Chapter 4

Region Algebra Implementation

This chapter describes how the region algebra is implemented in LAPIS. The key aspect of the implementation is the representation chosen for region sets, which has two parts:

- A *region rectangle* is a rectangle in region space. Region rectangles are a compact way to represent the result of applying a relational operator to a region.
- A *rectangle collection* represents a region set as a union of region rectangles. We can draw on research in computational geometry to find efficient data structures for rectangle collections.

Section 4.6 describes how the basic implementation can be optimized to significantly improve its performance in special cases that are common in practice.

4.1 Region Rectangles

A *region rectangle* is a tuple of four integers, (s_1, e_1, s_2, e_2), which represents a set of regions as a closed rectangle in region space:

$$(s_1, e_1, s_2, e_2) \equiv \{[s, e] | s_1 \le s \le s_2 \wedge e_1 \le e \le e_2\}$$

Essentially, a region rectangle is a set of regions whose start point and end point must fall into the specified half-open interval. Figure 4.1 shows some region rectangles corresponding to region sets in a text string.

A few facts about region rectangles follow immediately from the definition:

- The single region $[s, e]$ corresponds to the region rectangle (s, e, s, e).
- The set of all possible regions Ω in a string of length n is the region rectangle $(0, 0, n, n)$.
- One region rectangle is a subset of another, $(s_1, e_1, s_2, e_2) \subseteq (s_1', e_1', s_2', e_2')$, if and only if $s_1' \le s_1 \le s_2 \le s_2'$ and $e_1' \le e_1 \le e_2 \le e_2'$.
- One region rectangle intersects another, $(s_1, e_1, s_2, e_2) \cap (s_1', e_1', s_2', e_2') \ne \emptyset$, if and only if $s_1 \le s_2'$, $s_1' \le s_2$, $e_1 \le e_2'$, and $e_1' \le e_2$.

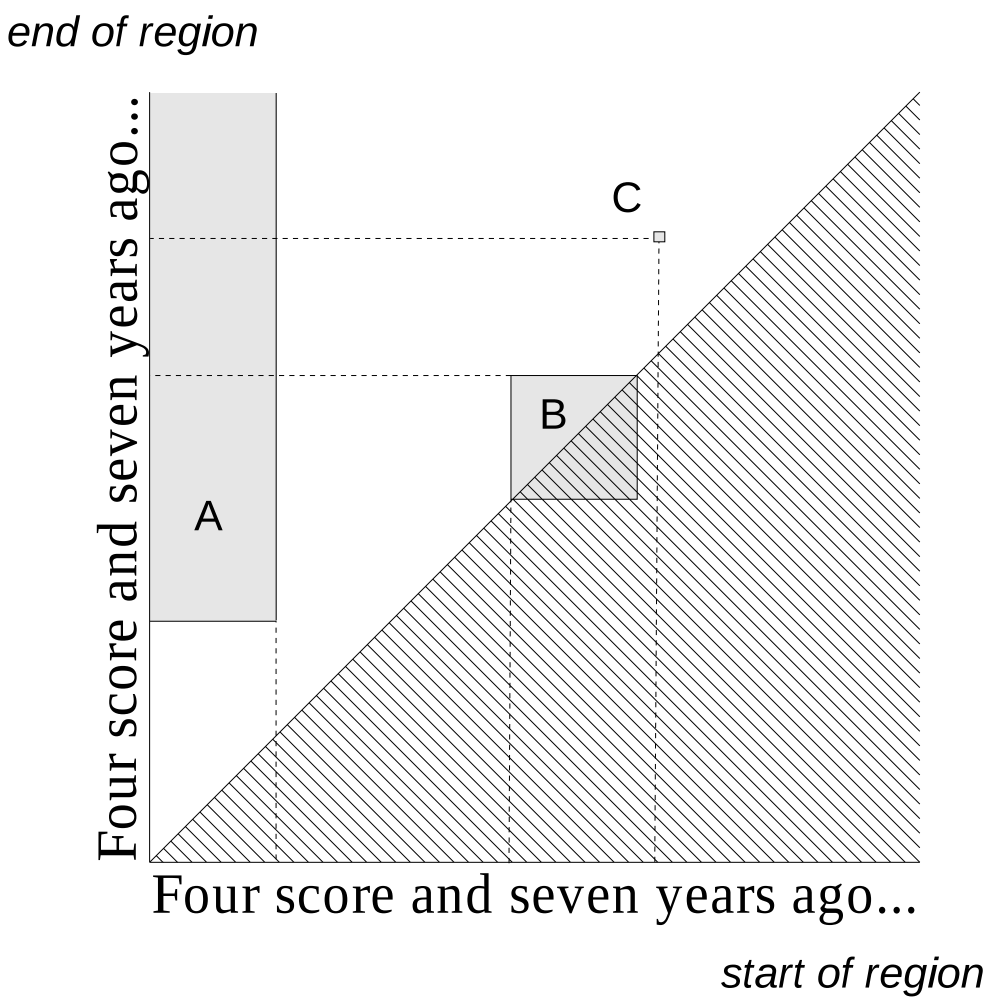
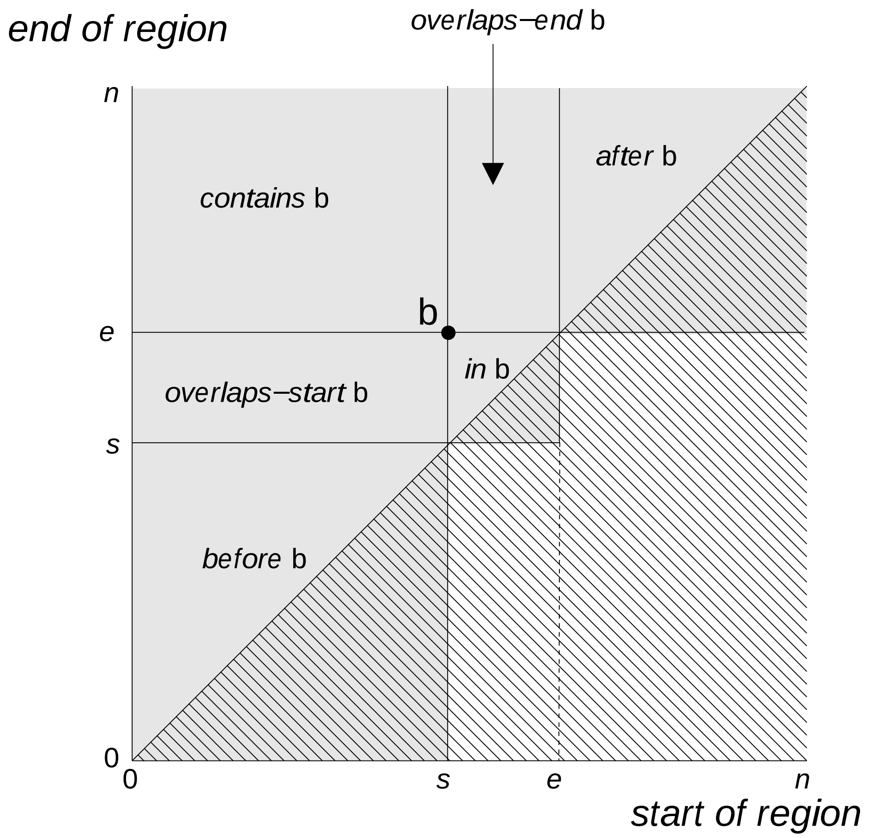

end of region
Four score and seven years ago...
C
B
A
Four score and seven years ago...
start of region

Figure 4.1: Region rectangles depicted in a text string and in region space. Region rectangle A is the set of all regions that *contains* the word “score”, B is the set of regions that are *in* the word “seven”, and C consists of the single region “years”.



$$b = [s, e]$$

before b	$=$	$(0, 0, s, s)$	*after* b	$=$	(e, e, n, n)
overlaps-start b	$=$	$(0, s, s, e)$	*overlaps-end* b	$=$	(s, e, e, n)
contains b	$=$	$(0, e, s, n)$	*in* b	$=$	(s, s, e, e)

Figure 4.2: Region relations can be represented by rectangles.

A region rectangle is capable of representing certain region sets very compactly. With only four integers, a region rectangle can describe a set as large as Ω. One particularly interesting kind of region set can be represented with a region rectangle: the result of a region relation operator. Recall from Figure 3.6 that a region space map shows how every region in region space is related to a given region b. The map has areas for all the fundamental relations: *before* b, *overlaps-start* b, *contains* b, *in* b, *overlaps-end* b, and *after* b. Each of these areas is the result of applying an algebraic operator to the set $\{b\}$.

The key insight is that every area in b's region space map can be represented by a rectangle, as shown in figure 4.2. Some areas are already rectangular (*contains* b, *overlaps-start* b, and *overlaps-end* b). Other areas (*before* b, *in* b, and *after* b) are triangular, cut by the $45°$ diagonal. But even these areas can be represented by a rectangular area, part of which extends below the diagonal, as long as the part below the diagonal is implicitly ignored. Thus, the result of applying any of the six region relation operators to a region $[s, e]$ can always be represented by one rectangle.

We can go beyond single regions $[s, e]$, however. Applying a relational operator to any region rectangle (s_1, e_1, s_2, e_2) will also produce a rectangle. In other words:

Claim 2. *The set of region rectangles is closed under the relational operators* before, after, overlaps-start, overlaps-end, in, *and* contains.

Figure 4.3 demonstrates a geometric proof of Claim 2.

This closure property is the reason why region rectangles are so useful as a fundamental representation. Thanks to the closure property, if a region set A is represented by N rectangles, then *op* A can also be represented by N rectangles, for any relational operator *op*. The closure property also extends to the derived relational operators that were defined in Chapter 3, including *just-before*, *just-after*, *starting*, *ending*, and *overlaps*. A few of these operators are illustrated in Figure 4.3.

Region rectangles are also closed under set intersection, since the intersection of two rectangles is also a rectangle. Region rectangles are not closed under union or difference, however.

4.2 Rectangle Collection

Any region set can be represented by a union of region rectangles. Some examples of rectangle collections for typical region sets are shown in Figures 4.4–4.6.

A *rectangle collection* is the fundamental representation for a region set in LAPIS. For now, we will treat a rectangle collection as an abstract data type representing a set of points in region space. Points can be queried, added, and removed as axis-aligned rectangles with integer coordinates. A rectangle collection C supports four operations:

- QUERY(C, r) searches C for all points that lie in the query rectangle r. The result is a stream of rectangles, not necessarily disjoint, so the same point can be returned more than once in different rectangles. All points in the rectangle collection are enumerated by QUERY(C, Ω).

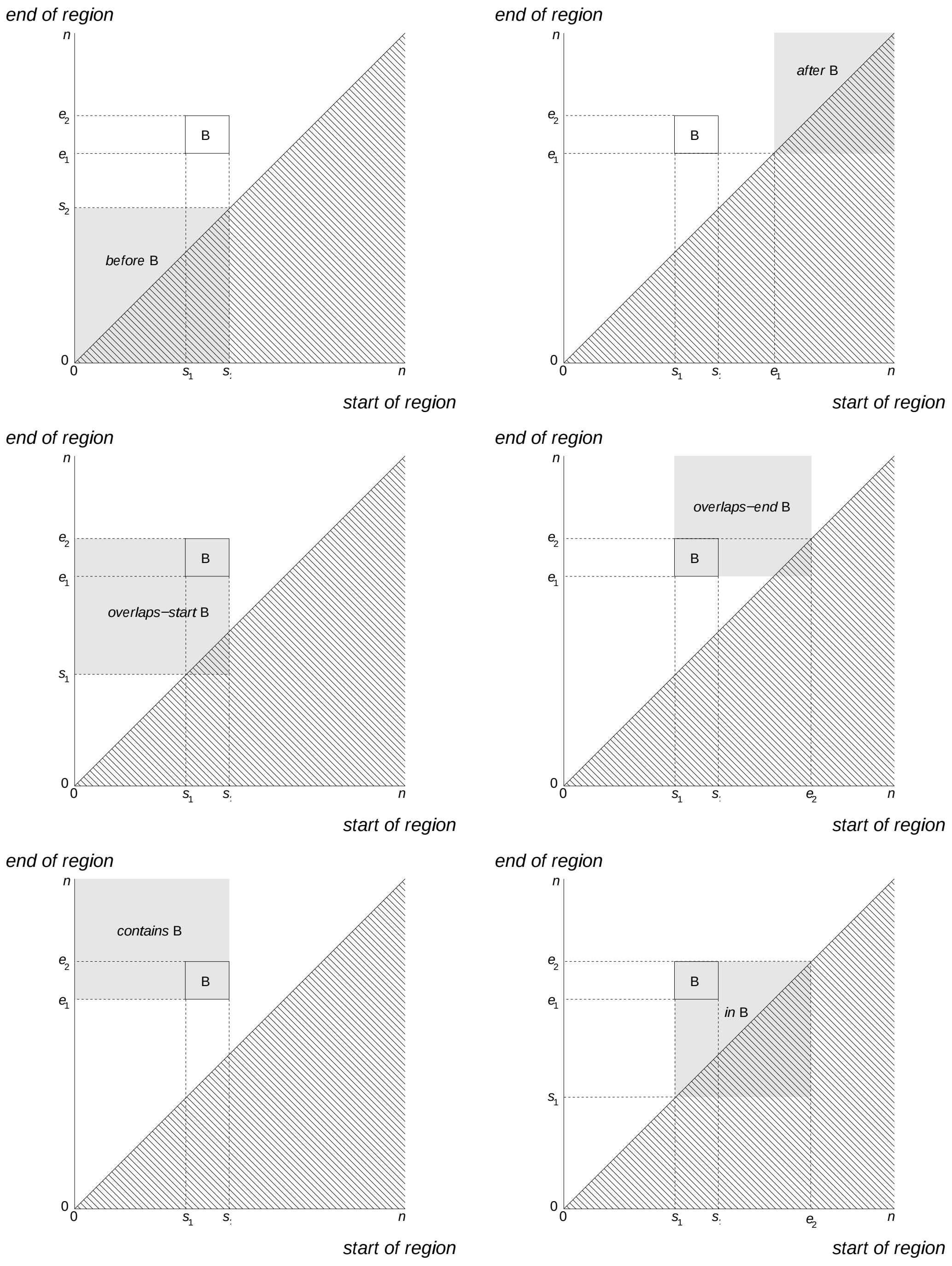
- INSERT(C, r) inserts a rectangle r into C, so that the set of points represented by C now includes all the points in r.

- DELETE(C, r) deletes a rectangle r from C, so that the set of points represented by C no longer includes any of the points in r.

- COPY(C) duplicates a rectangle collection, hopefully faster than enumerating its contents and inserting them into an empty collection.

These operations are extensions of the conventional membership test, insert, and delete operations for sets. Instead of taking a single element to query, insert, or delete, the operations take a region rectangle, a set of related elements.

A rectangle collection only needs to represent the region set faithfully, not the particular set of rectangles that were inserted to create it. A rectangle collection can merge rectangles, split rectangles, and throw away redundant rectangles, in order to store the collection more compactly or make queries faster. As a result, QUERY(C, Ω) need not return the same set of rectangles that were inserted into the collection, as long the union of the rectangles returned by QUERY is identical to the union of the inserted rectangles.



$$B = (s_1, e_1, s_2, e_2)$$

$$\begin{array}{rclrcl} \textit{before}\ B & = & (0, 0, s_2, s_2) & \textit{after}\ B & = & (e_1, e_1, n, n) \\ \textit{overlaps-start}\ B & = & (0, s_1, s_2, e_2) & \textit{overlaps-end}\ B & = & (s_1, e_1, e_2, n) \\ \textit{contains}\ B & = & (0, e_1, s_2, n) & \textit{in}\ B & = & (s_1, s_1, e_2, e_2) \end{array}$$

Figure 4.3: Applying a relational operator to a region rectangle always produces another rectangle.

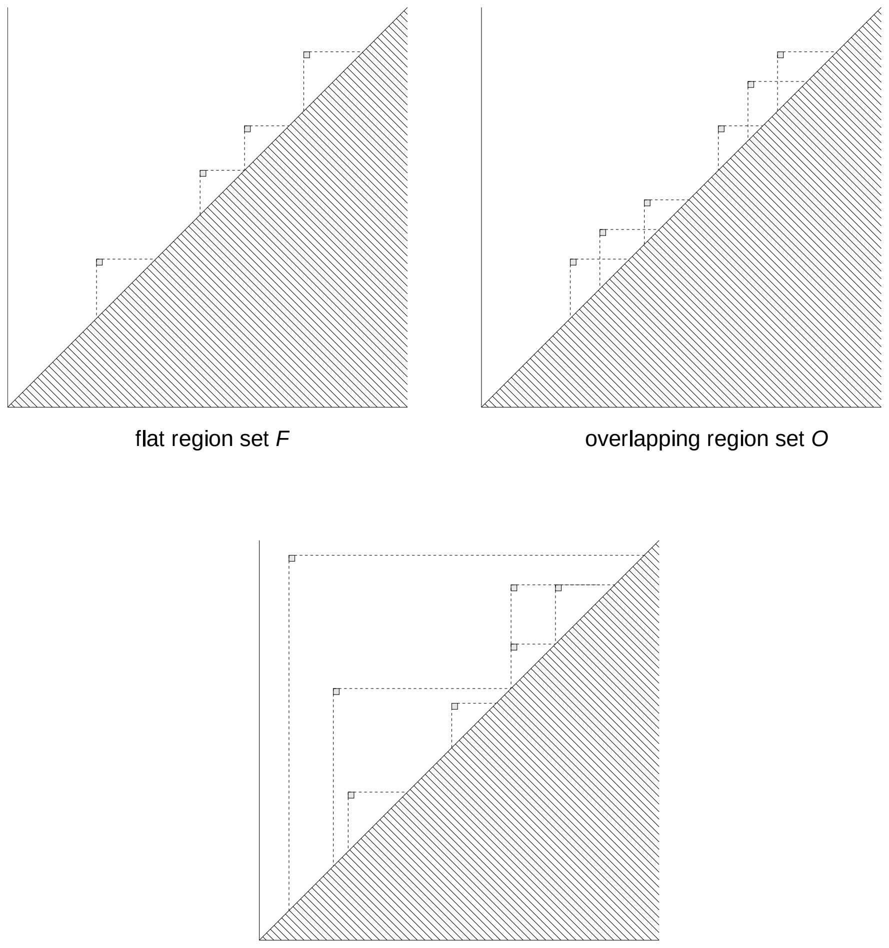


Figure 4.4: Rectangle collections for flat, overlapping, and nested region sets. Each rectangle covers only a single point. Dashed lines show that each region set meets the desired definition. Regions in F must be *before* or *after* each other. Regions in O may also *overlap-start* or *overlap-end*. Regions in N may be related by *before, after, in,* or *contains.*

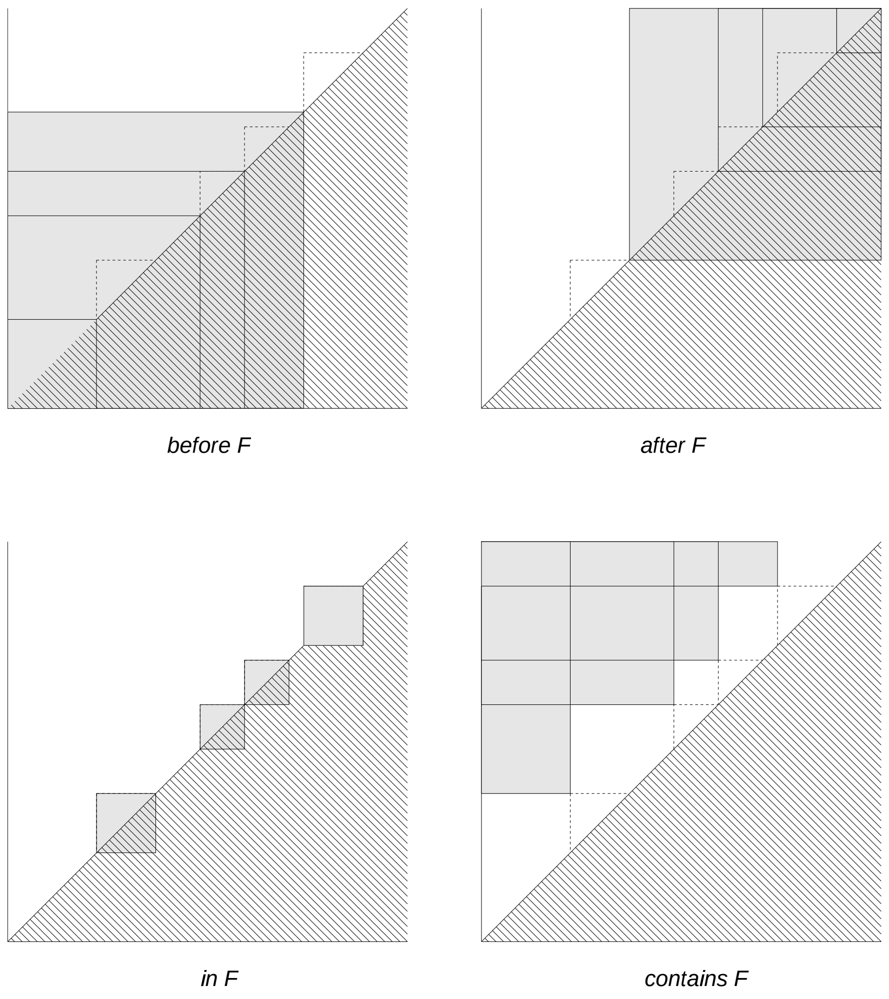


Figure 4.5: Rectangle collections for relational operators applied to the flat region set F from Figure 4.4. Each collection contains four rectangles, which may intersect. Dashed lines show the location of the original region set F.

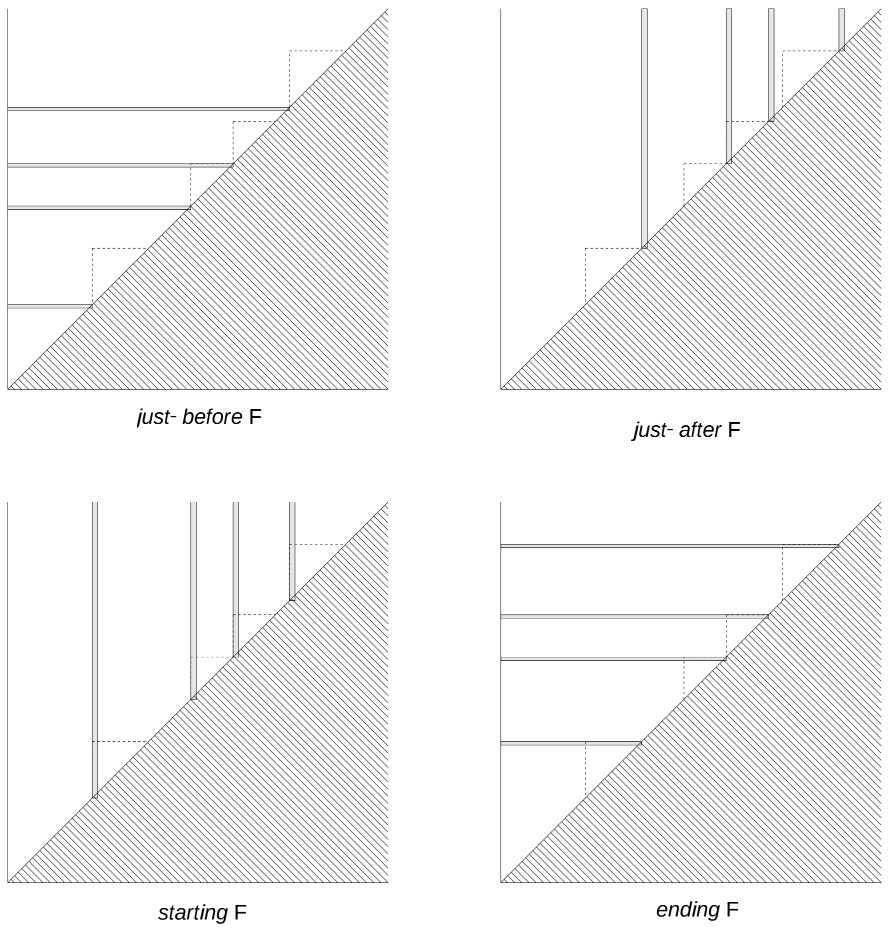


Figure 4.6: More relational operators applied to the flat region set F from Figure 4.4. Dashed lines show the location of the original region set F.

4.3 Implementing the Region Algebra

Now we are ready to implement the algebra operators using rectangle collections. Recall that the region algebra described in Section 3.5 has the following operators:

- Relational operators: *before, after, in, contains, overlaps-start, overlaps-end*.
- Set operators: $\cup$, $\cap$, $-$.
- Iteration operator: *forall*.

The relational operators are implemented by Algorithm 4.1. The key line of the algorithm is line 3, in which the relational operator *op* is applied to a rectangle, producing another rectangle which is then inserted into the result. Figure 4.3 shows the rectangle produced by each relational operator.

Algorithm 4.1 RELATION applies a relational operator to a rectangle collection.

```
RELATION( op , $A$)
1  $U \leftarrow$ new RECTANGLECOLLECTION
2  for each $r$ in $A$
3  do INSERT($U$, op $r$)
4  return $U$
```

Set union, intersection, and difference are implemented by Algorithms 4.2–4.4. Since union and intersection are commutative, A and B can be passed in any order. UNION and INTERSECTION are written under the assumption that $|A| > |B|$, in the sense that enumerating the rectangles in A takes longer than enumerating the rectangles in B. Thus UNION copies A instead of enumerating it, and INTERSECTION queries A instead of enumerating it. If the caller can cheaply determine which collection is larger, then this optimization may save some time.

Algorithm 4.2 UNION finds the union of two rectangle collections.

```
UNION($A, B$)
1  $U \leftarrow$ COPY($A$)
2  for each $r$ in $B$
3  do INSERT($U, r$)
4  return $U$
```

Finally, the *forall* operator is shown in Algorithm 4.5. FORALL iterates through all the rectangles in a rectangle collection and applies a function f to every rectangle. Each application of f returns a stream of rectangles, which are inserted into a new rectangle collection and returned. The function f takes a rectangle (s_1, e_1, s_2, e_2) and produces a stream of rectangles that result from applying the body of the *forall* expression to every region $[s, e]$ in the rectangle. Although in general this may require iterating through all pairs $[s, e]$ such that $s_1 \leq s < s_2$ and $e_1 \leq e < e_2$, in practice f can compute a result for the entire rectangle at once. All the uses of *forall* in the examples in Chapter 3 behave this way.

Algorithm 4.3 INTERSECTION finds the intersection of two rectangle collections.

```
INTERSECTION($A, B$)
1  $U \leftarrow$ new RECTANGLECOLLECTION
2  for each $r$ in $B$
3  do for each $r'$ in QUERY($A, r$)
4         do INSERT($U, r'$)
5  return $U$
```

Algorithm 4.4 DIFFERENCE finds the difference between two rectangle collections.

```
DIFFERENCE($A, B$)
1  $U \leftarrow$ COPY($A$)
2  for each $r$ in $B$
3  do DELETE($U, r$)
4  return $U$
```

Algorithm 4.5 FORALL applies a function f to every rectangle in collection C.

```
FORALL($A, f$)
1  $U \leftarrow$ new RECTANGLECOLLECTION
2  for each $r$ in $A$
3  do for each $r'$ in $f(r)$
4         do INSERT($U, r'$)
5  return $U$
```

4.4 Data Structures

We have reduced the problem of implementing the region algebra to finding an efficient data structure for a rectangle collection that supports querying, insertion, and deletion. Research in computational geometry and multidimensional databases has resulted in a variety of suitable data structures. Samet [Sam90] gives a good survey.

This section describes three classes of spatial data structures that are useful for rectangle collections:

- *R-trees* divide the rectangle collection recursively. Each leaf of an R-tree is one rectangle in the collection, and each internal node stores the bounding box of the rectangles in its subtree. Variants of the R-tree variants use different insertion heuristics to minimize overlap between nodes, which speeds up query operations.

- *Quadtrees* divide region space recursively. Each quadtree node covers a fixed area of region space, and each node has four children, one for each equal quadrant of its area. Variants of the quadtree differ in how deeply the tree is subdivided and whether rectangles are stored only in the leaves or in all nodes.

- *Point-based methods* represent each rectangle as a four-dimensional point. The points are stored in a point data structure, such as a k-d tree, quadtree, or range tree.

Each class of data structure is described below. Discussion will focus on how to implement the three key operations for rectangle collections: querying for intersections with a rectangle, inserting a rectangle, and deleting a rectangle. The running time of these operations and the storage cost of the data structure will also be discussed.

Only one of these data structures is implemented in LAPIS: a variant of the R-tree that I call an *RB-tree*.

4.4.1 R-Trees

An *R-tree* [Gut84] is a balanced tree that stores an arbitrary collection of rectangles. The R-tree is based on the B-tree [CLR92], in that every internal node (other than the root) has between m and M children for some constants m and M, and the tree is kept in balance by splitting overflowing nodes and merging underflowing nodes. Each leaf node represents one rectangle in the collection, and each internal node stores the bounding box of all the rectangles in its subtree. An example R-tree is shown in Figure 4.7.

Querying for a rectangle in an R-tree uses Algorithm 4.6. The algorithm compares the query rectangle recursively against the bounding box of each node (*T.bbox)*. If the query rectangle does not intersect a node's bounding box, then the node's subtree is pruned from the search. When the search reaches a leaf of the tree, it returns the intersection of the query rectangle with the rectangle stored in the leaf (*T.rectangle*). For convenience, the pseudocode in Algorithm 4.6 uses the `yield` keyword from CLU [Lis81] to return each rectangle. Unlike `return`, `yield` implicitly saves a continuation so that the traversal can be resumed at the same point to generate the next rectangle.

Inserting a rectangle into an R-tree is similar to insertion into a B-tree. The new rectangle is inserted as a leaf. If the leaf's parent overflows (i.e., has $M + 1$ children), the parent is split into

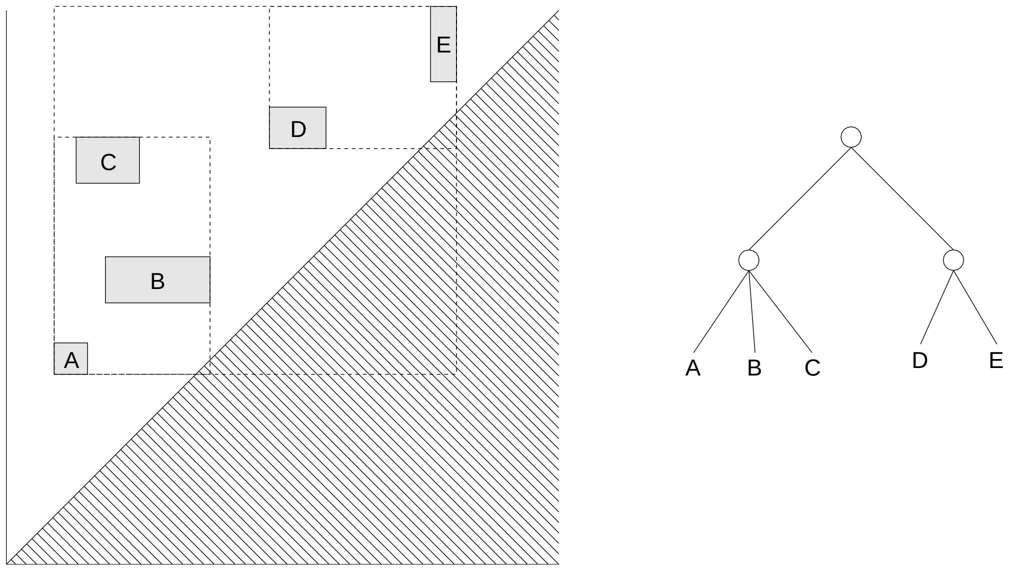


Figure 4.7: An R-tree containing 5 rectangles, $A - E$.

Algorithm 4.6 QUERY (T, r) traverses an R-tree T to find all rectangles that intersect the rectangle r.

```
QUERY($(T, r)$)
1  if $r$ doesn't intersect $T.bbox$
2     then return
3  if $T$ is leaf
4     then yield $r \cap T.rectangle$
5     else for each $C$ in $T.children$
6             do QUERY($(C, r)$)
```

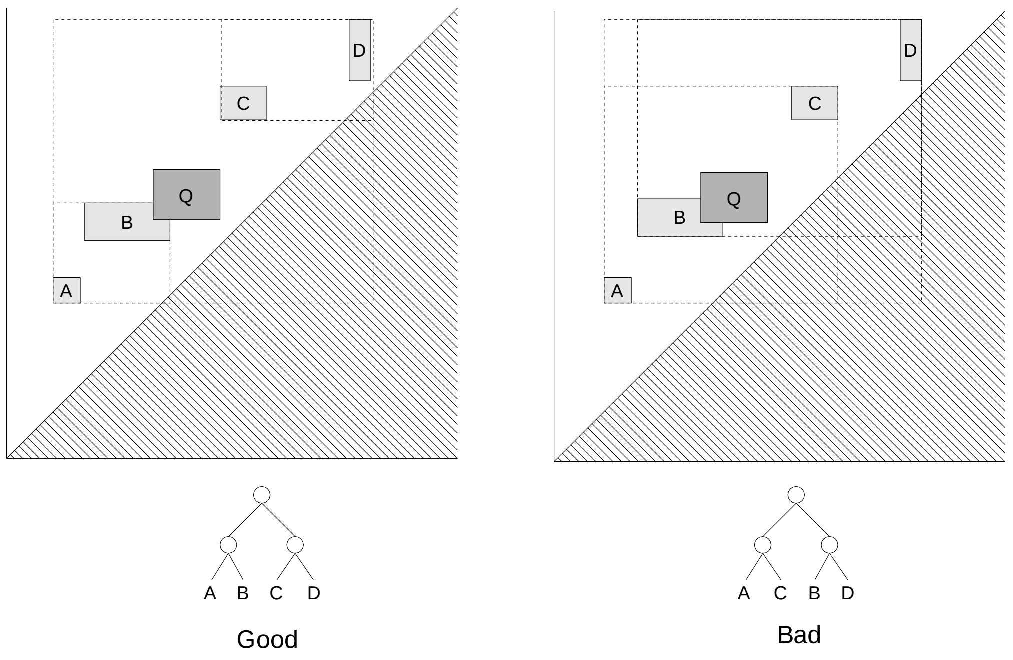


Figure 4.8: Different R-trees for the same set of rectangles can have different querying performance. A query for rectangle Q can avoid visiting the C-D subtree in the R-tree on the left, but it must visit all nodes in the R-tree on the right.

two nodes, which are then inserted into the grandparent. Overflows propagate up the tree as far as necessary. When the root overflows, it is split and a new root node is created, increasing the height of the tree.

Unlike a B-tree, however, an R-tree has no fixed rule for placing rectangles in its leaves. A rectangle can be inserted as *any* leaf without violating the R-tree's invariant properties. Bad placement leads to slower querying, however, because the internal nodes' bounding boxes become larger, more likely to overlap, and less likely to be pruned from the search. Figure 4.8 shows an example. Ideally, good placement should minimize both the area of the nodes and the overlap between sibling nodes.

Variants of the R-tree differ in the heuristics they use to achieve good placement. Two decisions are made heuristically. First is the *insertion heuristic*, which determines where to insert a new rectangle. Second is the *node-splitting heuristic*, which partitions the children of an overflowing node into two new nodes.

The original R-tree proposed by Guttman [Gut84] uses heuristics that minimize the area of bounding boxes. The insertion heuristic traverses the tree recursively, choosing the node whose bounding box would be expanded the least (in area) by the new rectangle. Ties are broken by choosing the node with the smallest area. For the node-splitting heuristic, Guttman offered three possibilities:

- an exponential heuristic that does an exhaustive search of the 2^M possible partitions, searching for the partition that produces the smallest nodes. This heuristic is generally impractical, but serves as a good baseline for measuring the performance of other heuristics.

- a quadratic heuristic that chooses two rectangles as seeds and greedily adds the remaining rectangles to the node whose bounding box needs the least enlargement. The seeds are chosen so that their bounding rectangle R maximizes $\mathit{area}(R) - \mathit{area}(\mathit{seed}_1) - \mathit{area}(\mathit{seed}_2)$. Implementing the second heuristic requires testing all M^2 possible pairs of seeds.

- a linear heuristic identical to the quadratic heuristic except that the chosen seeds are the rectangles with maximum separation (in some dimension), which can be computed in $O(M)$ time.

Guttman's experiments suggested that the linear heuristic was as good as the other two, but later experimenters [BKSS90] argue that the quadratic heuristic is superior to the linear heuristic in many cases.

The R*-tree [BKSS90] uses another set of heuristics. For non-leaf nodes, the R*-tree uses the same insertion heuristic as the R-tree, minimum area enlargement. For leaf nodes, however, the R*-tree insertion heuristic chooses the leaf node whose overlap with its siblings would be enlarged the least. The cost of computing each node's overlap with its siblings is $O(M)$, so this heuristic takes $O(M^2)$ time. For the node-splitting heuristic, the R*-tree sorts the rectangles separately along each axis, chooses one axis for splitting, and then splits the sorted list into two groups with minimum overlap. The sorting axis is chosen to minimize the perimeters of the two groups. This heuristic is $O(MlogM)$. The R*-tree heuristics were empirically shown to be efficient for random collections of rectangles [BKSS90].

LAPIS originally used the R*-tree heuristics. Rectangle collections used by the region algebra are *not* particularly random, however. They tend to be nonoverlapping and distributed linearly along some dimension of region space, such as the x-axis, y-axis, or $45°$ line. For such sets, a fixed lexicographic ordering of rectangles works just as well and avoids expensive placement calculations entirely. The revised R-tree data structure in LAPIS, which I call an *RB-tree*, orders its leaves lexicographically by (s_1, e_1, s_2, e_2). Each internal node in an RB-tree is augmented with a pointer to the lexicographically smallest leaf in its subtree, which allows the insertion heuristic to find the correct place for a new rectangle among the leaves. (This is equivalent to the way a conventional B-tree intersperses keys with child pointers in internal nodes.) The node-splitting heuristic simply divides the children in half, preserving their order.

A rectangle r is deleted from an RB-tree by querying the tree for r. Every rectangle that intersects r is either deleted from the tree outright using the B-tree deletion algorithm, or else split into one or more new rectangles which are reinserted into the tree. The cases for each kind of overlap are shown in Figure 4.9.

The time to query an RB-tree in which N rectangles have been inserted is $O(N)$ in the worst case, because the query rectangle may intersect the bounding box of every internal node, even if it does not intersect any of the leaves. Inserting a rectangle in the tree takes $O(\log N)$ time. Deleting a rectangle must query the tree, so it takes $O(N)$ time in the worst case. The average case is better, as the performance measurements in Section 4.7 show; querying, insertion, and deletion are typically $O(\log N)$.

The storage required by an RB-tree containing N rectangles is $O(N)$.

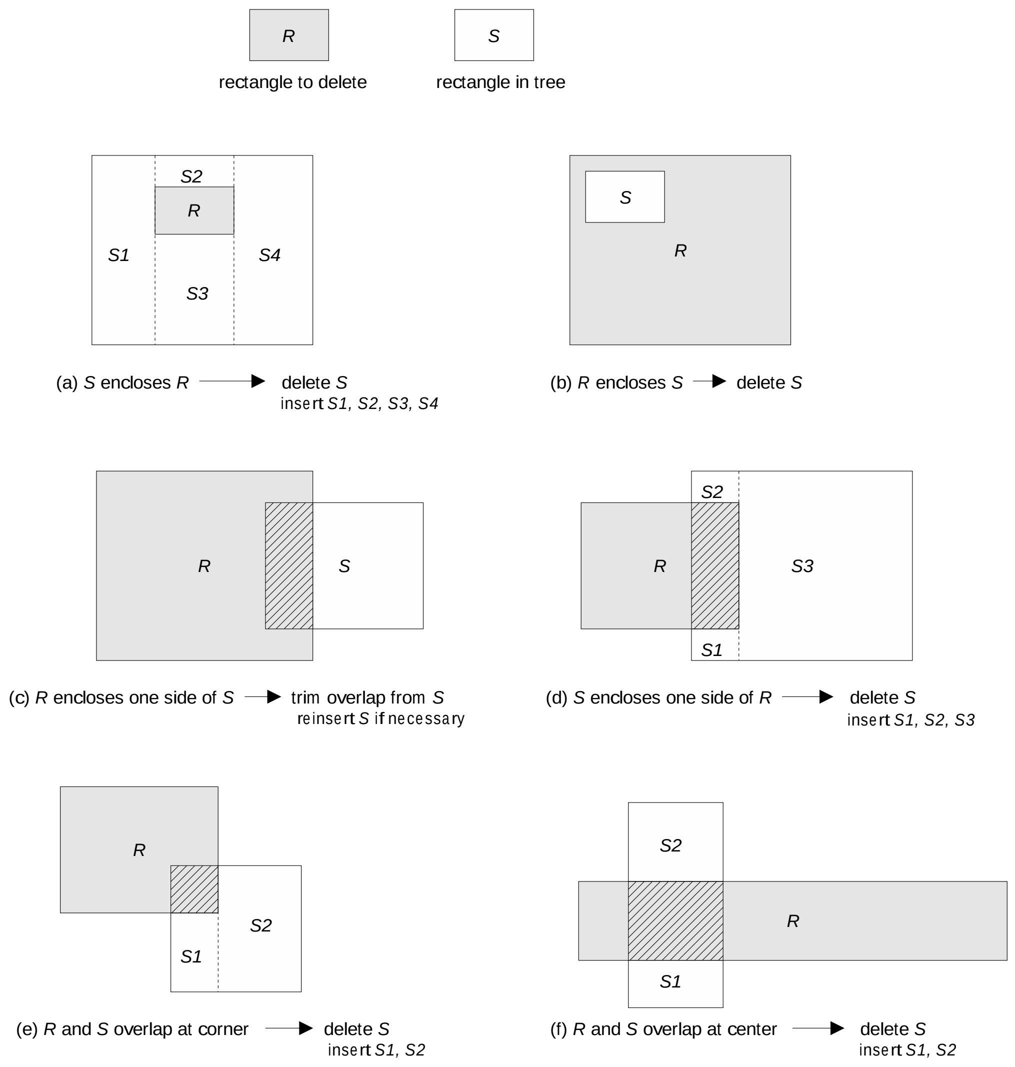

R
rectangle to delete
S
rectangle in tree
S2
R
S1
S4
S3
(a) S encloses R → delete S
insert S1, S2, S3, S4
S
R
(b) R encloses S → delete S
R
S
(c) R encloses one side of S → trim overlap from S
reinsert S if necessary
S2
R
S3
S1
(d) S encloses one side of R → delete S
insert S1, S2, S3
R
S2
S1
(e) R and S overlap at corner → delete S
insert S1, S2
S2
R
S1
(f) R and S overlap at center → delete S
insert S1, S2

Figure 4.9: Deleting a rectangle from an RB-tree.

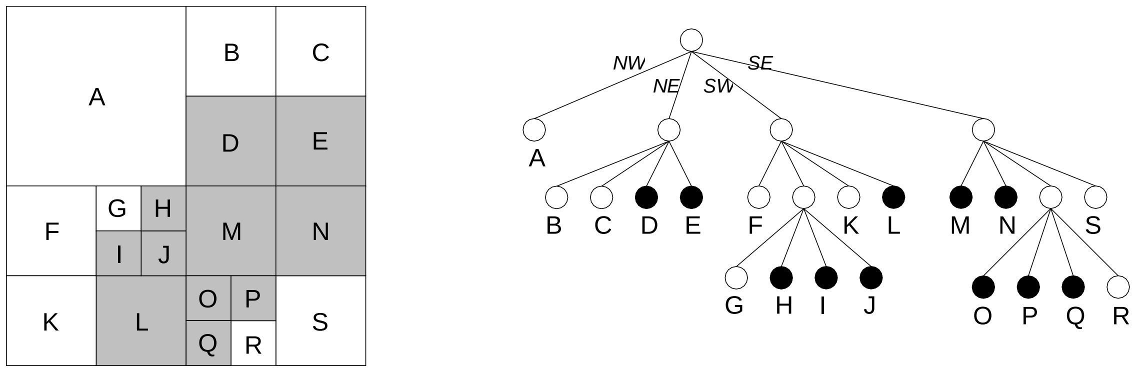


Figure 4.10: Example of a region quadtree (after Figure 1.1 from Samet [Sam90]).

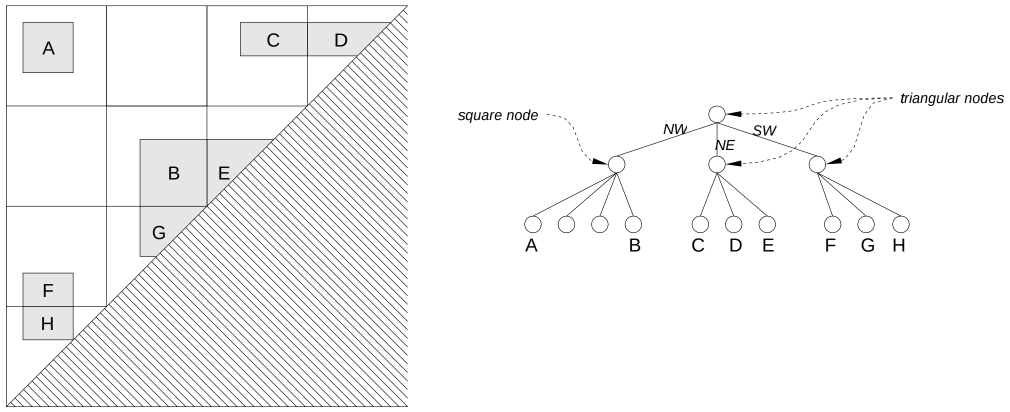


Figure 4.11: A quadtree used as a rectangle collection.

4.4.2 Quadtrees

A *quadtree* is a recursive data structure over 2D space. Each quadtree node represents a fixed area of space. The root node's rectangle is the entire space. Every node has four children, which divide the node's rectangle into four equal quadrants.

The most familiar kind of quadtree is the *region quadtree*, which is used to represent a set of points in the plane, such as a shape or a set of pixels in a raster image. A region quadtree is subdivided until its leaves are all homogeneous — either all points covered by the leaf are included in the set, or all points are excluded. A single bit in each leaf indicates which is the case. An example of a region quadtree is shown in Figure 4.10.

A region quadtree can store a rectangle collection by storing the set of points in the union of the inserted rectangles. The leaves need not be completely homogeneous, however. It is enough to subdivide until every leaf contains a rectangular set of points (or no points at all). Furthermore, region space is actually *triangular*, not square. As a result, quadtree nodes that intersect the $45°$ line only need three children, not four. These optimizations produce quadtrees like the one shown in Figure 4.11.

Querying a quadtree for a rectangle uses the same algorithm as the R-tree (Algorithm 4.6). One important difference is that a quadtree does not need to store node bounding boxes explicitly, because every node's bounding box is uniquely determined by its path from the root. Recursive algorithms like QUERY calculate bounding boxes on the fly as the quadtree is traversed.

Inserting a rectangle into a quadtree uses a similar recursive search for all nodes that intersect the new rectangle (Algorithm 4.7). If the new rectangle completely covers a quadtree node, then the node is simply converted to a leaf. Otherwise, if the node is a leaf, INSERT first tries to extend the leaf's rectangle to include the new rectangle. If the resulting shape would be nonrectangular, the leaf is split using SPLIT (Algorithm 4.8), and the new rectangle is inserted into its children. Node splitting is guaranteed to terminate, because eventually it will reach a 1×1 node which is either inside or outside the new rectangle. After inserting the new rectangle in the children of a node, the algorithm calls MERGE (Algorithm 4.9) to test whether the node now represents a simple rectangular area, in which case its children can be discarded and the node converted into a leaf. SPLIT and MERGE grow and shrink the quadtree to preserve the invariant that nodes are subdivided only as far as necessary to make every leaf contain a simple rectangular region.

Algorithm 4.7 INSERT (T, r) inserts a rectangle r into a quadtree T.

```
INSERT($T, r$)
 1  if $r$ doesn't intersect $T.bbox$
 2     then return
 3  $r' \leftarrow r \cap T.bbox$
 4  if $r' = T.bbox$
 5     then delete children of $T$
 6          $T.rectangle \leftarrow r'$
 7          return
 8  if $T$ is leaf
 9     then if $T.rectangle \cup r'$ is rectangular
10             then $T.rectangle \leftarrow T.rectangle \cup r'$
11                  return
12             else SPLIT($T$)
13  for each $C$ in $T.children$
14  do INSERT($C, r$)
15  MERGE($T$)
```

Algorithm 4.8 SPLIT (T) converts a quadtree leaf into a node with children.

```
SPLIT($T$)
1  create children of $T$
2  INSERT($T, T.rectangle$)
3  $T.rectangle \leftarrow \emptyset$
```

Deleting a rectangle from a quadtree follows the same pattern as insertion. The only difference is that the query rectangle is subtracted from the tree's leaf rectangles, instead of added.

The running times of these algorithms depend on the height of the tree, which in turn depends on the size of region space. In a string of length n, region space has dimension $n \times n$, and any

Algorithm 4.9 MERGE (T) converts a quadtree node into a leaf if its children are all leaves and the union of their rectangles is rectangular.

```
MERGE($T$)
1  $r \leftarrow \emptyset$
2  for each $C$ in $T.children$
3  do if $C$ is not a leaf or $r \cup C.rectangle$ is not rectangular
4        then return
5     $r \leftarrow r \cup C.rectangle$
6  delete all children of $T$
7  $T.rectangle \leftarrow r$
```

quadtree over region space has $O(\log n)$ height. Querying or deleting a rectangle from the quadtree takes $O(F \log n)$ time, where F is the number of leaves in the tree that intersect the query. Inserting a rectangle takes $O(n)$ time in the worst case, because the quadtree may need to drill down to its maximum depth ($\log n$) in order to separate two rectangles into different nodes, and drilling down to maximum depth in one dimension can create up to n nodes in the other dimension (Figure 4.12). The storage required by a quadtree is $O(\min(nN, n^2))$ in the worst case.

This dependency on the size of region space n, rather than just the number of inserted rectangles N, is an important difference between quadtrees and RB-trees. The bounding box of each quadtree node is fixed, so a quadtree may be less efficient than an RB-tree at storing a collection of regions localized to a small part of a long string. On the other hand, the quadtree can completely eliminate overlaps between rectangles. Quadtrees are not implemented in LAPIS, so comparing the practical performance of RB-trees and quadtrees is left for future work.

The region quadtree is not the only kind of quadtree that could be used to implement a rectangle collection. For example, the MX-CIF quadtree [Sam90] associates each rectangle with the smallest quadtree node that completely contains it. This raises the question of how to organize the possibly-large list of rectangles stored on each node. One approach is an unordered list, but a more interesting approach reduces the dimension of the problem by 1: each rectangle is intersected with the node's horizontal and vertical axis, producing two sets of intervals that are stored in a pair of 1-dimensional MX-CIF quadtrees. Another quadtree is the RR quadtree [Sam90], which comes in two variants. The RR_1 quadtree splits nodes until each leaf intersects just one rectangle or a *clique*, a set of rectangles that are all mutually intersecting. The RR_2 quadtree splits until each leaf intersects a single rectangle or a chain of intersecting rectangles. Unlike the region quadtree, both MX-CIF and RR quadtrees guarantee to preserve the identities of the inserted rectangles — a guarantee which is not important for our rectangle collections.

4.4.3 Rectangles as Points

Another way to look at a rectangle (s_1, e_1, s_2, e_2) is as a point in four-dimensional space. This is the same change in perspective that led from regions as intervals in a one-dimensional string to points in two-dimensional region space (Section 3.3).

Representing rectangles as points makes it possible to store a rectangle collection in a data structure designed for points. In order to implement the QUERY operation, the point data structure

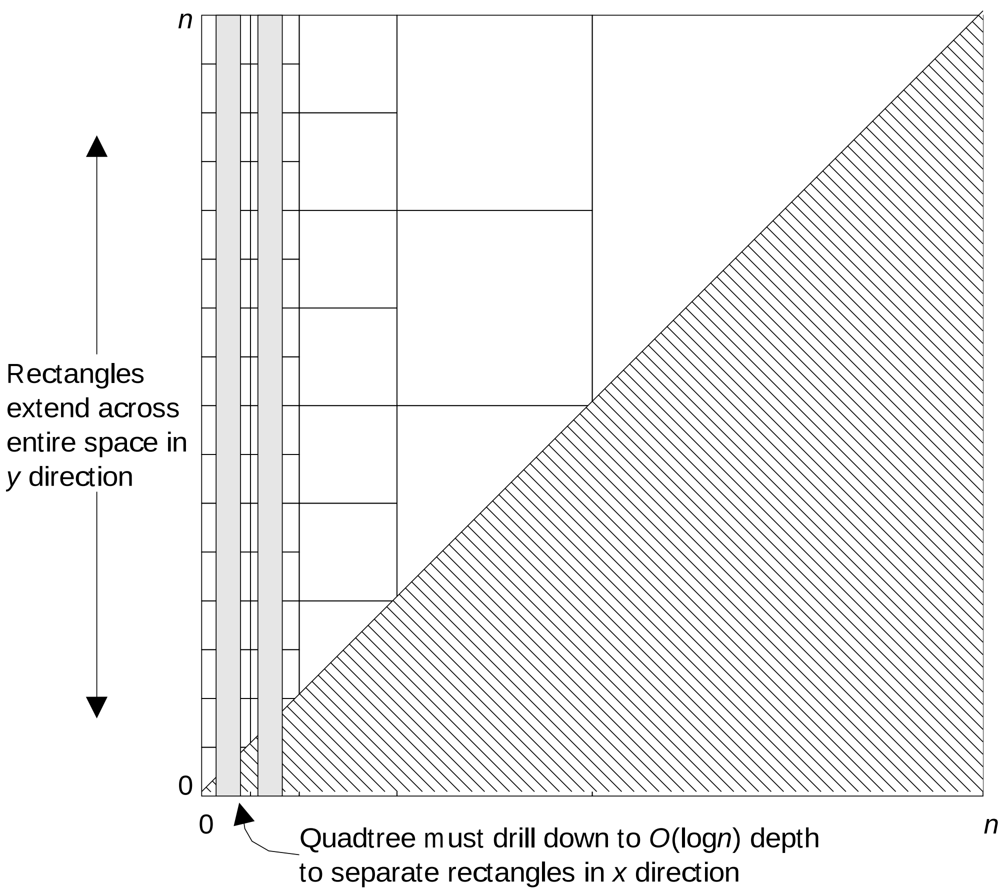

n
0
Rectangles extend across entire space in y direction
0
n
Quadtree must drill down to O(logn) depth to separate rectangles in x direction

Figure 4.12: Bad case for a quadtree: two long but closely-spaced rectangles force the quadtree to drill down to depth $O(\log n)$, requiring $O(n)$ quadtree nodes.

must support *range queries*. A range query returns all points that lie in a four-dimensional hyper-rectangle — the Cartesian product of four intervals, one for each dimension. For example, to find all the 2D rectangles (4D points) that are enclosed by the 2D query rectangle (s_1, e_1, s_2, e_2), one would use the range query $[s_1, s_2] \times [e_1, e_2] \times [s_1 : s_2] \times [e_1 : e_2]$. To implement QUERY, which searches for all the rectangles that intersect the query rectangle, one would use the range query $[-\infty, s_2] \times [-\infty, e_2] \times [s_1, \infty] \times [e_1, \infty]$.

A menagerie of data structures have been developed for k-dimensional points with range queries. Here are a few:

- The *k-d tree* [Ben75] is a binary tree in which each node stores a point, and each level of the tree compares a different dimension of a query point with its stored point. For example, in a 2-d tree, nodes at even-numbered depths might compare the x coordinates of the query point and the stored point, while nodes at odd-numbered depths compare the y coordinate. A new point is inserted by traversing the tree for the new point's correct position and adding it as a leaf. Like binary search trees, the performance of a k-d tree is very sensitive to the order in which the points are presented. An optimal k-d tree can be built in $O(kN \log N)$ time if all N points are known in advance. Range queries in an optimal k-d tree take $O(kN^{1-1/k})$ time in the worst case [LW77]. Since $k = 4$ for the purpose of this section, this means that querying takes $O(N^{3/4})$ worst-case time.

- A quadtree can also be used to store points. The 2D quadtree described in the previous section is generalized to k dimensions by splitting each node into 2^k children, one for each hyperquadrant. Many nodes will not need all 16 children, however, since some parts of 4-dimensional space can never contain a 4D point representing a 2D rectangle.

- The *point quadtree[FB74]* is a kind of quadtree that provides an adaptive decomposition of space. Each node stores one point from the set, and the node is split into 2^k children by hyperplanes passing through the stored point (instead of the node's centroid, as in the standard quadtree). A point quadtree is like a k-d tree in which each level discriminates on all k dimensions at once, instead of just one dimension at a time. A variant of the point quadtree chooses an arbitrary point to partition each node, not necessarily a point in the collection. Points in the collection are stored only in the leaves, which makes them easier to delete. Range queries in point quadtrees take $O(kN^{1-1/k})$ worst-case time, just like k-d trees [LW77].

- The *range tree [BM80]* is a data structure designed for good worst-case query performance at the cost of more storage. A one-dimensional range tree is a balanced binary search tree with the points stored in leaf nodes. The leaves are linked in sorted order by a doubly-linked list. A range query $[l, h]$ is done by searching the range tree for a leaf $\geq l$, then following the linked list until reaching a leaf $\geq h$. A k-d range tree uses a 1D range tree to index the x coordinate, and every node of the tree points to a $(k-1)$-d range tree indexing the remaining coordinates of every point in the node's subtree. The resulting data structure uses $O(N \log^{k-1} N)$ storage, and range queries can be satisfied in $O(\log^k N)$ time.

Although the point data structures have provably better asymptotic behavior than R-trees and quadtrees, using 4D makes the data structures more complicated and significantly increases the constant factors. Furthermore, points representing region rectangles are not uniformly distributed in 4D space. In many rectangle collections generated by the region algebra, the rectangles all share one or more coordinates, making those coordinates useless for discrimination — but point data structures give all coordinates equal weight. For these reasons, one would expect 4D point data structures to perform worse than R-trees or quadtrees on the average. No 4D data structures are implemented in LAPIS, so this comparison is left for future work.

As a special case, the 2D versions of these point data structures could be used to store rectangle collections where every rectangle is just a point. Flat, overlapping, and nested region sets all satisfy this requirement.

4.5 Specialized Data Structures

The rectangle-collection data structures that have been presented to this point — RB-trees, quadtrees, and 4D point collections — are suitable for storing arbitrary rectangle collections. For some important kinds of rectangle collections, however, there are simpler data structures.

- A *region array* is an array of rectangles in lexicographic order. Region arrays can store a *monotonic* rectangle collection, in which all rectangle coordinates increase monotonically. Flat and overlapping region sets are monotonic. Region arrays have guaranteed $O(\log N+F)$ query time.

- A *syntax tree* extends the familiar abstract syntax tree by storing a rectangular bounding box on each node. Syntax trees can be used to store nested region sets.

These data structures are described in more detail in the next two sections.

4.5.1 Region Arrays

For some rectangle collections, an RB-tree has too much overhead. In many cases, we can throw away the internal nodes of the RB-tree, leaving only the leaves, a list of rectangles sorted in lexicographic order. I call the resulting data structure a *region array*. Eliminating the internal nodes saves space, but only a constant factor since leaves always outnumber internal nodes. More importantly, however, it can be shown that a query on a region array always takes $O(\log N + F)$ time, where F is the number of intersecting rectangles found, which is an improvement over the $O(N)$ worst-case bound for RB-trees.

A region array can be used whenever the rectangle collection satisfies the following property. A rectangle collection is *monotonic* if, when the collection is sorted in increasing lexicographic order, the coordinates of the rectangles are also sorted in increasing order. In other words, if r and r' are a pair of rectangles in the collection such that $r \leq r'$ in lexicographic order, then $r.s_1 \leq r'.s_1$, $r.s_2 \leq r'.s_2$, $r.e_1 \leq r'.e_1$, and $r.e_2 \leq r'.e_2$. Many rectangle collections generated by the region algebra are monotonic. In particular, all flat and overlapping region sets are monotonic. The rectangle collection produced by applying a unary relational operator to a flat or overlapping region set is also monotonic. See Figures 4.4–4.6 for examples of these kinds of rectangle collections.

A nested region set is *not* monotonic in general, however. Figure 4.4 includes a nested set which is nonmonotonic. Region arrays cannot be used to store general nested region sets.

Monotonicity makes it easy to find the bounding box for any contiguous range of a region array. Suppose a region array A contains N monotonic rectangles in lexicographic order, so $A[1] \leq A[2] \leq \cdots \leq A[N]$. Because the collection is monotonic, we know that $A[1]$ has the minimum s and e values of all the rectangles, and $A[N]$ has the maximum values, so the bounding box of the entire collection is $(A[1].s_1, A[1].e_1, A[N].s_2, A[N].e_2)$. In general, for any $i \leq j$, the bounding box of $A[i \ldots j]$ is $(A[i].s_1, A[i].e_1, A[j].s_2, A[j].e_2)$.

We can use this fact to query a region array as if it were a binary RB-tree whose internal nodes are constructed on the fly. Algorithm 4.10 demonstrates this approach. The algorithm is very similar to the query algorithm for RB-trees (Algorithm 4.6), but bounding boxes and children are calculated on the fly. This algorithm uses $O(\log N)$ space to traverse the region array recursively, however, and it isn't clear how many bounding boxes may unnecessarily intersect the query rectangle.

Algorithm 4.10 QUERY (A, i, j, r) searches a segment of a region array, $A[i...j]$, for all rectangles that intersect the rectangle r.

```
QUERY($A, i, j, r$)
1  $bbox \leftarrow (A[i].s_1, A[i].e_1, A[j].s_2, A[j].e_2)$
2  **if** $r$ doesn't intersect $bbox$
3     **then return**
4  **if** $i = j$
5     **then yield** $r \cap bbox$
6     **else** $m \leftarrow \lfloor (i + j)/2 \rfloor$
7           QUERY($A, i, m, r$)
8           QUERY($A, m + 1, j, r$)
```

A better algorithm uses a separate binary search on each coordinate. Recall from Section 4.4.3 that a rectangle intersection query is equivalent to the range query $[-\infty, r.s_2] \times [-\infty, r.e_2] \times [r.s_1, \infty] \times [r.e_1, \infty]$. In a region array, each of these intervals corresponds to a bound on the array indexes of intersecting rectangles. For example, consider the first interval, which stipulates that the s_1 coordinate of an intersecting rectangle must be less than or equal to $r.s_2$. Using a binary search on the s_1 coordinates of the region array (which are guaranteed to be in order because the rectangles are monotonic), find the largest index k_1 such that $A[k_1].s_1 \leq r.s_2$. Then $A[1 \ldots k_1]$ is the set of rectangles that satisfy the first interval. Similar binary searches for the other dimensions find $A[1 \ldots k_2]$ satisfying the second interval, $A[k_3 \ldots N]$ satisfying the second interval, and $A[k_4 \ldots N]$ satisfying the fourth interval. Putting all four constraints together, we find that the set of intersecting rectangles is precisely the rectangles in $A[\max(k_3, k_4) \ldots \min(k_1, k_2)]$. Algorithm 4.11 restates this procedure in pseudocode. This algorithm takes $O(\log N + F)$ time and $O(1)$ working space in all cases.

Region arrays are not designed for dynamic construction, so they do not support the INSERT operation. To construct a region array, LAPIS accumulates the rectangles resulting from an algebra operation. After the entire collection has been generated, the rectangles are sorted, and then a single pass over the collection tests whether the collection is monotonic. If so, the sorted list can be used directly as a region array. If not, the region array can be converted to an RB-tree by using the sorted list as a foundation and erecting the tree above it in $O(N)$ time.

Algorithm 4.11 QUERY (A, r) searches a region array for all rectangles that intersect the rectangle r, using a binary search on each coordinate.

```
QUERY($A, r$)
1  $k_1 \leftarrow$ BINARYSEARCH($A.s_1, r.s_2 + 1$) $- 1$
2  $k_2 \leftarrow$ BINARYSEARCH($A.e_1, r.e_2 + 1$) $- 1$
3  $k_3 \leftarrow$ BINARYSEARCH($A.s_2, r.s_1$)
4  $k_4 \leftarrow$ BINARYSEARCH($A.e_2, r.e_1$)
5  for $i \leftarrow max(k_3, k_4)$ to $min(k_1, k_2)$
6  do yield $A[i]$

BINARYSEARCH($L, x$)
1  do a binary search on array $L$ for value $x$
2  return the index $i$ such that $L[1 \ldots i-1] < x \leq L[i \ldots N]$
```

Deleting rectangles from a region array is possible only in certain cases. Deleting entire rectangles is easy, since a rectangle can be removed without affecting the monotonicity of the collection. A rectangle can be removed from the array in $O(1)$ amortized time by just marking it deleted. The array is reallocated and compacted only when at least half of its entries have been deleted. Harder cases arise when deletion would split or shrink a rectangle. Although it might be possible in some cases to split or shrink the rectangle in such a way that the collection is still monotonic (and doesn't require another $O(N \log N)$ sort), in general it is necessary to convert the region array to an RB-tree and then perform the deletion.

A region array of N rectangles takes $O(N)$ space. A region array saves up to a factor of 2 relative to an RB-tree by eliminating the internal nodes. LAPIS saves more space by representing the rectangles as an array of integers, eliminating the overhead of representing each rectangle as a Java object (12 bytes per object in Java 1.3). In LAPIS, a region array consumes about 16 bytes per rectangle, while an RB-tree averages 33 bytes per rectangle (counting the cost of internal nodes, with the tree's minimum and maximum branching factors set to 4 and 12, respectively). More space could be saved by delta-encoding or otherwise compressing the coordinates of the rectangles, at the cost of increased access time.

4.5.2 Syntax Trees

Context-free parsing often produces an abstract syntax tree representing the parse. If this syntax tree is augmented with the region from which each node was parsed, then the tree can be queried as if it were a rectangle collection.

In general, a *syntax tree* can represent any nested region set, not just the output of a context-free parser. In a syntax tree, any node may store a region, not just the leaves. When a region r is associated with a node, then the node's subtree must contain all the regions in the set that are *in* r. Nodes may have an arbitrary number of children, sorted in lexicographic order. Some nodes may have no associated region; such nodes are *empty*. Empty nodes arise when regions are deleted from the set. Empty nodes may also be used to ensure that the tree has a constant branching factor. Every node, empty or not, stores the bounding box of all the regions in its subtree.

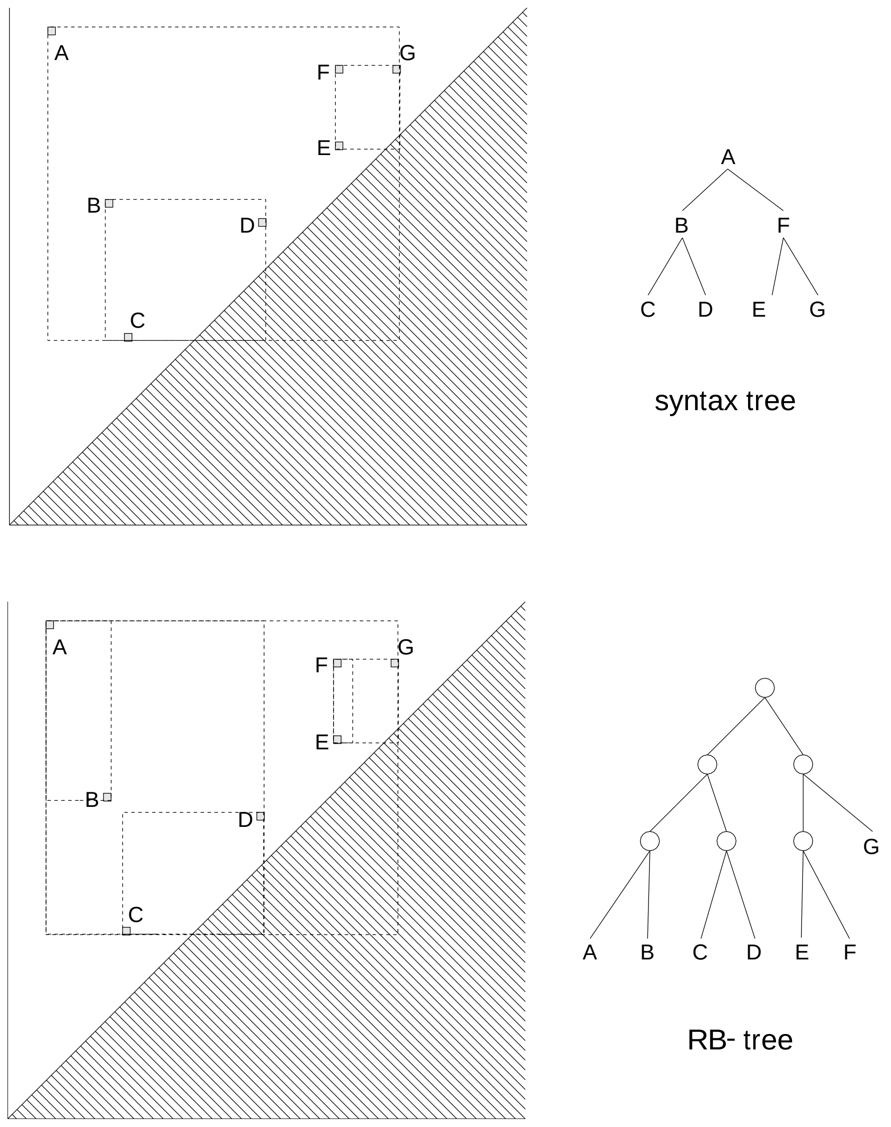


Figure 4.13: Syntax tree and RB-tree representations of a nested region set.

Figure 4.13 shows a syntax tree and an RB-tree for the same nested set. One key difference is that the syntax tree can store a region in any node, while the RB-tree stores regions only in leaves. Another difference is the shape of the node bounding boxes. RB-trees pack regions into the leaves in lexicographic order, without much concern for whether the regions are near each other in region space. Syntax trees can sometimes achieve more locality by exploiting the nested nature of the region set.

A syntax tree is queried by Algorithm 4.12. Every node in the syntax tree has two attributes: $T.bbox$ is the bounding box of the regions in T's subtree, and $T.region$ is the region stored in T itself. Since the bounding boxes of a node's children form a monotonic rectangle collection, the exhaustive search in line 5 can be replaced by a binary search as in region arrays, but this would help only when nodes have many children.

Algorithm 4.12 QUERY (T, r) searches a syntax tree for all regions that intersect the rectangle r.

```
QUERY($T, r$)
1  **if** $r$ doesn't intersect $T.bbox$
2     **then return**
3  **if** $r$ intersects $T.r$
4     **then yield** $r \cap T.r$
5  **for each** $C$ **in** $T.children$
6  **do** QUERY($C, r$)
```

A new region r can be inserted in a syntax tree by drilling down through nodes that *contain* r until finding a node whose children's bounding boxes are all *before* or *after* r, then inserting r in sorted order among the children. If some child's bounding box *overlaps-start* or *overlaps-end* r, then the resulting region set will no longer be nested, so the syntax tree must be converted to an RB-tree. The conversion is done by traversing the syntax tree in lexicographic order (Algorithm 4.13), from which an RB-tree can be built in $O(N)$ time.

Algorithm 4.13 LEXORDER (T) generates the regions from a syntax tree in lexicographic order. The algorithm is basically a preorder traversal, except that all descendants of T that start at the same point as T must be returned before T itself.

```
LEXORDER($T$)
1  PRE($T$)
2  POST($T$)

PRE($T$)
1  **if** $T$ is not a leaf and $T.children[1].bbox.s_1 = T.bbox.s_1$
2     **then** PRE($T.children[1]$)
3  **if** $T.region \neq \emptyset$
4     **then yield** $T.region$

POST($T$)
1  **if** $T$ is not a leaf and $T.children[1].bbox.s_1 = T.bbox.s_1$
2     **then** POST($T.children[1]$)
3  **for each** $C$ **in** $T.children$ such that $C.bbox.s_1 > T.bbox.s_1$
4  **do** LEXORDER($C$)
5
```

A region can be deleted from a syntax tree by removing it from its node (i.e., setting $T.region$ to $\emptyset$) and updating its ancestors' bounding boxes. If a subtree becomes completely empty, with no regions associated with any of its nodes, it can be pruned from the tree.

Syntax trees are not currently implemented in LAPIS.

4.6 Optimizations

The basic region algebra implementation should now be clear. A region set is stored in a rectangle collection data structure, such as an RB-tree or a region array. Algebra operators combine the region sets by applying relational operators to rectangles, intersecting rectangle collections, merging rectangle collections, or deleting rectangles from a rectangle collection.

The basic implementation can be tweaked in many ways to improve its performance. The following optimizations are discussed in this section:

- Trimming mutually-intersecting rectangles in a rectangle collection.
- Generating rectangle collections in lexicographic order wherever possible to avoid sorting.
- Doing set operations on collections of single-point rectangles using a sorted merge in $O(N + M)$ time.
- Intersecting rectangle collections by traversing both trees in tandem.
- Doing set operations on quadtrees by traversing both trees in tandem.
- Intersecting rectangle collections using the optimal plane-sweep algorithm.

Some of these optimizations are implemented in LAPIS, but others are left for future work.

4.6.1 Trimming Overlaps

Query performance is dramatically reduced when many of the rectangles in a collection intersect each other. For example, as shown in Figure 4.5, the *before, after,* and *contains* operators produce rectangle collections in which all N rectangles are mutually intersecting. Querying one of these rectangle collections, even with a small query rectangle, may produce up to $O(N)$ rectangles as a result.

This problem can be addressed by trimming rectangles when they are inserted into the collection, in order to eliminate as much overlap as possible between the new rectangle and existing rectangles. Figure 4.14 illustrates the trimming heuristics:

1. If the new rectangle is completely enclosed by some rectangle already in the collection, then the new rectangle is not inserted (Figure 4.14(a)).
2. If some rectangle in the tree is completely enclosed by the new rectangle, then the enclosed rectangle is deleted from the tree (Figure 4.14(b)).
3. If some existing rectangle encloses one whole side of the new rectangle, then the overlapping part is subtracted from the new rectangle (Figure 4.14(c)).
4. If the new rectangle encloses a side of an existing rectangle, then the existing rectangle is trimmed and reinserted (Figure 4.14(d)).

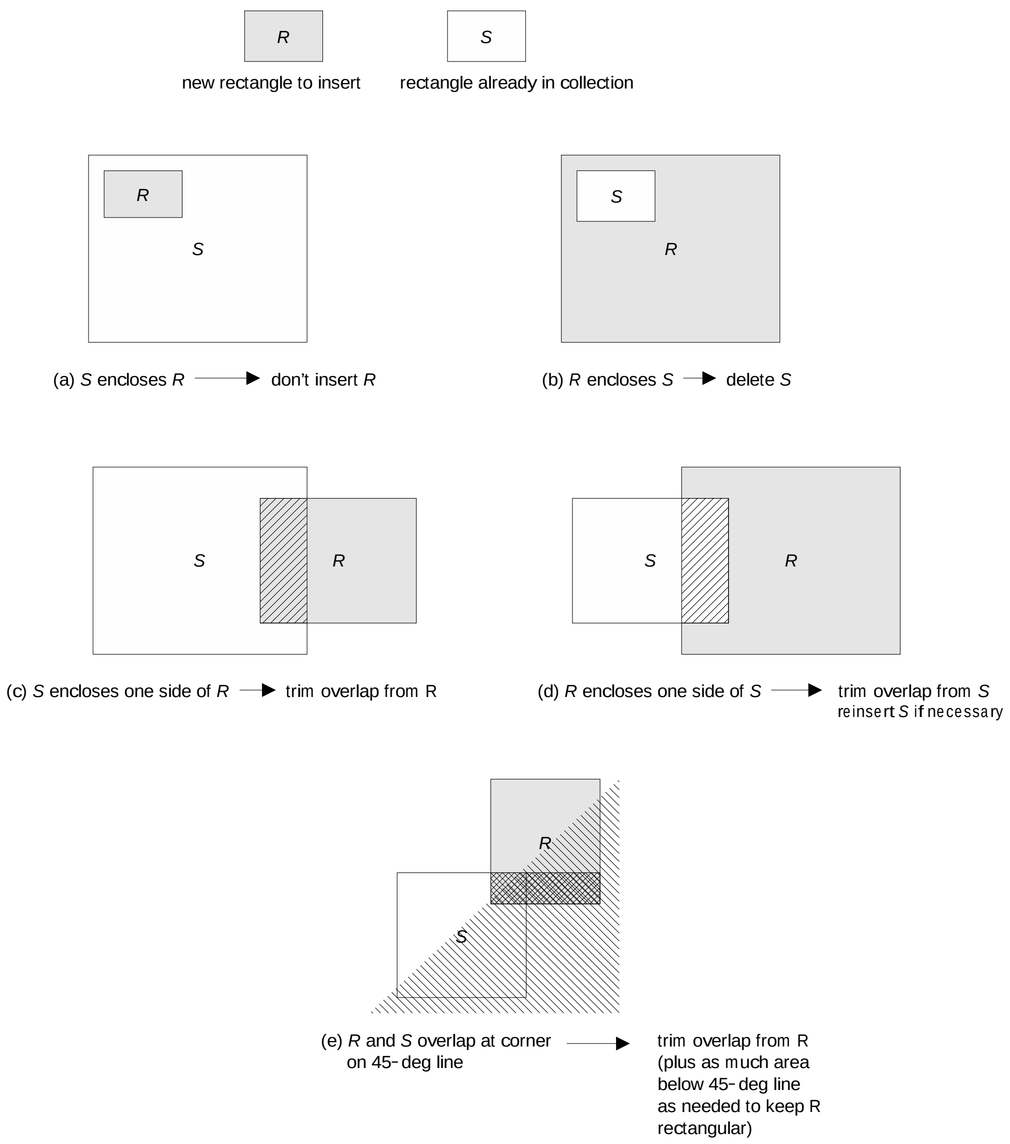

R
S
new rectangle to insert
rectangle already in collection
R
S
(a) S encloses R → don't insert R
S
R
(b) R encloses S → delete S
S
R
(c) S encloses one side of R → trim overlap from R
S
R
(d) R encloses one side of S → trim overlap from S reinsert S if necessary
R
S
(e) R and S overlap at corner on 45-deg line → trim overlap from R (plus as much area below 45-deg line as needed to keep R rectangular)

Figure 4.14: Heuristics for reducing overlap in a rectangle collection.

5. If the new rectangle and the old rectangle intersect at a corner which is cut by the $45°$ line of region space, then the overlapping part is subtracted from the new rectangle, along with enough area below the $45°$ line to keep it rectangular (Figure 4.14(e)).

Not all overlaps can be eliminated by these heuristics, since rectangles can still overlap at a corner away from the the $45°$ line, or through the center, but these heuristics help reduce overlap in the common cases produced by the region algebra.

Trimming all rectangles against each other might take $O(N^2)$ time in general, since each rectangle must be queried against the rest of the collection. LAPIS takes a simpler approach that works well for rectangle collections produced by the region algebra. After generating a list of rectangles and sorting them lexicographically, LAPIS makes a single pass through the sorted list, trimming each pair of rectangles. The effects of the heuristics on some common rectangle collections are shown in Figure 4.15.

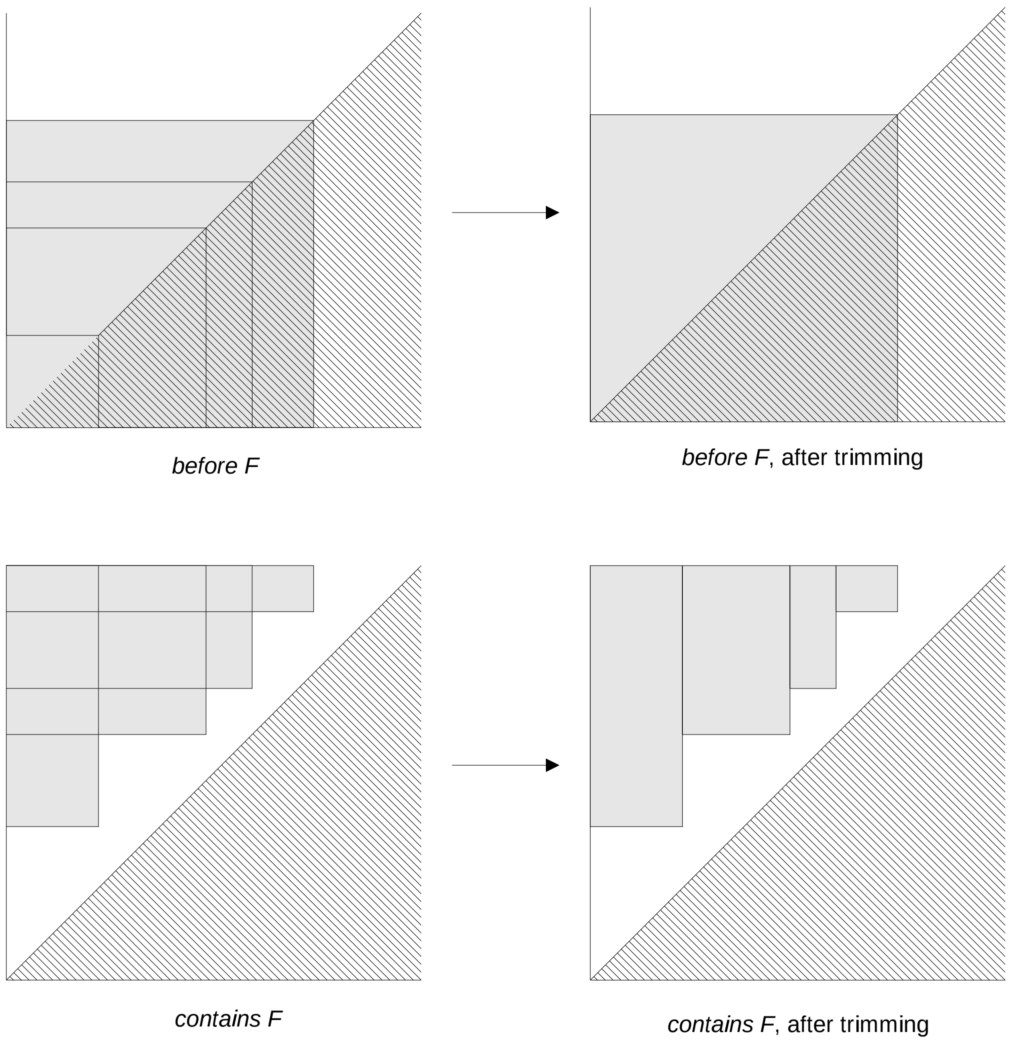


Figure 4.15: Rectangle collections after reducing overlap.

4.6.2 Preserving Lexicographic Order

Most of the cost of constructing an RB-tree or a region array is sorting the rectangle collection. If we can guarantee that the process generating the rectangles generates them in sorted order, then the sorting step can be omitted, and the RB-tree or region array can be built in $O(N)$ time.

Literal string matching and regular expression matching are two processes that naturally generate regions in lexicographic order. Each of these processes does a left-to-right scan over the string, generating all matches that start at position i before considering position $i + 1$.

Context-free parsing does *not* naturally generate regions in lexicographic order. For example, consider parsing the expression $a + b$ with conventional left-to-right, shift-reduce parsing. The parser first shifts a onto its stack. It then reduces a to an *Expression* nonterminal and emits a as a region for the *Expression* region set. Next, the parser shifts $+$ and b, reduces b to an expression and emits b as an *Expression* region, and then finally reduces *Expression*+*Expression* and emits $a + b$ as an *Expression* region. The resulting stream of regions — a, b, $a + b$ — is not in lexicographic order, because $a + b$ should precede b lexicographically.

Nevertheless, a context-free parser can produce a sorted stream of regions in $O(N)$ time if it first creates a syntax tree representing the parse. The tree is then scanned using Algorithm 4.13 to produce the region set in lexicographic order.

Applying a relational operator like *in*, *contains*, *starting*, or *ending* to a monotonic rectangle collection in lexicographic order always produces its result in lexicographic order. In fact, the result is itself a monotonic rectangle collection, as the following argument shows. Take any two rectangles $r < r'$ in the monotonic rectangle collection. Because of monotonicity, every coordinate of r is less than or equal to the corresponding coordinate of r'. Applying any relational operator *op* to r produces a new rectangle $\mathit{op}\, r$ that just rearranges the coordinates of r (possibly substituting 0 or n for some coordinates). This fact can be verified in Figure 4.3. Thus every coordinate of $\mathit{op}\, r$ is less than or equal to the corresponding coordinate of $\mathit{op}\, r'$, so $\mathit{op}\, r \leq \mathit{op}\, r'$ lexicographically and $\mathit{op}\, r$ and $\mathit{op}\, r'$ are monotonic. As a consequence, a relational operator can be applied to a region array simply by copying the region array and replacing each rectangle r with $\mathit{op}\, r$. Alternatively, the relational operator can be implemented lazily by a wrapper that applies *op* only when a rectangle is requested from the array. It takes only $O(1)$ time to apply the wrapper. LAPIS uses the lazy approach.

In general, the intersection, union, and difference operators do not necessarily generate their results in lexicographic order. One important case in which they do is when one or both operands is a collection of *one-point rectangles*. This case is described in the next section.

4.6.3 Point Collections

A *point collection* is a rectangle collection consisting entirely of one-point rectangles (s, e, s, e). Technically, any rectangle collection can be converted to a point collection by exploding its rectangles into individual points, but in general this would result in a quadratic explosion in the size of the collection. I will restrict the use of the term *point collection* to region sets that are naturally represented as single points. For example, nested, flat, and overlapping region sets are point collections. Unions of (a small number of) point collections are also naturally represented as point collections. The intersection or difference of a point collection and any other rectangle collection is always a point collection.

The intersection, union, or difference of two point collections can be found in linear time by traversing both collections simultaneously in lexicographic order. Algorithm 4.14 illustrates how intersection is done. Union and difference are similar. Not only does this algorithm take only $O(M + N)$ time, compared to $O(M \log N)$ average time for set operations on general rectangle collections, but the result is guaranteed to be in lexicographic order.

Algorithm 4.14 INTERSECTION(C_1, C_2) intersects two point collections by traversing them in lexicographic order.

```
INTERSECTION($C_1, C_2$)
 1  $U \leftarrow$ new RECTANGLECOLLECTION
 2  $p_1 \leftarrow$ first point in $C_1$
 3  $p_2 \leftarrow$ first point in $C_2$
 4  while $p_1 \neq \emptyset$ and $p_2 \neq \emptyset$
 5  do  if $p_1 = p_2$
 6        then INSERT($U, p_1$)
 7             $p_1 \leftarrow$ next point in $C_1$
 8             $p_2 \leftarrow$ next point in $C_2$
 9      else if $p_1 < p_2$
10        then $p_1 \leftarrow$ next point in $C_1$
11        else $p_2 \leftarrow$ next point in $C_2$
12  return $U$
```

Set operations involving point collections are more predictable (in the worst case) than set operations on general rectangle collections. We can exploit this fact by transforming an algebra expression into an equivalent expression that applies intersection, union, or difference to point collections instead of arbitrary rectangle collections, as much as possible. For example, the expression

$$\textit{Line} \cap ((\textit{starts}\ \text{"From:"} \cup \textit{starts}\ \text{"Sender:"} \cup \textit{contains}\ \text{"cmu.edu"}) \tag{4.1}$$

refers to four point collections: *Line* and the three quoted literals. In this expression, the union operators combine arbitrary rectangle collections generated by the unary relational operators *starts* and *contains*. The equivalent expression

$$((\textit{Line} \cap \textit{starts}\ \text{"From:"}) \cup (\textit{Line} \cap \textit{starts}\ \text{"Sender"})) \cup (\textit{Line} \cap \textit{contains}\ \text{"cmu.edu"}) \tag{4.2}$$

ensures that every union involves a point collection. In general, if we denote an expression known to return a point collection by P and other expressions by E, the transformation is described by the following rules, applied repeatedly until none match:

$$\begin{aligned} P \cap (E_1 \cap E_2) &\rightarrow (P \cap E_1) \cap E_2 \\ P \cap (E_1 \cup E_2) &\rightarrow (P \cap E_1) \cup (P \cap E_2) \\ P - E &\rightarrow P - (P \cap E) \end{aligned}$$

This transformation does not necessarily improve performance, however. In the example given, suppose that *Line* is much larger than the other region sets (“From”, “Sender”, “cmu.edu”). Then expression 4.2, which queries the *Line* region set three times, may actually run slower than the expression 4.1, which combines the three literals before querying *Line*. LAPIS does not apply the transformation automatically. An expert user might use it, however, to optimize the performance of a pattern.

Other pattern-matching systems, such as Proximal Nodes [NBY95] and WebL [KM98], can perform set operations only on point collections. The pattern languages in these systems are constrained (primarily by the absence of unary relational operators) so that only patterns like 4.2 can be written.

4.6.4 Reordering Operands

For commutative operators like intersection and union, the implementation is free to change the order of operands. LAPIS uses this fact to optimize both $A \cap B$ and $A \cup B$. The intersection operator enumerates the rectangles in the smaller collection and queries each rectangle against the larger collection. The union operator uses COPY to duplicate the larger collection (which never takes more than linear time, and so can be faster than rebuilding it) and then inserts rectangles from the smaller collection into the larger.

Optimizing the intersection operator also optimizes the binary relational operators, since they are implemented using intersection. Consider the expression:

Sentence contains "Gettysburg"

In a typical document, *Sentence* would have far more matches than "Gettysburg", so it would be faster to enumerate the matches to "Gettysburg" and query them against the *Sentence* region set. Since the internal representation of this expression is

Sentence $\cap$ *contains* "Gettysburg"

this optimization happens automatically. The intersection operator observes that *contains* "Gettysburg" contains fewer rectangles than *Sentence*, so it queries *contains* "Gettysburg" against *Sentence*.

The same optimization works for the complementary expression

"Gettysburg" *in Sentence*

LAPIS still queries the smaller region set derived from "Gettysburg" against the larger region set derived from *Sentence*, but this time it returns "Gettysburg" regions instead of *Sentence* regions. Note that, since *Sentence* is a flat region set, *in Sentence* can be computed from *Sentence* in $O(1)$ time by placing a wrapper around it, as described in Section 4.6.2.

4.6.5 Tandem Traversal

The simple algorithm for $A \cap B$ (Algorithm 4.3) takes each rectangle in B and traverses A recursively to find the intersections. I call this algorithm *iterative intersection.* The iterative algorithm takes no advantage of the fact that B is also a tree, hopefully organizing its rectangles with some locality. *Tandem intersection* exploits this by traversing both A and B simultaneously.

The tandem intersection algorithm is shown in Algorithm 4.15. The key line is line 1, which tests whether the bounding boxes of the entire trees A and B intersect. If their bounding boxes do not intersect, then we know that none of the rectangles stored in A can possibly intersect the rectangles stored in B. Thus, a single comparison high in the tree may be able to eliminate many comparisons at the leaves.

If the bounding boxes of A and B intersect after all, then the algorithm recursively drills into either A or B. The algorithm drills into A only if A's bounding box encloses B's bounding box, or B has no children; otherwise it drills into B.

Algorithm 4.15 TANDEMINTERSECTION finds the intersection of two tree-shaped rectangle collections by traversing both trees at the same time.

```
TANDEM($A, B$)
1  if $A.bbox$ doesn't intersect $B.bbox$
2     then return
3  if $A$ and $B$ are leaves
4     then yield $A.rectangle \cap B.rectangle$
5  else if $A.bbox \supseteq B.bbox$ or $B$ is a leaf
6     then for each $C$ in $A.children$
7             do TANDEM($C, B$)
8     else for each $C$ in $B.children$
9             do TANDEM($A, C$)
```

Tandem intersection is not guaranteed to be faster than iterative intersection. In the worst case, tandem intersection may do more work, because it can compare nodes of A with any node of B, while iterative intersection only compares nodes of A with leaves of B. However, the extra work is bounded by at most a factor of 2, as the following argument shows. We will compare the number of recursive calls made by tandem intersection with the number of recursive calls made by iterative intersection. Recall that the iterative intersection of A and B calls QUERY(A, r) for all rectangles $r \in B$. The following lemma relates the number of TANDEM calls to the number of QUERY calls:

Lemma 1. *For any two tree nodes A and B, if* TANDEM(A, B) *is called during a tandem intersection of two rectangle collections, then for all rectangles $r \in B$,* QUERY(A, r) *is called in the iterative intersection of the same collections.*

Proof. By induction on the depth of recursion. For the top-level call to TANDEM(A, B), A is the root of the tree, so iterative intersection must also start by calling QUERY(A, r) for all rectangles $r \in B$. For the induction step, we assume the hypothesis true for a call to TANDEM at recursion depth k, then show that the hypothesis is true for any recursive call it makes at recursion depth $k + 1$. There are four cases, corresponding to the conditions tested by Algorithm 4.15:

- $A.bbox \cap B.bbox = \emptyset$. In this case, tandem intersection makes no recursive calls to TANDEM, so there is nothing to prove.

- $A.bbox \supseteq B.bbox$. In this case, tandem intersection recursively calls TANDEM(C, B) for every child C of A. From the induction hypothesis, we know that iterative intersection calls

Query(A, r) for every $r \in B$. Since $r \subseteq B.bbox \subseteq A.bbox$, each of these Query calls must call Query(C, r) for all children C of A as well, so the hypothesis is proved for depth $k + 1$.

- B is a leaf. Like the previous case, tandem intersection drills into A. But there is only one rectangle in B, namely B itself. Thus, by the induction hypothesis, iterative intersection calls Query(A, B). Since $A.bbox$ and $B.bbox$ intersect, Query(A, B) must recursively call Query(C, B) for all children C of A, so the hypothesis is proved for depth $k + 1$.

- Otherwise, tandem intersection drills into B, recursively calling Tandem(A, C) for every child C of B. Since the induction hypothesis implies that Query(A, r) is called for all $r \in B$, and C is a subtree of B, we trivially have Query(A, r) for all $r \in C$. Thus the hypothesis is proved for depth $k + 1$.

□

Using Lemma 1, an amortized analysis shows that tandem intersection makes no more than twice as many calls to Tandem as iterative intersection would make to Query. We will charge the cost of calling Tandem(A, B) (not including its recursive calls) to all the Query(A, r) calls made with the rectangles in the leaves of B. Lemma 1 guarantees that all these Query calls are made. The share of the cost assigned to Query(A, r) is proportional to the depth of r in the tree, so that if B has branching factor m at every node and r is stored at depth d, then r is charged $1/m^d$ of the cost. Since every leaf of B contains a rectangle, the sum of these costs is 1. Turning the analysis around to look at it from the perspective of a Query call, Query(A, r) may be charged for some part of a Tandem(A, B) call for each B on the path from r to the root. If the minimum branching factor of the tree is m, then the cost charged to Query(A, r) is at most

$$1 + \frac{1}{m} + \frac{1}{m^2} + \frac{1}{m^3} + \cdots < \frac{m}{m-1}$$

which is at most 2 if $m \geq 2$. Thus the sum of the costs charged to the Query calls is at most twice the number of Query calls. Since this total cost is the same as the number of Tandem calls, there can be at most twice as many Tandem calls as Query calls.

So even in the worst case, tandem intersection is at most a factor of two worse than iterative intersection. In the average case, however, tandem intersection takes only $O(N)$, as the performance measurements in Section 4.7 show. This is significantly better than the $N \log N$ average-case time for iterative intersection. LAPIS uses tandem intersection exclusively.

Tandem intersection can be applied to any tree-like rectangle collection, including RB-trees, quadtrees, region arrays, and syntax trees. Since not all leaves in a quadtree contain a rectangle, however, the amortized analysis does not work for quadtrees, so tandem intersection on a quadtree may be more than a factor of 2 worse than iterative intersection. The next section discusses a form of tandem traversal specialized to quadtrees.

Tandem intersection also works when A and B are different data structures. For example, an RB-tree can be tandem-intersected with a region array or a syntax tree.

4.6.6 Quadtree Traversal

Quadtrees are particularly amenable to tandem traversal, because every quadtree decomposes region space in exactly the same way. To intersect two quadtrees A and B whose bounding boxes are the same, it suffices to recursively intersect only the four pairs of corresponding children, $A.children[i]$ with $B.children[i]$, as i ranges from 1 to 4. Algorithm 4.16 shows how this is done. As long as A and B are not leaves, QUADTREEINTERSECT traverses them in tandem. Whenever one tree reaches a leaf, the algorithm copies the other tree and calls INTERSECTWITH to intersect the leaf's rectangle with all the nodes in the other tree. When both functions return from their recursive traversal, they call MERGE (Algorithm 4.9) to test whether the intersected children can be merged into one node containing a single rectangle. The time to run the overall algorithm is proportional to the number of nodes in the quadtrees, which is $O(\min(nN, n^2))$ in the worst case.

Algorithm 4.16 QUADTREEINTERSECT finds the intersection of two quadtrees by tandem traversal.

```
QUADTREEINTERSECT($A, B$)
 1  if $A.children = \emptyset$
 2     then $U \leftarrow$ COPY$B$
 3          INTERSECTWITH($U, A.rectangle$)
 4  else if $B.children = \emptyset$
 5     then $U \leftarrow$ COPY$A$
 6          INTERSECTWITH($U, B.rectangle$)
 7     else $U \leftarrow$ new QUADTREE
 8          for $i \leftarrow 1$ to $4$
 9          do $U.children[i] \leftarrow$ QUADTREEINTERSECT($A.children[i], B.children[i]$)
10          MERGE($U$) return $U$

INTERSECTWITH($A, r$)
 1  if $r \cap A.bbox \neq \emptyset$
 2     then if $A.children = \emptyset$
 3             then $A.rectangle \leftarrow r \cap A.rectangle$
 4             else for each $C$ in $A.children$
 5                     do INTERSECTWITH($C, r$)
 6                  MERGE($A$)
```

Similarly, tandem traversal can can be used for the union or difference of two quadtrees. These algorithms save time relative to the general union and difference algorithms, which used INSERT and DELETE, because the cost of traversing the tree to insert (or delete) a rectangle is amortized over all the rectangles to be processed.

4.6.7 Plane-Sweep Intersection

Finding the intersections in a collection of rectangles is a well-studied problem in computational geometry. Traditionally, the rectangle-intersection problem is formulated as follows: given a col-

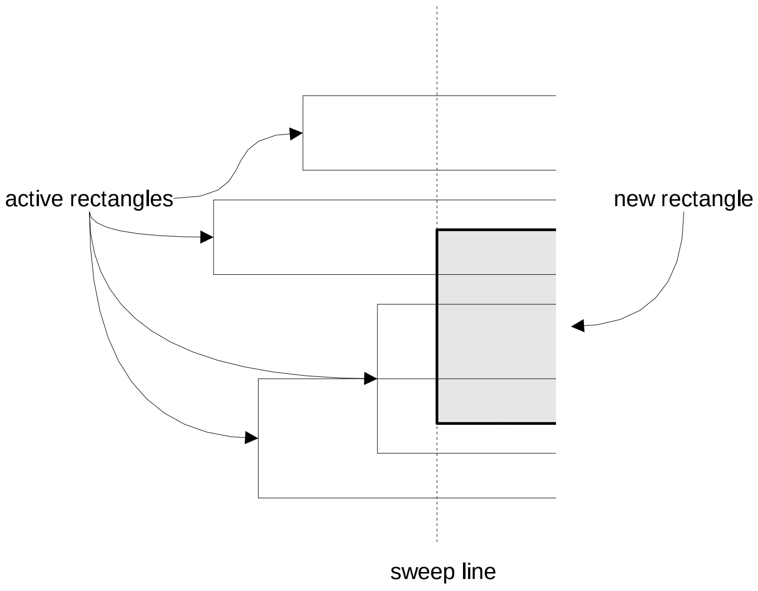


Figure 4.16: The plane-sweep intersection algorithm (after Preparata & Shamos [PS85], Figure 8.29).

lection of N axis-aligned rectangles, find all intersecting pairs of rectangles. The classic text by Preparata and Shamos [PS85] gives an optimal, $O(N \log N + F)$-time solution to this problem, where F is the number of intersections found. This section briefly outlines this algorithm, then describes how it can be used for region algebra intersection.

The algorithm uses the plane-sweep technique, passing a vertical sweep line through the rectangles from left to right (Figure 4.16). The sweep line stops at every x-coordinate of a rectangle, either its left side or its right side. The algorithm maintains a set of *active rectangles*, which are the rectangles currently intersecting the sweep line. When the left side of a rectangle is encountered by the sweep line, the rectangle is added to the active set. When its right side is encountered, it is deleted from the active set.

The active set is used to detect rectangle intersections. When a rectangle is added to the active set, the algorithm checks whether the new rectangle's y-interval intersects the y-interval of a currently-active rectangle. If so, then the rectangles are reported as an intersecting pair. In order to make this search fast, the active set's y-intervals are stored in an *interval tree*, a data structure that allows intervals to be inserted and deleted in $O(\log N)$ time, and handles range queries in $O(\log N + F)$ time. The interval tree was discovered independently by Edelsbrunner [Ede80] and McCreight [McC81]. See Preparata & Shamos [PS85] for more details.

To find the intersections in a collection of N rectangles, the plane sweep algorithm needs $O(N \log N)$ preprocessing time to separately sort the x-coordinates of the rectangles for the plane sweep and the y-coordinates for initializing the interval tree. The plane sweep itself takes $O(N \log N + F)$ time, so the overall time is $O(N \log N + F)$. Preparata and Shamos prove that this time is optimal for a decision-tree algorithm (i.e., one that only makes comparisons between rectangle coordinates).

For the region algebra, we want to solve a slightly different problem: given *two* collections of rectangles A and B, find all intersecting pairs (a, b) such that $a \in A$ and $b \in B$. One solution is to combine both collections, $A \cup B$, marking each rectangle according to whether it came from A or B (or both). Then find the intersecting pairs in the union and discard all but the (a, b) pairs. The problem with this approach is that finding and reporting the self-intersections, (a, a') and (b, b'), may dominate the running time of the algorithm, making it $O(N^2)$ in the worst case.

Instead, we maintain two active sets, one for A and one for B. The sweep line passes across the union of both collections. When an A rectangle becomes active, its y-interval is tested against B's active set to find intersecting pairs, but then inserted into A's active set. Vice versa for B. If the two collections have size M and N respectively, then this algorithm takes $O((M + N) \log(M + N))$ time to sort the x-coordinates of both collections together for the plane sweep. Querying to find the F intersecting pairs takes $O(M \log N + N \log M + F)$ time, inserting and deleting rectangles from active sets takes $O(M \log M + N \log N)$. The overall time is $O((M + N) \log(M + N) + F)$.

LAPIS does not implement the plane-sweep algorithm, so comparing its performance in practice is left for future work.

4.6.8 Counting, Min, and Max

Some of the operator definitions given in Chapter 3 are infeasible to implement directly, since they seem to require large intermediate results (e.g. forming $A - \{a\}$ for every region $a \in A$). This section describes how a few of the more important operators can be implemented efficiently in practice.

The *first* operator returns the first region in a region set in lexicographic order. The definition of this operator given in Section 3.6.5 would be very slow to implement:

$$\textit{first}\, A \equiv \textit{forall}\, (a : A) \, . \, a- > (A - a)$$

In practice, *first* is trivial to implement on a rectangle collection sorted in lexicographic order, like an RB-tree or region array. The result is the lower-left corner of the first rectangle in the collection.

The general counting operator *nth* returns the nth region in a region set. This operator is much harder to optimize, because the lexicographic order of regions does not necessarily correspond to the lexicographic ordering of the rectangles (Figure 4.17). Although the problem could be solved by a plane-sweep technique, LAPIS takes the simpler approach of defining *nth* only for point collections, so that the nth region always corresponds to the nth rectangle. Nested, overlapping, and flat region sets (and intersections, differences, and unions thereof) are always point collections, so this is not a serious limitation. Applying *nth* to a non-point rectangle collection raises an exception in LAPIS.

The *max* and *min* operators return the outermost and innermost regions in a set

$$\begin{aligned} \textit{max}\, A &\equiv \textit{forall}\, (a : A) \, . \, a - \textit{in}\, (A - a) \\ \textit{min}\, A &\equiv \textit{forall}\, (a : A) \, . \, a - \textit{contains}\, (A - a) \end{aligned}$$

LAPIS implements *max* by finding the *max* of each rectangle in A, which is just the rectangle's upper-left corner $[s_1, e_2]$, then querying the collection for *contains* $[s_1, e_2]$ to check that no other regions contain it. *Min* is implemented similarly.

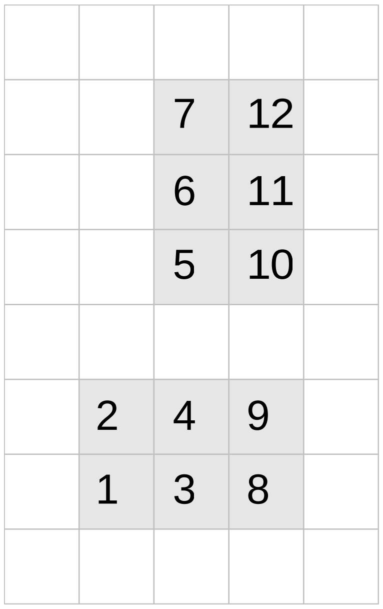


Figure 4.17: The lexicographic ordering of regions has no simple relationship with lexicographic ordering of rectangles. Finding the nth region in a rectangle collection may require jumping back and forth between rectangles.

Data structure	Region sets	Query	Insert	Delete	Space
RB-tree	any	$O(N)$	$O(\log N)$	$O(N+F)$	$O(N)$
quadtree	any	$O(F \log n)$	$O(n)$	$O(F \log n)$	$O(\min(nN, n^2))$
4D-tree	any	$O(N^{3/4}+F)$	$O(\log N)$	$O(\log N+F)$	$O(N)$
4D range tree	any	$O((\log N)^3+F)$	$O((\log N)^3)$	$O((\log N)^3+F)$	$O(N(\log N)^3)$
region array	monotonic	$O(\log N+F)$	$O(\log N)$	$O(\log N+F)$	$O(N)$
syntax tree	nested	$O(\log N+F)$	$O(\log N)$	$O(\log N+F)$	$O(N)$

Table 4.1: Rectangle-collection data structures. N is the number of rectangles in the collection, F is the number of rectangles found by a query or affected by a deletion, and n is the length of the string. Boldface indicates that the data structure is implemented in LAPIS.

4.7 Performance

The rectangle-collection data structures and algorithms discussed in this chapter are summarized in Tables 4.1–4.5. A boldfaced entry indicates that the data structure or algorithm is implemented in LAPIS. Some data structures are appropriate only to certain kinds of region sets. The type hierarchy of region sets is summarized in Figure 4.18.

These tables only give the worst-case time for each operation. To study the average case, the performance of the LAPIS implementation was measured directly with two experiments. The first experiment tested each algebra operator on random region sets of increasing size, in order to see how individual operators scale. The second experiment applied real patterns (the patterns from the LAPIS library) to a set of real documents (web pages, source code, and plain text), in order to gain a sense for how region algebra operators perform in practice.

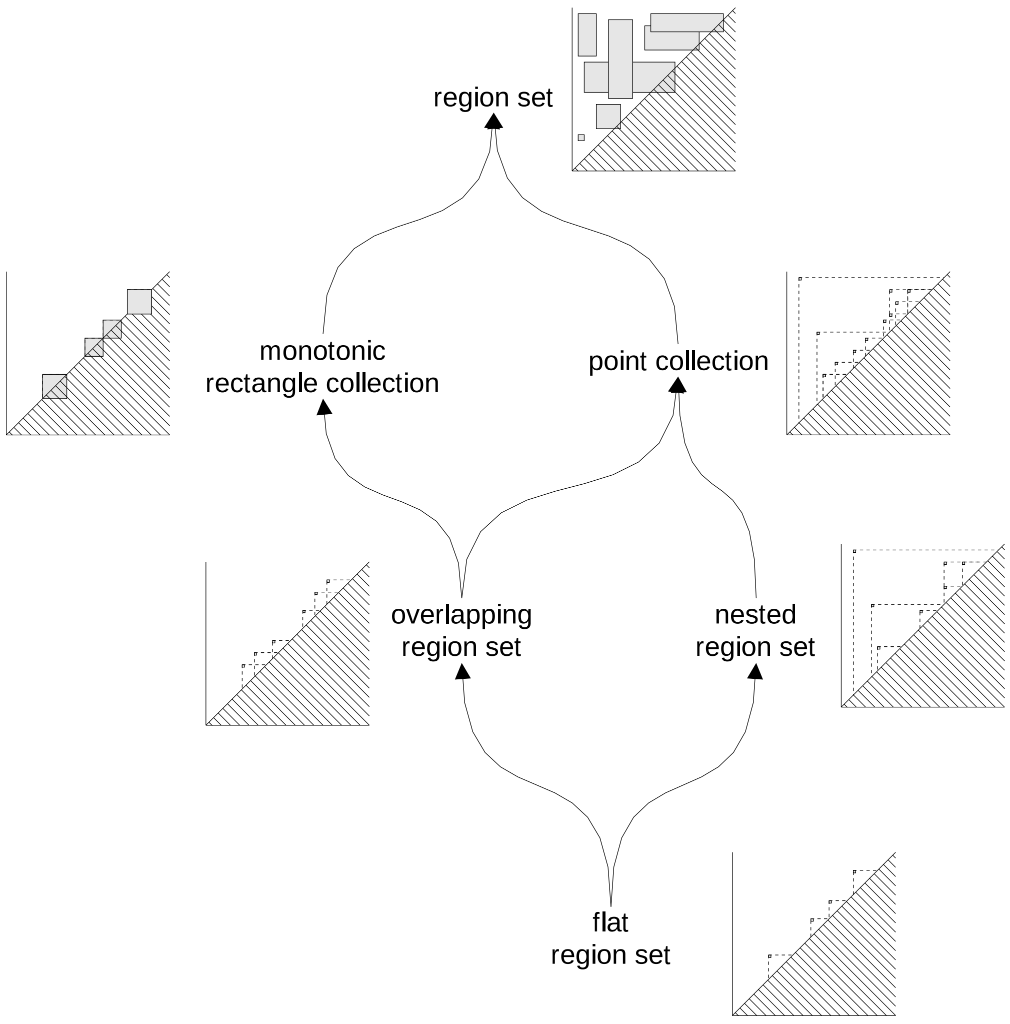


Figure 4.18: Hierarchy of region set types.

Expression	Time	Result
***op* RB-tree**	$O(N \log N)$	RB-tree
op quadtree	$O(\min(nN, n^2))$	quadtree
***op* region array**	$O(N)$	region array

Table 4.2: Unary relational operators, such as *in, contains, just-before, just-after,* etc.

Expression	Time	Result
RB-tree ∩ RB-tree (tandem)	$O(MN + F \log F)$	RB-tree
RB-tree ∩ RB-tree (plane-sweep)	$O((M+N)\log(M+N)+F)$	RB-tree
quadtree ∩ quadtree	$O(\min(nN, n^2))$	quadtree
point RB-tree ∩ region array	$O(M \log N)$	point RB-tree
point RB-tree ∩ point RB-tree	$O(M+N)$	point RB-tree

Table 4.3: Set intersection.

Expression	Time	Result
RB-tree ∪ **RB-tree**	$O((M+N)\log(M+N))$	RB-tree
quadtree ∪ quadtree	$O(\min(nN, n^2))$	quadtree
point RB-tree ∪ point RB-tree	$O(M+N)$	point RB-tree

Table 4.4: Set union.

Expression	Time	Result
RB-tree − **RB-tree**	$O(MN^2)$	RB-tree
quadtree − quadtree	$O(\min(nN, n^2))$	quadtree
point RB-tree − region array	$O(M \log N)$	point RB-tree
point RB-tree − point RB-tree	$O(M+N)$	point RB-tree

Table 4.5: Set difference.

4.7.1 Random Microbenchmarks

The first experiment tested LAPIS against randomly-generated region sets. No documents were needed for this experiment. Since region sets are just sets of integer pairs, they can be generated at random without reference to any document.

Region sets were generated in such a way in such a way that expected number of intersections between any two region sets of size N would be $O(N)$. Two kinds of region sets were generated: flat and nested. Random flat region sets were generated by first choosing $2N$ small lengths l_i uniformly from $0 \ldots 9$, then using the lengths to create a set of $2N$ endpoints $p_i = \sum_{j=1}^{i} l_i$, then connecting each pair of endpoints $[p_{2i}, p_{2i+1}]$ to create a flat set of N regions. Two flat sets of size N generated with this approach would share $N/100$ regions on the average. A random nested region set was generated by first generating $2N$ endpoints as for flat region sets, then proceeding in sorted order through the endpoints, making a uniform decision at each point whether to open a new parent (pushing the point on a stack), close the current parent (popping from the stack), or make a leaf (consuming both the point and its next point to produce a region). When one alternative was impossible (e.g. the stack was empty, or only enough points remained to close the stack), only the possible alternatives were chosen uniformly. This process produced nested region sets that shared $N/200$ regions on the average. Flat region sets were stored in a region array, and nested region sets in an RB-tree.

The timing tests were performed in LAPIS, compiled with Jikes 1.06 and running under Java 1.3.0 with the HotSpot just-in-time compiler, on an 850 MHz Pentium III running Linux. To limit interference from the garbage collector, the tests were run with 128MB of preallocated heap storage, and the garbage collector was forced to run synchronously before every test. Startup effects, such as class loading, just-in-time compiling, and instruction cache warmup, were eliminated by running each sequence of tests twice and keeping only the results of the second. Every test was repeated at least three times or for at least 1 second, whichever took longer, and the average value was reported.

The tests applied each algebra operator to random region sets of varying size. The results are shown in Figures 4.19–4.22. In each graph, the x-axis is the number of rectangles in one operand, N, on a logarithmic scale. The y-axis is the cost per input rectangle, computed by dividing the running time T by the number of rectangles N.

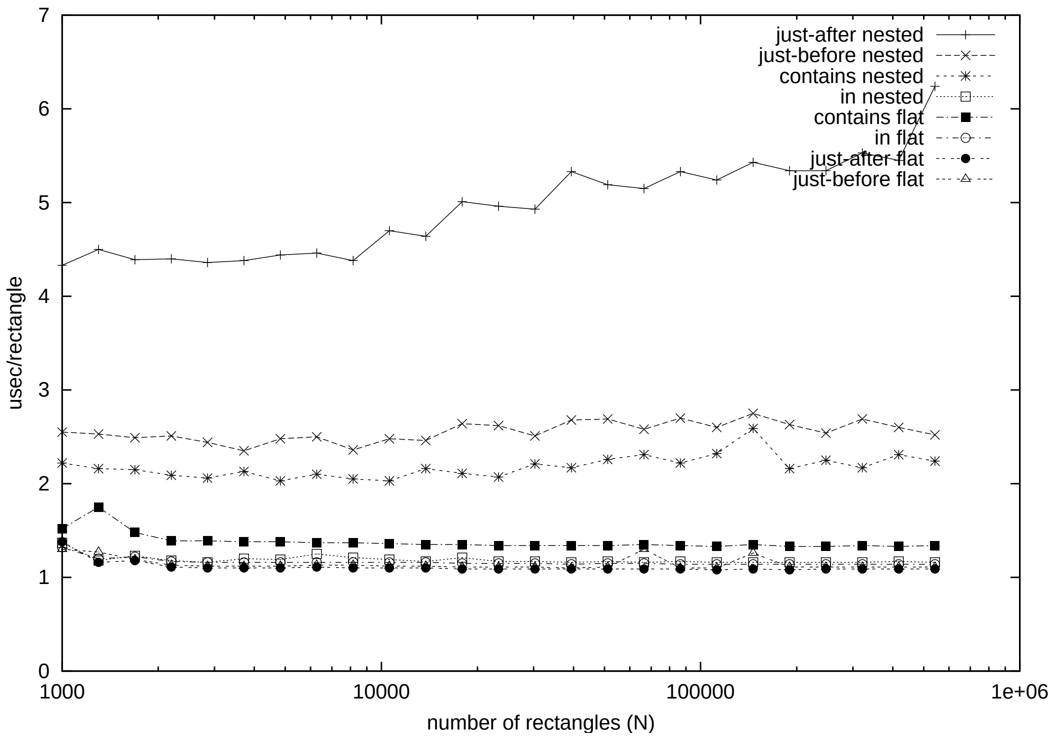


Figure 4.19: Relational operators applied to random region sets of size N.

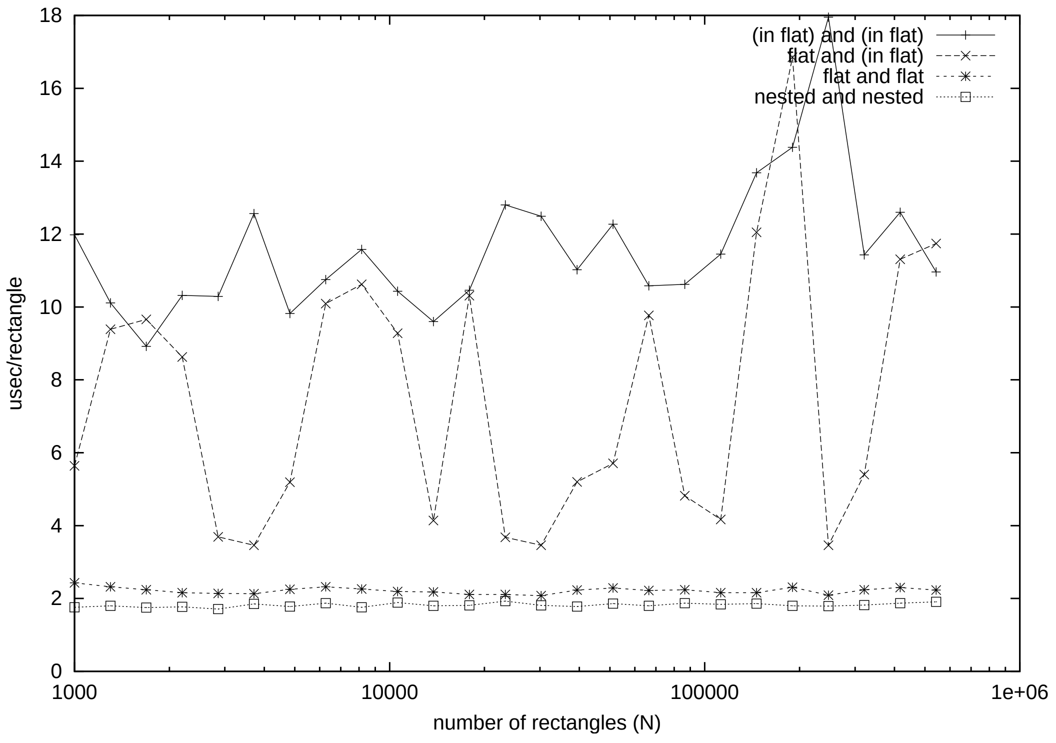


Figure 4.20: Intersection of random region sets of size N.

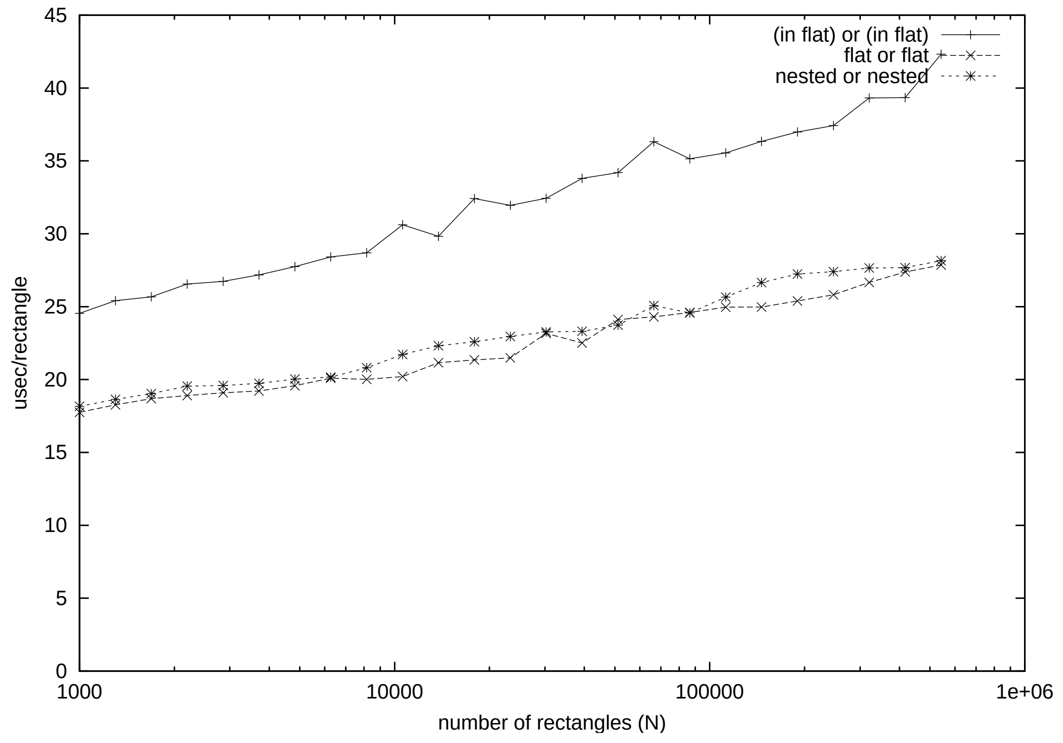


Figure 4.21: Union of random region sets of size N.

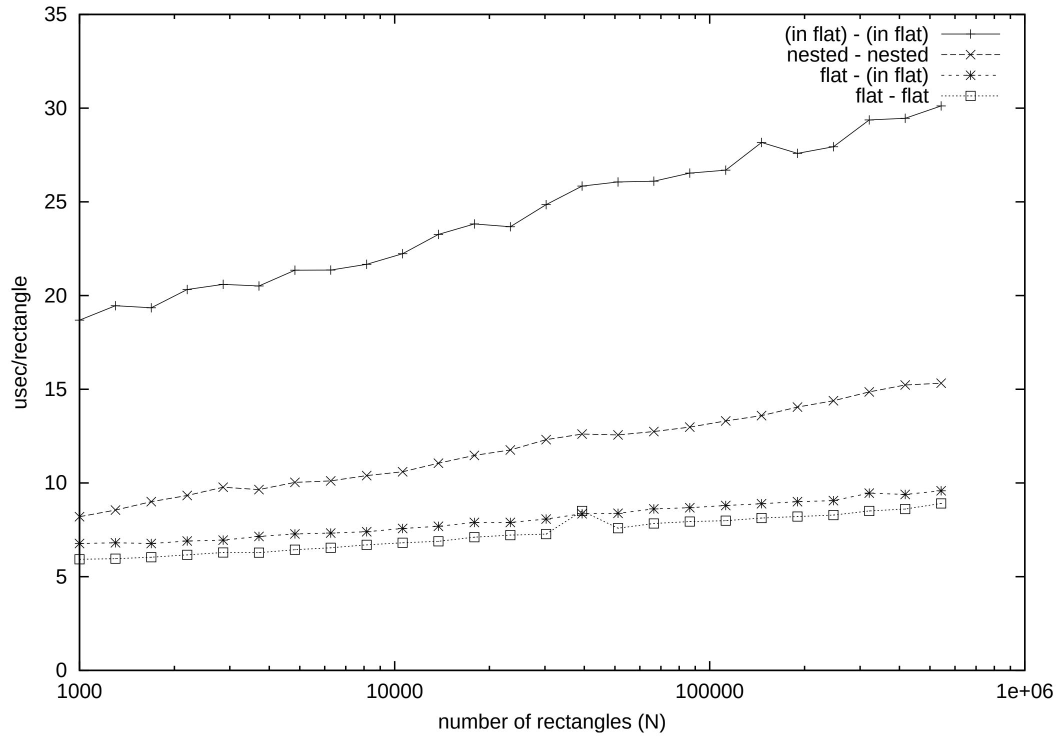


Figure 4.22: Difference of random region sets of size N.

The slope of the line indicates an operator's asymptotic behavior as N increases. When $T = O(N)$, the line is flat ($T/N = k$). When $T = O(N \log N)$, the line has constant slope ($T/N = k \log N$). When $T = O(N^{1+\varepsilon})$ for some $\varepsilon > 0$, the line has an exponential shape ($T/N = k2^{\varepsilon \log N}$).

Figure 4.19 shows some representative relational operators. Most of the relational operators behave like $O(N)$. The only exception is *just-after nested*, which produces its output rectangles in unsorted order, requiring $O(N \log N)$ time to sort them. Relational operators on a flat set (stored in a region array) are considerably faster than relational operators on a nested set (stored in an RB-tree), because the region-array operator produces a simple wrapper around the region array that takes no extra space.

Intersections are more complicated (Figure 4.20). The bottom two lines in the graph (*flat and flat*, *nested and nested*) show that intersections between point collections take linear time. The top two lines (*flat and (in flat)* and *(in flat) and (in flat)*) are much more complicated, because sometimes the result set doesn't require sorting and sometimes it does. The cost per rectangle varies between a constant lower bound and a logarithmic upper bound.

Set union and set difference operators (Figure 4.21 and 4.22) increase like $O(N \log N)$.

4.7.2 Realistic Benchmarks

The second experiment measured the time to evaluate all the TC patterns in the LAPIS library against real documents. The LAPIS library contains 82 TC patterns, which compile into 478 region algebra operators. These TC patterns include concepts like `Number`, `Line`, `Sentence`, `Time`, `Date`, `PhoneNumber`, `ZipCode`, `URL`, and `EmailAddress`, among others. A complete list of the named patterns in the library can be found in Appendix A. The library also contains three parsers (HTML, Java, and Characters), but the running time of the parsers was excluded from the measurements because the parsers are written in Java and do not use the region algebra.

Three sets of documents were used in the test:

- 339 Java source files from the LAPIS source code, ranging in size from 1 KB to 211 KB;
- 158 HTML web pages from the LAPIS distribution and other sources, ranging from 1 KB to 235 KB;
- 63 outputs of `ls -l` for various directories on a Linux workstation, ranging from 1 KB to 134 KB.

For each document, the total time to match all the TC patterns was measured, using the same techniques described in the previous section. The total number of input region rectangles processed by pattern operators was also recorded. The results are shown in Figures 4.23–4.25. In each graph, the x-axis is the total number of rectangles processed by all the patterns, on a logarithmic scale, and the y-axis is the cost per input rectangle, computed by dividing the total running time by the total number of rectangles. Each point represents a single document. The graphs suggest that the typical region algebra operator in these patterns takes between 3 and 6 μsec per rectangle on realistic data, which is in the same ballpark as the microbenchmarks from the previous section. No upward trend is detectable as documents grow larger.

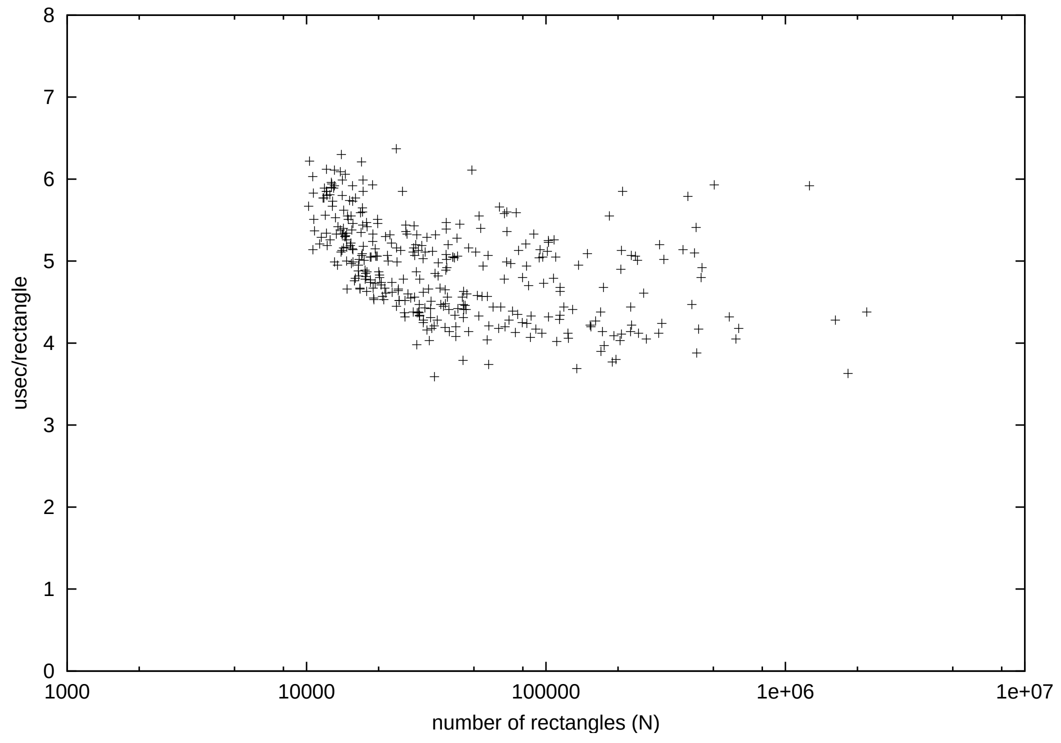


Figure 4.23: Performance of LAPIS library patterns on Java source files.

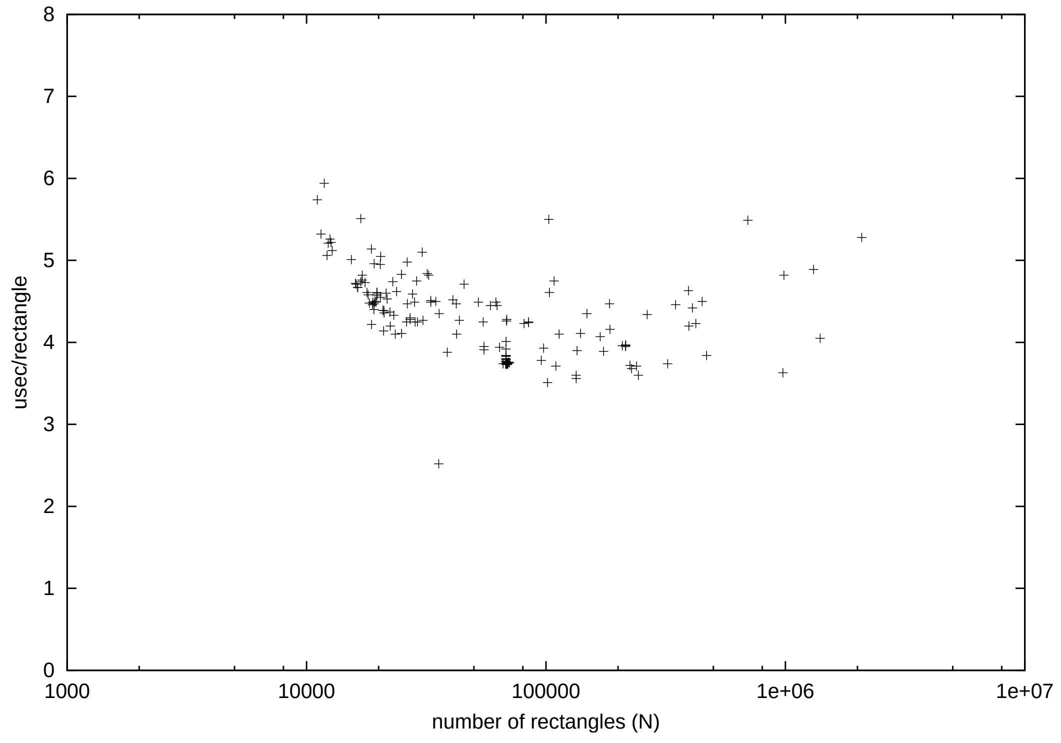


Figure 4.24: Performance of LAPIS library patterns on HTML web pages.

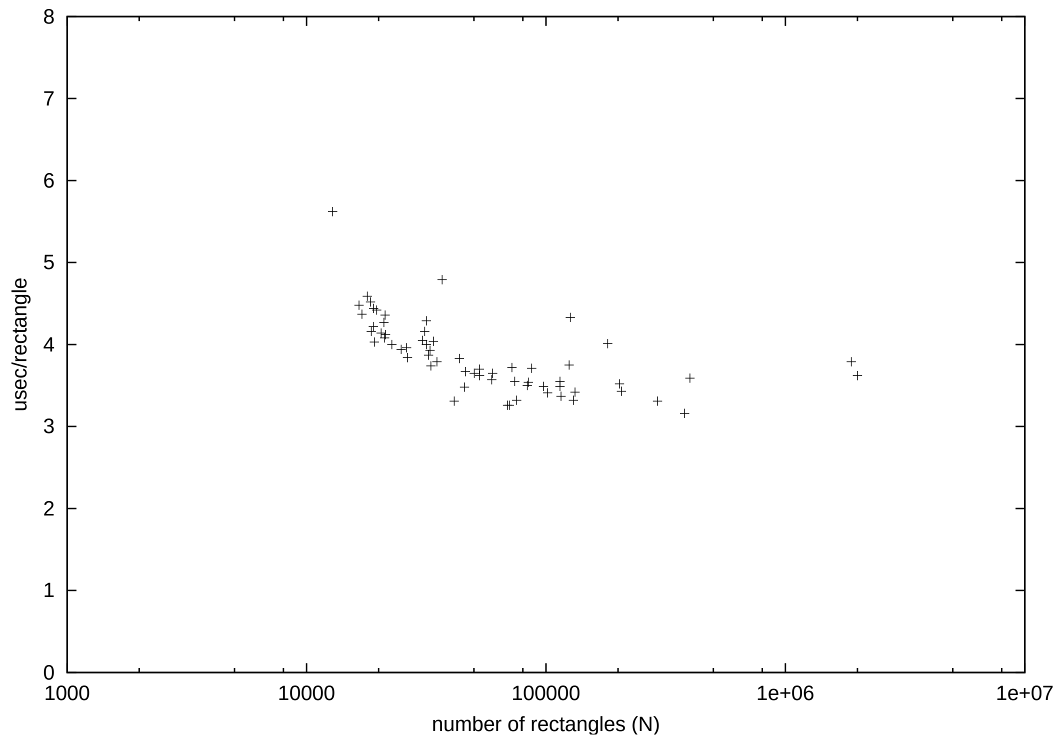


Figure 4.25: Performance of LAPIS library on `ls -l` outputs.

To get a sense for performance in terms of document size, Table 4.6 shows the pattern matching rate for each kind of benchmark document, computed by dividing the size of the document in kilobytes by the total running time on that document in seconds. From the table, a typical library pattern is matched at a rate of roughly 1,500 KB/sec. By comparison, the Jakarta regular expression package [Jak99] matches a simple word pattern (`\w+`) at roughly 750 KB/sec on the same hardware, compiler, and virtual machine. One possible reason for TC's faster performance is that most TC patterns process only a few regions relative to the size of the file, whereas a regular expression tests every character. Another possible reason is that the Jakarta package is not fully optimized, although it is the fastest Java regular expression matcher I have found. It would be interesting to compare the performance of region algebra expressions with regular expressions directly, using equivalent patterns in each language. A proper comparison would require using the fastest regular expression package available, however, which would certainly be written in C or C++. The region algebra implementation would also have to be rewritten in a higher-performance language to make the comparison fair, so this possibility remains future work.

Documents	All patterns in library	Per pattern	Per operator
Java source files	21.5 KB/sec (±2.7)	1,760 KB/sec (±220)	10,300 KB/sec (±1300)
HTML web pages	20.6 KB/sec (±2.0)	1,690 KB/sec (±160)	9,850 KB/sec (±960)
`ls -l` output	17.6 KB/sec (±1.6)	1,440 KB/sec (±130)	8,410 KB/sec (±760)

Table 4.6: Pattern matching rate for the benchmark documents (mean ± standard deviation). *All patterns in library* is the rate at which all TC patterns in the library are matched. *Per pattern* is the average rate for each of the 82 patterns. *Per operator* is the average rate for each of the 478 operators in the patterns.