Multidimensional Divide and Conquer 2 — Spatial Joins

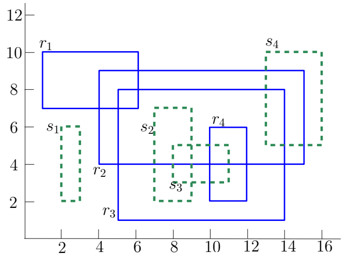
Yufei Tao

ITEE
University of Queensland

Today we will continue our discussion of the divide and conquer method in computational geometry. This lecture will discuss the spatial join, which is another fundamental problem on multidimensional rectangles. Central to our discussion is an output-sensitive technique, which aims to design an algorithm whose cost depends on the size of the result (i.e., how much do we need to output).

Spatial Join

Let R and S be sets of axis-parallel rectangles in $\mathbb{R}^2$. The objective of the spatial join problem is to output all pairs of rectangles $(r, s) \in R \times S$ such that r intersects s.



The result is $\{(r_2, s_2), (r_2, s_3), (r_2, s_4), (r_3, s_2), (r_3, s_3), (r_3, s_4), (r_4, s_3)\}$.

Applications

- "For each hotel, find all the restaurants that are within 5km".
- "Find all the intersection points between railways and roads".
- "Find all pairs of airplanes within a distance (in the 3D space) of 10km at 12pm of 1 Jan 2017".
- Consider a dating website where each man registers his age, height, and salary, and each woman specifies ranges on the age, height, and salary of her ideal significant other. "Find all pairs of (man, woman) such that the man satisfies the requirements of the woman".
- ...

Think: In the first 3 applications, where are the "rectangles"? Hint: Filter refinement.

Naive Solution Already Worst-Case Optimal

One simple algorithm solving the spatial join is simply to inspect all the pairs. Set $n = |S| + |T|$. The running time is $O(n^2)$.

In the worst case, however, every algorithm must incur $\Omega(n^2)$ time because there can be $n^2/4$ pairs to report! This happens when every rectangle in S intersects every rectangle in T.

In other words, $O(n^2)$ time is already asymptotically optimal.

Remedy: Output-Sensitive Algorithms

It would be really disappointing if we had to accept $O(n^2)$ as the best we could do—this complexity is horrible in practice. Fortunately, we do not have to. Notice that the "optimality proof" on the previous slide (although correct) is rather weak: it requires the result to have $\Omega(n^2)$ pairs, which seldom happens in practice. In other words, if we denote by k the number of result pairs, we often have $k \ll n^2$. Can we achieve good efficiency in those scenarios?

The answer is yes. We will learn an algorithm that solves the problem in $O(n \log n + k)$ time. Such a time complexity is output-sensitive by being "elastic" to the output size. It is clearly better than a non-elastic running time of $O(n^2)$—the two are equivalent only when $k = \Omega(n^2)$.

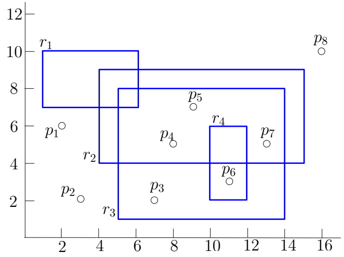
Note that $\Omega(n + k)$ is a trivial lower bound for any algorithm (why?). Therefore, $O(n \log n + k)$ is "nearly" optimal, up to only a factor of $O(\log n)$.

Next, we will explain how to apply divide and conquer to achieve the aforementioned performance guarantees. Our goal is (again) to reduce the dimensionality of the problem. Attention should be paid to two aspects:

- How to divide a problem recursively into two sub-problems—as we will see, this is done in a more sophisticated manner than in the skyline problem;
- The analysis, in particular, how the output-sensitive bound is established.

Rectangles-Join-Points

Let R be a set of axis-parallel rectangles, and P be a set of points, all in $\mathbb{R}^2$. The objective of the rectangles-join-points problem is to output all pairs of $(r, p) \in R \times P$ such that r intersects p.

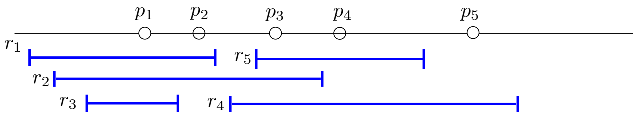


The result is $\{(r_2, p_4), (r_2, p_5), (r_2, p_7), (r_3, p_3), (r_3, p_4), (r_3, p_5), (r_3, p_6), (r_3, p_7), (r_4, p_6)\}$.

This problem cannot be harder than spatial join. An algorithm solving the latter problem efficiently must be able to do so on the former (why)?

1D Rectangles-Join-Points

Let us first consider the rectangles-join-points problem in 1D space. Here, R is a set of intervals, and P a set of points, both in $\mathbb{R}$.

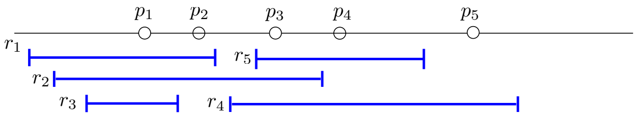


By resorting to the binary search tree, we can easily settle the problem in $O(n \log n + k)$ time (think: how). But this will not be enough for us to solve the spatial join problem fast enough.

It turns out that we can do better if the input sets have been sorted, as explained next.

1D **Sorted** Rectangles-Join-Points

Again, R is a set of intervals, and P a set of points, both in $\mathbb{R}$. The points of P have been sorted. Likewise, the intervals of R have been sorted by left endpoint.



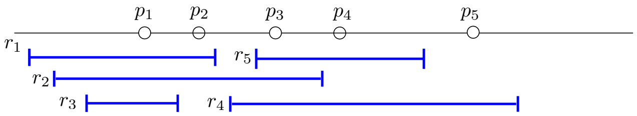
Sorted list on P: p_1, p_2, p_3, p_4, p_5.
Sorted list on R: r_1, r_2, r_3, r_4, r_5.

Combined sorted list: $r_1, r_2, r_3, p_1, p_2, r_4, r_5, p_3, p_4, p_5$.

- Can be obtained in $O(n)$ time from the sorted lists of P and R.

1D **Sorted** Rectangles-Join-Points: Algorithm

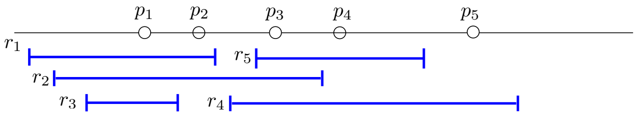
We will process the combined sorted list in ascending order. Whenever we encounter an interval, it is added to a linked list L.



In the above example, after processing r_1, r_2, r_3, we have $L = (r_1, r_2, r_3)$. Let us say that these intervals are alive.

1D **Sorted** Rectangles-Join-Points: Algorithm

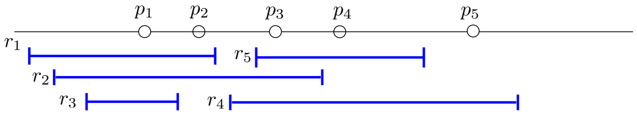
Whenever a point p is encountered, we go through L to check whether the intervals therein contain p. For every such interval r, report (r, p).



Continuing our example, next the algorithm processes point p_1. Since all the intervals in L cover p_1, we report (r_1, p_1), (r_2, p_1), and (r_3, p_1).

1D **Sorted** Rectangles-Join-Points: Algorithm

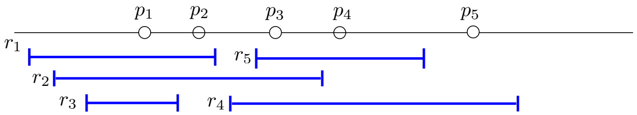
If we find an interval $r \in L$ such that r does not cover p, we delete r from L.



From the previous slide, the algorithm turns to point p_2. Since r_1, r_2 cover p_2, we report (r_1, p_2) and (r_2, p_2). However, r_3 is removed from L, after which $L = (r_1, r_2)$. In other words, r_3 is dead.

Think: Can r_3 cover any more points?

1D **Sorted** Rectangles-Join-Points: Algorithm



After processing r_4 and r_5, we have $L = (r_1, r_2, r_4, r_5)$. The processing of p_3 outputs (r_2, p_3), (r_4, p_3), and (r_5, p_3), but removes r_1 from L, which then becomes (r_2, r_4, r_5).

The rest of the algorithm proceeds in the same manner.

1D **Sorted** Rectangles-Join-Points: Analysis

Let us now analyze the running time of the algorithm.

First, apparently it takes only $O(1)$ time process each interval—all we need to do is to insert it into L.

Hence, the total cost of processing all the intervals is $O(|R|) = O(n)$.

1D **Sorted** Rectangles-Join-Points: Analysis

What is the time of processing the i-th point p_i $(1 \leq i \leq |P|)$?

Clearly it equals $O(|L|)$, namely, the number of alive intervals. The crux of the analysis is to break $|L|$ into two terms:

$$|L| = k_i + n_i^{del}$$

where

- k_i is the number of pairs reported when processing p_i.
- n_i^{del} is the number of intervals removed from L (i.e., the dead intervals).

1D **Sorted** Rectangles-Join-Points: Analysis

Therefore, the total cost of processing all the points is

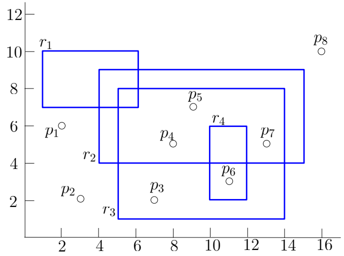
$$\begin{aligned} O\left(\sum_{i=1}^{|P|}(k_i + n_i^{del})\right) &= O\left(\sum_{i=1}^{|P|} k_i + \sum_{i=1}^{|P|} n_i^{del}\right) \\ &= O(k + |R|) \\ &= O(n + k). \end{aligned}$$

Note:

- $\sum_{i=1}^{|P|} k_i = k$ because each pair is reported exactly once.
- $\sum_{i=1}^{|P|} n_i^{del} \leq |R|$ because each interval can be deleted at most once.

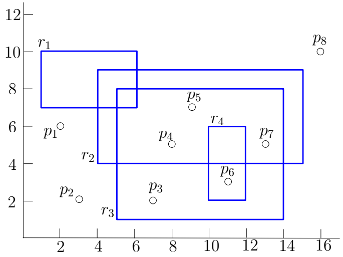
2D Rectangles-Join-Points

Having concluded that 1D sorted rectangles-join-points can be solved in $O(n + k)$ time, we will now apply divide and conquer to attack the 2D problem.



2D Rectangles-Join-Points: Algorithm

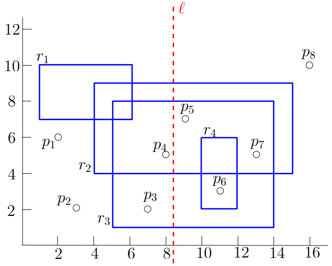
Let X be the set of all x-coordinates in the input sets (i.e., the x-coordinate of a left/right boundary of a rectangle, or the x-coordinate of a point). Call the values in X the raw x-coordinates. In the above example, $X = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16\}$.



To facilitate our discussion, let us assume that all the raw x-coordinates are distinct. Removing the assumption is easy and is left to you (hint: by some tie-breaking).

2D Rectangles-Join-Points: Algorithm

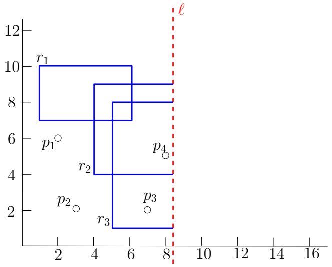
Divide the data space by a vertical line ℓ, such that there are one half of the raw x-coordinates on each side. In the example below, we place ℓ at $x = 8.5$.

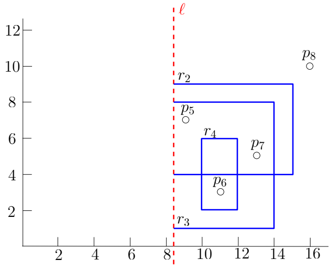


On the left hand side of ℓ, the set X_1 of raw x-coordinates is $\{1, 2, 3, 4, 5, 6, 7, 8\}$. On the right of ℓ, the set X_2 is $\{9, 10, 11, 12, 13, 14, 15, 16\}$.

2D Rectangles-Join-Points: Algorithm

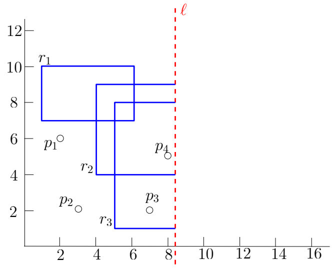
The line divides the problem into two sub-problems:

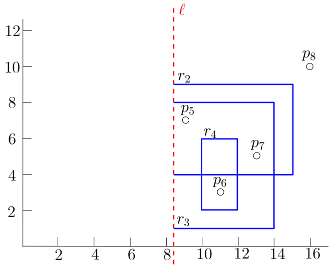




Note that some rectangles appear in both sub-problems: r_2 and r_3. We will then solve (i.e., conquer) each sub-problem. No “result merging” is needed (why?).

2D Rectangles-Join-Points: Algorithm



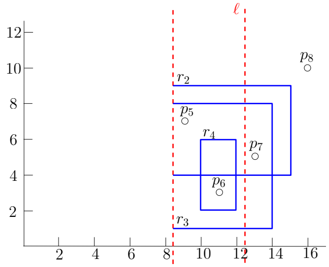


Consider first the left sub-problem, whose "raw x-coordinate" set is $X_1 = \{1, 2, 3, 4, 5, 6, 7, 8\}$. Notice that, $x = 8.5$, is not counted as a raw x-coordinate. Indeed, as far as the left sub-problem is concerned, the right boundaries of r_2 and r_3 can be regarded of being at $x = \infty$.

In this way, we guarantee that we never create new raw x-coordinates in recursively dividing the problem.

2D Rectangles-Join-Points: Algorithm

Consider dividing the right sub-problem into two:

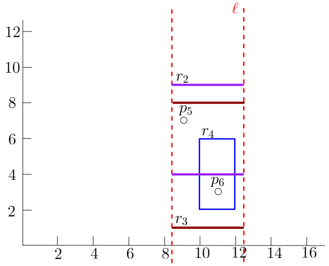


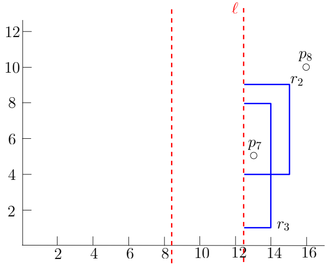
The left sub-sub-problem has raw x-coordinate set $X_{21} = \{9, 10, 11, 12\}$, while the right sub-sub-problem has raw x-coordinate set $X_{22} = \{13, 14, 15, 16\}$.

See the next slide for a clearer illustration of the two sub-sub-problems.

2D Rectangles-Join-Points: Algorithm

From the previous slide, $X_{21} = \{9, 10, 11, 12\}$ and $X_{22} = \{13, 14, 15, 16\}$.



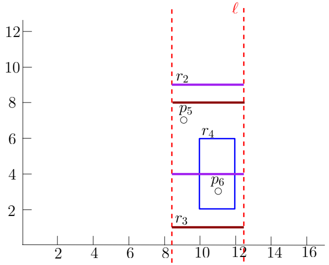


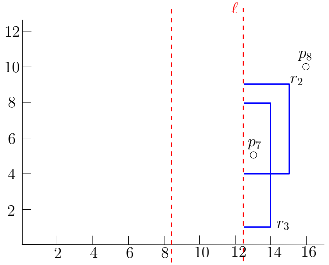
Something interesting happens in the left sub-sub-problem: r_2 and r_3 do not contribute any raw x-coordinates. Notice that their x-ranges "span" the x-range of this sub-sub-problem.

This is great news for us! We can immediately get rid of r_2 and r_3 by finding their result pairs via a reduction to a 1D problem.

2D Rectangles-Join-Points: Algorithm

From the previous slide, $X_{21} = \{9, 10, 11, 12\}$ and $X_{22} = \{13, 14, 15, 16\}$.

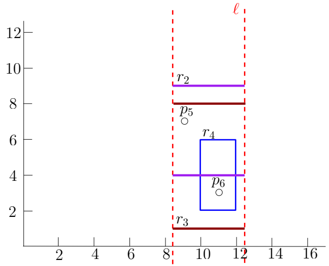


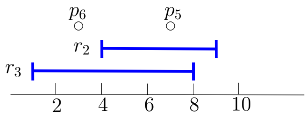


Something interesting happens in the left sub-sub-problem: r_2 and r_3 do not contribute any raw x-coordinates. Notice that their x-ranges "span" the x-range of this sub-sub-problem.

This is great news for us! We can immediately get rid of r_2 and r_3 by finding their result pairs via a reduction to a 1D problem.

2D Rectangles-Join-Points: Algorithm





From the left sub-sub-problem, we construct a 1D rectangles-join-points instance with an interval set $\{r_2, r_3\}$ and a point set $\{p_5, p_6\}$.

r_2 and r_3 are then excluded from the left sub-sub-problem.

2D Rectangles-Join-Points: Algorithm

We can now summarize a principle behind our divide-and-conquer:

> A sub-problem with a raw x-coordinate set X' includes only the rectangles and points that contribute at least one coordinate to X'.
>
> Rectangles that do not contribute to X' are dealt with directly with a 1D instance. They will not be passed further down into sub-sub-problems.

2D Rectangles-Join-Points: Algorithm

The previous discussion points to the following divide-and-conquer algorithm for solving the 2D rectangles-join-points problem (inputs: R and P, with raw x-coordinate set X):

1. Let R_{span} be the set of rectangles that do not contribute to X.
2. Construct a 1D rectangles-join-points instance with R' and P' where R' is the set of intervals obtained by projecting R_{span} onto the y-axis, and P the set of points obtained by projecting P onto the y-axis. Solve the 1D instance.
3. Divide X into two disjoint subsets X_1 and X_2 with the same size (by respecting the ordering) by a vertical line ℓ.
4. Let R_1 (or R_2) be the set of rectangles in R that intersect with the left (or right, resp.) side of ℓ. Let P_1 (or P_2) be the set of points in P that fall on the left (or right, resp.) side of ℓ.
5. Solve the left sub-problem with inputs R_1, P_1 and X_1, and the right sub-problem with inputs R_2, P_2, and X_2.

2D Rectangles-Join-Points: Analysis

Let $f(m)$ be the running time of our algorithm when the raw X-coordinate set has a size of m. It holds that:

$$f(m) \leq 2 \cdot f(m/2) + g(m)$$

where $g(m)$ is the cost of solving a 1D instance of size m.

When $m \leq 2$, obviously $f(m) = O(1)$.

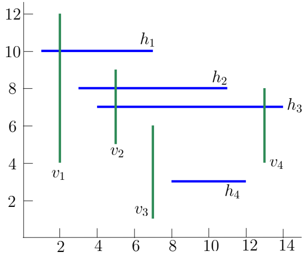
Plugging $g(m) = O(m)$ plus the linear output cost, we obtain $f(m) = O(m \log m + k)$.

Remark 1: Every result pair is reported only once (think: why?).
Remark 2: To ensure $g(m) = O(m)$ plus linear output time, we must ensure that the 1D instances are sorted. How to do so without increasing the time complexity?

The above also implies that our algorithm finishes in $O(n \log n + k)$ time, because $m \leq 2n$.

Segment Join

Let V be a set of vertical segments, and H be a set of horizontal segments, all in $\mathbb{R}^2$. The objective of the segment join problem is to output all pairs of $(v, h) \in V \times H$ such that v intersects h.

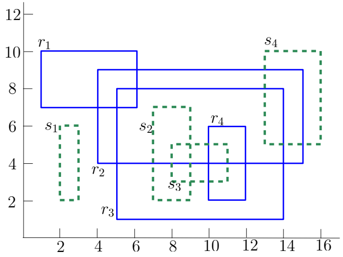


The result is $\{(v_1, h_1), (v_2, h_2), (v_2, h_3), (v_4, h_3)\}$.

This problem can also be solved in $O(n \log n + k)$ time, using essentially the same algorithm. The details are left as an exercise.

Spatial Join

Let R and S be sets of axis-parallel rectangles in $\mathbb{R}^2$. The objective of the spatial join problem is to output all pairs of rectangles $(r, s) \in R \times S$ such that r intersects s.



This problem can be reduced to a rectangles-join-points problem and a segment-join problem. The overall time complexity is $O(n \log n + k)$. The details are left as an exercise.

Spatial Join in d-dimensional Space

Let R and S be sets of axis-parallel rectangles in $\mathbb{R}^d$. The objective of the spatial join problem is to output all pairs of rectangles $(r, s) \in R \times S$ such that r intersects s.

Using divide and conquer, we can solve the problem in $O(n \log^{d-1} n + k)$ time. We will explore the details in an exercise.