Accelerating Point-Polygon Topological Relationship Queries with Hierarchical Indices

Ruthvik Kodati (2019101035)
International Institute of Information Technology Hyderabad

I. Abstract

This paper describes a point-polygon query program that alludes to the ACM SIGSPATIAL CUP 2013. The contest is about geo-fencing, which is a virtual perimeter for a real-world geographic area. Geo-fencing is used widely in location-based services, e.g., location-based advertisements (which send the targeted ads to the users when they are close to the shopping mall) and child location services (which notify parents when a child leaves a designated area).

In this paper, I propose an efficient algorithm to improve the scalability of geofencing. In the first stage of each sub-problem (INSIDE or WITHIN n), I utilize the R-tree data structure to quickly detect whether a point is inside the minimum bounding rectangle of a polygon. Based on the patterns of the training dataset, I built a hierarchy of indices including polygon index and edge index to help find polygons near a point, calculate the distance from a point to a polygon, and determine whether a point is inside a polygon, respectively. Furthermore, the cup provides a large number of point locations and spatial polygons along with some spatial predicates. The goal is to find every (point, polygon) pair in the dataset that satisfies the spatial predicate. Ultimately, I follow the use-case of the contest: there are way more points (up to one million) than polygons (up to 500).

II. Introduction

The geo-fencing problem posed by the GIS Cup 2013 can be formulated as an estimation whether a point is INSIDE or WITHIN a distance of a polygon. Both a point and a polygon may have multiple instances, each identified by an ID and a sequence number. A polygon contains one outer ring, and zero or more inner rings. A point may appear in several overlapping polygons.

Querying whether a point is inside a polygon, often called point-in-polygon query in the literature, or the "INSIDE" query of this competition, is one of the fundamental geometric problems. The classical algorithm (naive methodology) is the *ray casting algorithm*, which tests how many times a ray, starting from the point and going any fixed direction, intersects the edges of the polygon. If the point is on the outside of the polygon, the ray will intersect its edge an even number of times. If the point is inside the polygon, then it will intersect the edge an odd number of times (see Figure 1). Through some math, we can deduce the conditions for the ray casting algorithm. As you can see, all edges are checked in order to check how many times an emanating ray (in any direction) intersects the polygon (looping through edges - shown in Figure 2). Therefore, in the naive methodology, we apply the ray casting algorithm for each (point, polygon) pair to check if the point is INSIDE the polygon. The time complexity would be $O(n * p * e)$ where n is the number of points, p is the number of polygons, and e is the average number of edges across all polygons (computationally expensive).

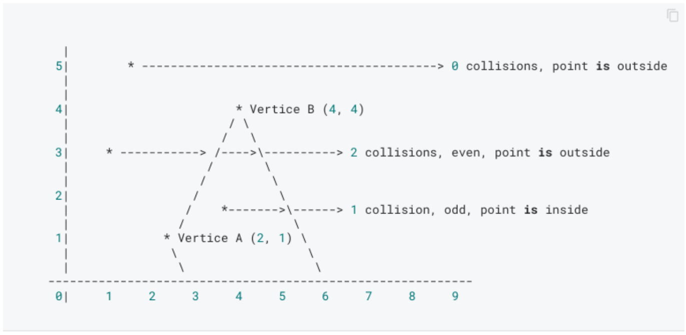
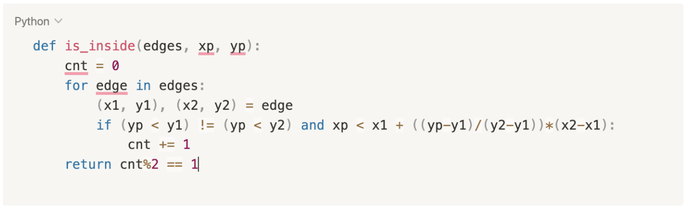

0 collisions, point is outside
Vertice B (4, 4)
2 collisions, even, point is outside
1 collision, odd, point is inside
Vertice A (2, 1)
0
1
2
3
4
5
6
7
8
9

Fig. 1. Ray Casting Algorithm



```python
def is_inside(edges, xp, yp):
    cnt = 0
    for edge in edges:
        (x1, y1), (x2, y2) = edge
        if (yp < y1) != (yp < y2) and xp < x1 + ((yp-y1)/(y2-y1))*(x2-x1):
            cnt += 1
    return cnt%2 == 1
```

Fig. 2. Ray Casting Algorithm Code

If a point is not inside a polygon, we may want to calculate its distance to the polygon, referred to as "WITHIN n" query of this competition. This can be done by calculating the distance between the point and each edge of the polygon, and returning the minimum value. While the complexity (both average and worst case) for the naive methodology is $O(n)$ (n being the number of edges of the polygon), in practice, spatial indices are the key to achieving optimal performance, especially when querying a large dataset.

Different datasets and different query patterns may require different indices for the best performance. For example, if we query each point and each polygon just once, there is no need to build any index at all. On the other hand, if we want to find the relationship between a set of points and a set of polygons, we can index either points or the polygons, e.g., depending on which set is smaller. Therefore, we first analyze the training dataset to determine the optimal indices to use. There are way more points, up to one million, than polygons, up to 500, so it is apparent that polygons should be indexed rather than points, using their minimum bounding rectangles. Moreover, the $O(n)$ complexity for the “INSIDE” or “WITHIN” query is not acceptable. Overall, this work seeks to improve upon the $O(n)$ complexity and improve the scalability of geo-fencing by utilizing indexing techniques.

III. Problem Statement and Objective

A. Problem Statement

Given a set of points and a set of possibly overlapping regions (polygons with and without interior rings), the goal is to pair each input point with one or more of the input polygons such that a given spatial condition is satisfied between the polygon and the point. The given spatial condition (spatial predicate) can be as follows:

1) INSIDE: this evaluates to TRUE if a point is inside the polygon. A point can be associated with one or more polygons (i.e., two overlapping polygons can contain the same point).
2) WITHIN n (ex. WITHIN a distance of 1000 units): this evaluates to true if the point is at less than 1000 units distance from the polygon. A point may be associated with one or more of the polygons as the same point can be within 1000 units distance from several polygons.

The input points are associated with an ID and the same point might appear multiple times in the input. So these input points can be thought of as moving points. Each instance of the input point may appear zero or more times in the output. If a point is not associated with any polygon, it does not appear in the output. Different instances of the input point may be associated with different polygons in the output. The polygons are also associated with a unique ID. Moreover, the location of a polygon can change over time: think of them as moving polygons. In the input sequence, the points should always look at the latest position of the polygon to find the right association.

B. Objectives

1) The main objective is to go over all (point, polygon) pairs and output the ones that satisfy the given spatial predicate (inside or within n distance). My methodology seeks to improve upon the naive methodology and accelerate point-polygon topological relationship queries with hierarchical indices. I maintain the use-case provided by the contest in my solution (there are way more points than polygons).
2) In addition to tackling the problem, I also provide techniques that would augment my solution.

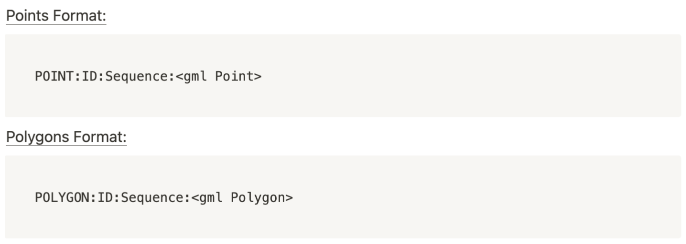
IV. Dataset Description

A. Assumptions

- Data is assumed to be in the spherical Mercator Projection format (the projection used in Google, Bing and OpenStreet maps). That means the calculations will be done in Cartesian space.
- Polygons are valid and are ordered CCW for outer-rings, and CW for inner rings.
- The only possible spatial predicates are: INSIDE and WITHIN n.
- Number of polygons will never be more than 500 polygons.
- Number of points will never be more than 1 Million.

B. Details

The competition provides two files in which the first file contains the input points and the second file contains polygons used to describe the regions. Both the points and the polygons are in GML 2.1.1 format.


Points Format:

```
POINT:ID:Sequence:<gml Point>
```

Polygons Format:

```
POLYGON:ID:Sequence:<gml Polygon>
```

Fig. 3. Point and Polygon Format

C. Timestamps

Consider the following output:

```
1:501:1:1

1:503:1:1

1:504:1:503
```

Note that the third instance of the first point is now associated with the second instance of Polygon 1 (with ID 503). The second point is not inside any of the polygons, so this point does not appear in the output. The sequence numbers are generated so that each instance of the data in a file gets a unique number. Here, we consider the sequence numbers to be logical timestamps as recorded by a system. So, no two points will have the same timestamp and no two polygons will have the same timestamp. When considering the association of points and polygons, the timestamp values should be taken into account. A point can never be associated with a polygon that has a higher (or equal) timestamp value. When a polygon gets a new timestamp value, the old instance of that polygon ceases to exist. In our example, after the timestamp 503 when

Polygon 1 gets a new instance, the old instance of Polygon 1 ceases to exist. So points with timestamp value 504 or higher can only see the new instance of the polygon with the timestamp value 503.

V. Methodology

A. Dataset Preprocessing

To read the point data for analysis, the following step is performed:

- Extract the coordinate values from each point (extracting the X and Y coordinates from each point) by parsing the values between the <gml: coordinates> tags. This was done using *lmxl* and *geopandas* libraries.

To read the polygon data for analysis, the following step is performed:

- Extract the coordinate values from each polygon (extracting the X and Y coordinates from each polygon) by parsing the values between the <gml: coordinates> tags. This was done using *lmxl* and *geopandas* libraries.

Overall, the tools used were *geopandas* and *lmxl* for dataset preprocessing, *rtree* (python library) for constructing the R-trees, and *matplotlib* for visualizing the data.

Through this data reading, it is also possible to convert the data into shapefiles for better visualization in GIS software (some points, polygons from the file shown in below figure).

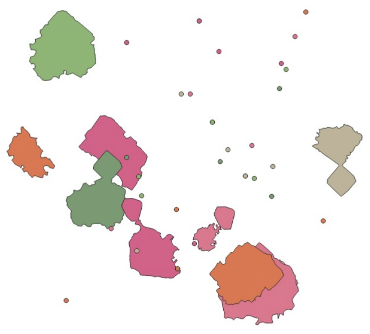


Fig. 4. QGIS Visualization

B. INSIDE Query

As compared to the complexity of the naive solution $O(n)$, my methodology is much faster as it utilizes spatial indices which are the key to achieving optimal performance, especially when querying a large dataset.

I note that the above-mentioned one million points and 500 polygons both may include multiple versions (or timestamps) of the same point or polygon ID, and it is required that each version of each point ID be queried against the latest versions of all polygon IDs (thinking of it as a location-based service where people move around and buildings relocate or reshape over time). Since the number of polygons, even including multiple versions, is several orders of magnitude smaller than the number of points, I built indices for each version of each polygon statically, and then filtered out older versions during the query process. Comparing with the implementation that changes the index as the time progresses, my static indices method is simpler for this use-case. Furthermore, it is also possible to venture into *trajectory-based approaches* which involve modeling the movements of objects over time as trajectories and using these trajectories to perform spatial queries. In the context of point-in-polygon problems with moving polygons, one approach is to model the movement of polygons as continuous trajectories and use these trajectories to efficiently answer point-in-polygon queries.

Steps for INSIDE Query:

1) Build an R-tree index over the set of polygons *(polygon indexing)* -

- Insert each polygon's MBR (minimum bounding rectangle) into the R-tree index, which recursively partitions space into smaller regions (can be done using the *rtree* package in Python).
- The minimum bounding rectangle (MBR) is the smallest rectangle that contains the polygon. To calculate the bounding box of a polygon, we need to find the minimum and maximum values of its x and y coordinates. The minimum values will correspond to the bottom-left corner of the bounding box, and the maximum values will correspond to the top-right corner of the bounding box. The bounds attribute returns a tuple of four values: the minimum x-coordinate, minimum y-coordinate, maximum x-coordinate, and maximum y-coordinate of the polygon.

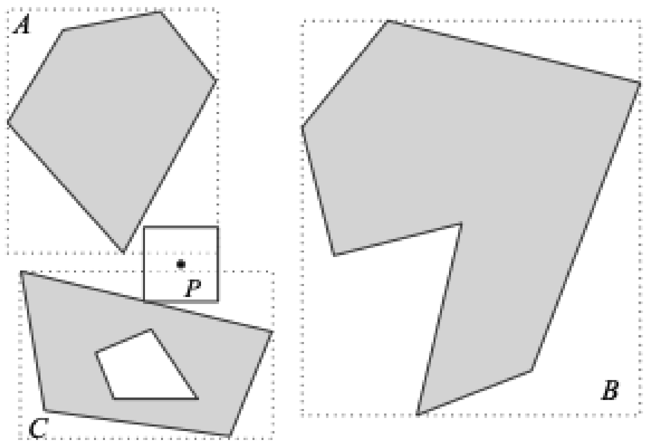


Fig. 5. Polygon Indexing

Figure 5 illustrates a illustrates a polygon index using the minimum bounding rectangle of each polygon. For “INSIDE” queries, we need to find all polygons whose bounding boxes contain the given point. For example, in the figure, no polygon needs to be further examined for the “INSIDE” query of point P.

2) For each query point, traverse the R-tree to identify the set of polygons that intersect with the point.

- R-trees allow you to perform range queries, which can be used to find all polygons that intersect with a given bounding box. For a point, the bounding box is simply a single point. The x and y coordinates of the point will be the same for the bottom-left and top-right corners of the bounding box. In Python, you can use the *idx.intersection()* method to perform the range query.
- After this, we get the set of polygons (candidate polygons) that intersect with the query point. As you can see, the number of polygons that must be checked will typically be lower than in the naive approach.

3) Skip if the polygon is not the latest version with respect to the point’s timestamp.

- When considering the association of points and polygons, the timestamp values should be taken into account.
- A point can never be associated with a polygon that has a higher (or equal) timestamp value.
- When a polygon gets a new timestamp value, the old instance of that polygon ceases to exist.

4) For each candidate polygon, perform the *ray casting algorithm* to determine whether the point lies within it.

5) Output the (point, polygon) pairs that satisfy the INSIDE problem.

Time Complexity:

- In the first step, I built an R-tree for polygon indexing. Generally, the time complexity of building an R-tree is $O(nlogn)$, where n is the number of objects being indexed.
- Now for each point, I’m traversing the R-tree and seeing which polygons intersect it. Searching through an R-tree has a time complexity of $O(logn+m)$, where n is the number of entries in the R-tree, and m is the number of entries that match the search criteria. For each intersecting polygon, I perform the ray casting algorithm. The time complexity for this would be $O(e)$.

Hence, the overall time complexity for my methodology is $O(p*logp+n*((logp+m)+me))) = O(n*m*e)$ where p is the number of polygons, n is the number of points, m is the number of intersecting polygons, and e is the average number of edges across all polygons.

The naive solution has a time complexity of $O(n*p*e)$ and my solution has a time complexity of $O(n*m*e)$ which is significantly better.

Code Snippet (Shortened):

```
# Creating R-tree index using the MBR's
of the polygons
idx = rtree.index.Index()
for i, polygon in enumerate(polygons):
    idx.insert(i, polygon.bounds)

# Performing a range query for each point
for index, point in enumerate(points):
   bbox = point.bounds
   intersected_polygons = [polygons[i] for
   i in idx.intersection(bbox)]
   for polygon in intersected_polygons:
        if ray_casting(point,intersected_polygons):
          return true
```

In the above shortened code snippet, we can see that the *rtree* library is used to create the R-tree. Then, *polygon.bounds* is utilized to create the MBR of the polygon. Finally, *idx.intersection* is used to perform the range query and there is a ray casting algorithm (not mentioned) to do the final point-in-polygon check.

C. WITHIN n Query

Steps for WITHIN n Query:

1) Create a 2n x 2n bounding box for each point (x, y). This is done by subtracting n from both x and y. The lower-left corner would have coordinates (x - n, y - n) and the upper-right corner of the bounding box would have coordinates (x + n, y + n).

2) Using the R-tree for polygon indexing, find all polygons whose bounding boxes intersect the bounding box of the point. These are the initial candidate polygons (same as step 2 in the INSIDE problem). As seen in figure 5, for “WITHIN n” queries, we construct a 2n × 2n square centered at the given point and find all polygons whose bounding boxes intersect the square. For example, in the figure, polygons A and C must be considered for the “WITHIN n” query.

3) Skip if the polygon is not the latest version with respect to the point’s timestamp.

- When considering the association of points and polygons, the timestamp values should be taken into account.
- A point can never be associated with a polygon that has a higher (or equal) timestamp value.
- When a polygon gets a new timestamp value, the old instance of that polygon ceases to exist.

4) For these initial candidate polygons, find the ones in which the point is not inside the polygon (can be done with ray casting algorithm). This is because the points inside the polygon cannot possibly be ‘within n’ distance from it.

5) For the remaining polygons (final candidate polygons in which the point is not inside the polygon), create an R tree for each that does edge indexing.

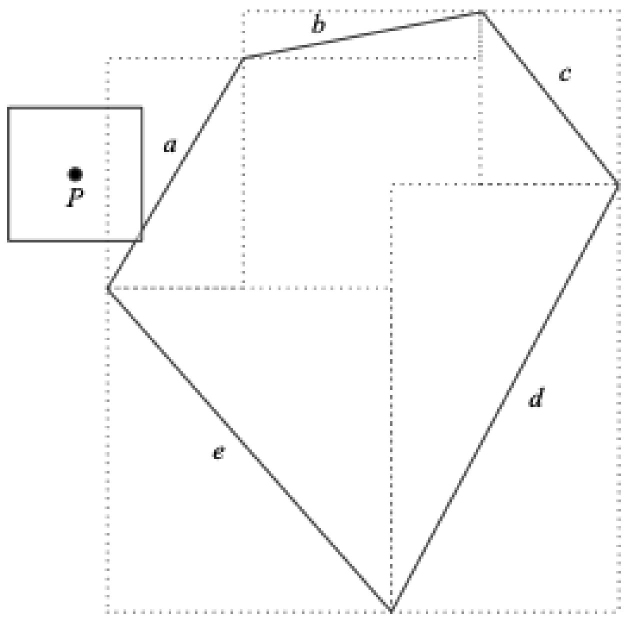


Fig. 6. Edge Indexing

To do edge indexing, we find the minimum bounding rectangle for each edge. Again, we construct a 2n × 2n bounding box to query all edges whose minimum bounding boxes intersect it. In this example, only edge a is returned by the query for the given point P. (shown in figure above)

6) Finally, with the 2n x 2n bounding box for the point, perform a range query on the R tree to check if the point is 'within n' distance from the polygon. (I assume that on the edge of the polygon is considered not inside)

A naive solution would check if a point is 'within n' distance for all (point, polygon) pairs. However, my methodology first filters out the polygons for which the point cannot be 'within n' distance from and then checks for the final candidate polygons. Therefore, it is much less computationally expensive as compared to a naive approach.

Code Snippet (Shortened):

```
# Creating R-tree index using the MBR's
of the polygons
idx = rtree.index.Index()
for i, polygon in enumerate(polygons):
    idx.insert(i, polygon.bounds)

# Defining distance threshold
n = 4000

# Finding candidate polygons for each point
# We are interested in not_in_polygons
for index, point in enumerate(points):
    # Create 2n x 2n bounding box for point
    bbox = (point.x - n, point.y - n,
    point.x + n, point.y + n)
    intersected_polygons = polygons in
    idx.intersection(bbox)
    candidate_polygons = intersected_poly with
    ray_casting[point, polygon]
    not_in_polygons = intersected_poly not
    with ray_casting[point, polygon]

# Creating an R-tree index for each not-in-polygon
    for polygon in not_in_polygons:
        edge_idx = rtree.index.Index()
        for i, segment in enumerate(segments):
            edge_idx.insert(i, segment.bounds)

        # Performing edge-based queries
        using the index
        intersections = [segment for segment in
        edge_idx.intersection(point.bounds]
        if intersections:
            return true
        else:
            return false
```

In the above shortened code snippet, we first utilize the *rtree* library to create the MBR's of the polygons. After that, we define the distance threshold 'n'. Next, we find the candidate polygons for each point (*not in polygons* in the code). Finally, edge-indexing is performed on these *not in polygons* and the corresponding points 'within n' distance are returned.

VI. Results

The ACM GIS Cup 2013 provides a training dataset which includes two point files (Point500 and Point1000) and two polygon files (Poly10 and Poly15). I conducted multiple experiments by considering each predicate and selecting (point, polygon) combinations. The main focus was to compare execution time with and without indexing as well as verify the correctness of the solution.

A. INSIDE Query

I utilized the points and polygons from the files provided by the cup to confirm that my algorithm was valid.

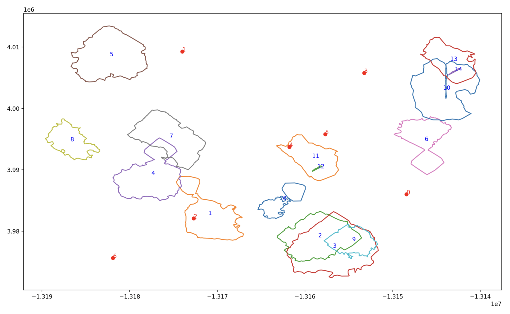


Fig. 7. Some Points, Polygons Read from File

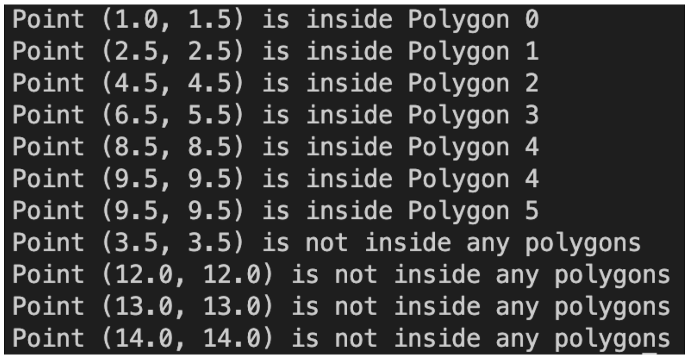

```
Point (1.0, 1.5) is inside Polygon 0
Point (2.5, 2.5) is inside Polygon 1
Point (4.5, 4.5) is inside Polygon 2
Point (6.5, 5.5) is inside Polygon 3
Point (8.5, 8.5) is inside Polygon 4
Point (9.5, 9.5) is inside Polygon 4
Point (9.5, 9.5) is inside Polygon 5
Point (3.5, 3.5) is not inside any polygons
Point (12.0, 12.0) is not inside any polygons
Point (13.0, 13.0) is not inside any polygons
Point (14.0, 14.0) is not inside any polygons
```

Fig. 8. Results from my Algorithm

As my use-case considered that there were many more points than polygons, I utilized static indices in the problem. In my implementation, I considered all polygons to be different and timestamps from the problem statement are not dealt with. However, as referred to in step 3 in my algorithm, polygons can easily be skipped if they are not the latest version with respect to the point's timestamp. My main focus was the point-in-polygon approach and these intricacies may be added later. My results were found to be 100 percent accurate. The difference between the naive methodology (without indexing) and my methodology (with indexing) is highlighted in the figure below:

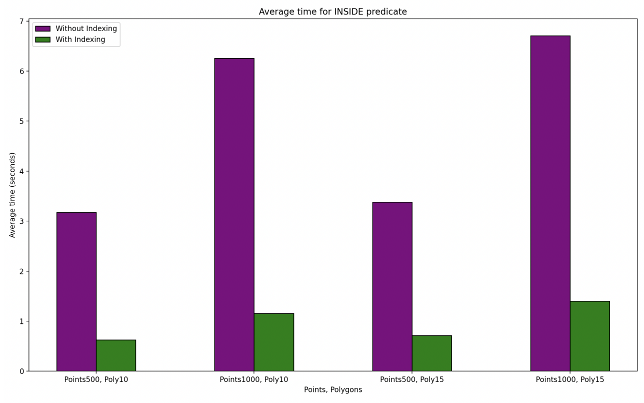


Fig. 9. Average time taken to pair points and polygons

As presented, there is a significant time difference between the naive methodology and my methodology (utilizing hierarchical indices).

B. WITHIN n Query

I utilized the points and polygons from the files provided by the cup to confirm that my algorithm was valid.

In figure 10, the distance threshold was set to 6000. In addition, the statement *'Point 4 is not within 6000 from polygon 1'* indicates that the point passed steps 1-5 in my algorithm but failed in step 6. Therefore, point 4's bounding box intersected with polygon 1's bounding box (and it is not inside), but it failed to intersect with

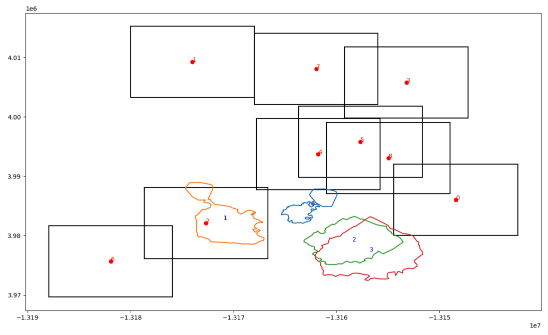


Fig. 10. Some Points, Polygons Read from File

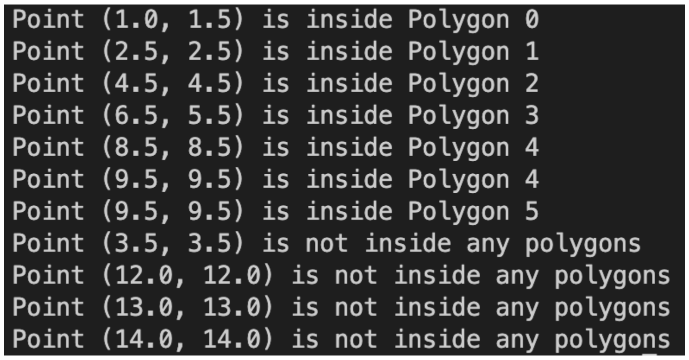

```
Point (1.0, 1.5) is inside Polygon 0
Point (2.5, 2.5) is inside Polygon 1
Point (4.5, 4.5) is inside Polygon 2
Point (6.5, 5.5) is inside Polygon 3
Point (8.5, 8.5) is inside Polygon 4
Point (9.5, 9.5) is inside Polygon 4
Point (9.5, 9.5) is inside Polygon 5
Point (3.5, 3.5) is not inside any polygons
Point (12.0, 12.0) is not inside any polygons
Point (13.0, 13.0) is not inside any polygons
Point (14.0, 14.0) is not inside any polygons
```

Fig. 11. Results from my Algorithm

the edge indices of the polygon. As mentioned before, I utilized static indices in the problem (due to my use-case) and timestamps from the problem statement are not dealt with. My main focus was on the approach and further intricacies can be implemented later. My results were found to be 100 percent accurate. The difference between the naive methodology (without indexing) and my methodology (with indexing) is highlighted in the figure below:

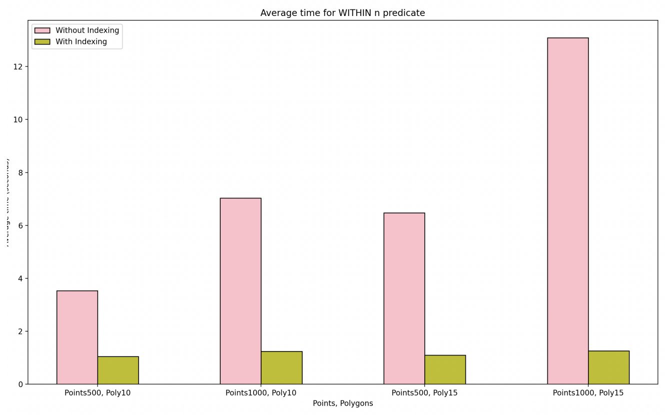


Fig. 12. Average time taken to pair points and polygons

As presented, there is a significant time difference between the naive methodology and my methodology (utilizing hierarchical indices).

VII. Improvements

A. Interval Indexing

In my solutions, I directly used the ray casting algorithm when I wanted to check if a point was inside a polygon. However, this can be computationally expensive. To speed up this process, *interval indexing* can be used.

Once we have constructed the interval trees for all polygons, we can use them to efficiently determine which polygons contain a given point (x, y). To do this, we cast a horizontal ray along the x-axis from the point, and then look for the intervals in the interval trees that contain the y-coordinate of the point. We only need to consider the edges corresponding to these intervals, since they are the only ones that could possibly intersect the ray.

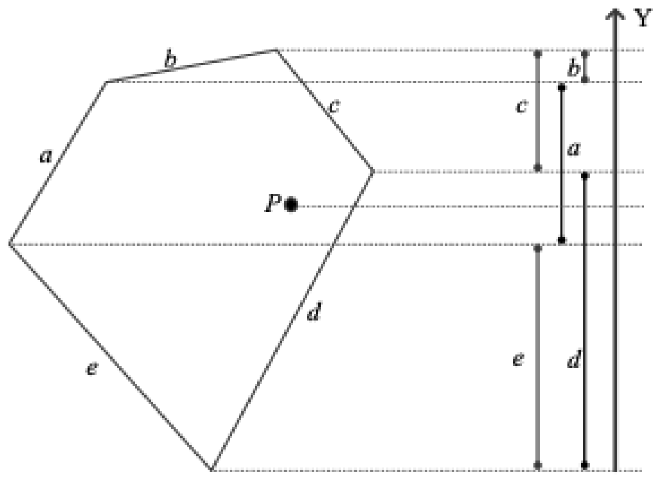


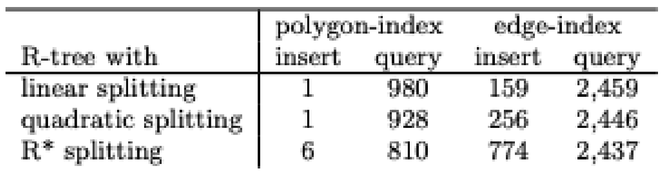
Fig. 13. Interval indexing involves creating an interval tree for each polygon, which includes all the projected intervals of the polygon's edges on the y-axis. Each interval in the tree represents a portion of the edge that lies between two y-coordinates. For example, in this figure, only edges a and d are considered for the "INSIDE" query of point P.

By using interval indexing, we can greatly reduce the number of edges that we need to consider for each query, which can result in significant performance improvements for large datasets.

B. R-tree variants

In the *rtree* library in Python, *quadratic splitting* algorithm is used to split nodes when they become too full. For polygon indexing, *R* trees* are found to be better because it seeks to minimize the overlap between the two resulting child nodes and also seeks to curtail the area of the bounding boxes of the child nodes.

For edge indexing, *linear splitting* is found to be better because it helps to evenly distribute the data across the index, resulting in a more balanced and efficient search.



	polygon-index		edge-index	
R-tree with	insert	query	insert	query
linear splitting	1	980	159	2,459
quadratic splitting	1	928	256	2,446
R* splitting	6	810	774	2,437

Fig. 14. Higher numbers indicate higher times

VIII. Conclusion

Overall, this paper's aim was to tackle the ACM SIGSPATIAL CUP 2013 problem which focused on two spatial predicates: INSIDE and 'WITHIN n'. I oriented my approach around the use-case that the number of points is significantly more than the number of points. In the future, I hope to create a more versatile algorithm that takes into account a use-case in which there are way more polygons than points. In such a case, utilizing static indices would be computationally expensive and timestamps of polygons would have to be dealt with in a different manner. Through my analysis, the idea behind geo-fencing became much more vivid. For example, users cam enter or exit geo-fences based on geo-fencing-enabled location preferences in mobile apps when notifications (e.g., advertisement territory, shopping mall, university campus) are sent to users or their network of friends. Ultimately, by analyzing the patterns of the training dataset and the performance of different indices, I found that indices are the key to top performance, and it is worthwhile to build a hierarchy of indices in exchange for optimal query performance.[1] [2] [3] [4] [5]

References

[1] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data - SIGMOD '84*, page 47, Boston, Massachusetts, 1984. ACM Press.

[2] Kai Hormann and Alexander Agathos. The point in polygon problem for arbitrary polygons. *Computational Geometry*, 20(3):131–144, Nov 2001.

[3] Suikai Li, Weiwei Sun, Renchu Song, Zhangqing Shan, Zheyong Chen, and Xinyu Zhang. Quick geo-fencing using trajectory partitioning and boundary simplification. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, page 580–583, Orlando Florida, Nov 2013. ACM.

[4] Yi Yu, Suhua Tang, and Roger Zimmermann. Edge-based locality sensitive hashing for efficient geo-fencing application. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, page 576–579, Orlando Florida, Nov 2013. ACM.

[5] Tianyu Zhou, Hong Wei, Heng Zhang, Yin Wang, Yanmin Zhu, Haibing Guan, and Haibo Chen. Point-polygon topological relationship query using hierarchical indices. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, page 572–575, Orlando Florida, Nov 2013. ACM.