Reimplementing Raster Intervals: A Detailed Study on Polygon Intersection Joins

Dimitrios Kyriakopoulos[*]
dimitriskpl@di.uoa.gr
National and Kapodistrian University of Athens
Athens, Greece

Dimitros Rontogiannis[*]
dronto@di.uoa.gr
National and Kapodistrian University of Athens
Athens, Greece

Abstract

This paper presents the implementation of the Raster Intervals (RI) technique for efficient polygon intersection joins, as initially proposed by Georgiadis et al. [1]. While following the original algorithm's steps, we introduced two novel algorithms for Minimum Bounding Rectangle (MBR) filtering, specifically designed for dense scenarios with a high number of intersections between polygons. One of these algorithms is particularly effective in dynamic situations where new polygons may appear, allowing for quick identification of non-intersecting polygons to efficiently ignore them. Moreover, we enhanced an approximation algorithm of this step to achieve exact results. Additionally, we implemented alternative clipping algorithm-Weiler-Atherton—for the rasterization process ensuring better handling of complex shapes during polygon rasterization. During our implementation, we also identified an ambiguity in the rasterization process of the original algorithm. We provide a detailed account of our steps, modifications, and the performance achieved.

1 Introduction

The computation of spatial joins, specifically polygon intersection joins, is a fundamental operation in many data science applications, particularly in Geographic Information Systems (GIS). The goal is to identify pairs of polygonal objects that intersect, meaning they share at least one common point. This task is computationally intensive due to the complex nature of polygonal objects and the large volume of data involved.

To tackle this, polygons are often approximated by their Minimum Bounding Rectangles (MBRs) to perform an initial filtering step. This step quickly identifies candidate pairs of objects whose MBRs intersect. However, the refinement step, which verifies the actual intersection of the polygons, remains a significant bottleneck due to its computational expense.

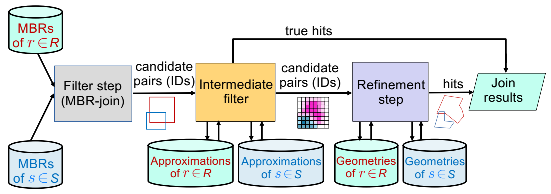
In this paper, we implement the Raster Intervals (RI) technique for polygon intersection joins, as initially proposed by Georgiadis et al [1]. Our contributions include:

- **Implementation of the RI Technique:** We faithfully followed the original algorithm's steps for the RI technique.
- **Novel MBR Filtering Algorithms:** We introduced two new algorithms for MBR filtering in dense scenarios.
- **Approximation step enhancement**: We enhanced an approximation algorithm use in MBR filtering to achieve exact results.
- **Alternative Polygon Rasterization:** We replaced the existing polygon rasterization algorithm with a more accurate one for concave polygons, addressing a gap in the original methodology.
- **Identification of Ambiguities:** During our implementation, we identified an ambiguity in the rasterization process and provided clarification for this step.

This paper is organized as follows: The **Background** section provides an overview of the original RI technique. The **Data and Preprocessing** section describes the datasets and preprocessing steps. Next, we present the two novel MBR filtering algorithms and the approximation enhancement in the **MBR Filtering** section. The **Rasterization** section discusses the rasterization of polygons, followed by the **Hilbert Curve** section which covers the Hilbert curve used for serialization. We then detail the serialization of the polygons in the **Serialization of Polygons** section, followed by a section on the **Raster Intervals Join** process. Then we present the **Refinement** step and our experimental results in the **Experiments** section. Finally, we include a **Build** section and conclude with a summary of our findings in the **Conclusion** section.

2 Background

The original paper by Georgiadis et al. [1] proposes the Raster Intervals (RI) technique to improve the efficiency of polygon intersection joins. This method addresses the computational complexity involved in verifying polygon intersections by introducing a multi-step approach.



Firstly, the polygons are approximated using their Minimum Bounding Rectangles (MBRs) to quickly identify potential intersecting pairs. The RI technique then employs a fine grid to rasterize the polygons, representing groups of nearby intersecting cells as intervals. These intervals are encoded with bitstrings that capture the overlap of each cell with the polygons. The bitstrings of these intervals can then be rapidly ANDed with those of other polygons to efficiently determine intersections.

3 Data and Preprocessing

As Georgiadis et al. [1] did, we used datasets from SpatialHadoop's collection. The first two datasets (T1 and T2) contain landmark and water areas, respectively, from the United States (conterminous states only). Additionally, we used two OpenStreetMap (OSM)

[*]Both authors contributed equally to this research.

datasets (O5 and O6), containing lakes and parks, respectively, from all over the world.

The polygons from each of the two OSM datasets were grouped by continent to create six pairs of datasets: North America (O5NA and O6NA), South America (O5SA and O6SA), Oceania (O5OC and O6OC), Europe (O5EU and O6EU), Asia (O5AS and O6AS), and Africa (O5AF and O6AF). Spatial joins were conducted only between datasets that refer to the same geographic regions.

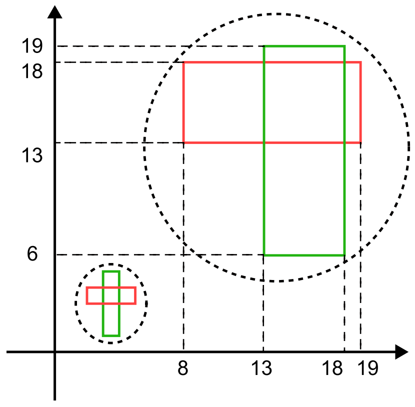


Figure 1: Coordinate Compression

For preprocessing, we discarded multipolygons (geometric objects composed of multiple distinct polygons, which can represent complex shapes with disjoint or overlapping parts) and kept only simple polygons (individual polygons that do not intersect themselves and consist of a single contiguous boundary). Additionally, in our MBR filtering algorithms, we performed coordinate compression (Figure 1) of the MBRs. This was achieved by collecting all coordinates of the corners of the MBRs, sorting these coordinates, and then assigning each coordinate its index in the sorted sequence. This compression helps in reducing the range of coordinates, making the filtering process more efficient.

4 MBR Filtering

The authors utilized the techniques outlined in the paper by Tsitsigkos et al. [4] for their MBR filtering algorithm. This approach involves partitioning the spatial data into smaller subsets using a grid-based strategy, allowing for efficient in-memory parallel processing of spatial joins. The method optimizes performance by balancing the load across partitions and using the Forward Scan based Plane Sweep algorithm to handle the joins within each partition.

We implemented the Forward Scan based Plane Sweep algorithm with uniform partitions and we developed two novel algorithms for MBR filtering. The specific details of these novel algorithms are presented in the following sections. Moreover, we enhanced an approximation step to find the exact result.

4.1 Forward Scan based Plane Sweep

For the Forward Scan based Plane Sweep algorithm we performed a simple uniform partition (Figure 2) of the compressed space by partitioning the X axis and Y axis into P intervals, where P ranged from 3 to 10, and selected the best P based on performance evaluations. As illustrated in the figure, instead of computing intersections between 13 rectangles, we now have 9 regions where we need to compute intersections between 2, 3, 1, 1, 3, 2, 2, 1, and 1 rectangles, respectively.

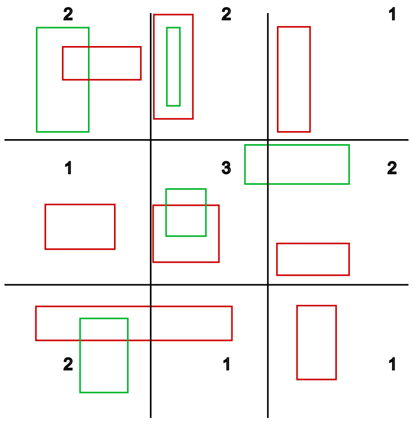


Figure 2: Partition of the plane

Additionally, we determined the optimal axis (either X or Y) on which to sweep the plane. This decision depends on the distribution of the MBRs; choosing the axis with the lesser extent of overlap minimizes the number of comparisons needed during the sweep (Figure 3), thereby improving the efficiency of the algorithm.

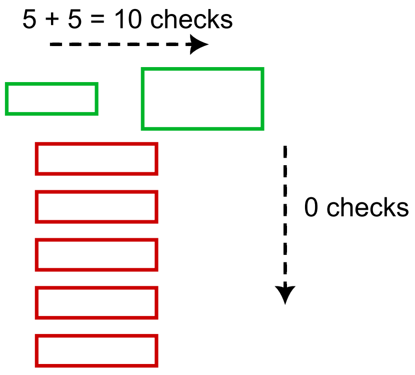


Figure 3: Axis selection

To estimate the number of intersecting projections per axis, the authors compute histogram statistics. Specifically, they subdivide the x and y projections into a predefined number of partitions k. Then, they count how many rectangles from R and S intersect each x-division. The procedure for y partitions is symmetric. This results in four histograms, H_x^R, H_y^R, H_x^S, and H_y^S, each with k buckets. The number of rectangles I_x in $R_T \times S_T$ that intersect along the x-axis can then be approximated by accumulating the product of the corresponding histogram buckets: $I_x = \sum_{i=1}^{k} H_x^R[i] \times H_x^S[i]$. Similarly, we can calculate the approximation for I_y.

Approximation Enhancement

Since the authors use a sample of rectangles to approximate the intersections, they might occasionally choose the wrong axis. However, we can determine the exact number of intersections for both projections efficiently using the following method:

Consider the problem as having $N+M$ segments on the x-axis (or y-axis similarly) and computing the total number of intersections between the two sets of segments. For each segment $(x1, x2)$ from one set, we need to find how many segments $(x1', x2')$ from the other set satisfy the following condition:

$$(x1' \leq x1 \text{ and } x2' \geq x1) \text{ or } (x1' > x1 \text{ and } x1' \leq x2)$$

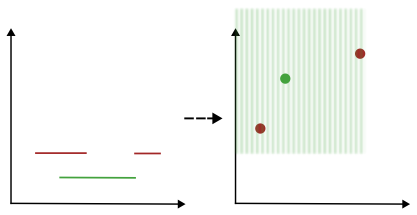


Figure 4: Segment to point transformation

Representing our segments as points in the 2D plane, for a querying segment/point $(x1, x2)$ of one set, we need to find how many points lie in the half-plane $[1, y1] \times [x1, +\infty]$. An example is illustrated in Figure 4, where the segments (displayed in 2D for better visualization) are transformed into 2D points, and the query half-plane of the green segment is displayed.

To solve this efficiently, we build a segment tree on the x-axis, where each node contains the y-coordinates of points with x in the range the node is responsible for. We can then query ranges and find the total number of points quickly by binary searching the sets of y values in all nodes with ranges completely inside the querying range.

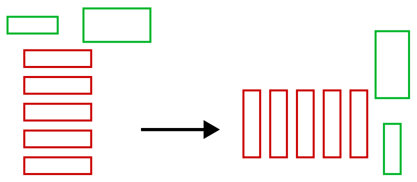


Figure 5: Axis rotation

After identifying the axis with the minimum number of intersections, if the y-axis is the optimal choice, we can simply swap the x and y values (Figure 5). This allows us to run the same algorithm without any modifications.

The Forward Scan based Plane Sweep algorithm 1 operates by first sorting the input rectangles by their lower x-endpoints. It then concurrently scans the sorted lists, maintaining active intervals of intersecting rectangles. As the sweep line progresses, it quickly identifies intersecting pairs by comparing the active intervals' y-ranges. This method is efficient for detecting intersections because it leverages the sorted order to limit the number of comparisons needed, reducing the overall computational complexity.

ALGORITHM 1: Forward Scan based Plane Sweep

```
Input  : collections of rectangles $R$ and $S$
Output : set $J$ of all intersecting rectangles $(r, s) \in R \times S$
1  sort $R$ and $S$ by lower $x$-endpoint $x_l$;
2  $r \leftarrow$ first rectangle in $R$;
3  $s \leftarrow$ first rectangle in $S$;
4  while $R$ and $S$ not depleted do
5      if $r.x_l < s.x_l$ then
6          $s' \leftarrow s$;
7          while $s' \neq null$ and $r.x_u \geq s'.x_l$ do
8              if $r.y$ intersects $s'.y$ then
9                  output $(r, s')$;                ▷ update result
10             $s' \leftarrow$ next rectangle in $S$;  ▷ scan forward
11         $r \leftarrow$ next rectangle in $R$;
12     else
13         $r' \leftarrow r$;
14         while $r' \neq null$ and $s.x_u \geq r'.x_l$ do
15             if $r'.y$ intersects $s.y$ then
16                 output $(r', s)$;                ▷ update result
17             $r' \leftarrow$ next rectangle in $R$;  ▷ scan forward
18         $s \leftarrow$ next rectangle in $S$;
```

4.2 2D Segment Tree Join

In our implementation, we employed a 2D segment tree approach to efficiently manage and query spatial data. A 2D segment tree (Figure 6) is an extension of the 1D segment tree, designed to handle range queries in two dimensions. A segment tree can be thought of as a binary tree, where each node represents an interval of the range and stores information pertinent to that interval. The root node covers the entire range, and each subsequent level of the tree divides the range into smaller intervals, down to the leaf nodes which cover single elements.

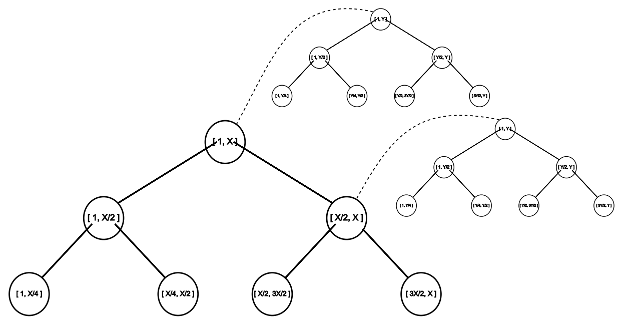


Figure 6: 2D segment tree

For the 2D segment tree, we first build a 1D segment tree to manage the X-values of the dataset. Each node of this 1D segment tree then contains another segment tree that manages the Y-values (and a specific segment tree that manages the Y-values for a particular case). This hierarchical structure allows us to efficiently handle 2D range queries.

ALGORITHM 2: 2D Segment Tree Algorithm

```
Input  : collections of rectangles $R$ and $S$ in compressed space
Output : set $J$ of all intersecting rectangles $(r, s) \in R \times S$
1 Initialize segment tree $T$ for $X$ and $Y$ axis ;
2 foreach rectangle $r \in R$ do
3     $min_x, max_x, min_y, max_y \leftarrow$ compressed coordinates of $r$'s mbr;
4     update $T$ with $X$ range $[min_x, max_x]$ and $Y$ range $[min_y, max_y]$;
5 foreach rectangle $s \in S$ do
6     $min_x, max_x, min_y, max_y \leftarrow$ compressed coordinates of $s$'s mbr;
7     if query $T$ with $X$ range $[min_x, max_x]$ and $Y$ range $[min_y, max_y]$ then
8         output $s$ ;
```

The algorithm begins by constructing the 2D segment tree using the Minimum Bounding Rectangles (MBRs) of the polygons from set *R*. For each MBR in *S*, it then queries this segment tree to identify intersections. The segment tree is constructed as follows: When updating the tree with a rectangle, we locate all nodes in the X segment tree that are responsible for intervals containing the range $(\min_x, \max_x)$ of the rectangle. For each of these nodes, the corresponding Y segment trees are updated to indicate that there is an interval present in the range $(\min_y, \max_y)$. If a node in the X segment tree has an interval that is fully covered by the range $(\min_x, \max_x)$, we update the other Y segment tree for that range.

The same process is then repeated in the opposite direction: we build a segment tree using the MBRs from set *S* and query it with the MBRs from *R*.

The process of querying a rectangle from the querying set in the segment tree works as follows: For all nodes in the X segment tree whose ranges are fully covered by the querying rectangle's $(\min_x, \max_x)$, we query their corresponding Y segment trees to check for overlap with $(\min_y, \max_y)$. Additionally, while traversing the segment tree to locate these nodes, if we encounter a node that has a non-empty specific Y segment tree—indicating it was updated by a rectangle spanning the entire X range—we also query this Y segment tree for overlap, even if the node's range is not completely covered by $(\min_x, \max_x)$.

When querying the Y segment tree, if we are at a node that was marked as a whole covered during the update of an MBR, we can safely return that there is an intersection. Otherwise, for all nodes where the query Y range completely overlaps the node's range, we simply check if the node was marked as true from an updated range.

It is important to note that this method does not directly find all pairs of intersecting MBRs. Instead, it identifies MBRs that intersect with at least one other MBR from the opposite set. Consequently, in the subsequent step, we must examine all possible pairs of intersecting MBRs. While this might be nearly as expensive as examining all pairs of MBRs, in dense cases the rasterization join algorithm performs similarly, but without the significant performance gain in the initial step.

Additionally, if a new polygon appears, we can quickly query the 2D segment tree to determine if there are no intersections, allowing us to ignore it and avoid unnecessary computations. This fast querying capability makes the 2D segment tree particularly useful in dynamic scenarios where the dataset may frequently change.

4.3 Combined Plane Sweep Join

4.3.1 Missed Inner Intersections. In this approach, the algorithm first sweeps along the X axis to determine which polygons from *R* intersect with polygons from *S*. The process begins by detecting the starting and ending segments of the Minimum Bounding Rectangles (MBRs) from the collections. When the algorithm encounters a segment from *R*, it queries a data structure that maintains the active Y intervals of *S*. If a starting segment of an MBR from *S* is found, it is added to this data structure, and if an ending segment is found, it is removed. When intervals share the same X coordinate and belong to different sets, priority is given to the starting segment from *S* so that it can be queried for *R* in the next iteration. Conversely, if the segment from *S* is ending, priority is given to the segment from *R* to ensure it is queried before the segment from *S* is removed. The same procedure is repeated to find which polygons from *S* intersect with polygons from *R*.

For the data structure, we have used a Binary Indexed Tree (BIT). A BIT is essentially an array *B*, where the value at each index *i* provides information about the range $[g(i), i]$, as dipicted in Figure 7 (in our case, this range represents a segment on the y-axis). When selecting an appropriate function *g*, the BIT offers efficient methods for quickly adding or subtracting values at specific positions and calculating the sum up to a certain index. With some modifications and by adding another Binary Indexed Tree, we can also perform range updates along with range queries. This efficiency in both updates and queries makes the BIT an ideal choice for maintaining and querying active Y intervals in our algorithm.

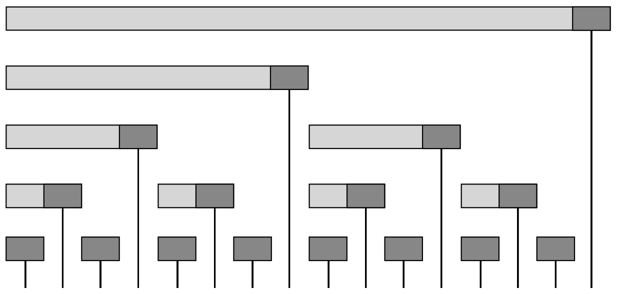


Figure 7: Binary Indexed Tree

This method misses intersections where a rectangle from the querying set starts and ends entirely within the x-interval of a rectangle from the other set. However, we can find additional intersections by applying the same methodology with a sweep along the Y axis. By doing that we can detect cases where a rectangle from the querying set starts and ends entirely within the x-interval of a rectangle from the other set, provided the y-intervals of the rectangles are not contained within each other.

ALGORITHM 3: Missed Inner Intersections

```
Input  : collections of rectangles $R$ and $S$ in compressed space
Output : set $J$ of all intersecting rectangles $(r, s) \in R \times S$
1  Initialize data structure $T$ for active Y intervals of $S$;
2  foreach segment in sorted intervals of $R$ and $S$ do
3      if segment is a starting/ending segment of $R$ then
4          $[min_y, max_y] \leftarrow$ segment;
5          query $T$ for intersection with $[min_y, max_y]$;
6          if intersection found then
7              add to result set $J$;
8      else if segment is a starting segment of $S$ then
9          add segment to $T$;
10     else
11         remove segment from $T$;
12 Repeat the process for segments in $S$ with active Y intervals of $R$;
13 Repeat the entire process with sweeps along the Y axis;
```

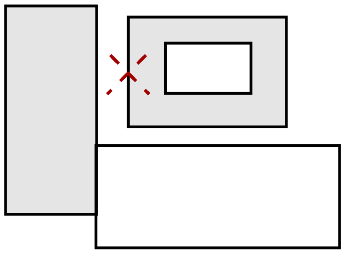


Figure 8: Missing inside case

As we can see, this approach misses only the cases where a rectangle from one collection has only intersections with rectangles from the other collection that are completely inside it. For example, in Figure 8, the above algorithm will find all polygons as intersecting except the one crossed by the red X. This occurs because the polygon from the other set that is inside it has starting and ending intervals for both x and y coordinates completely within the starting and ending intervals of the red crossed rectangle. Consequently, during the sweep line process on both axes, it will be added and removed before any query can detect the intersection.

4.3.2 Missed Outer Intersections. This approach aims to identify intersections between polygons from two collections, R and S, but it may miss intersections where one rectangle from the querying set fully covers another rectangle.

We build a segment tree using the coordinates of the MBRs from S and then query it with the MBRs from R. For each MBR from S, we insert each of its corners (x, y) into a segment tree. This involves performing a point update at position x. While traversing the tree from the root node to the leaf node responsible for position x, we add the y coordinate to each node's set that we pass. After all the updates are done, we ensure that the y values are maintained in sorted order in all the sets at each node of the segment tree. For each MBR from R, we perform a range query on the x coordinates $[min_x, max_x]$ and check if there are any y values within the range $[min_y, max_y]$. While traversing the tree, we check all nodes responsible for a range that is completely overlapped by the querying $[min_x, max_x]$ (this check is performed quickly using binary search due to the sorted nature of the set). If such y values are found, it indicates an intersection. The same procedure is then repeated by building the data structure for R and identifying the intersecting rectangles in S.

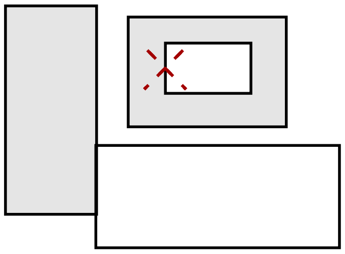


Figure 9: Missing outside case

The above algorithm fails to identify a rectangle as intersecting if it is completely covered by other rectangles from the opposite set (i.e., the examined rectangle is entirely inside them). This occurs because none of the points of the covering rectangles will fall within the query range for the examined rectangle. We can see an example of this in Figure 9, where the rectangle marked with a red cross will not be identified as intersecting, because the only polygon that intersects with it completely covers it.

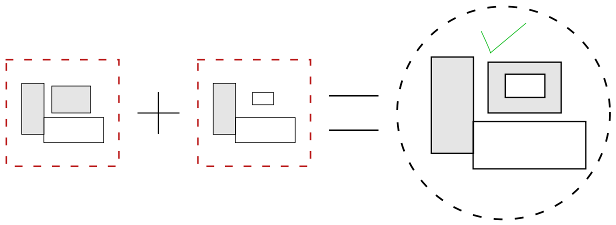


Figure 10: Combined Sweep

By combining the outputs of the two approaches (Figure 10)—Missed Inner Intersections and Missed Outer Intersections—we achieve a comprehensive solution that captures all possible intersections between polygons from collections R and S. Together, these methods provide a robust mechanism for accurately determining all intersecting polygons, overcoming the limitations inherent in each individual algorithm.

Again, it is important to note that this method does not directly find all pairs of intersecting MBRs. In dense cases we have a significant performance gain in the initial mbr filtering. Moreover, this combined approach generally outperforms the 2D segment tree method

5 Rasterization

The rasterization process converts polygons into a grid-based representation to efficiently detect intersections within a spatial dataset.

ALGORITHM 4: General Rasterization Algorithm

```
Input  : Polygon $P$, Grid parameters
Output : Clipped polygon segments within grid cells
1 Initialize polygon borders and grid cells;
2 for each vertical grid line do
3     Clip polygon against the line;
4     Carry over remaining parts to next column;
5 for each horizontal grid line do
6     Clip results from vertical clipping;
7     Store final clipped segments in grid cells;
```

This grid is a $2^N \times 2^N$ structure, which overlays the minimum bounding rectangle (MBR) of the dataset. Each polygon is clipped against the grid lines to determine which grid cells it intersects, and the cells are classified as **full**, **strong**, or **weak** based on the extent of overlap.

We implemented two clipping algorithms Hodgman Clipping [3], which was utilized by Georgiadis et al. [1] and Weiler Atherton Clipping [5] to perform this task. Both algorithms follow the same general rasterization process: vertical clipping is applied first, followed by horizontal clipping.

The General Rasterization Algorithm 4 performs by first clipping the polygon against vertical grid lines from left to right. Remaining parts of the polygon on the right are carried over to be clipped against the next vertical line. The results from vertical clipping are then clipped against horizontal grid lines from top to bottom. The final clipped polygon segments are stored in the corresponding grid cells.

Due to the fixed structure of the grid boundaries, if a new dataset of polygons that is not inside the grid boundaries, should be tested with another one to find intersecting pairs, the grid should be reconstructed from scratch to include the new dataset too and all rasterized polygons should undergo the same rasterization process since the cells coordinates have changed.

5.1 Hodgman Clipping

The Sutherland–Hodgman 5 algorithm clips polygons by sequentially processing each edge of the clipping polygon, selecting only vertices on the visible side. It creates new vertices at each intersection for each grid segment (column or row), the polygon is clipped against the segment line. Points inside the clipping line are kept, and new vertices are created at intersections.

This method is limited to convex clip polygons according to Günther Greiner et al. [2]. Concave polygons may be displayed with extraneous lines as demonstrated in the figure 11, which depict the clipping of a concave polygon first vertically and then horizontally as described in 4.

5.2 Weiler-Atherton Clipping

The Weiler–Atherton algorithm 6 clips polygons by managing intersections comprehensively, allowing for clipping by an arbitrarily shaped region. It handles concave polygons and multiple intersections by constructing new polygons through managing these intersections. For each grid segment (column or row), all intersections between the polygon and the segment line are identified and sorted. The algorithm then walks through polygon vertices and intersection points to build new polygon parts that fit within the grid cells.

ALGORITHM 5: Hodgman Clipping

```
Input  : Polygon $P$, Grid segment line
Output : Clipped polygon segments
1 Initialize new polygon vertices list;
2 for each edge of polygon $P$ do
3     Determine if vertices are inside or outside the clipping
        line;
4     Add intersection points as new vertices;
5     Keep vertices on the visible side;
6 Return clipped polygon segments;
```

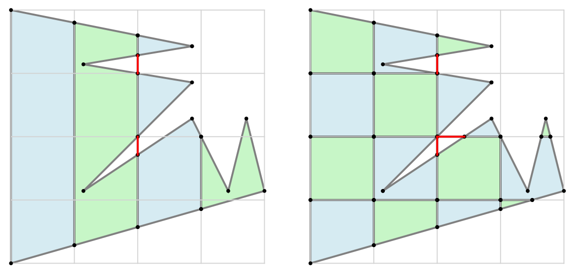


Figure 11: Rasterization Of Concave Polygon with Hodgman

ALGORITHM 6: Weiler-Atherton Clipping

```
Input  : Polygon $P$, Grid segment line
Output : Clipped polygon segments
1 Identify intersections between polygon $P$ and grid segment
    line;
2 Sort intersection points;
3 Walk through vertices and intersection points to build new
    segments;
4 Return clipped polygon segments;
```

Weiler-Atherton Clipping effectively manages complex, concave polygons by handling intersections comprehensively, ensuring accurate results even for intricate shapes. In figure 12 we can see the same concave polygon rasterization using the Weiler-Atherton Clipping algorithm first vertically and then horizontally as described in 4. The result may consist of multiple polygons in the same column (for vertical clipping) as in the third column of the example or in the same cell (for the horizontal clipping). Therefore we end up with one or more polygons in a cell without leaving any residue behind unlike Sutherland – Hodgman polygon clipping algorithm 5.

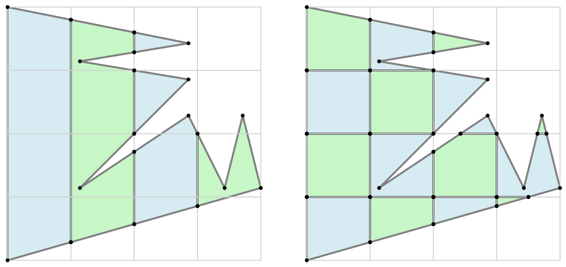


Figure 12: Rasterization Of Concave Polygon with Weiler

5.3 Cells Classification

After clipping the polygons against the grid, we classify the intersecting cells into three categories: **full**, **strong**, and **weak**. This classification is based on the extent of overlap between each cell and the data object, helping to approximate the spatial presence of the object. Here's how each type of cell is determined:

- **Full Cells**: These are cells where the data object completely covers the cell. Full cells provide the highest certainty that the object is present within the cell.
- **Strong Cells**: These cells are partially covered by the data object. They indicate a significant presence of the object but not a complete coverage.
- **Weak Cells**: These cells have a minimal overlap with the data object. They represent the least degree of overlap and, consequently, the lowest certainty of the object's presence.

In order to determine the area of the polygon that is clipped for cell boundaries we use the Shoelace formula.

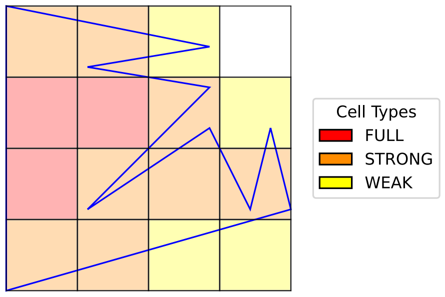


Figure 13: Rasterization of a polygon in a grid $2^2 \times 2^2$. Red indicates full cell type, orange indicates strong, and yellow indicates weak.

In Figure 13, we can see the rasterization of a polygon in a grid $2^2 \times 2^2$, with the red color indicating full cell type, the orange indicating strong, and the yellow indicating weak.

5.4 Encoding of Cell Types

We employ a 3-bit encoding scheme for the cell types to facilitate efficient spatial joins according to Georgiadis et al. [1]. The encoding differs based on whether the object comes from join input R or S. Table 1 illustrates the encoding:

Cell Type	Input R Code	Input S Code
Full	011	101
Strong	101	011
Weak	100	010

Table 1: 3-bit type codes for each input dataset

5.5 Properties of the Encoding

The encoding has two critical properties:

(1) **Intersection Detection**: If for a cell c, the bitwise AND of the codes of an object $r \in R$ and an object $s \in S$ is non-zero, we can conclude that r and s intersect in c. This is particularly useful for cases where at least one type is full, or both are strong.
(2) **Input Role Swapping**: The encoding allows swapping the roles of R and S in a join operation. To convert the cell code for an object from input R to input S, we XOR the code with the mask $m = 110$. For example, the encoding for a full cell in R (011) becomes 101 in S after XORing with the mask.

This flexibility is important when the rasterization of a dataset has been precomputed according to the R encoding and needs to be used as the right join input S. This XOR operation is performed on-the-fly during the serialization phase.

5.6 Ambiguity in Rasterization Process

In our exploration of the rasterization process, we identified an ambiguity in the original algorithm. Consider the scenario where two polygons are separated by a distance equal to the side length of a grid cell. Applying the clipping algorithm results in the intersection of each polygon with the intermediate neighboring cell, forming either a point or a line segment parallel to one of the cell's sides.

Following the guidelines from the paper, such a cell is classified as **weak** for each polygon, since each object covers at most 50% of the cell. According to the encoding scheme, applying the logical AND operation between the binary representations of weak cell types results in an **indecisive** outcome. Consequently, even though determining the intersection in this cell involves merely comparing a line/point with another line/point, the encoding implies indecisiveness, meaning the polygons will not be filtered in the intermediate step.

This classification leads to the polygons being passed to the final, computationally expensive refinement step, which could have been avoided. We hypothesize that this scenario was not adequately considered due to the constraints of a 3-bit binary representation. Addressing this ambiguity might necessitate extending the binary representation from 3 to 4 bits to accommodate a new cell type for accurate identification and filtering of these cases.

Figure 14 illustrates this ambiguity. In this example, the cell in the second column and third row is determined as weak for both polygons, resulting in an indecisive outcome when joining the two cell types.

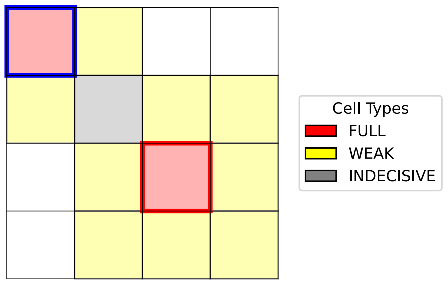


Figure 14: Ambiguity Example: Intermediate cell classified as weak for both polygons, leading to indecisiveness

6 Hilbert Curve

The Hilbert curve is a continuous fractal space-filling curve that maps a one-dimensional space into a two-dimensional space. It is used in various applications such as image compression, data sorting, and improving locality of reference in data structures. The Hilbert curve maintains locality, meaning points that are close in the one-dimensional space are also close in the two-dimensional space.

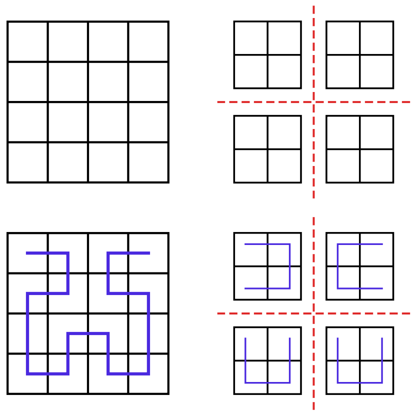


Figure 15: Recursive hilbert curve generation

The Hilbert curve can be generated using a recursive algorithm. The basic idea is to divide the space into four quadrants and recursively apply the Hilbert curve to each quadrant, with appropriate rotations to maintain continuity.

As illustrated in Figure 15, we partition the $2^N \times 2^N$ grid into four $2^{N-1} \times 2^{N-1}$ grids and find the Hilbert curves for each smaller grid. Subsequently, we add segments joining the starting and ending endpoints of these smaller curves to form the final Hilbert curve.

7 Serialization

To serialize polygons efficiently for spatial operations, we employ the methodology that leverages Hilbert curves and bitmasks, as described by Georgiadis et al. [1]. This approach allows us to represent the spatial relationships of polygons in a compact and structured format. The process can be broken down into the following steps (Algorithm 8):

We start with rasterized polygon information for the polygons that have passed the MBR filtering. Each polygon is represented by its raster cells, and for each cell, we have a bitmask representing the type of the cell (Strong, Full, Weak). For each polygon, we iterate through its raster cells identified by their coordinates (i, j), which are then mapped to the corresponding Hilbert value using a precomputed Hilbert map. This map associates each cell (i, j) with the corresponding Hilbert value index.

ALGORITHM 7: Hilbert Curve Generation and Sorting

```
Input  : Order of hilbert curve $N$
Output : Map of coordinates to Hilbert values
Initialize empty list $H$;
Initialize $x = 0, y = 2^N - 1$;
Function hilbert($level$):
    if $level == 0$ then
        append $(x, y)$ to $H$;
    Rotate appropriately;
    hilbert($level - 1$);
    Move forward;
    Rotate appropriately;
    hilbert($level - 1$);
    Move forward;
    hilbert($level - 1$);
    Rotate appropriately;
    Move forward;
    hilbert($level - 1$);
    Rotate appropriately;
```

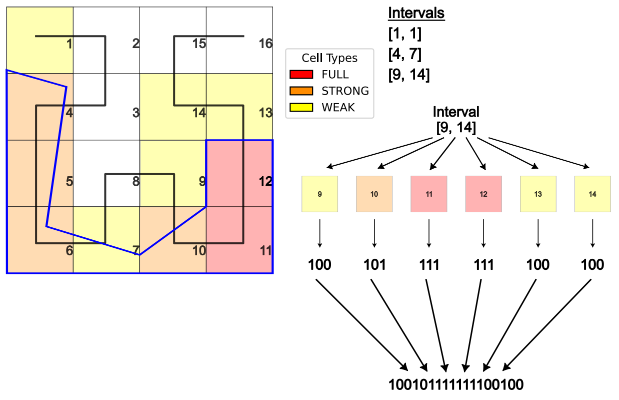


Figure 16: Polygon Serialization

Next, we create intervals of continuous Hilbert values of the rasterized polygon and their corresponding bitmasks. For each interval of Hilbert values, we construct a bitmask that aggregates the bitmasks from all raster cells within that interval. Now, each polygon is represented by a small number of pairs in the form $[\text{start}_i, \text{end}_i]$ and bitmask_i.

ALGORITHM 8: Polygon Serialization Algorithm

```
Input  : collections of raster polygon info $L$ and $R$,
         Hilbert map $H$
Output : serialized polygons $L'$ and $R'$
foreach set of polygon info $set \in \{L, R\}$ do
    foreach polygon info $P \in set$ do
        Initialize $hilbert\_values \leftarrow \emptyset$;
        foreach raster cell $(i, j)$ in $P$ do
            append $H[(i, j)]$ to $hilbert\_values$
        Sort $hilbert\_values$;
        Initialize $S' \leftarrow \emptyset$;
        $start, end \leftarrow$ NULL;
        $bitmask \leftarrow$ empty bitmask;
        foreach pair $(hval, cell\_type) \in hilbert\_values$ do
            if $end + 1 = hval$ then
                $end \leftarrow hval$;
            else
                if $end \neq NULL$ then
                    append $(start, end, bitmask)$ to $S'$;
                $start \leftarrow hval$;
                $end \leftarrow hval$;
                clear $bitmask$;
            if $set = R$ then
                XOR $cell\_type$ with 110;
            append $cell\_type$ to $bitmask$;
        append $(start, end, bitmask)$ to $S'$;
        $serialized\_polygon \leftarrow$ SerializedPolygon$(S', P.id)$;
        append $serialized\_polygon$ to $set'$;
```

As we can see, during the serialization, we XOR the bitmasks of the set R with the bitmask 110 since they have the same encoding as S.

In Figure 16, we can see an example of a polygon that is serialized with three intervals. For the interval $[9, 14]$, we observe the aggregation of the cells' bitmasks into one larger bitmask representing the entire interval.

8 Raster Intervals Join

The Raster Interval (RI) Join algorithm (Algorithm 9) efficiently identifies intersecting polygons through a detailed process of interval comparison and bitmask analysis. The methodology involves several key steps, ensuring accurate detection of spatial overlaps.

First, we consider each pair of candidate intersecting polygons, as determined by their Minimum Bounding Rectangles (MBRs). These candidates are identified in a preprocessing step, ensuring that only potentially intersecting polygons are evaluated further.

For each pair of polygons, we leverage their serialized representations, which include intervals of continuous Hilbert values and corresponding bitmasks. The intervals are sorted to facilitate efficient comparison. The algorithm then iterates through the sorted intervals of both polygons, systematically checking for overlapping ranges.

ALGORITHM 9: Raster Interval Join Algorithm

```
Input  : collections of serialized polygons $L'$ and $R'$
Output : set $J$ of all intersecting polygons
         $(l', r') \in L' \times R'$
foreach pair $(l', r') \in final\_result$ do
    $i, j \leftarrow 0, 0$;
    $overlap \leftarrow$ false;
    while $i < size(l'.bitmasks)$ and $j < size(r'.bitmasks)$
     do
        $l'_{start}, l'_{end} \leftarrow$
          $l'.bitmasks[i].start, l'.bitmasks[i].end$;
        $r'_{start}, r'_{end} \leftarrow$
          $r'.bitmasks[j].start, r'.bitmasks[j].end$;
        $l'_{bm} \leftarrow l'.bitmasks[i].bitmask$
         $r'_{bm} \leftarrow r'.bitmasks[j].bitmask$
        if $l'_{end} \geq r'_{start}$ and $r'_{end} \geq l'_{start}$ then
            $overlap \leftarrow$ true;
            $start \leftarrow \max(l'_{start}, r'_{start})$;
            $end \leftarrow \min(l'_{end}, r'_{end})$;
            $lhs\_bitmask \leftarrow$
              truncate$(l'_{bm}, start, end, l'_{start}, l'_{end})$;
            $rhs\_bitmask \leftarrow$
              truncate$(r'_{bm}, start, end, r'_{start}, r'_{end})$;
            if $lhs\_bitmask$ AND $rhs\_bitmask$ is non-zero
              then
                output $(l', r')$ to $result$;
        if $l'_{end} \leq r'_{end}$ then
            $i \leftarrow i + 1$;
        else
            $j \leftarrow j + 1$;
    if $overlap$ and $(l', r')$ not in result then
        add $(l', r')$ to $indecisive$;
```

When an overlap between intervals is detected, we focus on the intersecting segment. The corresponding bitmasks for these intervals are truncated to the overlap range, ensuring that only the relevant portion of each bitmask is considered. This truncation involves shifting and resizing the bitmasks to align them precisely with the overlap interval.

Next, we perform a bitwise AND operation on the truncated bitmasks. This step is crucial as it determines whether there is an actual intersection within the overlapping interval. If the result of the bitwise AND operation is non-zero, it indicates that the polygons intersect in the specified range.

If an intersection is confirmed, the pair of polygons is recorded as intersecting. If the bitwise AND operation does not reveal an intersection, but there was an overlap in intervals, the pair is marked as indecisive for further examination in the refinement process.

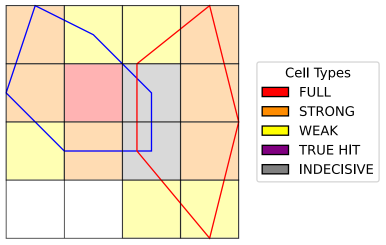
8.1 The effect of N in RI

The decision of N regarding the size of the grid has huge impact in the raster join performance. Larger value of N creates a finer grid, leading to more precise approximations. This can be illustrated by comparing two cases: one using a $2^2 \times 2^2$ grid and the other using a $2^3 \times 2^3$ grid.

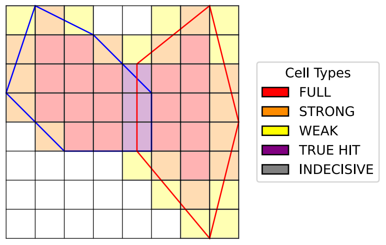
In the first case with the $2^2 \times 2^2$ grid, the coarse granularity results in no true hits as shown in 17(a). The cells are mostly indecisive due to weak rasterization; they are the result of a raster join between weak/strong cells and weak cells. This necessitates a costly refinement step to test for the intersection of the two polygons.

In contrast, the second case with the $2^3 \times 2^3$ grid shows a higher granularity, leading to true hits as depicted in 17(b). Here, the cells are more accurately classified, often as full or strong types. These cells result from raster joins between full and strong/full cells or strong and strong cells, thereby avoiding the need for the expensive refinement step.

However, increasing N also demands more storage space for the interval endpoints.



(a) Raster join by cell type of 2 polygons in a grid $2^2 \times 2^2$



(b) Raster join by cell type of 2 polygons in a grid $2^3 \times 2^3$

Figure 17: Raster join by cell type of 2 polygons in different granularities

9 Refinement

After identifying indecisive pairs of polygons through the initial filtering and joining processes, we proceed to the refinement stage to determine actual intersections.

For every indecisive pair, we perform a comprehensive intersection check by following a two-step process:

- **Segment Intersection Check:**
  We first check for intersections between all pairs of line segments from the two polygons. This involves iterating through each edge of the first polygon and checking if it intersects with any edge of the second polygon. The intersection of two line segments is determined using the orientation of points, checks for collinearity and overlaps.

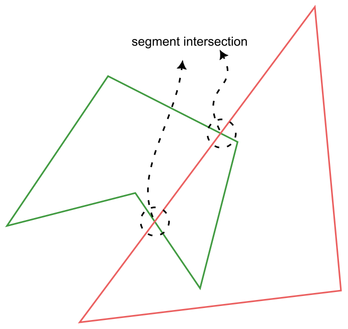


Figure 18: Segment Intersection

- **Point-Inside-Polygon Check:**
  If no segment intersections are found, we further verify whether any vertex of one polygon lies inside the other polygon using the ray-casting algorithm. This situation may occur when a very small polygon (smaller than a cell of the grid) is situated within another polygon, but does not fully cover the cell, causing the raster interval join to mark them as indecisive. To check if a point is inside a polygon, we cast a horizontal ray from the point and count the number of times it intersects with the polygon's edges (Figure 19). If the count is odd, the point is inside the polygon; if even, it is outside.

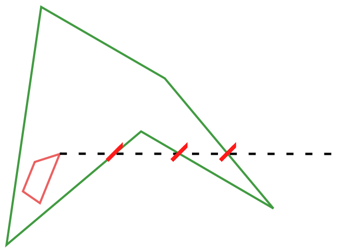


Figure 19: Ray Casting

10 Experiments

The experiments begin with data preprocessing of polygon data from the provided files. For each polygon, we compute the Minimum Bounding Rectangles (MBRs). To determine the overall grid boundaries, we find the minimum and maximum corners from both datasets.

The initial filtering process uses MBRs to quickly identify potential intersecting polygons. We employ the forward scan algorithm to accomplish this.

Next, we rasterize the filtered polygons onto the grid using the rasterization process.

Following rasterization, we serialize the rasterized polygons.

The Raster Interval (RI) Join algorithm is then applied to the serialized polygons.

Finally, we address indecisive pairs—those for which the RI Join could not conclusively determine intersection. For each indecisive pair, we perform a detailed geometric intersection check.

We measure various metrics (Table 2) throughout the pipeline, including the durations of the MBR filter, rasterization, serialization, RI join, and refinement steps. Additionally, we compute the ratios of true hits, false hits, and indecisive pairs to assess the performance and accuracy of the methodology.

11 Build

To configure, build, and create the executables for this project, follow the commands below:

```
conda create -n boostenv cmake boost-cpp
conda activate boostenv
git clone <repository-url>
cd <repository-url>
cmake -B build -S .
cmake --build build
```

12 Conclusion

Our implementation of the Raster Intervals (RI) technique for polygon intersection joins demonstrates improvements in handling dense scenarios with many intersections in MBR filtering and ensures better performance on some edge cases on the rasterization process. Future work could explore extending the binary representation from 3 to 4 bits to resolve ambiguities in the rasterization process and further enhance the accuracy and efficiency of the technique.

References

[1] Thanasis Georgiadis, Eleni Tzirita Zacharatou, and Nikos Mamoulis. 2024. Raster Interval Object Approximations for Spatial Intersection Joins. arXiv:2307.01716

[2] Günther Greiner and Kai Hormann. 1998. Efficient clipping of arbitrary polygons. *ACM Transactions on Graphics* 17, 2 (April 1998), 71–83.

[3] I. E. Sutherland and G. W. Hodgman. 1974. Reentrant polygon clipping. *Commun. ACM* 17, 1 (Jan 1974), 32–42.

[4] Dimitrios Tsitsigkos, Panagiotis Bouros, Nikos Mamoulis, and Manolis Terrovitis. 2019. Parallel In-Memory Evaluation of Spatial Joins. In *Proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL '19)*. ACM. https://doi.org/10.1145/3347146.3359343

[5] K. Weiler and P. Atherton. 1977. Hidden surface removal using polygon area sorting. *Computer Graphics* 11, 2 (Jul 1977), 214–222. Proceedings of SIGGRAPH.

Metric	**T1 T2**	**OC**	**SA**	**AF**	**AS**
Total polygons	2389922	146179	257671	135975	978325
Average number of vertices	32	45	48	50	45
MBR filter duration (s)	24.73	1.26	2.41	1.15	11.99
Rasterization duration (s)	18.04	3.84	29.20	13.01	19.12
Serialization duration (s)	0.38	0.19	1.27	0.39	1.17
RI join duration (s)	0.16	0.10	0.19	0.06	0.71
Refinement step duration (s)	327.93	385.23	122.91	31.27	1158.13
True hits percentage	4.35	1.08	5.00	6.43	1.29
False hits percentage	14.40	20.87	36.88	24.84	12.66
Indecisive percentage	81.26	78.06	58.12	68.73	86.05

Table 2: Comparison of various metrics across different datasets