Rectangle Partitioning Algorithm: Finding Connected Components

Overview

This document explains the algorithm behind `Rect.partitions()` and `Rect.bounding_boxes()`, which efficiently partition a collection of rectangles into groups of transitively intersecting rectangles and compute their bounding boxes.

Problem Statement

Given a set of axis-aligned rectangles, we want to:

1. **Partition** them into disjoint groups where rectangles in each group are "connected"
2. **Compute bounding boxes** for each group

Two rectangles are **connected** if:

- They intersect directly, OR
- There exists a chain of rectangles connecting them through intersections

This is an equivalence relation that partitions the rectangles into connected components.

The Graph-Theoretic Perspective

> "Point of view is worth 80 IQ points." — Alan Kay

The key insight is to view this as a **graph problem**:

- Each rectangle is a **node** in a graph
- Two nodes are **connected by an edge** if their rectangles intersect
- We want to find all **connected components** in this graph

This transforms a geometric problem into a well-studied graph theory problem!

Why Rectangle Intersection = Graph Connectivity?

Two rectangles $R_1 = (l_1, t_1, r_1, b_1)$ and $R_2 = (l_2, t_2, r_2, b_2)$ intersect if and only if:

$$l_1 < r_2 \text{ AND } r_1 > l_2 \text{ AND } t_1 < b_2 \text{ AND } b_1 > t_2$$

This can be decomposed into two independent conditions:

1. **Horizontal overlap**: $[l_1, r_1)$ overlaps $[l_2, r_2)$
2. **Vertical overlap**: $[t_1, b_1)$ overlaps $[t_2, b_2)$

Two rectangles intersect **if and only if** they overlap both horizontally AND vertically.

This decomposition is crucial for the algorithm's efficiency.

The Algorithm

Step 1: Filter and Deduplicate

```python
rects = frozenset(filter(None, rects))
```

- Remove `EMPTY` rectangles (represented as `None` or falsy values)
- Convert to a `frozenset` to eliminate duplicates

Why remove EMPTY? In lattice terms, `⧄` is the identity element for join, so it doesn't contribute to any bounding box. Geometrically, it has no area and doesn't connect anything.

Step 2: Build Interval Trees

```python
htree = ITree(Interval(rect, *horizontal(rect)) for rect in rects)
vtree = ITree(Interval(rect, *vertical(rect)) for rect in rects)
```

We build two **interval trees**:

- `htree`: indexes rectangles by their horizontal extent `[left, right)`
- `vtree`: indexes rectangles by their vertical extent `[top, bottom)`

Interval trees support efficient queries: "find all intervals that overlap with a given interval."

- **Construction**: O(n log n) for n rectangles
- **Query**: O(log n + k) where k is the number of overlapping intervals

Step 3: Find Neighbors Using Dual Interval Tree Queries

```python
neighbors = {}
for rect in rects:
    neighbors[rect] = frozenset(
        {i.rect for i in htree.search(Interval(rect, *horizontal(rect)))}
        & {i.rect for i in vtree.search(Interval(rect, *vertical(rect)))}
    )
```

For each rectangle, we find all rectangles that intersect with it by:

1. Query `htree` for rectangles with horizontal overlap → set `H`
2. Query `vtree` for rectangles with vertical overlap → set `V`
3. Intersecting rectangles are exactly `H ∩ V`

Why does this work?

As established earlier, two rectangles intersect ⟺ they overlap both horizontally AND vertically. So:

```
Intersecting rectangles = (Horizontally overlapping) ∩ (Vertically overlapping)
```

Complexity for this step:

For each of n rectangles:

- Horizontal query: O(log n + k_h) where k_h is horizontal overlaps
- Vertical query: O(log n + k_v) where k_v is vertical overlaps
- Set intersection: O(min(k_h, k_v))

Total: O(n log n + total overlaps)

Step 4: Find Connected Components

python

```python
def component(node):
    todo = set([node])
    while todo:
        node = todo.pop()
        seen.add(node)
        todo |= neighbors[node] - seen
        yield node

seen = set()
for node in neighbors:
    if node not in seen:
        yield set(component(node))
```

This is a standard **depth-first search (DFS) / breadth-first search (BFS)** for finding connected components:

1. Start from an unvisited node
2. Explore all its neighbors
3. Recursively explore their neighbors
4. Mark all visited nodes as part of the same component
5. Repeat for any remaining unvisited nodes

Complexity: O(n + e) where n is the number of rectangles and e is the number of edges (intersection pairs).

Since we already computed all neighbors in Step 3, this is essentially just traversing the adjacency list.

Total Time Complexity

Let's break down the full algorithm:

1. **Filter/deduplicate**: O(n)
2. **Build interval trees**: O(n log n)
3. **Find all neighbors**: O(n log n + k) where k is the total number of intersection pairs
4. **Connected components**: O(n + k)

Total: O(n log n + k)

where:

- n = number of distinct rectangles
- k = number of rectangle intersection pairs

This is efficient because:

- We avoid the naive $O(n^2)$ approach of checking every pair
- Interval trees reduce the search space significantly
- We only pay for actual intersections, not all possible pairs

The `bounding_boxes()` Method

python

```python
@classmethod
def bounding_boxes(cls, rects):
    for region in cls.partitions(rects):
        yield cls.bounding_box(*region)
```

This method combines partitioning with the lattice join operation:

1. Find each connected component (partition)
2. Compute the **supremum** (join) of all rectangles in that component

In lattice terms: for each connected component C, compute $\bigvee C$ (the least upper bound).

Since the join operation in the Rect lattice is the bounding box operation, this gives us the smallest rectangle containing all rectangles in the component.

Connection to Lattice Theory

From the formal proofs document, we know:

- The **join** ($|$) of two rectangles is their bounding box
- The **meet** ($\&$) of two rectangles is their intersection
- Rect forms a **complete lattice**, meaning arbitrary joins exist

The `partitions()` algorithm exploits the meet operation:

- Two rectangles are in the same partition if $R_1 \,\&\, R_2 \neq \varnothing$ (they intersect)
- Or transitively: if there exists a chain $R_1 \,\&\, R_2 \neq \varnothing$, $R_2 \,\&\, R_3 \neq \varnothing$, ..., $R_{n-1} \,\&\, R_n \neq \varnothing$

The `bounding_boxes()` operation computes:

$$\bigvee \{ R \mid R \in \text{partition} \}$$

This is well-defined because Rect is a complete lattice (Corollary 9.3 in the proofs document).

Example Walkthrough

Consider these rectangles:

$R_1 = (0, 0, 3, 2)$
$R_2 = (2, 1, 5, 3)$ [intersects R_1]
$R_3 = (4, 2, 7, 4)$ [intersects R_2 but not R_1]
$R_4 = (10, 0, 12, 2)$ [disjoint from all others]

Step 1: Build neighbor adjacency

$\text{neighbors}[R_1] = \{R_1, R_2\}$ [R_1 intersects R_2]
$\text{neighbors}[R_2] = \{R_1, R_2, R_3\}$ [R_2 intersects both R_1 and R_3]
$\text{neighbors}[R_3] = \{R_2, R_3\}$ [R_3 intersects R_2]
$\text{neighbors}[R_4] = \{R_4\}$ [R_4 is disjoint]

Step 2: Find connected components

Starting from R_1:

- Visit R_1, add its neighbors $\{R_1, R_2\}$
- Visit R_2, add its neighbors $\{R_1, R_2, R_3\}$
- Visit R_3, add its neighbors $\{R_2, R_3\}$
- All visited: $\{R_1, R_2, R_3\}$

Starting from R_4:

- Visit R_4, add its neighbors $\{R_4\}$
- All visited: $\{R_4\}$

Result: Two partitions: $\{R_1, R_2, R_3\}$ and $\{R_4\}$

Step 3: Compute bounding boxes

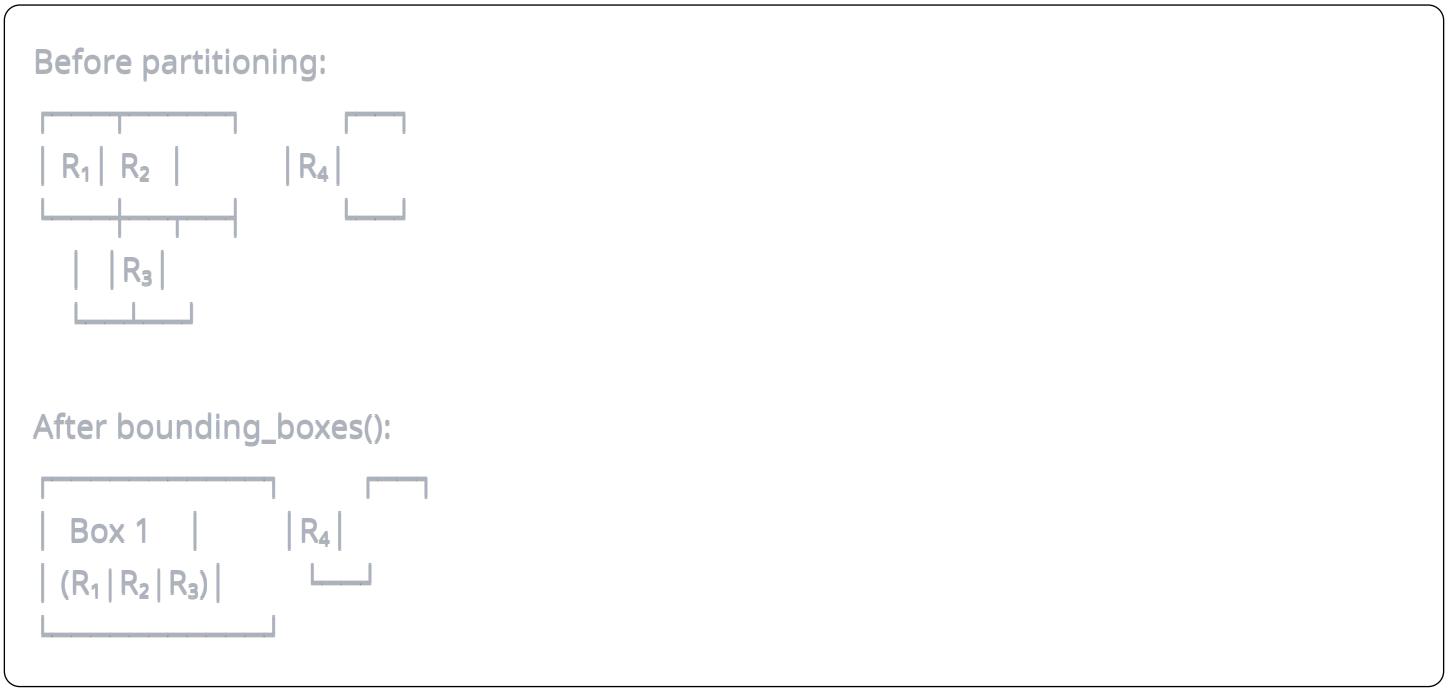
For $\{R_1, R_2, R_3\}$:

```
R₁ | R₂ | R₃ = (0, 0, 3, 2) | (2, 1, 5, 3) | (4, 2, 7, 4)
          = (min(0,2,4), min(0,1,2), max(3,5,7), max(2,3,4))
          = (0, 0, 7, 4)
```

For $\{R_4\}$:

```
R₄ = (10, 0, 12, 2)  [identity: R | ⧄ = R]
```

Visual Example



Even though R_1 and R_3 don't directly intersect, they're in the same partition because they're both connected to R_2.

Why Two Interval Trees?

You might wonder: why not use a single spatial index that handles 2D queries directly?

The two-tree approach has several advantages:

1. **Simplicity**: 1D interval trees are simpler to implement and reason about than 2D spatial structures
2. **Decomposition**: Separates the problem into two independent 1D overlap problems
3. **Flexibility**: Each tree can be optimized independently
4. **Correctness**: The intersection of results is exactly what we need

More sophisticated spatial data structures (like R-trees, quadtrees, or kd-trees) could potentially be faster for certain distributions of rectangles, but the dual interval tree approach is elegant and efficient for general cases.

Edge Cases

Empty Input

python

```
partitions([]) → yields nothing
bounding_boxes([]) → yields nothing
```

Single Rectangle

python

```
partitions([R₁]) → yields {R₁}
bounding_boxes([R₁]) → yields R₁
```

All Disjoint

python

```
partitions([R₁, R₂, R₃])  # all disjoint
→ yields {R₁}, {R₂}, {R₃}

bounding_boxes([R₁, R₂, R₃])
→ yields R₁, R₂, R₃
```

All Intersecting

```python
partitions([R₁, R₂, R₃])  # all mutually intersecting
→ yields {R₁, R₂, R₃}

bounding_boxes([R₁, R₂, R₃])
→ yields R₁ | R₂ | R₃
```

With EMPTY Rectangles

```python
partitions([R₁, ⧄, R₂, ⧄])
→ same as partitions([R₁, R₂])

# EMPTY is always filtered out
```

Implementation Notes

Why `frozenset`?

```python
rects = frozenset(filter(None, rects))
```

Using `frozenset` serves two purposes:

1. **Deduplication**: Identical rectangles are automatically merged
2. **Hashability**: We can use rectangles as dictionary keys in `neighbors`

The `Interval` Helper Class

```python
@dataclass(eq=True, frozen=True, slots=True)
class Interval:
    rect: Rect
    start: Real
    end: Real
```

This wraps a rectangle with its 1D projection (either horizontal or vertical) for the interval tree. The `dataclass` with `frozen=True` makes it immutable and hashable.

The `component()` Generator

```python
def component(node):
    todo = set([node])
    while todo:
        node = todo.pop()
        seen.add(node)
        todo |= neighbors[node] - seen
        yield node
```

This is an **iterative DFS** that yields nodes as they're discovered. Using a generator allows the caller to consume components lazily without building all of them in memory at once.

The `todo |= neighbors[node] - seen` line is particularly elegant:

- Add all neighbors of the current node
- But subtract those we've already seen
- This prevents infinite loops and redundant work

Conclusion

The rectangle partitioning algorithm demonstrates how viewing a geometric problem through the lens of graph theory can lead to an elegant and efficient solution. By decomposing 2D rectangle intersection into two 1D overlap problems and leveraging interval trees, we achieve $O(n \log n + k)$ time complexity—much better than the naive $O(n^2)$ approach.

The connection to lattice theory provides additional insight: we're computing equivalence classes under the transitive closure of the meet operation, then computing the join (supremum) for each class.