Optimal (Multiway) Spatial Joins*

RU WANG and YUFEI TAO, The Chinese University of Hong Kong, China

In a *spatial join*, we are given a constant number $k \geq 2$ of sets — denoted as $R_1, R_2, ..., R_k$ — containing axis-parallel rectangles in a 2D space. The objective is to report all k-tuples $(r_1, r_2, ..., r_k) \in R_1 \times R_2 \times ... \times R_k$ where the rectangles $r_1, r_2, ..., r_k$ have a non-empty intersection, i.e., $r_1 \cap r_2 \cap ... \cap r_k \neq \emptyset$. The problem holds significant importance in spatial databases and has been extensively studied in the database community. In this paper, we show how to settle the problem in $O(n \log n + \text{OUT})$ time — regardless of the constant k — where $n = \sum_{i=1}^{k} |R_i|$ and OUT is the result size (i.e., the total number of k-tuples reported). The runtime is asymptotically optimal in the class of comparison-based algorithms, to which our solution belongs. Previously, the state of the art was an algorithm with running time $O(n \log^{2k-1} n + \text{OUT})$.

CCS Concepts: • **Theory of computation → Design and analysis of algorithms**.

Additional Key Words and Phrases: Multiway Spatial Joins; Computational Geometry; Theory

ACM Reference Format:
Ru Wang and Yufei Tao. 2024. Optimal (Multiway) Spatial Joins. *Proc. ACM Manag. Data* 2, 5 (PODS), Article 210 (November 2024), 25 pages. https://doi.org/10.1145/3695828

1 Introduction

This paper studies the *spatial join* (SJ) problem formulated as follows. Let $k \geq 2$ be a constant integer. In the *k-SJ* problem, the input comprises k sets — denoted as $R_1, R_2, ..., R_k$ — of axis-parallel rectangles[1] in $\mathbb{R}^2$. The goal is to find all k-tuples $(r_1, r_2, ..., r_k)$ where

- $r_i \in R_i$ for each $i \in [1, k]$; and
- $r_1 \cap r_2 \cap ... \cap r_k \neq \emptyset$, namely, the k rectangles $r_1, r_2, ..., r_k$ have a non-empty intersection.

We represent the set of k-tuples described above as $\mathcal{J}(R_1, R_2, ..., R_k)$, referred to as the *join result*. Set $n = \sum_{i=1}^{k} |R_i|$, i.e., the input size, and $\text{OUT} = |\mathcal{J}(R_1, R_2, ..., R_k)|$, i.e., the output size.

SJ is a fundamental operation in spatial databases (SDB), which manage geometric entities such as land parcels, service areas, habitat zones, commercial districts, administrative boundaries, etc. The operation plays a crucial role in implementing the *filter-refinement mechanism*, which is the dominant approach for computing overlay information in an SDB. To explain this mechanism, first note that a geometric entity is typically modeled as a polygon. Determining whether two entities overlap amounts to deciding if two polygons intersect, which can be exceedingly expensive when the polygons have complex boundaries. To mitigate the issue, an SDB stores, for each polygon γ, its *minimum bounding rectangle* (MBR) defined as the smallest axis-parallel rectangle enclosing γ; this way, each set Γ of geometric entities spawns a set R of MBRs. Consider k sets of geometric entities $\Gamma_1, \Gamma_2, ..., \Gamma_k$, and the corresponding sets of MBRs $R_1, R_2, ..., R_k$. To compute overlays from $\Gamma_1, \Gamma_2, ..., \Gamma_k$,

*This work was supported in part by GRF projects 14207820, 14203421, and 14222822 from HKRGC.

[1] A rectangle is *axis-parallel* if it has the form $r = [x_1, x_2] \times [y_1, y_2]$.

Authors' Contact Information: Ru Wang, rwang21@cse.cuhk.edu.hk; Yufei Tao, taoyf@cse.cuhk.edu.hk, The Chinese University of Hong Kong, Hong Kong, Shatin, China.

ACM 2836-6573/2024/11-ART210
https://doi.org/10.1145/3695828

filter-refinement first executes (i) a "filter step", which performs an SJ to obtain $\mathcal{J}(R_1, R_2, ..., R_k)$, and (ii) a "refinement step", which, for each $(r_1, r_2, ..., r_k) \in \mathcal{J}(R_1, R_2, ..., R_k)$, examines if $\gamma_1, \gamma_2, ..., \gamma_k$ indeed have a non-empty intersection, where γ_i ($i \in [1, k]$) is the entity in Γ_i whose MBR is r_i.

Math Conventions. For any integer $x \geq 1$, we use $[x]$ to represent the set $\{1, 2, ..., x\}$. Given $k \geq 2$ sets $S_1, S_2, ..., S_k$ (of arbitrary elements), we often treat a k-tuple $(e_1, e_2, .., e_k)$ in the Cartesian product $S_1 \times S_2 \times ... \times S_k$ as a k-dimensional vector $\boldsymbol{t}$ with $\boldsymbol{t}[i] = e_i$ for each $i \in [k]$. Unless otherwise stated, every mention of the word "rectangle" henceforth will refer to an axis-parallel rectangle in $\mathbb{R}^2$. All logarithms have base 2 by default.

1.1 Previous Results

SJs have been extensively studied in the database-system community, leading to the development of numerous methods that, although lacking strong theoretical guarantees, exhibit good empirical performance in real-world applications. We refer interested readers to [3, 4, 7, 8, 10–15, 18, 19] as entry points into the literature.

From the perspective of theory, SJs are best understood when $k = 2$, i.e., the *pairwise* scenario, where it is folklore that the problem can be solved by a comparison-based algorithm in $O(n \log n + \text{OUT})$ time (e.g., by planesweep [5]). However, the problem becomes much more challenging for $k \geq 3$, known as the *multiway* scenario. All the solutions developed before 2022 (see [7, 13, 14, 18] and the references therein) suffer from a worst-case time complexity of $O(n^k)$, offering essentially no improvement over the naive method that enumerates the entire cartesian product $R_1 \times R_2 \times ... \times R_k$.

Year 2022 witnessed two independent works [9, 21] that, although not tackling k-SJ directly, imply provably fast k-SJ algorithms. Specifically, in [21], Tao and Yi studied several variants of "interval intersection joins" under updates. Most relevant to our context is the variant where the input includes, for each $i \in [k]$, a set $\mathcal{I}_i$ of 1D intervals in $\mathbb{R}$, and the join result comprises all k-tuples $(I_1, I_2, ..., I_k) \in \mathcal{I}_1 \times \mathcal{I}_2 \times ... \times \mathcal{I}_k$ with $\bigcap_{i=1}^{k} I_i \neq \emptyset$. The objective is to design a data structure, which, given the insertion (resp., deletion) of an interval in one of the k sets, can identify all the newly-appearing (resp., disappearing) k-tuples in the join result in $O((1 + \Delta) \cdot \text{polylog}\, n)$ time, where $n = \sum_{i=1}^{k} |\mathcal{I}_i|$ and Δ is the number of such k-tuples. Tao and Yi [21] presented a structure of $O(n \,\text{polylog}\, n)$ space achieving the purpose. Combining their structure with planesweep, one can obtain an algorithm for solving the k-SJ problem in $O((n + \text{OUT}) \cdot \text{polylog}\, n)$ time.

In [9], Khamis et al. investigated a type of joins that extends the conventional equi-join in two ways. First, each attribute value in a relation is an interval (rather than a real value); second, each equality predicate in an equi-join is replaced with a "non-empty intersection" predicate on the attributes involved. The k-SJ problem can be converted to a join under the framework of [9] as defined next. For each $i \in [k]$, define R_i as a relation over two attributes X and Y. For each tuple $\boldsymbol{t} \in R_i$, its values $\boldsymbol{t}(X)$ and $\boldsymbol{t}(Y)$ on the two attributes are both intervals (effectively defining a rectangle). The objective is to output all k-tuples $(\boldsymbol{t}_1, \boldsymbol{t}_2, ..., \boldsymbol{t}_k) \in R_1 \times R_2 \times ... \times R_k$ satisfying $\bigcap_{i=1}^{k} \boldsymbol{t}_i(X) \neq \emptyset$ and $\bigcap_{i=1}^{k} \boldsymbol{t}_i(Y) \neq \emptyset$. It is clear that there is one-one correspondence between the result of this join and that of k-SJ. Khamis et al. [9] developed an algorithm that can process the join in $O(n \log^{2k-1} n + \text{OUT})$ time.

$\Omega(n \log n)$ is a lower bound on the runtime of any comparison-based k-SJ algorithms even for $k = 2$. This can be established via a reduction from the *element distinctness* problem; see [6].

1.2 Our Results

In this paper, we solve the k-SJ problem with a comparison-based algorithm that runs in $O(n \log n + \text{OUT})$ time regardless of the constant k. The time complexity is asymptotically optimal in the class of comparison-based algorithms.

k	method	runtime	remark
2	folklore	$O(n \log n + \text{OUT})$	optimal
≥ 3	before 2022	$O(n^k)$	
≥ 3	[21]	$O((n + \text{OUT}) \cdot \text{polylog}\, n)$	
≥ 3	[9]	$O(n \log^{2k-1} n + \text{OUT})$	
≥ 3	ours	$O(n \log n + \text{OUT})$	optimal

Table 1. Result comparison on k-SJ problem for a constant k

Our primary technical contribution is the revelation of a new property on the problem's mathematical structure. Fix any $k \geq 3$ and an *arbitrary* algorithm $\mathcal{A}$ for the $(k-1)$-SJ problem. Define function $F_{k-1}(n, \text{OUT})$ to return the worst-case running time of $\mathcal{A}$ on any instance of the $(k-1)$-SJ problem having input size at most n and output size at most OUT. We will establish:

Theorem 1.1. *Equipped with the algorithm $\mathcal{A}$ as described above, the k-SJ problem with $k \geq 3$ can be solved in time*

$$O(k^3) \cdot \left(F_{k-1}(n, \text{OUT}) + n \log n + k \cdot \text{OUT}\right) \tag{1}$$

where n (resp., OUT*) is the input (resp., output) size of the problem. Furthermore, if $\mathcal{A}$ is comparison-based, the obtained k-SJ algorithm is also comparison-based.*

The theorem implies a recursive nature of k-SJ. Indeed, we will see that an k-SJ instance with input size n and output size OUT can be converted to $O(k^3)$ instances of the $(k-1)$-SJ problem — all having input size at most n and output size at most OUT — plus an additional cost of $O(k^3) \cdot (n \log n + k \cdot \text{OUT})$. For 2-SJ, we can set $\mathcal{A}$ to the "folklore algorithm" mentioned in Section 1.1, which ensures $F_2(n, \text{OUT}) = O(n \log n + \text{OUT})$. Combining this with (1) gives a recurrence that relates the time complexity of k-SJ to that of $(k-1)$-SJ. Solving the recurrence yields:

Theorem 1.2. *For $k \geq 3$, we can settle k-SJ with a comparison-based algorithm in*

$$O(c^k \cdot (k!)^3 \cdot (n \log n + k \cdot \text{OUT}))$$

time, where $c > 1$ is a positive constant.

When $k = O(1)$, the time complexity becomes $O(n \log n + \text{OUT})$, as promised; the space consumption of our algorithm is $O(n + \text{OUT})$. Now that Theorem 1.2 offers a satisfactory k-SJ result for $k = O(1)$ in 2D space, it is natural to wonder whether the constraint on dimensionality 2 is necessary. Interestingly, the answer is "yes" as far as $k \geq 3$ is concerned, subject to the absence of breakthroughs on a classical problem in graph theory. Specifically, if the 3D version of the 3-SJ problem (which we will formally define in Appendix E) could be solved in $O((n + \text{OUT}) \cdot \text{polylog}\, n)$ time, we would be able to detect the presence of a triangle (i.e., 3-clique) in a graph of m edges in $O(m\, \text{polylog}\, m)$ time, which would make a remarkable breakthrough because the state of the art needs $O(m^{1.41})$ time [2]. This reduction can be inferred from an argument in [9] used to prove a more generic result. We simplify the argument for 3D 3-SJ and present the full reduction in Appendix E.

2 Preliminaries in Geometry

This section will first introduce some definitions and notations to be frequently used in our presentation and then formulate several computational geometry problems, whose solutions will serve as building bricks for our k-SJ algorithm.

Terminology. A *horizontal* segment is a segment of the form $[x_1, x_2] \times y$, and a *vertical* segment is a segment of the form $x \times [y_1, y_2]$. We say that a horizontal segment h_1 is *lower* (resp., *higher*)

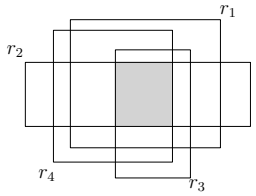


Fig. 1. For 4-tuple $\boldsymbol{t} = \{r_1, r_2, r_3, r_4\}$, $B_{\boldsymbol{t}}$ is the rectangle in gray, left-guard($\boldsymbol{t}$) = r_3 and bot-guard($\boldsymbol{t}$) = r_2.

than another horizontal segment h_2 if the y-coordinate of h_1 is smaller (resp., larger) than that of h_2. Similarly, a vertical segment v_1 is *to the left* (resp., *right*) *of* another vertical segment v_2 if the x-coordinate of v_1 is smaller (resp., larger) than that of v_2.

Given a horizontal segment $h = [x_1, x_2] \times y$, we say that a rectangle r is a *left-end covering rectangle* of h if r contains the left endpoint of h (i.e., $(x_1, y) \in r$). A horizontal/vertical segment s *crosses* a rectangle r if $s \cap r \neq \emptyset$ but r covers neither of the two endpoints of s. A rectangle r *contains* a horizontal/vertical segment s if r covers both endpoints of s.

Let S be a set of segments where either all segments are horizontal or all are vertical. Given a rectangle r, we define

$$\text{cross}_S(r) = \{s \in S \mid s \text{ crosses } r\}; \tag{2}$$

namely, $\text{cross}_S(r)$ is the set of segments in S crossing r. Let R be a set of rectangles. Given a horizontal segment h, we define

$$\text{contain}_R(h) = \{r \in R \mid r \text{ contains } h\}; \tag{3}$$

namely, $\text{contain}_R(h)$ is the set of rectangles in R containing h.

Given a rectangle $r = [x_1, x_2] \times [y_1, y_2]$, we define $\text{left}(r) = x_1$, $\text{right}(r) = x_2$, $\text{bot}(r) = y_1$, and $\text{top}(r) = y_2$. Consider a k-tuple $\boldsymbol{t} = (r_1, r_2, ..., r_k)$ where $k \geq 2$ and each $\boldsymbol{t}[i] = r_i$ ($i \leq [k]$) is a rectangle. We define

$$B_{\boldsymbol{t}} = \bigcap_{i=1}^{k} r_i; \tag{4}$$

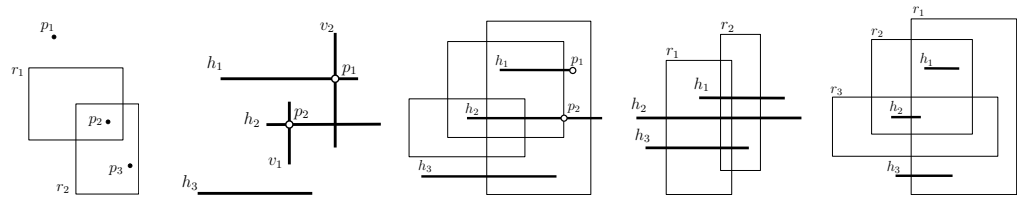
namely, $B_{\boldsymbol{t}}$ is the intersection of the rectangles in $\boldsymbol{t}$ (note: $B_{\boldsymbol{t}}$ is a rectangle itself). Also, if $B_{\boldsymbol{t}}$ is not empty, define:

- left-guard($\boldsymbol{t}$) as the rectangle r_i, $i \in [k]$, satisfying $\text{left}(r_i) = \text{left}(B_{\boldsymbol{t}})$. In case multiple values in $[k]$ fulfill the condition, let i be the smallest of such values.
- bot-guard($\boldsymbol{t}$) as the rectangle r_i, $i \in [k]$, satisfying $\text{bot}(r_i) = \text{bot}(B_{\boldsymbol{t}})$. In case multiple values in $[k]$ fulfill the condition, let i be the smallest of such values.

See Figure 1 for an illustration. It is worth mentioning that since a horizontal segment h is a degenerated rectangle, notations such as $\text{left}(h)$ and $\text{right}(h)$ are well-defined.

Problem $\mathscr{A}$. The input involves a set P of 2D points and set R of rectangles. In the *detection version* of Problem $\mathscr{A}$, the goal is to output, for each point $p \in P$, whether it is covered by at least one rectangle in R. Figure 2a gives an example where $P = \{p_1, p_2, p_3\}$ and $R = \{r_1, r_2\}$; the output is "yes" for p_2 and p_3 and "no" for p_1. The problem can be solved in $O(n \log n)$ time where $n = |P| + |R|$ as shown in Appendix A.

In the *reporting version* of Problem $\mathscr{A}$, the goal is to output, for each point $p \in P$, all the rectangles $r \in R$ containing p; if no such r exists, report nothing for p. In Figure 1a, for instance, the output is $\{(p_2 : r_1, r_2), (p_3 : r_2)\}$. As shown in Appendix A, the problem can be solved in $O(n \log n + \text{OUT})$ time, where OUT is the number of pairs $(p, r) \in P \times R$ such that $p \in r$.



(a) Problem $\mathscr{A}$ (b) Problem $\mathscr{B}$ (c) Problem $\mathscr{C}$ (d) Problem $\mathscr{D}$ (e) Problem $\mathscr{E}$

Fig. 2. Five geometric building brick problems

Problem $\mathscr{B}$. The input involves a set H of horizontal segments and a set V of vertical segments. The goal is to report, for each segment $h \in H$, the leftmost point p on h such that p is on some vertical segment in V. If h does not intersect with any segment in V, report nothing for h. Figure 2b gives an example where $H = \{h_1, h_2, h_3\}$ and $V = \{v_1, v_2\}$; the output is $\{(h_1, p_1), (h_2, p_2)\}$. The problem can be solved in $O(n \log n)$ time where $n = |H| + |V|$, as shown in Appendix A.

Problem $\mathscr{C}$. The input involves a set H of horizontal segments and a set R of rectangles. The goal is to report, for each segment $h \in H$, the rightmost point p on h such that p is covered by at least one left-end covering rectangle of h in R — formally, for $h = [x_1, x_2] \times y$, we aim to find the maximum $x \in [x_1, x_2]$ such that at least one rectangle $r \in R$ covers both the point (x_1, y) and the point (x, y). If the point p exists (i.e., h has at least one left-end covering rectangle in R), we should output a tuple (h, p); otherwise, output nothing for h. Figure 2c gives an example where $H = \{h_1, h_2, h_3\}$ and R includes the three rectangles shown; the output is $\{(h_1, p_1), (h_2, p_2)\}$. The problem can be solved in in $O(n \log n)$ time where $n = |H| + |R|$, as shown in Appendix A.

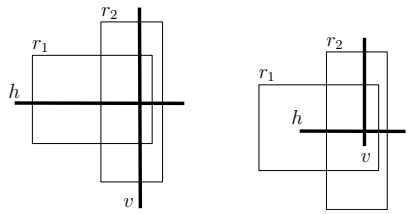
Problem $\mathscr{D}$. The input involves a set H of horizontal segments and a set R of rectangles. In the *find-lowest* version of the problem, the goal is to report, for each rectangle $r \in R$, the lowest segment in $\text{cross}_H(r)$; see (2) for the definition of $\text{cross}_H(r)$. If no segment in H crosses r, output nothing for r. Figure 2d gives an example where $H = \{h_1, h_2, h_3\}$ and $R = \{r_1, r_2\}$; the output is $\{(r_1, h_3), (r_2, h_2)\}$. The problem can be solved in $O(n \log n)$ time where $n = |H| + |R|$, as shown in Appendix A.

In the *find-all-sorted* version of the problem, the goal is to report, for each rectangle $r \in R$, the entire $\text{cross}_H(r)$ sorted by y-coordinate. Formally, if $\text{cross}_H(r) = \{h_1, h_2, ..., h_z\}$ for some $z \geq 1$, we output $(r : h_1, h_2, ..., h_z)$, provided that $y_i \geq y_{i-1}$ for each $i \in [2, z]$ where y_i (resp., y_{i-1}) is the y-coordinate of h_i (resp., h_{i-1}). In the example of Figure 2d, the output is $\{(r_1 : h_3, h_2), (r_2 : h_2, h_1)\}$. In Appendix A, we explain how to solve the problem in $O(n \log n + \text{OUT})$ time where OUT is the number of pairs $(h, r) \in H \times R$ such that h crosses r.

Problem $\mathscr{E}$. The input involves a set H of horizontal segments and a set R of rectangles. The goal is to report, for each segment $h \in H$, the set $\text{contain}_R(h)$ — defined in (3) — where the rectangles are sorted by their right boundaries; if $\text{contain}_R(h)$ is empty, output nothing for h. Formally, if $r_1, r_2, ..., r_z$ for some $z \geq 1$ are all the rectangles in $\text{contain}_R(h)$, we output $(h : r_1, r_2, ..., r_z)$, provided that $\text{right}(r_i) \geq \text{right}(r_{i-1})$ for each $i \in [2, z]$. Figure 2e gives an example where $H = \{h_1, h_2, h_3\}$ and $R = \{r_1, r_2, r_3\}$; the output is $\{(h_1 : r_2, r_1), (h_2 : r_3, r_2)\}$. In Appendix A, we explain how to solve the problem in $O(n \log n + \text{OUT})$ time where $n = |H| + |R|$ and OUT is the number of pairs $(h, r) \in H \times R$ such that r contains h.

3 The Core: H-V Multiway Spatial Joins

Recall that the input of k-SJ comprises k sets of rectangles: $R_1, R_2, ..., R_k$. We now formulate a special version of k-SJ, named the *H-V k-SJ problem*. The special nature is reflected in the introduction of three constraints: (i) $k \geq 3$, (ii) R_{k-1} should be a set of horizontal segments, and (iii) R_k should be



(a) Type 1 (b) Type 2

Fig. 3. Classifying H-V k-SJ result tuples ($k = 4$)

a set of vertical segments. For better clarity, we will represent the input sets as $R_1, R_2, ..., R_{k-2}, H$ ($= R_{k-1}$), and V ($= R_k$). The goal is to output the join result $\mathcal{J}(R_1, ..., R_{k-2}, H, V)$, including every k-tuple $(r_1, ..., r_{k-2}, h, v) \in R_1 \times ... \times R_{k-2} \times H \times V$ such that $h \cap v \cap \bigcap_{i=1}^{k-2} r_i$ is not empty.

Our objective is to prove that H-V k-SJ can be efficiently reduced to $(k-1)$-SJ (note: it is $(k-1)$-SJ here, rather than H-V $(k-1)$-SJ). To ensure the soundness of our notation system, let us formulate the "1-SJ" as the trivial problem where the input is a set R of n rectangles, and the goal is simply to enumerate each rectangle of R; the problem can obviously be "solved" in $O(n)$ time. We assume the existence of an algorithm $\mathcal{A}$ that can settle κ-SJ for all $\kappa \in [1, k-1]$. Denote by $F_\kappa(n, \text{OUT})$ the worst-case runtime of $\mathcal{A}$ on any instance of κ-SJ that has input size n and output size OUT. We consider that $F_\kappa(n, \text{OUT}) \leq F_{\kappa+1}(n, \text{OUT})$ for any $\kappa \geq 1$, that is, its overhead on κ-SJ should not be larger than that on $(\kappa+1)$-SJ.

We will establish:

Lemma 3.1. *Equipped with the algorithm $\mathcal{A}$ described above, the H-V k-SJ problem can be solved in*

$$O(k) \cdot \left(F_{k-1}(n, \text{OUT}) + n \log n + k \cdot \text{OUT}\right)$$

time where n (resp., OUT*) is the input (resp., output) size of the problem. Furthermore, if $\mathcal{A}$ is comparison-based, the H-V k-SJ algorithm obtained is also comparison-based.*

The part of the paper from this point till the end of Section 5 will be devoted to proving the above lemma. This is the most challenging step in solving the general k-SJ problem optimally, as will be discussed in Section 6, where we will prove Theorems 1.1 and 1.2 based on Lemma 3.1.

Consider any k-tuple $(r_1, ..., r_{k-2}, h, v)$ in the join result $\mathcal{J}(R_1, ..., R_{k-2}, H, V)$. We classify the tuple into one of the two types below:

- **Type 1:** h crosses all of $r_1, ..., r_{k-2}$ and, at the same time, v crosses all of $r_1, ..., r_{k-2}$;
- **Type 2:** either h or v fails to cross at least one rectangle in $\{r_1, r_2, ..., r_{k-2}\}$. Equivalently, at least a rectangle r_i (for some $i \in [k-2]$) covers an endpoint of either h or v or both.

Figure 3 illustrates a result tuple of each type, assuming $k = 4$. In Section 4 (resp., 5), we will explain how to produce the result tuples of Type 1 (resp., 2) in the time complexity claimed in Lemma 3.1.

Remark. In [20], Rahul et al. studied the problem of storing a set H of horizontal segments and a set V of vertical segments in a data structure such that, given a query rectangle r, all the pairs $(h, v) \in H \times V$ satisfying $h \cap v \cap r \neq \emptyset$ can be reported efficiently. They gave a structure of $O(n \log n)$ space that can be built in $O(n \log n)$ time and can be used to answer a query in $O(\log n + K)$ time, where $n = |H| + |V|$ and K is the number of pairs reported. Their structure can be utilized to solve H-V 3-SJ in $O(n \log n + \text{OUT})$ time. Oh and Ahn [17] developed a structure for solving a problem more general than that of [20]; however, in the specific scenario of [20], the structure of [17] offers the same guarantees as [20]. We are unaware of a way to extend these solutions to handle H-V k-SJ of $k > 3$. Our method for proving Lemma 3.1 is based on drastically different ideas even for $k = 3$.

4 H-V k-SJ: Result Tuples of Type 1

As before, let $R_1, ..., R_{k-2}, H$, and V be the input sets of the H-V k-SJ problem. Denote by $\mathcal{J}_1$ the set of type-1 result tuples defined in Section 3. In this section, we aim to compute a set $\mathcal{J}^*$ satisfying

$$\mathcal{J}_1 \subseteq \mathcal{J}^* \subseteq \mathcal{J}(R_1, ..., R_{k-2}, H, V) \tag{5}$$

where $\mathcal{J}(R_1, ..., R_{k-2}, H, V)$, let us recall, is the join result of the (whole) H-V k-SJ. Remember that the output size OUT is defined as $|\mathcal{J}(R_1, ..., R_{k-2}, H, V)|$. From $\mathcal{J}^*$, we will report only those k-tuples belonging to $\mathcal{J}_1$ and ignore the rest.

Example 4.1. To illustrate our algorithm, we will utilize the running example in Figure 4a, where $k = 4$, and $R_1 = \{\alpha\}$ (the solid rectangle), $R_2 = \{\beta_1, \beta_2\}$ (the dashed rectangles), $H = \{h_1, h_2, ..., h_6\}$, and $V = \{v_1, v_2, ..., v_5\}$. The set $\mathcal{J}_1$ contains the following tuples: $(\alpha, \beta_2, h_2, v_3)$, $(\alpha, \beta_2, h_2, v_5)$, $(\alpha, \beta_2, h_5, v_3)$, and $(\alpha, \beta_2, h_5, v_5)$. □

Sets $R'_1, R'_2, ..., R'_{k-2}$. Fix any $i \in [k-2]$. For each rectangle $r \in R_i$, we compute four segments:

- $h_\perp$ (resp., $h_\top$): the lowest (resp., highest) segment in H that crosses r;
- $v_\vdash$ (resp., $v_\dashv$): the leftmost (resp., rightmost) segment in V that crosses r.

Define $r' = [x_\vdash, x_\dashv] \times [y_\perp, y_\top]$, where $x_\vdash$ (resp., $x_\dashv$) is the x-coordinate of $v_\vdash$ (resp., $v_\dashv$), and $y_\perp$ (resp., $y_\top$) is the y-coordinate of $h_\perp$ (resp., $h_\top$). We say that r' is the *trimmed rectangle* of r, and conversely, r is the *full rectangle* of r'. Note that r' exists if and only if r is crossed by at least one horizontal segment in H and by at least one vertical segment in V.

Construct

$$R'_i \quad = \quad \{r' \mid r \in R_i \text{ and its trimmed rectangle } r' \text{ exists}\}. \tag{6}$$

Computing the "segment $h_\perp$" for each $r \in R_i$ is an instance of Problem $\mathscr{D}$ (the find-lowest version, with H and R_i as the input). By symmetry, so is computing the $h_\top, v_\vdash$, and $v_\dashv$ segments for each $r \in R_i$. It thus follows from Section 2 that $R'_1, R'_2, ..., R'_{k-2}$ can be produced in $O(kn \log n)$ total time.

We now solve a $(k-2)$-SJ problem on the input $\{R'_1, R'_2, ..., R'_{k-2}\}$ using the algorithm $\mathcal{A}$ supplied (see Lemma 3.1). This $(k-2)$-SJ clearly has an input size at most n, and let us represent its result as $\mathcal{J}(R'_1, R'_2, ..., R'_{k-2})$. We prove in Appendix B:

Lemma 4.1. $|\mathcal{J}(R'_1, R'_2, ..., R'_{k-2})| \leq \text{OUT}$.

As a corollary of Lemma 4.1, the $(k-2)$-SJ can be settled in $F_{k-2}(n, \text{OUT})$ time.

Example 4.2. Figure 4b shows the rectangles in $R'_1 = \{\alpha'\}$ and $R'_2 = \{\beta'_1, \beta'_2\}$. For instance, α', which is trimmed from rectangle α, is decided by $h_\perp = h_2$, $h_\top = h_6$, $v_\vdash = v_3$, and $v_\dashv = v_5$. The $(k-2)$-SJ on R'_1 and R'_2 returns $\mathcal{J}(R'_1, R'_2) = \{(\alpha', \beta'_1), (\alpha', \beta'_2)\}$. □

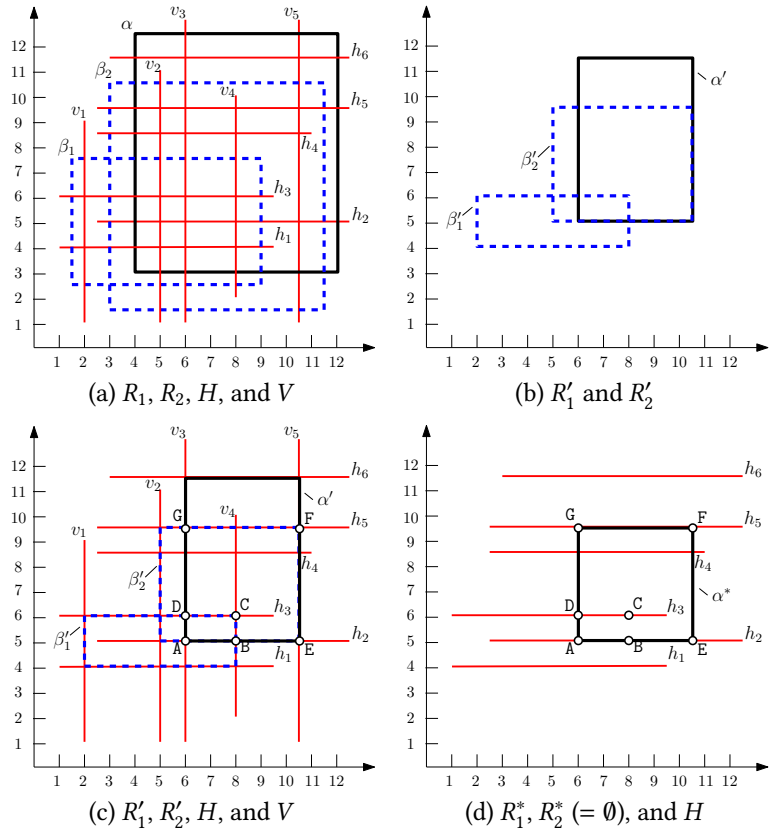
Generating $\mathcal{J}^*$. Take any $(k-2)$-tuple $\boldsymbol{t} = (r'_1, r'_2, ..., r'_{k-2}) \in \mathcal{J}(R'_1, R'_2, ..., R'_{k-2})$. The reader should recall from Section 2 that

- $B_{\boldsymbol{t}}$ is $\bigcap_{i=1}^{k-2} \boldsymbol{t}[i] = \bigcap_{i=1}^{k-2} r'_i$;
- left-guard($\boldsymbol{t}$) is the r'_i $(1 \leq i \leq k-2)$ with $\text{left}(r'_i) = \text{left}(B_{\boldsymbol{t}})$;
- bot-guard($\boldsymbol{t}$) is the r'_i $(1 \leq i \leq k-2)$ with $\text{bot}(r'_i) = \text{bot}(B_{\boldsymbol{t}})$.

We now introduce:

$$\begin{aligned} &\text{d-cross}_H(\boldsymbol{t}) = \{h \in H \mid h \text{ crosses both } B_{\boldsymbol{t}} \text{ and bot-guard}(\boldsymbol{t})\} \\ &\text{d-cross}_V(\boldsymbol{t}) = \{v \in V \mid v \text{ crosses both } B_{\boldsymbol{t}} \text{ and left-guard}(\boldsymbol{t})\}. \end{aligned} \tag{7}$$

The prefix "d-" stands for "double". These sets have important properties as stated in the next lemma, whose proof can be found in Appendix B:



(a) R_1, R_2, H, and V (b) R'_1 and R'_2

(c) R'_1, R'_2, H, and V (d) R^*_1, R^*_2 (= $\emptyset$), and H

Fig. 4. Finding H-V k-SJ result tuples of type 1 ($k = 4$)

Lemma 4.2. *All the following statements are true:*

(1) *Consider any* $(k-2)$*-tuple* $\boldsymbol{t} \in \mathcal{J}(R'_1, R'_2, ..., R'_{k-2})$*. Let* r_i $(i \in [k-2])$ *be the full rectangle of* $\boldsymbol{t}[i]$*. Then, for any* $h \in \text{d-cross}_H(\boldsymbol{t})$ *and any* $v \in \text{d-cross}_V(\boldsymbol{t})$*, the* k*-tuple* $(r_1, r_2, ..., r_{k-2}, h, v)$ *must belong to* $\mathcal{J}(R_1, ..., R_{k-2}, H, V)$.

(2) *Consider any* k*-tuple* $(r_1, r_2, ..., r_{k-2}, h, v) \in \mathcal{J}_1$*. Let* r'_i $(i \in [k-2])$ *be the trimmed rectangle of* r_i*, and set* $\boldsymbol{t} = (r'_1, r'_2, ..., r'_{k-2})$*. Then, we must have*
- $\boldsymbol{t} \in \mathcal{J}(R'_1, R'_2, ..., R'_{k-2})$*;*
- $h \in \text{d-cross}_H(\boldsymbol{t})$ *and* $v \in \text{d-cross}_V(\boldsymbol{t})$.

(3) $\sum_{\boldsymbol{t}} |\text{d-cross}_H(\boldsymbol{t})| \leq \text{OUT}$ *and* $\sum_{\boldsymbol{t}} |\text{d-cross}_V(\boldsymbol{t})| \leq \text{OUT}$*, where the two summations are over all* $\boldsymbol{t} \in \mathcal{J}(R'_1, R'_2, ..., R'_{k-2})$.

Example 4.3. Let us examine, in turn, the two 2-tuples $\boldsymbol{t}_1 = (\alpha', \beta'_1)$ and $\boldsymbol{t}_2 = (\alpha', \beta'_2)$ in $\mathcal{J}(R'_1, R'_2)$. For $\boldsymbol{t}_1$, $B_{\boldsymbol{t}_1}$ is the rectangle ABCD in Figure 4c, and left-guard$(\boldsymbol{t}_1)$ = bot-guard$(\boldsymbol{t}_1) = \alpha'$. Accordingly, d-cross$_H(\boldsymbol{t}_1) = \{h_2\}$ and d-cross$_V(\boldsymbol{t}_1) = \{v_3\}$. For $\boldsymbol{t}_2$, $B_{\boldsymbol{t}_2}$ is the rectangle AEFG, and left-guard$(\boldsymbol{t}_2)$ = bot-guard$(\boldsymbol{t}_2) = \alpha'$. Accordingly, d-cross$_H(\boldsymbol{t}_2) = \{h_2, h_4, h_5\}$ and d-cross$_V(\boldsymbol{t}_2) = \{v_3, v_5\}$. □

Equipped with Lemma 4.2, we generate our target $\mathcal{J}^*$ as follows:

algorithm generate-$\mathcal{J}^*$
1. $\mathcal{J}^* = \emptyset$
2. **for** each $(k-2)$-tuple $\boldsymbol{t} \in \mathcal{J}(R'_1, ..., R'_{k-2})$ **do**
3. $\quad r_i \leftarrow$ the full rectangle of $\boldsymbol{t}[i]$, for each $i \in [k-2]$

4. **for** each $(h, v) \in \text{d-cross}_H(\boldsymbol{t}) \times \text{d-cross}_V(\boldsymbol{t})$ **do**
5. add $(r_1, ..., r_{k-2}, h, v)$ to $\mathcal{J}^*$

By statements (1) and (2) of Lemma 4.2, the set $\mathcal{J}^*$ thus computed indeed satisfies (5). Furthermore, if we are given $\text{d-cross}_H(\boldsymbol{t})$ and $\text{d-cross}_V(\boldsymbol{t})$ for each $\boldsymbol{t}$, the above algorithm runs in $O(1 + k \cdot |\mathcal{J}(R'_1, ..., R'_{k-2})| + k \cdot |\mathcal{J}^*|) = O(1 + k \cdot \text{OUT})$ time, where the derivation used (5) and Lemma 4.1.

The rest of the section will focus on how to prepare the sets $\text{d-cross}_H(\boldsymbol{t})$ of all $\boldsymbol{t} \in \mathcal{J}(R'_1, ..., R'_{k-2})$ in $O(kn \log n + k \cdot \text{OUT})$ time. An analogous method can be used to compute the sets $\text{d-cross}_V(\boldsymbol{t})$ of all $\boldsymbol{t}$ within the same time complexity.

Example 4.4. In our running example, the $\mathcal{J}^*$ computed includes 7 tuples: $(\alpha, \beta_1, h_2, v_3)$, and $\{\alpha\} \times \{\beta_2\} \times \{h_2, h_4, h_5\} \times \{v_3, v_5\}$. All these 7 tuples belong to $\mathcal{J}(R_1, R_2, H, V)$ and include the 4 tuples in $\mathcal{J}_1$ (see Example 4.1). □

Sets $R_1^*, R_2^*, ..., R_{k-2}^*$. Fix any $i \in [k-2]$. Define for each $r' \in R'_i$:

$$\text{maxtop}(r') = \max_{\boldsymbol{t} \in \mathcal{J}(R'_1, ..., R'_{k-2}): \text{bot-guard}(\boldsymbol{t}) = r'} \text{top}(B_{\boldsymbol{t}}). \tag{8}$$

We set $\text{maxtop}(r')$ to $-\infty$ if no $\boldsymbol{t} \in \mathcal{J}(R'_1, ..., R'_{k-2})$ has r' as the $\text{bot-guard}(\boldsymbol{t})$. When $\text{maxtop}(r') \neq -\infty$, assuming $r' = [x_1, x_2] \times [y_1, y_2]$, we introduce a rectangle

$$r^* = [x_1, x_2] \times [y_1, \text{maxtop}(r')]. \tag{9}$$

and call it the *top-sliced rectangle* of r'.

Example 4.5. Recall from Example 4.3 that rectangle α' is both $\text{bot-guard}(\boldsymbol{t}_1)$ and $\text{bot-guard}(\boldsymbol{t}_2)$. Thus, $\text{maxtop}(\alpha') = \max\{\text{top}(B_{\boldsymbol{t}_1}), \text{top}(B_{\boldsymbol{t}_2})\} = \max\{6, 9.5\} = 9.5$. The top-sliced rectangle of α' is the rectangle α^* in Figure 4d. Rectangles β'_1 and β'_2 do not have top-sliced rectangles. □

Next, we construct from R'_i a new set of rectangles:

$$R_i^* = \{r^* \mid r' \in R'_i \text{ and its top-sliced rectangle } r^* \text{ exists}\}. \tag{10}$$

In Appendix B, we show how to compute $R_1^*, ..., R_{k-2}^*$ altogether in $O(n + k \cdot \text{OUT})$ total time.

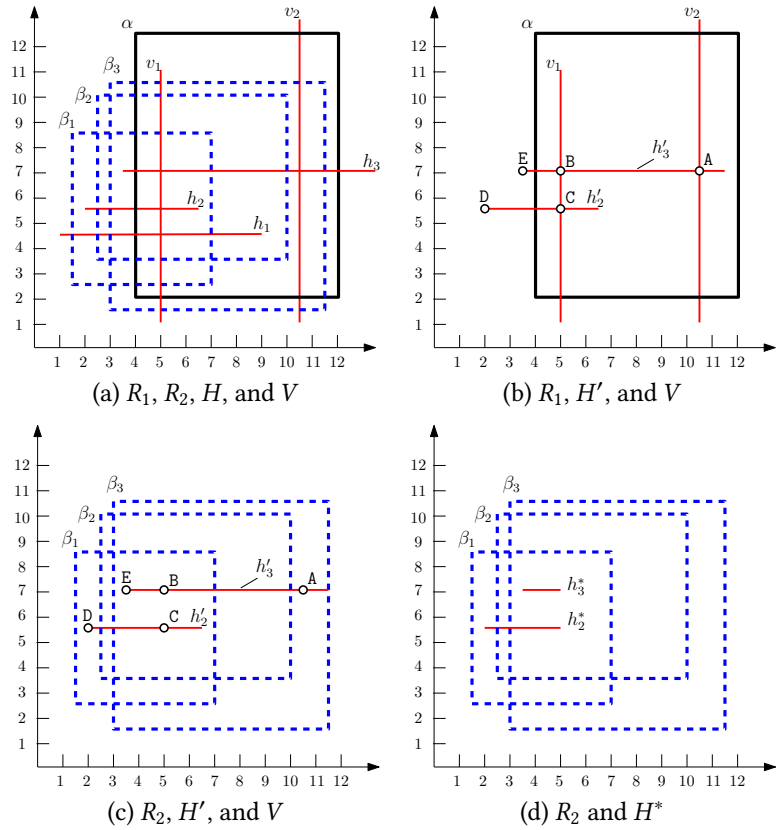
Our interest lies specifically in the sets $\text{cross}_H(r^*)$ of the rectangles r^* in R_i^*, where $\text{cross}_H(r^*)$ — defined in (2) — is the set of segments in H crossing r^*. The following lemma, proven in Appendix B, presents some useful properties of these sets.

Lemma 4.3. *Both statements below are true:*

(1) $\sum_{i=1}^{k-2} \sum_{r^* \in R_i^*} |\text{cross}_H(r^*)| \leq \text{OUT}$.
(2) *Consider any tuple $\boldsymbol{t} \in \mathcal{J}(R'_1, ..., R'_{k-2})$. Let $r' = \text{bot-guard}(\boldsymbol{t})$ and r^* be the top-sliced rectangle of r'. Then, we have $\text{d-cross}_H(\boldsymbol{t}) \subseteq \text{cross}_H(r^*)$. Furthermore, if the (horizontal) segments of $\text{cross}_H(r^*)$ are sorted in ascending order of their y-coordinates, then $\text{d-cross}_H(\boldsymbol{t})$ includes a prefix of the sorted order.*

Example 4.6. It is clear from Figure 4d that $\text{cross}_H(\alpha^*)$ contains h_2, h_4, and h_5, sorted in ascending order of their y-coordinates. Recall that $\text{bot-guard}(\boldsymbol{t}_1) = \text{bot-guard}(\boldsymbol{t}_2) = \alpha'$. Both $\text{d-cross}_H(\boldsymbol{t}_1) = \{h_2\}$ and $\text{d-cross}_H(\boldsymbol{t}_2) = \{h_2, h_4, h_5\}$ are indeed prefixes of the sorted $\text{cross}_H(\alpha^*)$, as stated in Lemma 4.3. □

Finding the $\text{cross}_H(r^*)$ sets of all $r^* \in R_i^*$ is an instance of the find-all-sorted version of Problem $\mathcal{D}$ (with H and R_i^* as the input). Statement (1) of Lemma 4.3, as well as the discussion in Section 2, assures us that the total time to do so for all $R_1^*, ..., R_{k-2}^*$ is bounded by $O(kn \log n + \text{OUT})$. Note that, for each $\text{cross}_H(r^*)$ computed, the (horizontal) segments therein have been sorted in ascending order of y-coordinate.



(a) R_1, R_2, H, and V (b) R_1, H', and V

(c) R_2, H', and V (d) R_2 and H^*

Fig. 5. Finding H-V k-SJ result tuples of type 2 ($k = 4$)

Computing the "d-cross" Sets. We are ready to compute d-cross$_H(\boldsymbol{t})$, defined in (7), for any $\boldsymbol{t} \in \mathcal{J}(R'_1, ..., R'_{k-2})$, thanks to Statement (2) of Lemma 4.3. First, compute $B_{\boldsymbol{t}}$, obtain the rectangle $r' =$ bot-guard$(\boldsymbol{t})$, and fetch the (already computed) top-sliced rectangle r^* of r'; these steps require $O(k)$ time. Then, scan the segments in cross$_H(r^*)$ in ascending order of their y-coordinates. For each segment h scanned, check whether h belongs to d-cross$_H(\boldsymbol{t})$, namely, whether h crosses $B_{\boldsymbol{t}}$ (the reader can verify that h must cross bot-guard$(\boldsymbol{t})$); this can be done in constant time. Abort the scan as soon as $h \notin$ d-cross$_H(\boldsymbol{t})$. This way, we produce d-cross$_H(\boldsymbol{t})$ in $O(k + |\text{d-cross}_H(\boldsymbol{t})|)$ time. Doing so for all $\boldsymbol{t} \in \mathcal{J}(R'_1, ..., R'_{k-2})$ takes $O(k \cdot |\mathcal{J}| + \sum_{\boldsymbol{t}} |\text{d-cross}_H(t)|) = O(k \cdot \text{OUT})$ time, where the derivation used Lemma 4.1 and statement (3) of Lemma 4.2.

We conclude that $\mathcal{J}_1$ — the set of type-1 result tuples — can be computed in $F_{k-2}(n, \text{OUT}) + O(kn \log n + k \cdot \text{OUT})$ time.

5 H-V k-SJ: Result Tuples of Type 2

Still, denote by $R_1, ..., R_{k-2}, H$, and V the input sets of the H-V k-SJ problem. This section will explain how to find the result tuples of Type 2 as defined in Section 3.

As mentioned before, for a result tuple $(r_1, ..., r_{k-2}, h, v)$ of this type, a rectangle r_i, for some $i \in [k-2]$, covers an endpoint of h or v or both. As (i) there are $k-2$ choices for i and (ii) h and v together have four endpoints, we can divide Type 2 further into $4(k-2)$ "sub-types": in subtype 1 (resp., 2), r_1 covers the left (resp., right) endpoint of h, in subtype 3 (resp., 4), r_1 covers the bottom (resp., top) endpoint of v, in subtype 5 (resp., 6), r_2 covers the left (resp., right) endpoint of h, etc. It

is possible for the result tuple to belong to multiple sub-types simultaneously. Next, we will focus on producing the result tuples of a particular sub-type:

$$\mathcal{J}_2 = \{(r_1, ..., r_{k-2}, h, v) \in \mathcal{J}(R_1, ..., R_{k-2}, H, V) \mid r_{k-2} \text{ covers the left endpoint of } h\}. \quad (11)$$

The other sub-types can be found analogously.

A remark is in order about duplicate removal. By finding each sub-type separately, we may see the same result tuple multiple times (precisely, up to $4(k-2)$ times) in the whole algorithm. However, this does not mean that the tuple needs to be reported multiple times. Whenever a type-2 result tuple is found, we can immediately decide in $O(k)$ time all the sub-types it belongs to. To avoid outputting the tuple more than once, we can enforce a policy to designate a specific sub-type for outputting. One such policy is the following: among all sub-types that the tuple belongs to, identify the one with the smallest sub-type number t (an integer from 1 to $4(k-2)$); report the tuple only when we are computing the particular sub-type t.

Example 5.1. To illustrate our algorithm, we will utilize the running example in Figure 5a, where $k = 4$, and $R_1 = \{\alpha\}$, $R_2 = \{\beta_1, \beta_2, \beta_3\}$, $H = \{h_1, h_2, h_3\}$, and $V = \{v_1, v_2\}$. The set $\mathcal{J}_2$ contains the following tuples: $(\alpha, \beta_1, h_2, v_1)$, $(\alpha, \beta_1, h_3, v_1)$, $(\alpha, \beta_2, h_3, v_1)$, $(\alpha, \beta_3, h_3, v_1)$, and $(\alpha, \beta_3, h_3, v_2)$. □

Set H'. Take any horizontal segment $h = [x_1, x_2] \times y \in H$. Recall from Section 2 that a left-end covering rectangle of h is a rectangle covering the left endpoint of h. Let p be the rightmost point on h such that at least one left-end covering rectangle of h in R_{k-2} covers p. This p exists if and only if h has at least one left-end covering rectangle in R_{k-2}. If p exists and has coordinates (x, y), we refer to the segment $h' = [x_1, x] \times y$ as the *trimmed segment* of h; conversely, we call h the *full segment* of h'.

Construct

$$H' = \{h' \mid h \in H \text{ and its trimmed segment } h' \text{ exists}\}. \quad (12)$$

The construction is an instance of Problem $\mathscr{C}$ (with H and R_{k-2} as the input) and finishes in $O(n \log n)$ time based on the discussion in Section 2.

Now, solve a $(k-1)$-SJ problem on the input $\{R_1, ..., R_{k-3}, H', V\}$ using the algorithm $\mathcal{A}$ supplied (by Lemma 3.1). Let $\mathcal{J}(R_1, ..., R_{k-3}, H', V)$ represent the result of this $(k-1)$-SJ, whose input size is at most n. Given the lemma below (which is proved in Appendix C), we assert that $\mathcal{J}(R_1, ..., R_{k-3}, H', V)$ can be computed in $F_{k-1}(n, \text{OUT})$ time.

Lemma 5.1. $|\mathcal{J}(R_1, ..., R_{k-3}, H', V)| \leq \text{OUT}$.

Example 5.2. Segment h_1 has no left-covering rectangle in R_2 (see Figure 5a) and thus has no trimmed segment. Segment h_2 has one left-covering rectangle in R_2, which is β_1. As the entire h_2 is covered by β_1, it is equivalent to its trimmed segment h_2'; see Figure 5b. Segment h_3 has two left-covering rectangles in R_2, which are β_2 and β_3. The right endpoint of its trimmed segment h_3', as shown in Figure 5b, is decided by the right edge of β_3. Therefore, $H' = \{h_2', h_3'\}$. It is clear from Figure 5b that $\mathcal{J}(R_1, H', V)$ has 3 tuples: $\boldsymbol{t}_1 = (\alpha, h_2', v_1)$, $\boldsymbol{t}_2 = (\alpha, h_3', v_1)$, and $\boldsymbol{t}_3 = (\alpha, h_3', v_2)$. □

Generating $\mathcal{J}_2$. Take any $(k-1)$-tuple $\boldsymbol{t} = (r_1, ..., r_{k-3}, h', v) \in \mathcal{J}(R_1, ..., R_{k-3}, H', V)$. Note that $B_{\boldsymbol{t}}$ — defined in (4) — is the point $h' \cap v$ (the intersection of h' and v). Suppose that $h' = [x_1, x_2] \times y$ and $B_{\boldsymbol{t}} = (x, y)$; we define the *effective horizontal segment* of $\boldsymbol{t}$ as the horizontal segment $[x_1, x] \times y$. This allows us to define

$$\text{contain}_{R_{k-2}}(\boldsymbol{t}) = \{r \in R_{k-2} \mid r \text{ contains the effective horizontal segment of } \boldsymbol{t}\} \quad (13)$$

The above should not be confused with (3), where the "contain" function takes a segment as the parameter, rather than a tuple.

Example 5.3. Consider the tuples $\boldsymbol{t}_1$, $\boldsymbol{t}_2$, and $\boldsymbol{t}_3$ of $\mathcal{J}(R_1, H', V)$ given in Example 5.2. For $\boldsymbol{t}_1 = (\alpha, h'_2, v_1)$, its effective horizontal segment is DC (see Figure 5b). For $\boldsymbol{t}_2 = (\alpha, h'_3, v_1)$, its effective horizontal segment is EB. For $\boldsymbol{t}_3 = (\alpha, h'_3, v_2) \in \mathcal{J}(R_1, H', V)$, its effective horizontal segment is EA. Accordingly, as can be seen from Figure 5c, $\text{contain}_{R_{k-2}}(\boldsymbol{t}_1) = \text{contain}_{R_2}(\boldsymbol{t}_1) = \{\beta_1\}$, $\text{contain}_{R_2}(\boldsymbol{t}_2) = \{\beta_1, \beta_2, \beta_3\}$, and $\text{contain}_{R_2}(\boldsymbol{t}_3) = \{\beta_3\}$. □

We prove the next lemma in Appendix C (the reader may want to be reminded that, for each $\boldsymbol{t} \in \mathcal{J}(R_1, ..., R_{k-3}, H', V)$, $\boldsymbol{t}[k-2]$ is a horizontal segment and $\boldsymbol{t}[k-1]$ is a vertical segment).

Lemma 5.2. *All the following statements are true:*

(1) *Consider any $(k-1)$-tuple $\boldsymbol{t} \in \mathcal{J}(R_1, ..., R_{k-3}, H', V)$. Denote by h the full segment of $\boldsymbol{t}[k-2]$. Then, for any $r \in \text{contain}_{R_{k-2}}(\boldsymbol{t})$, the k-tuple $(\boldsymbol{t}[1], ..., \boldsymbol{t}[k-3], r, h, \boldsymbol{t}[k-1])$ belongs to $\mathcal{J}_2$.*
(2) *Consider any k-tuple $(r_1, ..., r_{k-2}, h, v) \in \mathcal{J}_2$. Let h' be the trimmed segment of h and set $\boldsymbol{t} = (r_1, ..., r_{k-3}, h', v)$. Then, $\boldsymbol{t} \in \mathcal{J}(R_1, ..., R_{k-3}, H', V)$ and $r_{k-2} \in \text{contain}_{R_{k-2}}(\boldsymbol{t})$.*
(3) *$\sum_{\boldsymbol{t}} |\text{contain}_{R_{k-2}}(\boldsymbol{t})| \leq \text{OUT}$, where the summation is over all $\boldsymbol{t} \in \mathcal{J}(R_1, ..., R_{k-3}, H', V)$.*

Equipped with Lemma 5.2, we generate our target $\mathcal{J}_2$ as follows:

algorithm generate-$\mathcal{J}_2$
1. $\mathcal{J}_2 = \emptyset$
2. **for** each $(k-2)$-tuple $\boldsymbol{t} \in \mathcal{J}(R_1, ..., R_{k-3}, H', V)$ **do**
3. $\quad h \leftarrow$ the full segment of $\boldsymbol{t}[k-2]$
4. $\quad$ **for** each $r \in \text{contain}_{R_{k-2}}(\boldsymbol{t})$ **do**
5. $\quad\quad$ add $(\boldsymbol{t}[1], ..., \boldsymbol{t}[k-3], r, h, \boldsymbol{t}[k-1])$ to $\mathcal{J}_2$

The correctness of the algorithm follows from statements (1) and (2) of Lemma 5.2. Furthermore, if we are given $\text{contain}_{R_{k-2}}(\boldsymbol{t})$ for each $\boldsymbol{t}$, statement (3) of Lemma 5.2 assures us that the algorithm runs in $O(1 + k \cdot |\mathcal{J}(R_1, ..., R_{k-3}, H', V)| + k \sum_{\boldsymbol{t}} |\text{contain}_{R_{k-2}}(\boldsymbol{t})|) = O(1 + k \cdot \text{OUT})$ time, where the derivation used Lemma 5.1 and statement (3) of Lemma 5.2.

Example 5.4. For $\boldsymbol{t}_1 = (\alpha, h'_2, v_1)$, the full segment of h'_2 is h_2. As β_1 is the only rectangle in $\text{contain}_{R_2}(\boldsymbol{t}_1)$, Line 5 of the algorithm adds tuple $(\alpha, \beta_1, h_2, v_1)$ to $\mathcal{J}_2$. For $\boldsymbol{t}_2 = (\alpha, h'_3, v_1)$, the full segment of h'_3 is h_3. As $\text{contain}_{R_2}(\boldsymbol{t}_2) = \{\beta_1, \beta_2, \beta_3\}$, Line 5 adds $(\alpha, \beta_1, h_3, v_1)$, $(\alpha, \beta_2, h_3, v_1)$, and $(\alpha, \beta_3, h_3, v_1)$ to $\mathcal{J}_2$. Finally, the processing of $\boldsymbol{t}_3 = (\alpha, h'_3, v_2)$ adds $(\alpha, \beta_3, h_3, v_2)$ to $\mathcal{J}_2$. □

Set H^*. For each segment $h' \in H'$, define

$$\text{minleft}(h') = \min_{\substack{\boldsymbol{t} \in \mathcal{J}(R_1, ..., R_{k-3}, H', V): \\ \boldsymbol{t}[k-2] = h'}} x\text{-coordinate of } \boldsymbol{t}[k-1]. \tag{14}$$

We set $\text{minleft}(h')$ to ∞ if no $\boldsymbol{t} \in \mathcal{J}(R_1, ..., R_{k-3}, H', V)$ has h' in its field $\boldsymbol{t}[k-2]$. When $\text{minleft}(h') \neq \infty$, assuming $h' = [x_1, x_2] \times y$, we introduce a horizontal segment

$$h^* \quad = \quad [x_1, \text{minleft}(h')] \times y.$$

and call it the *minimal segment* of h'.

Example 5.5. As mentioned, $\mathcal{J}(R_1, H', V)$ has 3 tuples $\boldsymbol{t}_1$, $\boldsymbol{t}_2$, and $\boldsymbol{t}_3$. Both $\boldsymbol{t}_2 = (\alpha, h'_3, v_1)$ and $\boldsymbol{t}_3 = (\alpha, h'_3, v_2)$ have h'_3 as the horizontal segment. Therefore, $\text{minleft}(h'_3)$ equals 5, which is the smaller between the x-coordinate of v_1 and that of v_2. The minimal segment h^*_3 of h'_3 is shown in Figure 5(c). On the other hand, it is easy to verify that $\text{minleft}(h'_2)$ is the x-coordinate of v_1. The minimal segment h^*_2 of h'_2 is also shown in Figure 5(c). □

Next, we construct a new set of horizontal segments:

$$H^* = \{h^* \mid h' \in H' \text{ and its minimal segment } h^* \text{ exists}\}. \tag{15}$$

This can be done in $O(n + k \cdot \text{OUT})$ time, as shown in Appendix C.

We are interested in the sets $\text{contain}_{R_{k-2}}(h^*)$ of the segments h^* in H^*, where $\text{contain}_{R_{k-2}}(h^*)$ — defined in (3) — is the set of rectangles in R_{k-2} containing h^*. These sets have some useful properties:

Lemma 5.3. *Both statements below are true:*

(1) $\sum_{h^* \in H^*} |\text{contain}_{R_{k-2}}(h^*)| \leq \text{OUT}$.
(2) *Consider any tuple* $\boldsymbol{t} \in \mathcal{J}(R_1, ..., R_{k-3}, H', V)$. *Set* $h' = \boldsymbol{t}[k-2]$ *and let* h^* *be the minimal segment of* h'. *Then,*

$$\text{contain}_{R_{k-2}}(\boldsymbol{t}) \subseteq \text{contain}_{R_{k-2}}(h^*).$$

Furthermore, if the rectangles r *in* $\text{contain}_{R_{k-2}}(h^*)$ *are sorted in descending order of* $\text{right}(r)$, *then* $\text{contain}_{R_{k-2}}(\boldsymbol{t})$ *includes a prefix of the sorted order.*

The proof can be found in Appendix C.

Example 5.6. It is clear from Figure 5(d) that $\text{contain}_{R_2}(h_3^*)$ has rectangles $\beta_3, \beta_2, \beta_1$, sorted in descending order of their right boundaries' x-coordinates. Consider $\boldsymbol{t}_2 = (\alpha, h_3', v_1)$ and $\boldsymbol{t}_3 = (\alpha, h_3', v_2)$. Segment h_3^* is the minimal segment of h_3'. As stated in Lemma 5.3, both $\text{contain}_{R_2}(\boldsymbol{t}_2) = \{\beta_1, \beta_2, \beta_3\}$ and $\text{contain}_{R_2}(\boldsymbol{t}_3) = \{\beta_1\}$ are prefixes of the sorted order of $\text{contain}_{R_2}(h_3^*)$. Regarding h_2^*, it is the minimal segment of h_2', and $\text{contain}_{R_2}(h_2^*)$ contains only β_1. For $\boldsymbol{t}_1 = (\alpha, h_2', v_1)$, $\text{contain}_{R_2}(\boldsymbol{t}_1) = \{\beta_1\}$ is a (trivial) prefix of $\text{contain}_{R_2}(h_2^*)$, as is also consistent with the lemma. □

Finding the $\text{contain}_{R_{k-2}}(h^*)$ sets of all $h^* \in H^*$ is an instance of Problem $\mathscr{E}$ (with H^* and R_{k-2} as the input). The cost is $O(n \log n + \text{OUT})$ according statement (1) of Lemma 4.3 and the discussion in Section 2. Note that, for each $\text{contain}_{R_{k-2}}(h^*)$ computed, the rectangles r therein have been sorted in descending order of $\text{right}(r)$.

Computing the "$\text{contain}_{R_{k-2}}(\boldsymbol{t})$" Sets. Statement (2) of Lemma 5.3 allows us to produce $\text{contain}_{R_{k-2}}(\boldsymbol{t})$ — defined in (13) — for each $\boldsymbol{t} \in \mathcal{J}(R_1, ..., R_{k-3}, H', V)$ as follows. First, fetch the (already computed) minimal segment h^* of $\boldsymbol{t}[k-2]$ in $O(1)$ time. Then, scan the rectangles r of $\text{contain}_{R_{k-2}}(h^*)$ in descending order of $\text{right}(r)$. For each r scanned, check whether $r \in \text{contain}_{R_{k-2}}(\boldsymbol{t})$, or equivalently, whether r covers $B_{\boldsymbol{t}}$ (recall that $B_{\boldsymbol{t}}$ is a point); the cost of this inspection is $O(1)$. Abort the scan as soon as $r \notin \text{contain}_{R_{k-2}}(\boldsymbol{t})$. This way, $\text{contain}_{R_{k-2}}(t)$ can be decided in $O(k + |\text{contain}_{R_{k-2}}(\boldsymbol{t})|)$ time. Doing so for all $\boldsymbol{t} \in \mathcal{J}(R_1, ..., R_{k-3}, H', V)$ takes $O(k \cdot |\mathcal{J}| + \sum_{\boldsymbol{t}} |\text{contain}_{R_{k-2}}(\boldsymbol{t})|) = O(k \cdot \text{OUT})$ time, where the derivation used Lemma 5.1 and statement (3) of Lemma 5.2.

We conclude that $\mathcal{J}_2$ — see (11) — can be computed in $F_{k-1}(n, \text{OUT}) + O(n \log n + k \cdot \text{OUT})$ time. Remember that, to generate the entire type-2 result, we need to repeat the algorithm $4(k-2)$ times (one for each sub-type). The total running time is therefore $O(k) \cdot (F_{k-1}(n, \text{OUT}) + n \log n + k \cdot \text{OUT})$, as claimed in Lemma 3.1.

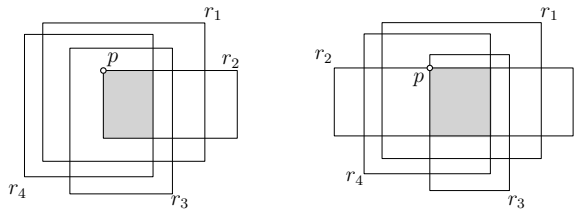
6 Settling k-SJ

This section will tackle the k-SJ problem in its general form, where the input comprises $k \geq 3$ sets of rectangles $R_1, R_2, ..., R_k$. The join result $\mathcal{J}(R_1, R_2, ..., R_k)$ is the set of k-tuples $\boldsymbol{t} = (r_1, r_2, ..., r_k) \in R_1 \times R_2 \times ... \times R_k$ satisfying the condition that $B_{\boldsymbol{t}}$ — which is $\bigcap_{i=1}^{k} r_i \neq \emptyset$ (see (4)) — is non-empty.

Consider any result tuple $\boldsymbol{t} = (r_1, r_2, ..., r_k) \in \mathcal{J}(R_1, R_2, ..., R_k)$, and let p be the top-left corner of $B_{\boldsymbol{t}}$. Depending on how p is determined, we classify $\boldsymbol{t}$ into one of the two categories below:

- **Cat. 1:** p is the top-left corner of r_i for some $i \in [k]$.
- **Cat. 2:** p is not a corner of any of $r_1, ..., r_k$. This means p must be the intersection point between the top edge of some rectangle r_i and the left edge of another rectangle r_j, where $i, j \in [k]$ and $i \neq j$.

Figure 6 illustrates a tuple of each category, assuming $k = 4$.



(a) Category 1 (b) Category 2

Fig. 6. Classifying k-SJ result tuples ($k = 4$)

The rest of this section serves as a proof of Theorem 1.1. Theorem 1.2 is a corollary of Theorem 1.1, as proved in Appendix D. As stated in Theorem 1.1, we are given an algorithm $\mathcal{A}$ that can solve any $(k-1)$-SJ problem in $F_{k-1}(n, \text{OUT})$ time, where n and OUT are the input and output sizes, respectively. Equipped with $\mathcal{A}$, we will show how to find the result tuples of each category within the time complexity of (1).

Category 1. Given an $i \in [k]$, we denote by $\mathcal{J}_i^{\text{cat1}}$ the set of k-tuples $\boldsymbol{t} = (r_1, ..., r_k) \in \mathcal{J}(R_1, ..., R_k)$ such that the top-left corner of $B_{\boldsymbol{t}}$ is the top-left corner of $\boldsymbol{t}[i] = r_i$. We will show how to compute $\mathcal{J}_i^{\text{cat1}}$ for $i = k$; the set $\mathcal{J}_i^{\text{cat1}}$ of every other i can be produced in the same manner.

For every $\boldsymbol{t} = (r_1, r_2, ..., r_k) \in \mathcal{J}_k^{\text{cat1}}$, the top-left corner of r_k must be covered by all of $r_1, ..., r_{k-1}$. This observation motivates us to find $\mathcal{J}_k^{\text{cat1}}$ as follows. First, collect the set P of top-left corners of all the rectangles in R_k. Remove from P every point p with the property that, there exists at least one $j \in [k-1]$ such that p is covered by no rectangle in R_j. This requires solving $k-1$ instances of the detection version of Problem $\mathscr{A}$ (in each instance, the input includes P together with a different R_j, $j \in [k-1]$); the cost is $O(kn \log n)$ by the discussion in Section 2.

Let P' be the set of remaining points in P after the aforementioned removal. Next, for each $j \in [k-1]$, solve the reporting version of Problem $\mathscr{A}$ by feeding P' and R_j as the input. This produces the set $\text{contain}_{R_j}(p)$ for each point $p \in P'$, where $\text{contain}_{R_j}(p)$ is defined in (3) (treating p as a degenerated "horizontal segment") and includes all rectangles of R_j covering p. By the discussion in Section 2, the total cost of this step is bounded by

$$O\Big(kn \log n + \sum_{p \in P'} \sum_{j \in [k-1]} |\text{contain}_{R_j}(p)|\Big). \tag{16}$$

We are ready to generate $\mathcal{J}_k^{\text{cat1}}$. Take any point $p \in P'$, and let $r \in R_k$ be the rectangle with p as the top-left corner. For every $(k-1)$-tuple

$$(r_1, ..., r_{k-1}) \in \text{contain}_{R_1}(p) \times ... \times \text{contain}_{R_{k-1}}(p)$$

we add $(r_1, ..., r_{k-1}, r)$ to $\mathcal{J}_k^{\text{cat1}}$. Performing the above for all $p \in P'$ generates the whole $\mathcal{J}_k^{\text{cat1}}$ in $O(1 + k \cdot |\mathcal{J}_k^{\text{cat1}}|)$ time. The way P' is computed ensures that $\text{contain}_{R_j}(p) \neq \emptyset$ for each $j \in [k-1]$. Hence, $\sum_{j \in [k-1]} |\text{contain}_{R_j}(p)| \leq \prod_{j \in [k-1]} |\text{contain}_{R_j}(p)|$, which implies that (16) is bounded by $O(kn \log n + k \cdot |\mathcal{J}_k^{\text{cat1}}|) = O(kn \log n + k \cdot \text{OUT})$.

Therefore, the total time of computing all of $\mathcal{J}_1^{\text{cat1}}, ..., \mathcal{J}_k^{\text{cat1}}$ is $O(k) \cdot (kn \log n + k \cdot \text{OUT})$. A category-1 result tuple $\boldsymbol{t}$ may be seen more than once (this happens if the top-left corner of $B_{\boldsymbol{t}}$ is the top-left corner of more than one rectangle in $\boldsymbol{t}$). Duplicate removal can be implemented at no extra cost asymptotically, following the ideas explained in Section 5.

Category 2. Given $i, j \in [k]$ with $i \neq j$, we denote by $\mathcal{J}_{i,j}^{\text{cat2}}$ the set of k-tuples $\boldsymbol{t} = (r_1, ..., r_k) \in \mathcal{J}(R_1, ..., R_k)$ such that the top-left corner of $B_{\boldsymbol{t}}$ is the intersection between the top edge of r_i and the left edge of r_j. The Category 2 of result tuples is the union of the $\mathcal{J}_{i,j}^{\text{cat2}}$ of all possible i, j.

The computation of $\mathcal{J}_{i,j}^{\text{cat2}}$ is an instance of the H-V k-SJ problem. Specifically, collect the top-edges of all rectangles of R_i into a set H, and collect the left-edges of all the rectangles of R_j into a set V. This yields an H-V k-SJ instance whose input comprises all the R_z with $z \in [k] \setminus \{i, j\}$, H, and V. Each result tuple consists of a rectangle $r_z \in R_z$, for $z \in [k] \setminus \{i, j\}$, a horizontal segment $h \in H$, and a vertical segment $v \in V$ such that $h \cap v \cap \bigcap_{z \in [k] \setminus \{i,j\}} r_z \neq \emptyset$. There is one-one correspondence between the output of the H-V k-SJ and $\mathcal{J}_{i,j}^{\text{cat2}}$. Thus, by Lemma 3.1, the H-V k-SJ can be solved in $O(k) \cdot (F_{k-1}(n, |\mathcal{J}_{i,j}^{\text{cat2}}|) + n \log n + k \cdot |\mathcal{J}_{i,j}^{\text{cat2}}|)$ time. Converting the output into $\mathcal{J}_{i,j}^{\text{cat2}}$ takes another $O(k \cdot |\mathcal{J}_{i,j}^{\text{cat2}}|)$ time. Applying $|\mathcal{J}_{i,j}^{\text{cat2}}| \leq \text{OUT}$, we know that $\mathcal{J}_{i,j}^{\text{cat2}}$ can be produced in $O(k) \cdot (F_{k-1}(n, \text{OUT}) + n \log n + k \cdot \text{OUT})$ time

Performing the above for all $i, j \in [k]$ with $i \neq j$ leads to a total time complexity of $O(k^3) \cdot (F_{k-1}(n, \text{OUT}) + n \log n + k \cdot \text{OUT})$. A category-2 result tuple $\boldsymbol{t}$ may be seen more than once (this can happen if, for example, more than one rectangle in $\boldsymbol{t}$ has the same top-edge). Again, duplicate removal can be achieved at no extra cost asymptotically.

We now complete the proof of Theorem 1.1.

Appendix

A Building Brick Algorithms

Terminology. Each point (x, y) is said to *define* a y-coordinate y, a horizontal segment $[x_1, x_2] \times y$ is said to *define* a y-coordinate y, and each rectangle $r = [x_1, x_2] \times [y_1, y_2]$ is said to *define* two y-coordinates y_1 and y_2. These definitions permit us to conveniently specify a set of y-coordinates using expressions like "the set of four y-coordinates defined by point p, horizontal segment h, and rectangle r".

Fundamental Data Structures. The *interval tree* [5] stores a set S of intervals in $\mathbb{R}$ using $O(|S|)$ space such that, given any real value q, the intervals of S containing q can be found in $O(\log |S| + K)$ time, where K is the number of intervals reported. It can also be used to detect whether S has at least one interval containing q in $O(\log |S|)$ time. The structure supports insertions and deletions on S in $O(\log |S|)$ time.

Now, let us assume that each interval of S is associated with a real-valued *weight*. Given a real value q, a *stabbing max* query returns the maximum weight of all the intervals in S covering q (if no such intervals exist, the query returns $-\infty$). We can store S in a structure of [1] using $O(|S|)$ space that can answer such a query in $O(\log |S|)$ time. The structure supports insertions and deletions on S in $O(\log |S|)$ amortized time.

The *priority search tree* (PST) [16] stores a set P of points using $O(|P|)$ space such that, given a 3-sided rectangle $q = [x_1, x_2] \times [y, \infty)$, the points of S covered by q can be found in $O(\log |P| + K)$ time, where K is the number of points reported. The structure supports insertions and deletions on S in $O(\log |S|)$ time.

The PST can be deployed to answer queries on intervals. Let S be a set of intervals in $\mathbb{R}$. Given an interval $q = [z_1, z_2]$, a *containment query* reports all the intervals of S that are contained by q. We can store S in a PST of $O(|S|)$ space that solves such a query in $O(\log |S| + K)$ time, where K is the number of intervals reported. To see why, observe that an interval $[x, y]$ is contained by another $[z_1, z_2]$ if and only if the point (x, y) falls in the 3-sided rectangle $[z_1, \infty) \times (-\infty, z_2]$. Thus, we create from S a point set $P = \{(x, y) \mid [x, y] \in S\}$ and store P in a PST. Given an interval $q = [z_1, z_2]$, we can answer the containment query by using the PST to find all the points in P covered by $[z_1, \infty) \times (-\infty, z_2]$ and, for each such point (x, y), report $[x, y]$.

Another closely related query is the *reverse-containment query*, which, given an interval $q = [z_1, z_2]$, finds all the intervals of S that contain q (rather than "being contained by q"). Again, we

can store S in a PST of $O(|S|)$ space that answers such a query in $O(\log |S| + K)$ time, where K is the number of intervals reported. In general, an interval $[x, y]$ contains another $[z_1, z_2]$ if and only if the point (x, y) falls in the 3-sided rectangle $(-\infty, z_1] \times [z_2, \infty)$. Thus, we create from S a point set $P = \{(x, y) \mid [x, y] \in S\}$ and store P in a PST. Given an interval $q = [z_1, z_2]$, we use the PST to find all the points in P covered by $(-\infty, z_1] \times [z_2, \infty)$ and, for each such point (x, y), report $[x, y]$.

Algorithm for Problem $\mathscr{A}$. Consider first the detection version. Sort the set of y-coordinates defined by the points of P and the rectangles of R. Next, sweep (conceptually) a horizontal line ℓ from $y = -\infty$ to $y = \infty$. At all times, maintain the set R_ℓ of rectangles in R that intersect with ℓ. Let S_ℓ be the set of x-ranges of the rectangles in R_ℓ; we store S_ℓ in an interval tree $\mathcal{T}$. Specifically, when ℓ hits the bottom (resp., top) edge of a rectangle $r = [x_1, x_2] \times [y_1, y_2]$ of R, we insert (resp., delete) $[x_1, x_2]$ into (resp., from) $\mathcal{T}$, which can be done in $O(\log n)$ time. When ℓ hits a point $p = (x, y)$ of P, search $\mathcal{T}$ to determine if any interval in S_ℓ contains the value x. If so, point p is covered by at least one rectangle in R; otherwise, it is not. The overall running time is $O(n \log n)$.

The algorithm for the reporting version of Problem $\mathscr{A}$ is similar. The only difference is that, when ℓ hits a point $p = (x, y)$ of P, we use $\mathcal{T}$ to report all the intervals in S_ℓ that contain x; the cost is $O(n \log n + K_p)$, where K_p is the number of such intervals. Every interval corresponds to a rectangle in R that contains p. The total running time is $O(n \log n + \sum_p K_p) = O(n \log n + \text{OUT})$.

Algorithm for Problem $\mathscr{B}$. Sort the set of y-coordinates defined by all the segments of H and V. Next, sweep a horizontal line ℓ from $y = -\infty$ to $y = \infty$. At all times, maintain the set V_ℓ of segments in V that intersect with ℓ. Let S_ℓ be the set of x-coordinates of the segments in V_ℓ; we store S_ℓ in a binary search tree (BST) $\mathcal{T}$. Specifically, when ℓ hits the lower (resp., upper) endpoint of a vertical segment $v = x \times [y_1, y_2]$ of V, we insert (resp., delete) the value x into (resp., from) $\mathcal{T}$, which can be done in $O(\log n)$ amortized time. When ℓ hits a horizontal segment $h = [x_1, x_2] \times y$ of H, search $\mathcal{T}$ to determine the successor x' of x_1 in S_ℓ. If $x' \leq x_2$, then we output a pair (h, p), where p is the point (x', y); otherwise, output nothing for h. The overall running time is $O(n \log n)$.

Algorithm for Problem $\mathscr{C}$. Sort the set of y-coordinates defined by all the segments in H and all the rectangles in R. Next, sweep a horizontal line ℓ from $y = -\infty$ to $y = \infty$. At all times, maintain the set R_ℓ of rectangles in R that intersect with ℓ. Let S_ℓ be the set of x-ranges of the rectangles in R_ℓ; we store S_ℓ in a stabbing-max structure $\mathcal{T}$ of [1]. Specifically, when ℓ hits the bottom (resp., top) edge of a rectangle $r = [x_1, x_2] \times [y_1, y_2]$ of R, we insert (resp., delete) $[x_1, x_2]$ with *weight* x_2 into (resp., from) $\mathcal{T}$, which can be done in $O(\log n)$ time. When ℓ hits a horizontal segment $h = [x_1, x_2] \times y$ of P, search $\mathcal{T}$ to determine the maximum weight w of all the intervals in S_ℓ containing the value x_1. If $w \neq -\infty$, we output (h, p) where the point p is defined in a way depending on w: if $w \leq x_2$, then $p = (w, y)$; otherwise $p = (x_2, y)$. The overall running time is $O(n \log n)$.

Algorithm for Problem $\mathscr{D}$. Consider first the find-lowest version. Sort the set of y-coordinates defined by all the horizontal segments and rectangles. In the outset, all rectangles of R are marked as inactive. During our algorithm, the status of each rectangle will turn from inactive to active at some point, turn from active back to inactive at a later point, and then stay that way forever.

Sweep a horizontal line ℓ from $y = -\infty$ to $y = \infty$. At all times, we maintain the set R_ℓ of *active* rectangles in R that intersect with ℓ. Let S_ℓ be the set of x-ranges of the rectangles in R_ℓ; we store S_ℓ in a PST $\mathcal{T}$. Specifically, when ℓ hits the top edge of a rectangle $r = [x_1, x_2] \times [y_1, y_2]$ of R, we insert $[x_1, x_2]$ into $\mathcal{T}$ and mark r as active, which can be done in $O(\log n)$ time. The rectangle r will be referred to as the *host* of $[x_1, x_2]$ and is stored together with $[x_1, x_2]$ in $\mathcal{T}$.

When ℓ hits a horizontal segment $h = [z_1, z_2] \times y$ of P, perform a containment query on $\mathcal{T}$ to find all the intervals in S_ℓ that are contained by $[z_1, z_2]$; if K_h is the number of such intervals, this retrieval takes $O(\log n + K_h)$ time. For each retrieved interval $[x_1, x_2]$, we also obtain its host rectangle r (stored along with $[x_1, x_2]$ in $\mathcal{T}$). As can be verified shortly, h is the lowest segment in

H that crosses r; we therefore output (r, h). After that, r is marked as inactive, and accordingly, its x-range $[x_1, x_2]$ is deleted from $\mathcal{T}$ in $O(\log n)$ time. As r will remain inactive in the rest of the execution, its x-range will not be retrieved again by another containment query in the future. This implies that h is indeed the lowest segment in H crossing r.

When ℓ hits the bottom edge of a rectangle $r = [x_1, x_2] \times [y_1, y_2]$ of R, we check whether r is active. If so, delete $[x_1, x_2]$ from $\mathcal{T}$ and mark r as inactive; otherwise, do nothing.

Overall, each rectangle of R necessitates one insertion and one deletion in $\mathcal{T}$. All these insertions and deletions take $O(n \log n)$ time in total. Each segment h of H performs a containment query on $\mathcal{T}$, which has a cost of $O(\log n + K_h)$. All these queries demand a total cost of $O(n \log n + \sum_h K_h)$. Recall that the x-range of a rectangle in R can be retrieved by at most one containment query. Hence, $\sum_h K_h \leq |R| \leq n$ and the runtime of our algorithm is $O(n \log n)$.

Next, we consider the find-all-sorted version of Problem $\mathscr{D}$. For each $r \in R$, we keep a linked list, which at the end of our algorithm will store the horizontal segments of $\text{cross}_H(r)$ in ascending order of their y-coordinates. In the outset, all linked lists are empty. Unlike the detection version, we will not need to keep the active status for the rectangles.

Again, sweep a horizontal line ℓ from $y = -\infty$ to $y = \infty$. At all times, we maintain the set R_ℓ of rectangles in R that intersect with ℓ. Let S_ℓ be the set of x-ranges of the rectangles in R_ℓ; we store S_ℓ in a PST $\mathcal{T}$. Specifically, when ℓ hits the bottom (resp., top) edge of a rectangle $r = [x_1, x_2] \times [y_1, y_2]$ of R, we insert (resp., delete) $[x_1, x_2]$ into (resp., from) $\mathcal{T}$, which can be done in $O(\log n)$ time. Again, the rectangle r — the host of $[x_1, x_2]$ — is stored together with $[x_1, x_2]$ in $\mathcal{T}$.

When ℓ hits a horizontal segment $h = [z_1, z_2] \times y$ of P, perform a containment query on $\mathcal{T}$ to find all the intervals in S_ℓ that are contained by $[z_1, z_2]$; the query cost is $O(\log n + K_h)$, where K_h is the number of intervals reported. For each retrieved interval $[x_1, x_2]$, we also obtain its host rectangle r. It is clear that h is a segment crossing r and is thus appended to the linked list of r. Note that h is higher than all the segments already in that linked list.

Overall, each rectangle of R necessitates one insertion and one deletion in $\mathcal{T}$. All these insertions and deletions take $O(n \log n)$ time in total. Each segment h of H performs a containment query on $\mathcal{T}$, which has a cost of $O(\log n + K_h)$. All these queries demand a total cost of $O(n \log n + \sum_h K_h)$. However, unlike the detection version, the sum $\sum_h K_h$ here is equal to the total size of $\text{cross}_H(r)$ for all the $r \in R$. The total size is equivalent to OUT.

Algorithm for Problem $\mathscr{E}$. Sort all the rectangles $r \in R$ in ascending order of right(r) (namely, the x-coordinate of the right edge of r). To each $r \in R$, we assign an ID $i \in [|R|]$ if r is at the i-th position of the sorted list.

Next, we aim to produce, for each pair $(h, r) \in H \times r$ such that h crosses r, a pair (h, λ) where λ is the ID of r. These pairs may be output in an arbitrary order. Sort the set of y-coordinates defined by all the horizontal segments and rectangles. Sweep a horizontal line ℓ from $y = -\infty$ to $y = \infty$. At all times, we maintain the set R_ℓ of rectangles in R that intersect with ℓ. Let S_ℓ be the set of x-ranges of the rectangles in R_ℓ; we store S_ℓ in a PST $\mathcal{T}$. Specifically, when ℓ hits the bottom (resp., top) edge of a rectangle $r = [x_1, x_2] \times [y_1, y_2]$ of R, we insert (resp., delete) $[x_1, x_2]$ into (resp., from) $\mathcal{T}$, which can be done in $O(\log n)$ time. We call the rectangle r the *host* of $[x_1, x_2]$ and store its ID together with $[x_1, x_2]$ in $\mathcal{T}$. When ℓ hits a horizontal segment $h = [z_1, z_2] \times y$ of H, perform a reverse-containment query on $\mathcal{T}$ to find all the intervals in S_ℓ that contain $[z_1, z_2]$; the query cost is $O(\log n + K_h)$, where K_h is the number of intervals reported. For each retrieved interval $[x_1, x_2]$, we also obtain the ID λ of its host rectangle r, and output the pair (h, λ).

Each rectangle of R necessitates one insertion and one deletion in $\mathcal{T}$. All these insertions and deletions take $O(n \log n)$ time in total. Each segment h of H performs a reverse-containment query on $\mathcal{T}$, which has a cost of $O(\log n + K_h)$. All these queries demand a total cost of $O(n \log n + \sum_h K_h)$.

The sum $\sum_h K_h$ here is equal to the total size of $\text{contain}_R(h)$ for all the $h \in H$. The total size is equivalent to OUT. The cost so far is therefore $O(n \log n + \text{OUT})$.

Let L be the list of (h, λ) pairs produced (the size of L is OUT). We now proceed to sort L in ascending order of the λ-field (which is a rectangle ID), breaking ties arbitrarily. Because the IDs are integers created by the algorithm, we are permitted to sort L using *counting sort* without violating the comparison-based requirements. The counting sort finishes in $O(|R| + |L|) = O(n + \text{OUT})$ time, recalling that all the IDs are in $[|R|]$.

Finally, we generate, for each h, its set $\text{contain}_R(h)$ (i.e., the set of rectangles covering h), where the rectangles r are sorted by $\text{right}(r)$. To start with, initialize an empty linked list for every $h \in H$. Inspect the pairs $(h, \lambda) \in L$ in ascending order of the ID-field λ. For each pair (h, λ) examined, identify the rectangle r whose ID is λ and add r to the linked list of h. By the way the rectangle IDs were generated, it is clear that $\text{right}(r)$ is larger than or equal to the x-coordinates of the right edges of the rectangles already in the linked list. The whole scan over L finishes in $O(1 + \text{OUT})$ time, and produces the correct output for Problem $\mathscr{E}$.

B Supplementary Proofs for Section 4

We start by presenting two properties underneath the procedures designed in Section 4. These properties will enable us to construct simpler proofs later.

Proposition B.1. *Consider any rectangle r taken from $R_1, R_2, ..., \text{or } R_{k-2}$. Let r' be the trimmed rectangle of r defined in Section 4.*

- *If a horizontal segment $h \in H$ crosses r, then h must also cross r'.*
- *If a vertical segment $v \in V$ crosses r, then v must also cross r'.*

Proof. We will prove only the first bullet due to symmetry. Let us represent h as $[x_1, x_2] \times y$. The fact of h crossing r indicates that $[x_1, x_2]$ contains the x-range of r. Since the x-range of r contains that of r', we know that $[x_1, x_2]$ must also contain the x-range of r'. To prove that h crosses r', we still need to show $y \in [\text{bot}(r'), \text{top}(r')]$. Recall that $\text{bot}(r')$ is the y-coordinate of the lowest segment in H crossing r. This implies $y \geq \text{bot}(r')$ because h itself is a segment in H crossing r. Analogously, it also holds that $y \leq \text{top}(r')$. We can now conclude that h crosses r'. □

Proposition B.2. *Consider the sets $R'_1, R'_2, ..., R'_{k-2}$ defined in (6). Let $\boldsymbol{t}$ be any $(k-2)$-tuple in $\mathcal{J}(R'_1, R'_2, ..., R'_{k-2})$. Then, neither $\text{d-cross}_H(\boldsymbol{t})$ nor $\text{d-cross}_V(\boldsymbol{t})$ can be empty.*

Proof. Due to symmetry, we will give the proof only for $\text{d-cross}_H(\boldsymbol{t}) \neq \emptyset$. Set $r' = \text{bot-guard}(\boldsymbol{t})$, and let r be the full rectangle of r'. Define h as the lowest segment crossing r (note that h definitely exists because otherwise r has no trimmed rectangle, contradicting the definition of r'). We will show that $h \in \text{d-cross}_H(\boldsymbol{t})$, which indicates $\text{d-cross}_H(\boldsymbol{t}) \neq \emptyset$. For this purpose, we should explain why h crosses both $B_{\boldsymbol{t}}$ and r'. However, by Proposition B.1, h crossing r directly implies h crossing r'. It remains to prove that h crosses $B_{\boldsymbol{t}}$.

By the definitions of r' and h, the bottom edge of r' must be contained in h (note that h is one of the segments used to trim r into r'). Because $r' = \text{bot-guard}(\boldsymbol{t})$, the bottom edge of $B_{\boldsymbol{t}}$ is contained in the bottom edge of r' and thus also contained in h. This means that $h \cap B_{\boldsymbol{t}} \neq \emptyset$. On the other hand, the x-range of h must cover that of r' (because h crosses r'), which in turn must cover that of $B_{\boldsymbol{t}}$ (because r' covers $B_{\boldsymbol{t}}$). Thus, the x-range of h covers that of $B_{\boldsymbol{t}}$. This together with $h \cap B_{\boldsymbol{t}} \neq \emptyset$ tells us that h must cross $B_{\boldsymbol{t}}$. □

We now proceed to elaborate the proofs postponed from Section 4. The order of the subsequent proofs will not strictly follow the sequence in which they are referenced in Section 4. In particular,

we will prove Lemma 4.2 before Lemma 4.1, because the claims of the former lemma can be used to produce a succinct argument for the latter.

Proof of Lemma 4.2. We will prove each statement in turn.

Proof of Statement (1). Take any $(k-2)$-tuple $\boldsymbol{t} \in \mathcal{J}(R'_1, R'_2, ..., R'_{k-2})$. Let r_i ($i \in [k-2]$) be the full rectangle of $\boldsymbol{t}[i]$. Fix any $h \in \text{d-cross}_H(\boldsymbol{t})$ and any $v \in \text{d-cross}_V(\boldsymbol{t})$. The segments h and v both cross $B_{\boldsymbol{t}}$, as can be seen directly from the definitions of $\text{d-cross}_H(\boldsymbol{t})$ and $\text{d-cross}_V(\boldsymbol{t})$. Thus, $h \cap v$ is a point in $B_{\boldsymbol{t}} = \bigcap_{i=1}^{k-2} \boldsymbol{t}[i]$. As r_i ($i \in [k-2]$) is the full rectangle of $\boldsymbol{t}[i]$, we know that $\bigcap_{i=1}^{k-2} r_i$ covers $B_{\boldsymbol{t}} = \bigcap_{i=1}^{k-2} \boldsymbol{t}[i]$. Hence, point $h \cap v$ falls in $\bigcap_{i=1}^{k-2} r_i$, indicating that $h \cap v \cap \bigcap_{i=1}^{k-2} r_i \neq \emptyset$. It follows that $(r_1, ..., r_{k-2}, h, v)$ is a result tuple in $\mathcal{J}(R_1, ..., R_{k-2}, H, V)$.

Proof of Statement (2). Take any k-tuple $(r_1, ..., r_{k-2}, h, v) \in \mathcal{J}_1$. By definition of $\mathcal{J}_1$, segments h and v cross each of the rectangles $r_1, ..., r_{k-2}$. By Proposition B.1, segments h and v must also cross each of the trimmed rectangles $r'_1, ..., r'_{k-2}$. Hence, $B_{\boldsymbol{t}} = \bigcap_{i=1}^{k-2} r'_i$ is non-empty as it contains the point $h \cap v$, which proves the first claim $\boldsymbol{t} = (r'_1, ..., r'_{k-2}) \in \mathcal{J}(R'_1, ..., R'_{k-2})$.

Next, we prove the second claim, i.e., $h \in \text{d-cross}_H(\boldsymbol{t})$ and $v \in \text{d-cross}_V(\boldsymbol{t})$. It suffices to show only the former due to symmetry. For that purpose, we need to argue that h crosses bot-guard($\boldsymbol{t}$) and $B_{\boldsymbol{t}}$. The first part, h crossing bot-guard($\boldsymbol{t}$), is done because as mentioned h crosses each of $r'_1, ..., r'_{k-2}$, and bot-guard($\boldsymbol{t}$) is merely one of those rectangles. To prove that h crosses $B_{\boldsymbol{t}}$, first note that $h \cap B_{\boldsymbol{t}} \neq \emptyset$ because as explained before $B_{\boldsymbol{t}}$ contains $h \cap v$. On the other hand, as bot-guard($\boldsymbol{t}$) covers $B_{\boldsymbol{t}}$, the fact of h crossing bot-guard($\boldsymbol{t}$) indicates that the x-range of h contains that of $B_{\boldsymbol{t}}$. Combining this with $h \cap B_{\boldsymbol{t}} \neq \emptyset$ shows that h crosses $B_{\boldsymbol{t}}$.

Proof of Statement (3). We will prove

$$\sum_{\boldsymbol{t} \in \mathcal{J}(R'_1, ..., R'_{k-2})} |\text{d-cross}_H(\boldsymbol{t})| \cdot |\text{d-cross}_V(\boldsymbol{t})| \leq \text{OUT} \tag{17}$$

which implies statement (3) because, by Proposition B.2, $|\text{d-cross}_H(\boldsymbol{t})| \geq 1$ and $|\text{d-cross}_V(\boldsymbol{t})| \geq 1$ for any $\boldsymbol{t}$ in the summation. Our proof resorts to the algorithm generate-$\mathcal{J}^*$ given in Section 4. This algorithm adds to $\mathcal{J}^*$ as many tuples as calculated by the left hand side of (17). By statement (1) of Lemma 4.2, the $\mathcal{J}^*$ produced must be a subset of $\mathcal{J}(R_1, ..., R_{k-2}, H, V)$, whereas $\text{OUT} = |\mathcal{J}(R_1, ..., R_{k-2}, H, V)|$. This establishes the inequality in (17).

Proof of Lemma 4.1. Our proof again resorts to the algorithm generate-$\mathcal{J}^*$ in Section 4. By Proposition B.2, both $\text{d-cross}_H(\boldsymbol{t})$ and $\text{d-cross}_V(\boldsymbol{t})$ are non-empty for any $\boldsymbol{t} \in \mathcal{J}(R'_1, ..., R'_{k-2})$. Therefore, in processing this $\boldsymbol{t}$, the algorithm adds at least one new tuple to $\mathcal{J}^*$. Hence, $|\mathcal{J}(R'_1, ..., R'_{k-2})| \leq |\mathcal{J}^*| \leq |\mathcal{J}(R_1, ..., R_{k-2}, H, V)| = \text{OUT}$, where the second inequality used Statement (1) of Lemma 4.2.

Computing $R_1^*, R_2^*, ..., R_{k-2}^*$. We consider, w.l.o.g., that each rectangle in the input $R_1 \cup ... \cup R_{k-2}$ is given a distinct integer ID in $[n]$. This allows us to create an array of size n and allocate an array cell to each $r \in R_1 \cup ... \cup R_{k-2}$. The cell can be accessed by the ID of r in constant time.

To compute $R_1^*, R_2^*, ..., R_{k-2}^*$, we start by deriving $\text{maxtop}(r')$ for each rectangle r' in $R'_1 \cup ... \cup R'_{k-2}$. For this purpose, first initialize $\text{maxtop}(r') = -\infty$ for each such r'. Recall that r' is the trimmed rectangle of some rectangle r in $R_1 \cup ... \cup R_{k-2}$. We store $\text{maxtop}(r')$ in the array cell allocated to r. Then, we scan $\mathcal{J}(R'_1, ..., R'_{k-2})$. For each tuple $\boldsymbol{t}$ therein, use $O(k)$ time to identify the rectangle $r' = \text{bot-guard}(\boldsymbol{t})$, and then update in constant time $\text{maxtop}(r')$ to the maximum between its current value and $\text{top}(r')$. The scan requires $O(n + k \cdot \text{OUT})$ time.

Finally, for each $i \in [k-2]$, we construct R_i^* by collecting the top-sliced rectangle (see definition in (9)) of every rectangle $r' \in R'_i$ with $\text{maxtop}(r') \neq -\infty$. This step takes $O(|R'_i|)$ time for each $i \in [k-2]$, or $O(n)$ total time for all i.

Proof of Lemma 4.3. We will prove each statement in turn.

Proof of Statement (1). We will map each r^* of $\bigcup_{i=1}^{k-2} R_i^*$ to a unique tuple $\boldsymbol{t}$ in $\mathcal{J}(R'_1, ..., R'_{k-2})$ satisfying $\text{cross}_H(r^*) \subseteq \text{d-cross}_H(\boldsymbol{t})$. The mapping allows us to derive

$$\sum_{i\in[k-2]} \sum_{r^*\in R_i^*} |\text{cross}_H(r^*)| \leq \sum_{\boldsymbol{t}\in\mathcal{J}(R'_1,...,R'_{k-2})} |\text{d-cross}_H(\boldsymbol{t})| \leq \text{OUT}$$

where the last step used statement (3) of Lemma 4.2.

The mapping is as follows. Consider an arbitrary $r^* \in \bigcup_{i=1}^{k-2} R_i^*$. Recall that r^* is the top-sliced of some rectangle r'. Specifically, if $r' = [x_1, x_2] \times [y_1, y_2]$, then $r^* = [x_1, x_2] \times [y_1, \text{maxtop}(r')]$. By the definition of $\text{maxtop}(r')$ in (9), there exists a tuple $\boldsymbol{t} \in \mathcal{J}(R'_1, ..., R'_{k-2})$ satisfying $\text{bot-guard}(\boldsymbol{t}) = r'$ and $\text{maxtop}(r') = \text{top}(B_{\boldsymbol{t}})$. We map r^* to $\boldsymbol{t}$.

Next, we will prove

Claim 1: If a segment $h \in \text{cross}_H(r^*)$, then $h \in \text{d-cross}_H(\boldsymbol{t})$.

For this purpose, we need to show that h crosses both $\text{bot-guard}(\boldsymbol{t}) = r'$ and $B_{\boldsymbol{t}}$.

- First, h crosses r' follows from the facts that (i) h crosses r^*, (ii) $r^* \subseteq r'$, and (iii) r^* and r' share the same x-range.
- Then, we explain why h crosses $B_{\boldsymbol{t}}$. Because $B_{\boldsymbol{t}} \subseteq r'$ and h crosses r' (just proved), no endpoint of h can fall in $B_{\boldsymbol{t}}$. Let us take the y-coordinate y_h of h. We will prove that the point $p = (\text{left}(B_{\boldsymbol{t}}), y_h)$ is on the segment h and is also in $B_{\boldsymbol{t}}$, suggesting $h \cap B_{\boldsymbol{t}} \neq \emptyset$. Once this is done, we can assert that h crosses $B_{\boldsymbol{t}}$ (as no endpoint of h falls in $B_{\boldsymbol{t}}$).
  - As $\text{bot-guard}(\boldsymbol{t}) = r'$, the bottom edge of $B_{\boldsymbol{t}}$ is contained in r'. Thus, $\text{left}(B_{\boldsymbol{t}}) \in [x_1, x_2]$. The fact of h crossing r^* tells us that $\text{left}(h) < x_1 \leq x_2 < \text{right}(h)$, which leads to $\text{left}(B_{\boldsymbol{t}}) \in [\text{left}(h), \text{right}(h)]$, suggesting that $p \in h$.
  - As $\text{bot-guard}(\boldsymbol{t}) = r'$ and $\text{maxtop}(r') = \text{top}(B_{\boldsymbol{t}})$, the y-range of $B_{\boldsymbol{t}}$ is $[y_1, \text{maxtop}(r')]$. Because $h \cap r^* \neq \emptyset$ (as h crosses r^*) and $[y_1, \text{maxtop}(r')]$ is also the y-range of r^*, we know that $y_h \in [y_1, \text{maxtop}(r')]$, suggesting that $p \in B_{\boldsymbol{t}}$.

It remains to show that no two distinct rectangles $r_1^*, r_2^* \in \bigcup_{i=1}^{k-2} R_i^*$ can be mapped to the same tuple in $\mathcal{J}(R'_1, ..., R'_{k-2})$. Assume, on the contrary, that r_1^* and r_2^* are mapped to the same tuple $\boldsymbol{t} \in \mathcal{J}(R'_1, ..., R'_{k-2})$. Suppose that r_1^* (resp., r_2^*) is the top-sliced rectangle of r'_1 (resp., r'_2). Under our mapping, it must be true that $r'_1 = r'_2 = \text{bot-guard}(B_{\boldsymbol{t}})$. However, the distinctness of r_1^* and r_2^* requires $r'_1 \neq r'_2$, thus giving a contradiction.

Proof of Statement (2). Take any tuple $\boldsymbol{t} \in \mathcal{J}(R'_1, ..., R'_{k-2})$. Set $r' = \text{bot-guard}(\boldsymbol{t})$, and let r^* be the top-sliced rectangle of r'. If we represent r' as $[x_1, x_2] \times [y_1, y_2]$, then r^* can be written as $[x_1, x_2] \times [y_1, \text{maxtop}(r')]$.

We will first prove $\text{d-cross}_H(\boldsymbol{t}) \subseteq \text{cross}_H(r^*)$ or equivalently: if a segment $h \in \text{d-cross}_H(\boldsymbol{t})$, then h crosses r^*. To do so, we need to explain why $\text{left}(h) < x_1 \leq x_2 < \text{right}(h)$ and $y_h \in [y_1, \text{maxtop}(r')]$.

- The fact of $h \in \text{d-cross}_H(\boldsymbol{t})$ tells us that h crosses $\text{bot-guard}(\boldsymbol{t})$, which is r'. As the x-range of r' is $[x_1, x_2]$, it must hold true that $\text{left}(h) < x_1 \leq x_2 < \text{right}(h)$.
- The fact of $h \in \text{d-cross}_H(\boldsymbol{t})$ also tells us that h crosses $B_{\boldsymbol{t}}$. Hence, $y_h \in [\text{bot}(B_{\boldsymbol{t}}), \text{top}(B_{\boldsymbol{t}})]$. From $r' = \text{bot-guard}(\boldsymbol{t})$, we get $\text{bot}(B_{\boldsymbol{t}}) = \text{bot}(r') = y_1$. By the definition of $\text{maxtop}(r')$ in (8), we know $\text{top}(B_{\boldsymbol{t}}) \leq \text{maxtop}(r')$. It thus follows that $y_h \in [y_1, \text{maxtop}(r')]$.

Next, assuming that the segments of $\text{cross}_H(r^*)$ are sorted in ascending order of their y-coordinates, we will prove that $\text{d-cross}_H(\boldsymbol{t})$ includes a prefix of the sorted order. It suffices to establish the following equivalent claim:

Claim 2: If a segment $h \in \text{cross}_H(r^*)$ has y-coordinate $y_h \leq \text{top}(B_t)$, then h must be in $\text{d-cross}_H(\boldsymbol{t})$.

To prove the above, we must explain why h crosses both r' and B_t.

- We first show that h crosses r', or equivalently: $\text{left}(h) < x_1 \leq x_2 < \text{right}(h)$ and $y_h \in [y_1, y_2]$. These conditions hold true because (i) h crosses $r^* = [x_1, x_2] \times [y_1, \text{maxtop}(r')]$, and (ii) the definition of a top-sliced rectangle in (9) tells us $\text{maxtop}(r') \leq y_2$.
- Next we show that h crosses B_t, or equivalently: $\text{left}(h) < \text{left}(B_t) \leq \text{right}(B_t) < \text{right}(h)$ and $y_h \in [\text{bot}(B_t), \text{top}(B_t)] = [y_1, \text{top}(B_t)]$.
  - The fact of h crossing r' tells us $\text{left}(h) < x_1 \leq x_2 < \text{right}(h)$. As $r' = \text{bot-guard}(B_t)$, the x-range of B_t must be contained in $[x_1, x_2]$. Thus, $\text{left}(h) < \text{left}(B_t) \leq \text{right}(B_t) < \text{right}(h)$.
  - The fact of h crossing r^* also tells us that $y_h \in [y_1, \text{maxtop}(r')]$. Moreover, Claim 2 explicitly gives us the condition $y_h \leq \text{top}(B_t)$. It thus follows that $y_h \in [y_1, \text{top}(B_t)]$.

C Supplementary Proofs for Section 5

We start by presenting two properties underneath the procedures designed in Section 5. These properties will enable us to construct simpler proofs later.

Proposition C.1. *Consider any k-tuple $(r_1, ..., r_{k-2}, h, v) \in \mathcal{J}_2$. Define h' as the trimmed segment of h. It holds that $h \cap v = h' \cap v$.*

Proof. Set $p = h \cap v$. As h' is contained in h, for proving $p = h' \cap v$, it suffices to explain why p is on h'. Let us represent h as $[x_1, x_2] \times y$. Accordingly, the segment h' can be written as $[x_1, x_2'] \times y$ for some $x_2' \in [x_1, x_2]$. By the definition of a trimmed segment, (x_2', y) is the rightmost point on h that falls in a left-end covering rectangle of h in R_{k-2}. As $(r_1, ..., r_{k-2}, h, v) \in \mathcal{J}_2$, we know that r_{k-2} is a left-end covering rectangle of h, and r_{k-2} covers p (which is a point on h). Therefore, the x-coordinate of p cannot exceed x_2', allowing us to assert that p is on h'. □

Proposition C.2. *For each tuple $\boldsymbol{t} \in \mathcal{J}(R_1, ..., R_{k-3}, H', V)$ (where H' is defined in (12)), the set $\text{contain}_{R_{k-2}}(\boldsymbol{t})$ is non-empty.*

Proof. Set $h' = \boldsymbol{t}[k-2]$. By the definition of a trimmed segment, there exists a rectangle $r_{k-2} \in R_{k-2}$ covering both endpoints of h'. Thus, the effective horizontal segment of $\boldsymbol{t}$ (defined in Section 5) — which must be contained in h' — must also be covered by r_{k-2}. Hence, $\text{contain}_{R_{k-2}}(\boldsymbol{t})$ contains r_{k-2} and thus cannot be empty. □

We now proceed to elaborate the proofs postponed from Section 5. The order of the proofs here will not strictly follow the sequence in which they are referenced in Section 5. In particular, we will prove Lemma 5.2 before Lemma 5.1, because the claims of the former lemma can be used to construct a simple argument for the latter.

Proof of Lemma 5.2. We will prove each statement in turn.

Proof of Statement (1). Take any tuple $\boldsymbol{t} = (r_1, ..., r_{k-3}, h', v) \in \mathcal{J}(R_1, ..., R_{k-3}, H', V)$. The point $h' \cap v$ is covered by all of $r_1, ..., r_{k-3}$. Let h be the full segment of h'. As h contains h', we have $h \cap v = h' \cap v$.

Consider any rectangle $r_{k-2} \in \text{contain}_{R_{k-2}}(\boldsymbol{t})$. From the definition of $\text{contain}_{R_{k-2}}(\boldsymbol{t})$, we know that r_{k-2} covers the effective horizontal segment of $\boldsymbol{t}$ whose right endpoint is $h' \cap v$. This means that $h \cap v$ falls in r_{k-2}. We now have $h \cap v \cap \bigcap_{i=1}^{k-2} r_i$ equals $h' \cap v$ and hence is non-empty, meaning that $(r_1, ..., r_{k-3}, r_{k-2}, h, v) \in \mathcal{J}(R_1, ..., R_{k-2}, H, V)$.

To show $(r_1, ..., r_{k-3}, r_{k-2}, h, v) \in \mathcal{J}_2$, we still need to explain why r_{k-2} covers the left endpoint of h. This follows immediately from the fact that r_{k-2} covers the effective horizontal segment of h' (which shares the same left endpoint as h).

Proof of Statement (2). Take any k-tuple $(r_1, ..., r_{k-2}, h, v) \in \mathcal{J}_2$. Set $p = h \cap v$, and define h' as the trimmed segment of h. We first argue that the $(k-1)$-tuple $\boldsymbol{t} = (r_1, ..., r_{k-3}, h', v)$ belongs to $\mathcal{J}(R_1, ..., R_{k-3}, H', V)$, or equivalently, the point $h' \cap v$ falls in all of $r_1, ..., r_{k-3}$. By Proposition C.1, $h' \cap v$ is the same point as p. From $(r_1, ..., r_{k-2}, h, v) \in \mathcal{J}(R_1, ..., R_{k-2}, H, V)$, we know that $p = h \cap v$ falls in all of $r_1, ..., r_{k-2}$. It thus follows that $r_1, ..., r_{k-3}$ all cover $h' \cap v$.

Next, we argue that $r_{k-2} \in \text{contain}_{R_{k-2}}(\boldsymbol{t})$. Let p' be the left endpoint of h. Because $h' \cap v = h \cap v = p$, the effective horizontal segment of $\boldsymbol{t}$ is the segment $p'p$. To prove $r_{k-2} \in \text{contain}_{R_{k-2}}(\boldsymbol{t})$, we need to explain why r_{k-2} covers segment $p'p$. This is true because:

- r_{k-2} covers p, as we already know;
- r_{k-2} covers p', as it is a left-end covering rectangle of h.

Proof of Statement (3). Our proof resorts to the algorithm generate-$\mathcal{J}_2$ (see Section 5). This algorithm adds to $\mathcal{J}_2$ exactly

$$\sum_{\boldsymbol{t} \in \mathcal{J}(R_1, ..., R_3, H', V)} |\text{contain}_{R_{k-2}}(\boldsymbol{t})|$$

tuples. By statement (2), the $\mathcal{J}_2$ produced must be a subset of $\mathcal{J}(R_1, ..., R_{k-2}, H, V)$, whose size is OUT. Therefore, the above expression is at most OUT, as claimed in statement (3).

Proof of Lemma 5.1. Again, we resort to the algorithm generate-$\mathcal{J}_2$. For each tuple $\boldsymbol{t} \in \mathcal{J}(R_1, ..., R_{k-3}, H', V)$, we know by Proposition C.2 that $\text{contain}_{R_{k-2}}(\boldsymbol{t}) \neq \emptyset$; thus, the algorithm adds at least one tuple to $\mathcal{J}_2$. Therefore, $|\mathcal{J}(R_1, ..., R_{k-3}, H', V)| \leq |\mathcal{J}_2| \leq \text{OUT}$, where the derivation used the definition of $\mathcal{J}_2$; see (11).

Computing H^*. We assume, w.l.o.g., that each segment in the input H is given a distinct integer ID in $[|H|]$. This allows us to create an array of size $|H| < n$ and allocate an array cell to each $h \in H$. The cell can be accessed by the ID of h in constant time.

To compute H^*, we start by deriving $\text{minleft}(h')$ for each segment h' in H'. For this purpose, first initialize $\text{minleft}(h') = \infty$ for each such h'. Recall that h' is the trimmed segment of some segment h in H. We store $\text{minleft}(h')$ in the array cell allocated to h. Then, we scan $\mathcal{J}(R_1, ..., R_{k-3}, H', V)$. For each tuple $\boldsymbol{t}$ therein, update in constant time $\text{minleft}(h')$ to the minimum between its current value and the x-coordinate of $\boldsymbol{t}[k-1]$. The scan requires $O(n + k \cdot \text{OUT})$ time.

Finally, we construct H^* by collecting the minimal segment (defined in (15)) of every segment $h' \in H'$ with $\text{minleft}(h') \neq \infty$. This step takes $O(|H'|) = O(n)$ time.

Proof of Lemma 5.3. We will prove each statement in turn.

Proof of Statement (1). We will map each segment $h^* \in H^*$ to a unique tuple $\boldsymbol{t} \in \mathcal{J}(R_1, ..., R_{k-3}, H', V)$ satisfying $\text{contain}_{R_{k-2}}(h^*) = \text{contain}_{R_{k-2}}(\boldsymbol{t})$. The mapping allows us to derive

$$\sum_{h^* \in H^*} |\text{contain}_{R_{k-2}}(h^*)| \leq \sum_{\boldsymbol{t} \in \mathcal{J}(R_1, ..., R_{k-3}, H', V)} |\text{contain}_{R_{k-2}}(\boldsymbol{t})| \leq \text{OUT}$$

where the last inequality used Statement (3) of Lemma 5.2.

The mapping is as follows. Consider any $h^* \in H^*$. Recall that h^* is the minimal segment of some segment $h' \in H'$. Specifically, if $h' = [x_1, x_2] \times y$, then $h^* = [x_1, \text{minleft}(h')] \times y$. By the definition of $\text{minleft}(h')$ in (15), there exists a tuple $\boldsymbol{t} \in \mathcal{J}(R_1, ..., R_{k-3}, H', V)$ satisfying $\boldsymbol{t}[k-2] = h'$ and $\text{minleft}(h') = x$-coordinate of the vertical segment $\boldsymbol{t}[k-1]$. We map h^* to $\boldsymbol{t}$.

Next, we will prove

Claim 1: a rectangle $r \in \text{contain}_{R_{k-2}}(h^*)$ if and only if $r \in \text{contain}_{R_{k-2}}(\boldsymbol{t})$.

According to the definitions of $\text{contain}_{R_{k-2}}(h^*)$ and $\text{contain}_{R_{k-2}}(\boldsymbol{t})$ — see (3) and (13), respectively — it suffices to show that h^* is the same as the effective horizontal segment of $\boldsymbol{t}$.

Denote by x the x-coordinate of $\boldsymbol{t}[k-1]$. Then, the point $h' \cap v$ can be written as (x, y), where as mentioned y is the y-coordinate of h'. The effective horizontal segment of $\boldsymbol{t}$ is therefore $[x_1, x] \times y$. Recall that $h^* = [x_1, \text{minleft}(h')] \times y$, where (by definition of $\boldsymbol{t}$) minleft(h') = x-coordinate of $\boldsymbol{t}[k-1] = x$. Therefore, h^* is the same as the effective horizontal segment of $\boldsymbol{t}$, proving claim 1.

It remains to show that no two distinct $h_1^*, h_2^* \in H^*$ can be mapped to the same tuple in $\mathcal{J}(R_1, ..., R_{k-3}, H', V)$. Assume, on the contrary, that h_1^* and h_2^* are both mapped to $\boldsymbol{t} \in \mathcal{J}(R_1, ..., R_{k-3}, H', V)$. Suppose that h_1^* (resp., h_2^*) is the minimal segment of h_1' (resp., h_2'). Under our mapping, it must hold that $h_1' = h_2' = \boldsymbol{t}[k-2]$. However, the distinctness of h_1^* and h_2^* requires $h_1' \neq h_2'$, which yields a contradiction.

Proof of Statement (2). Take any tuple $\boldsymbol{t} \in \mathcal{J}(R_1', ..., R_{k-3}', H', V)$. Set $h' = \boldsymbol{t}[k-2]$, and define h^* as the minimal segment of h'. Let us write h' as $[x_1, x_2] \times y$; accordingly, h^* can be written as $[x_1, \text{minleft}(h')] \times y$. Additionally, let x be the x-coordinate of the vertical segment $\boldsymbol{t}[k-1]$. The effective horizontal segment of $\boldsymbol{t}$ can then be represented as $[x_1, x] \times y$.

We will first prove $\text{contain}_{R_{k-2}}(\boldsymbol{t}) \subseteq \text{contain}_{R_{k-2}}(h^*)$, or equivalently, every rectangle $r_{k-2} \in \text{contain}_{R_{k-2}}(\boldsymbol{t})$ covers h^*. It suffices to show that h^* is contained in the effective horizontal segment of $\boldsymbol{t}$, or equivalently, minleft$(h') \leq x$. By the definition in (14), minleft(h') is the minimum x-coordinate of $\boldsymbol{t'}[k-1]$ among all $\boldsymbol{t'} \in \mathcal{J}(R_1', ..., R_{k-3}', H', V)$ with $\boldsymbol{t'}[k-2] = h'$, Thus, minleft$(h') \leq x$ holds because $\boldsymbol{t}$ is merely one such $\boldsymbol{t'}$.

Next, assuming that the rectangles $r \in \text{contain}_{R_{k-2}}(h^*)$ have been sorted in descending order of right(r), we will prove that $\text{contain}_{R_{k-2}}(\boldsymbol{t})$ includes a prefix of the sorted order. It suffices to establish the following equivalent claim:

> **Claim 2:** If a rectangle $r \in \text{contain}_{R_{k-2}}(h^*)$ satisfies the condition that right$(r) \geq x$ (where x is the x-coordinate of $\boldsymbol{t}[k-1]$), then $r \in \text{contain}_{R_{k-2}}(\boldsymbol{t})$.

To prove the above, we must explain why the rectangle r in the claim contains the effective horizontal segment of $\boldsymbol{t}$. Since $r \in \text{contain}_{R_{k-2}}(h^*)$, we know that r covers the segment $[x_1, \text{minleft}(h')] \times y$. Hence, $[x_1, \text{minleft}(h')] \subseteq [\text{left}(r), \text{right}(r)]$ and $y \in [\text{bot}(r), \text{top}(r)]$. Using also the condition right$(r) \geq x$ given in claim 2, we can derive $[x_1, x] \subseteq [\text{left}(r), \text{right}(r)]$. Therefore, $[x_1, x] \times y$, i.e. the effective horizontal segment of $\boldsymbol{t}$, is contained in r, establishing claim 2.

D Proof of Theorem 1.2

As mentioned in Section 1.1, 2-SJ can be solved in $F_2(n, \text{OUT}) = O(n \log n + \text{OUT})$ time using a comparison-based algorithm. By Theorem 1.1, in general, a comparison-based $(k-1)$-SJ algorithm with runtime $F_{k-1}(n, \text{OUT})$ spawns a comparison-based k-SJ algorithm whose running time $F_k(n, \text{OUT})$ obeys (1). Specifically, for $k \geq 3$, there is a constant $c \geq 2$ such that

$$
\begin{aligned}
& F_k(n, \text{OUT}) \\
\leq\ & c \cdot k^3 \cdot \big(F_{k-1}(n, \text{OUT}) + n \log n + k \cdot \text{OUT}\big) \\
=\ & c^2 \cdot k^3 (k-1)^3 \cdot (F_{k-2}(n, \text{OUT}) + n \log n + k \cdot \text{OUT}) + c \cdot k^3 \cdot (n \log n + k \cdot \text{OUT}) \\
<\ & c^2 \cdot k^3 (k-1)^3 \cdot F_{k-2}(n, \text{OUT}) + (c + c^2) \cdot k^3 (k-1)^3 \cdot (n \log n + k \cdot \text{OUT}) \\
\leq\ & c^3 \cdot k^3 (k-1)^3 (k-2)^3 \cdot F_{k-3}(n, \text{OUT}) + (c + c^2 + c^3) \cdot k^3 (k-1)^3 (k-2)^3 \cdot (n \log n + k \cdot \text{OUT}) \\
\leq \ldots \leq\ & c^{k-2} \cdot (k!)^3 \cdot F_2(n, \text{OUT}) + \Big(\sum_{i=1}^{k-2} c^i \Big) \cdot (k!)^3 \cdot (n \log n + k \cdot \text{OUT}) \\
\leq\ & 2c^{k-1} \cdot (k!)^3 \cdot O(n \log n + k \cdot \text{OUT})
\end{aligned}
$$

which completes the proof of Theorem 1.2.

E Hardness of 3-SJ in 3D Space

An axis-parallel rectangle in 3D space has the form $r = [x_1, x_2] \times [y_1, y_2] \times [z_1, z_2]$. We will refer to $[x_1, x_2]$ as the *x-projection* of r, and define its y- and *z-projections* analogously.

In the 3D 3-SJ problem, the input comprises three sets of axis-parallel rectangles in $\mathbb{R}^3$: R_1, R_2, and R_3. The goal is to report all 3-tuples $(r_1, r_2, r_3) \in R_1 \times R_2 \times R_3$ such that $r_1 \cap r_2 \cap r_3 \neq \emptyset$. Denote by $\mathcal{J}(R_1, R_2, R_3)$ the set of those 3-tuples. Define the input size as $n = |R_1| + |R_2| + |R_3|$ and the output size as $\text{OUT} = |\mathcal{J}(R_1, R_2, R_3)|$.

In the *triangle detection* problem, we are given an undirected graph $G = (V, E)$ and need to determine whether G has a triangle (a.k.a. 3-clique). Set $m = |E|$. We consider that G has no isolated vertices (namely, vertices with degree 0), and therefore $|V| \leq 2m$.

The subsequent discussion will show that if the 3D 3-SJ problem can be solved in $O((n + \text{OUT}) \cdot \text{polylog}\, n)$ time, then the triangle detection problem can be solved in $O(m \,\text{polylog}\, m)$ time. This would be truly surprising because as mentioned in Section 1.2 the state-of-the-art algorithm for triangle detection runs in $O(m^{1.41})$ time [2]. Thus, in the absence of such a breakthrough, no $O((n + \text{OUT}) \cdot \text{polylog}\, n)$ time algorithms can exist for the 3D 3-SJ problem.

Given a graph $G = (V, E)$ for triangle detection, we will construct an instance of the 3D 3-SJ problem with input size $3m$. W.l.o.g., let us assume that each vertex of V is represented as a unique integer in $[|V|]$. Initialize R_1, R_2, and R_3 as 3 empty sets of rectangles. For each edge $\{u, v\} \in E$ where $u < v$, we add

- a rectangle $(-\infty, \infty) \times [u, u] \times [v, v]$ to R_1;
- a rectangle $[v, v] \times (-\infty, \infty) \times [u, u]$ to R_2;
- a rectangle $[v, v] \times [u, u] \times (-\infty, \infty)$ to R_3.

Note that every rectangle in R_1 (resp., R_2 and R_3) has $(-\infty, \infty)$ as the x- (resp., y- and z-) projection.

The construction has the property that G has a triangle if and only if $\mathcal{J}(R_1, R_2, R_3) \neq \emptyset$. We can prove this with the following argument.

- Suppose that G has a triangle with vertices u, v, and w such that $u < v < w$. Then, by our construction, $r_1 = (-\infty, \infty) \times [u, u] \times [v, v] \in R_1$, $r_2 = [w, w] \times (-\infty, \infty) \times [v, v] \in R_2$, and $r_3 = [w, w] \times [u, u] \times (-\infty, \infty) \in r_3$. It is clear that (r_1, r_2, r_3) is a result tuple in $\mathcal{J}(R_1, R_2, R_3)$.
- Conversely, consider that $\mathcal{J}(R_1, R_2, R_3)$ is non-empty. Consider an arbitrary result tuple $(r_1, r_2, r_3) \in \mathcal{J}(R_1, R_2, R_3)$. Assume, w.l.o.g., that $r_1 = (-\infty, \infty) \times [u, u] \times [v, v]$, where $u < v$. Because the z-projection of r_2 must match that of r_1, we assert that r_2 must have the form $[w, w] \times (-\infty, \infty) \times [v, v]$ for some $w > v$. Because the x-projection of r_3 must match that of r_2 and the y-projection of r_3 must match that of r_1, it follows that r_3 must have the form $[w, w] \times [u, u] \times (-\infty, \infty)$. This means that the edges $\{u, v\}$, $\{v, w\}$, and $\{u, w\}$ must all exist in G, and thus form a triangle.

Now, assume that there exists an algorithm $\mathcal{A}$ capable of solving the 3D 3-SJ problem in $O((n + \text{OUT}) \cdot \text{polylog}\, n)$ time. For $\text{OUT} = 0$, this algorithm must perform at most $c \cdot n \,\text{polylog}\, n$ steps when the input size is n. We run $\mathcal{A}$ on the 3D 3-SJ instance R_1, R_2, R_3 constructed earlier in a *cost-monitoring manner*:

- If $\mathcal{A}$ terminates within $c \cdot (3m)\,\text{polylog}(3m)$ steps, we check whether it has output any result tuple in $\mathcal{J}(R_1, R_2, R_3)$. If so, a triangle has been found in G; otherwise, we declare that G has no triangles.
- If $\mathcal{A}$ has performed $1 + c \cdot (3m)\,\text{polylog}(3m)$ steps, we manually terminate the algorithm and declare that $\mathcal{J}(R_1, R_2, R_3) \neq \emptyset$, meaning that G must have at least one triangle.

The above strategy thus settles the triangle detection problem in $O(m \,\text{polylog}\, m)$ time.

References

[1] Pankaj K. Agarwal, Lars Arge, Haim Kaplan, Eyal Molad, Robert Endre Tarjan, and Ke Yi. An optimal dynamic data structure for stabbing-semigroup queries. *SIAM Journal of Computing*, 41(1):104–127, 2012.

[2] Noga Alon, Raphael Yuster, and Uri Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, 1997.

[3] Lars Arge, Octavian Procopiuc, Sridhar Ramaswamy, Torsten Suel, Jan Vahrenhold, and Jeffrey Scott Vitter. A unified approach for indexed and non-indexed spatial joins. In *Proceedings of Extending Database Technology (EDBT)*, volume 1777 of *Lecture Notes in Computer Science*, pages 413–429, 2000.

[4] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. Efficient processing of spatial joins using R-trees. In *Proceedings of ACM Management of Data (SIGMOD)*, pages 237–246, 1993.

[5] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 3rd edition, 2008.

[6] David P. Dobkin and Richard J. Lipton. On the complexity of computations under varying sets of primitives. *Journal of Computer and System Sciences (JCSS)*, 18(1):86–91, 1979.

[7] Himanshu Gupta, Bhupesh Chawda, Sumit Negi, Tanveer A. Faruquie, L. Venkata Subramaniam, and Mukesh K. Mohania. Processing multi-way spatial joins on map-reduce. In *Proceedings of Extending Database Technology (EDBT)*, pages 113–124, 2013.

[8] Edwin H. Jacox and Hanan Samet. Spatial join techniques. *ACM Transactions on Database Systems (TODS)*, 32(1):7, 2007.

[9] Mahmoud Abo Khamis, George Chichirim, Antonia Kormpa, and Dan Olteanu. The complexity of boolean conjunctive queries with intersection joins. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 53–65, 2022.

[10] Nick Koudas and Kenneth C. Sevcik. Size separation spatial join. In *Proceedings of ACM Management of Data (SIGMOD)*, pages 324–335, 1997.

[11] Ming-Ling Lo and Chinya V. Ravishankar. Spatial joins using seeded trees. In *Proceedings of ACM Management of Data (SIGMOD)*, pages 209–220, 1994.

[12] Ming-Ling Lo and Chinya V. Ravishankar. Spatial hash-joins. In *Proceedings of ACM Management of Data (SIGMOD)*, pages 247–258, 1996.

[13] Nikos Mamoulis and Dimitris Papadias. Constraint-based algorithms for computing clique intersection joins. In Robert Laurini, Kia Makki, and Niki Pissinou, editors, *Proceedings of ACM Symposium on Advances in Geographic Information Systems (GIS)*, pages 118–123, 1998.

[14] Nikos Mamoulis and Dimitris Papadias. Multiway spatial joins. *ACM Transactions on Database Systems (TODS)*, 26(4):424–475, 2001.

[15] Nikos Mamoulis and Dimitris Papadias. Slot index spatial join. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 15(1):211–231, 2003.

[16] Edward M. McCreight. Priority search trees. *SIAM Journal of Computing*, 14(2):257–276, 1985.

[17] Eunjin Oh and Hee-Kap Ahn. Finding pairwise intersections of rectangles in a query rectangle. *Comput. Geom.*, 85, 2019.

[18] Dimitris Papadias, Nikos Mamoulis, and Yannis Theodoridis. Processing and optimization of multiway spatial joins using R-trees. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 44–55, 1999.

[19] Jignesh M. Patel and David J. DeWitt. Partition based spatial-merge join. In *Proceedings of ACM Management of Data (SIGMOD)*, pages 259–270, 1996.

[20] Saladi Rahul, Ananda Swarup Das, Krishnan Sundara Rajan, and Kannan Srinathan. Range-aggregate queries involving geometric aggregation operations. In *Proceedings of International Conference on WALCOM: Algorithms and Computation*, volume 6552, pages 122–133, 2011.

[21] Yufei Tao and Ke Yi. Intersection joins under updates. *Journal of Computer and System Sciences (JCSS)*, 124:41–64, 2022.

Received May 2024; accepted August 2024