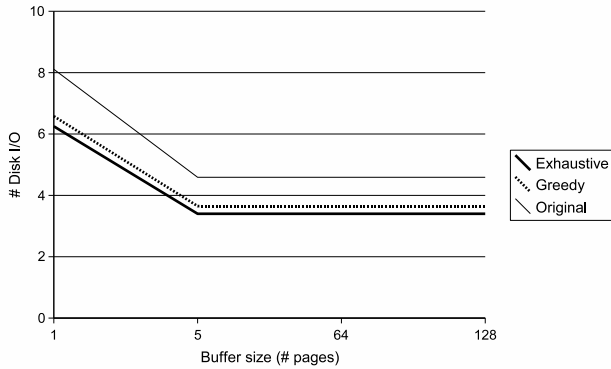


(a) the Postal dataset

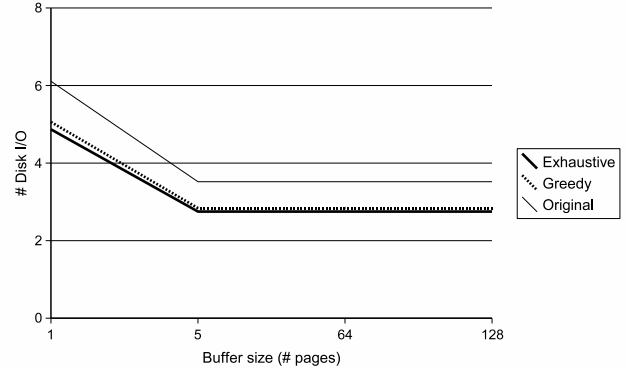


(b) the Street dataset

Figure 13: Range query performance varying the query size.



(a) the Postal dataset



(b) the Street dataset

Figure 14: Range query performance varying the buffer size.

In the second set of experiments, we compare the effects of the buffer sizes on the query performance. We performance this set of experiments by fixing the page sizes while varying the buffer size. The heights of the trees are also 4. At the query time, we fix the query size to be 0.01% of the whole space. We first use the *root buffer*, which always buffers the root page but nothing else. Then we use the *path buffer*, which could hold 5 pages and always contains the current search path. Besides, LRU buffers of larger sizes are also used. The results of our experiments are shown in Figure 14. For the range queries, the path buffer is as effective as larger LRU buffers. We also conclude that the performance difference between our algorithms and the original algorithm is almost unaffected by varying the buffer sizes.

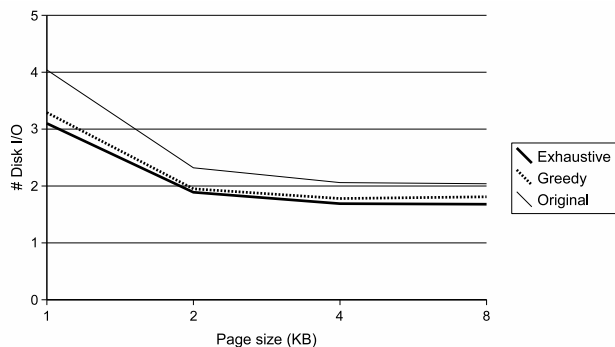
In the third set of experiments, we further compare the effects of the page sizes on the query performance. In these experiments, we generate the R*-trees using the original algorithm and our improved versions, with pages sizes of 1KB, 2KB, 4KB and 8KB. The capacities of those kinds of pages are about 50, 90, 170 and 340 entries, respectively. The height of a tree with 1KB page size is 4, and the heights of trees with 2KB, 4KB and 8KB page sizes are 3.

At the query time, we fix the query size to be 0.01% of the whole space. Figure 15 shows the results on the Postal dataset and the Street dataset. The number of disk I/Os decreases quite much from 1KB to 2KB pages sizes, due to the decrease of the tree height. Query performances on the trees with page sizes of 2K, 4K and 8K do not show large differences, since they have the same tree height. However, with the increase of the number of objects in each node, the difference between our algorithms and the original algorithms also decreases.

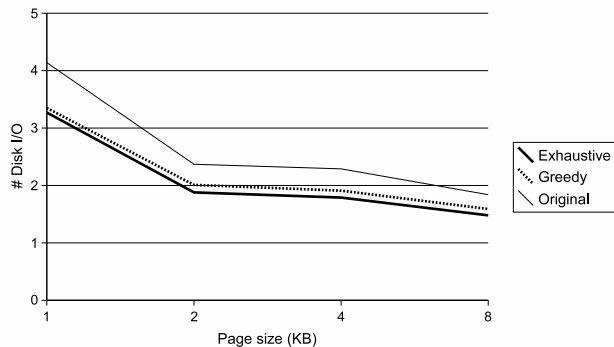
In addition, we compare the effects of various α values (used to determine the *quality*). Figure 16 shows range query performances with different α values. Generally, there is not much difference of disk access times among those α values, but the cases when $0.4 \leq \alpha \leq 0.6$ show more robust performance. We recommend to choose $\alpha = 0.5$.

5. ACKNOWLEDGEMENT

We would like to thank Prof. Betty Salzberg for reviewing the paper and providing helpful suggestions.



(a) the Postal dataset



(b) the Street dataset

Figure 15: Range query performance varying the page size.

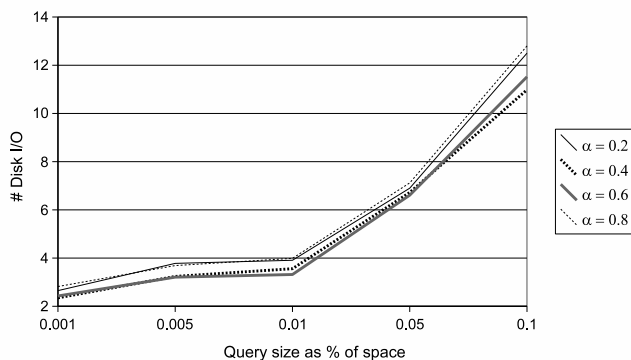


Figure 16: Comparison of α values.

6. CONCLUSIONS AND FUTURE WORK

In spatial index structures like the R*-tree, the MBRs should have high *quality*. That is, their areas should be small and their shapes should be close to squares. The R*-tree's reinsertion algorithm tries to achieve this goal by the following method. When a page overflows, some objects the farthest to the MBR's center are removed and reinserted into the index. In this paper we mathematically defined the *quality* of a rectangle and the *gain* to remove some objects from an MBR. Based on our definitions, we proposed algorithms to locate the set of objects to be removed with the maximal gain. These algorithms have better performances than that of the original R*-tree. We also improved the R*-tree in various other ways. The improved version of the R*-tree was experimentally compared with the original R*-tree on real spatial data sets. We have witnessed up to 20% improvement in range query performance. To extend this work, we are currently examining the idea to store selected objects in index nodes, thereby further shrink the size of MBRs at lower levels of the index.

7. REFERENCES

- [1] N. An, K. Kanth, and S. Ravada. Improving Performance with Bulk-Inserts in Oracle R-Trees. In *VLDB*, 2003.
- [2] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. In *ACM SIGMOD*, pages 322–331, 1990.
- [3] V. Gaede and O. Günther. Multidimensional Access Methods. *ACM Computing Surveys*, 30(2):170–231, 1998.
- [4] A. Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. In *ACM SIGMOD*, pages 47–57, 1984.
- [5] K. V. R. Kanth, A. El Abbadi, D. Agrawal, and A. K. Singh. Indexing Non-Uniform Spatial Data. In *Proc. of Int. Database Engineering & Applications Symposium (IDEAS)*, 1997.
- [6] K. V. R. Kanth, S. Ravada, and D. Abugov. Quadtree and R-tree Indexes in Oracle Spatial: A Comparison using GIS Data. In *ACM SIGMOD*, pages 546–557, 2002.
- [7] K. V. R. Kanth, S. Ravada, J. Sharma, and J. Banerjee. Indexing Medium-dimensionality Data in Oracle. In *ACM SIGMOD*, pages 521–522, 1999.
- [8] Y. Theodoridis. The R-tree-portal. <http://www.rtreeportal.org>, 2003.
- [9] J. S. Vitter. External Memory Algorithms and Data Structures. *ACM Computing Surveys*, 33(2):209–271, 2001.