Point-Polygon Topological Relationship Query using Hierarchical Indices

Tianyu Zhou Hong Wei Heng Zhang
Shanghai Jiao Tong University
Shanghai, China
{zhoutianyu007,keith.collens,shinedark}
@sjtu.edu.cn

Yin Wang
Facebook
Menlo Park, CA, USA
yinwang@fb.com

Yanmin Zhu Haibing Guan
Haibo Chen
Shanghai Jiao Tong University
Shanghai, China
{yzhu,hbguan,haibochen}
@sjtu.edu.cn

ABSTRACT

This paper describes a point-polygon query program we submitted to the ACM SIGSPATIAL Cup 2013. Point-polygon topological relationship query is one of the core functions for commercial spatial databases, and also an active research topic in academia. Spatial indices are the key to achieve top performance. However, different datasets or query patterns require different indices for optimal performance. Based on the patterns of the training dataset, we build a hierarchy of indices, including polygon index, edge index, and interval index, which 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. Using the provided training dataset, these three indices reduce the computation time of "WITHIN n" query by 90%, 10%, and 50%, respectively. We build a large dataset with more than 1 million samples and 520 polygons by cloning and offsetting the training dataset 15 and 13 times, respectively. Our program finishes the "WITHIN 1000" query in only one second on a 4-core 3.3GHz Xeon Processor.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Applications—*Spatial databases and GIS*

General Terms

Algorithms, Experimentation, Performance

Keywords

point-in-polygon, spatial index, spatial query

1. INTRODUCTION AND OVERVIEW

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 two classical algorithms are 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, and the winding number algorithm, which computes the given point's *winding number* with respect to the polygon [7]. 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) of all the above algorithms 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 [4, 5]. However, 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. Figure 1 visualizes the

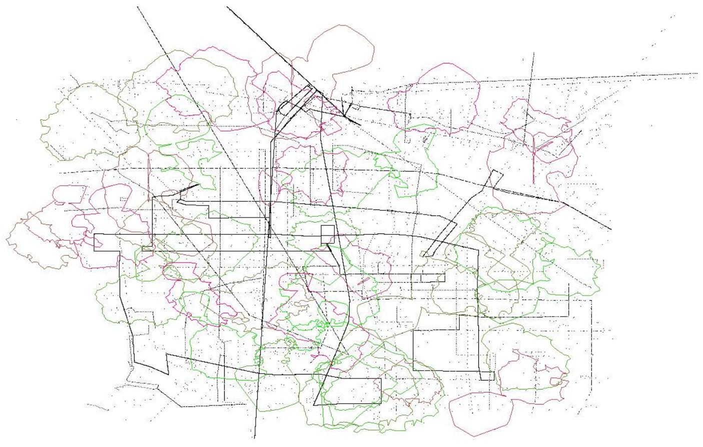
Figure 1: Visualization of the training dataset. Black points and traces represent points and their "movement". Polygons are shown in color.

SIGSPATIAL'13, Nov 05-08 2013, Orlando, FL, USA
ACM 978-1-4503-2521-9/13/11. http://dx.doi.org/10.1145/2525314.2527263

points and polygons in the provided training dataset. There are way more points, up to one million, than polygons, up to 500, so it is apparent that we should index polygons rather than points, using their minimum bounding rectangles; see Figure 2. Another crucial observation is that each polygon has lots of edges, 226 on average in the training dataset. The $O(n)$ complexity for the "INSIDE" or "WITHIN" query is not acceptable. We therefore build one edge index for each polygon using the minimum bounding rectangles of the edges, as shown in Figure 3. Therefore, "WITHIN n" query now only needs to iterate over those edges whose envelopes are within n to the given point. Finally, to speed up "INSIDE" query, we build one interval index for each polygon that includes all the projected intervals of its edges on the y-axis, as shown in Figure 4. Then, using the ray casting algorithm with a horizontal ray along the x-axis, we just need to consider those edges whose projected intervals contain y for a given point (x, y).

We 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, we build all three indices for each version of each polygon statically, and then filter out older versions during the query process. Comparing with the implementation that changes the index as the time progresses, our static indices allow simpler and more efficient implementation for concurrent queries using multi-threading.

In summary, our point-polygon relationship query algorithm consists of the following steps:

1. for each point P, query the polygon index to find polygons whose envelopes contain P (or within n to P if it is "WITHIN n" query). For each polygon returned by the query, perform the following steps in sequence;
2. skip if the polygon is not the latest version with respect to P's timestamp;
3. query whether P is inside the polygon using ray casting and the interval index;
4. for "WITHIN n" query, if P is not inside the polygon, calculate their distance using the edge index.

We discuss our hierarchy of indices and other optimization techniques in Section 2 and our experimental result in Section 3. Section 4 concludes this paper.

2. HIERARCHICAL INDICES

R-tree [3] and its variants are commonly used to index various geometric shapes by their minimum bounding rectangles. Minimization of both *coverage* and *overlap* is crucial to the performance of R-trees. Coverage is the entire area covered by tree nodes, and overlap is the area covered by two or more nodes. Minimizing coverage reduces the amount of "dead space" covered by the R-tree, and minimal overlap reduces the set of search paths to the leaves. Different R-tree variants employ different heuristics to split overflowing nodes, trying to minimize coverage and overlapping [6].

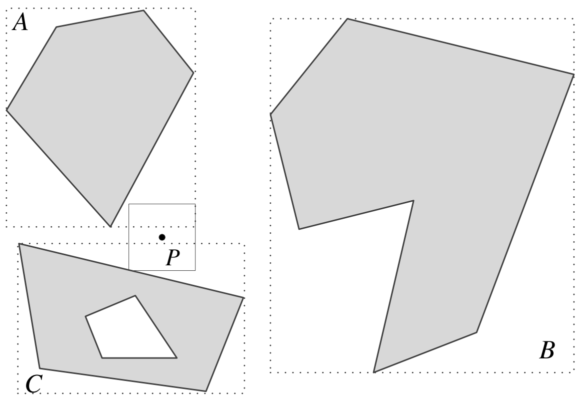

Figure 2: Polygon index example

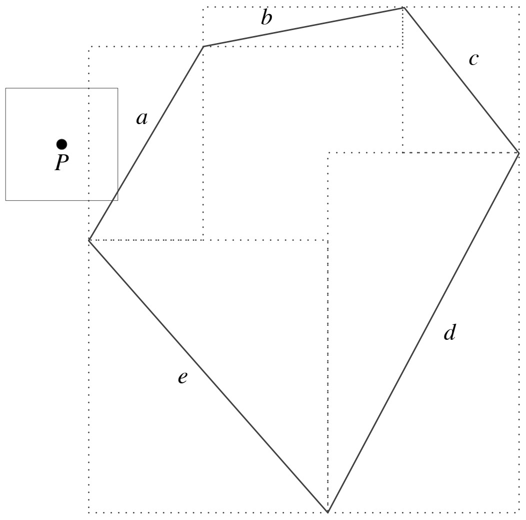

Figure 3: Edge index example

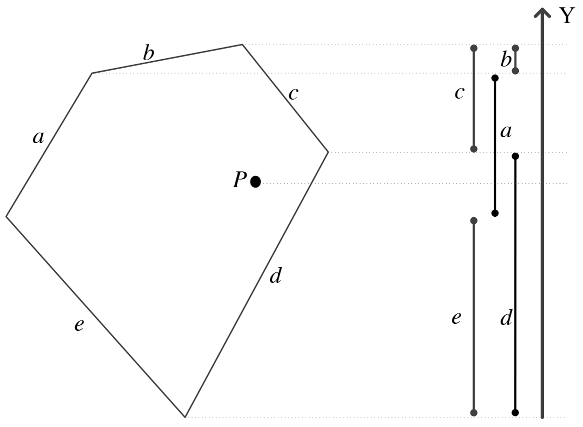

Figure 4: Interval index example

	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

Table 1: Computation time (ms) of R-tree variants using the large dataset we constructed.

Since the competition considers the overall computation time, we need to calculate both the tree construction and the query time. Intuitively, more advanced heuristics take longer time for tree construction but less for query. We compare different splitting strategies using the training dataset and select the best performers for our polygon index and edge index, respectively. For our interval index, we use *interval tree* which is essentially a degenerated R-tree for one dimension.

2.1 Polygon Index

Figure 2 illustrates our 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 "WITHIN n" queries, we construct a $2n \times 2n$ square centered at the given point, shown in the figure, and find all polygons whose bounding boxes intersect the square. For example in Figure 2, no polygon needs to be further examined for the "INSIDE" query of point P, and polygons A and B must be considered for the "WITHIN n" query.

We compare three node splitting strategies implemented in boost C++ library 1.54, linear split, quadratic split [3], and R*-tree topological split [1]. Table 1 shows the tree building time and the query time using our large dataset (see Section 3.1), which is consistent with the example computation time shown on boost documentation website, i.e., insert time increases in the order of linear splitting, quadratic splitting, and R*-tree splitting, while the query time decreases in the same order. However, the difference of tree building time among all three algorithms is negligible compared to the query time for the polygon index, because we have several orders of magnitude more points than polygons. It is worth the time to find as optimal splitting strategy as possible during tree construction in order to save query time. Therefore we use R*-tree for our polygon index.

2.2 Edge Index

Figure 3 shows our edge index using the minimum bounding rectangle of each edge. Again we construct a $2n \times 2n$ square to query all edges whose minimum bounding boxes intersect it. In this example, we only edge a is returned by the query for the given point P. We build one edge index for each version of each polygon to minimize the index building and query time.

Table 1 shows the total time for building all edge indices and queries using different splitting strategies. In this case, on the other hand, the difference of query time among different strategies is negligible compared to the insert time. We suspect the reason is that the edges of a polygon are much more evenly distributed over the space and there is very little overlapping. All three splitting strategies work well and therefore the query time is similar. R*-tree splitting spent more time on optimizing but the benefit is marginal. We use R-tree with linear splitting for our edge index.

2.3 Interval Index

Given a point and a polygon, the ray casting algorithm draws a ray from the point, going along any fixed direction, and tests how many times it intersects the edges of the polygon. If the point is not on any edge of the polygon, the point is inside the polygon if the number of crossings is an odd number, or outside if it is even. The result holds for polygons with one or more inner rings.

To simplify the calculation, it is common to construct the ray in parallel with the positive x-axis. The interval index is employed to quickly find edges whose projected intervals on y-axis contain the y coordinate of the point. Edges not returned by the query are either above or below the ray, and therefore do not intersect. (Here we define "contain" as such that one endpoint of the edge must be below the given point P and the other endpoint must be equal to or above P, which handles the corner case where the ray intersects a vertex of the polygon or completely overlaps an edge.) For example in Figure 4, we consider only edges a and d for the "INSIDE" query of point P.

Interval tree is an ordered binary tree data structure for storing and retrieving intervals. More specifically, it allows efficient query of intervals that overlap with any given interval or point. We use *centered interval tree* that is a binary tree where each node stores all intervals intersecting a *center point*. All intervals completely to the left of the center point are stored on the left subtree, and intervals to the right of the point are stored on the right subtree [2]. We build one interval tree for each version of each polygon. The intervals inserted into the tree are the projections of its edges to the y-axis, i.e., the y coordinates of the two endpoints of each edge, called y-intervals. For a given point P, the query returns all y-intervals that intersect its y coordinate. Edges whose y-intervals do not intersect the y coordinate of P are either above or below the x-parallel ray from P, and therefore excluded from the intersection counting. For those edges returned by the query, we further filter out edges whose x coordinates of both endpoints are less than the x coordinate of P, which are completely on the left of the ray. Likewise, edges whose x coordinates are both greater than x must intersect the ray so there is no need to perform the expensive intersection calculation either. Only those edges whose endpoints are on both sides of P with respect to x coordinates are included in the intersection calculation. For example in Figure 4, first our interval index filters out edges b, c, e, and then we skip edge a since it is on the left of P. The intersection calculation between edge d and the x-parallel ray from P returns true so the intersection count of P is one, and therefore it is inside the polygon.

2.4 Other Optimizations

In addition to the hierarchy of indices, we use multithreading for input parsing, index building, and query processing. For input parsing, we read the entire file at once and each thread is responsible for a portion of the file. These threads also build the edge and interval indices in parallel as they parse the input. Only the polygon index is built with a single thread, but its construction time is negligible as illustrated in Table 1. As mentioned in Section 1, all our indices are static (read-only) since we filter out older versions of polygons in the query results. Therefore concurrent query is straightforward to implement using a work queue.

There are other minor optimization techniques. For example, we simply scan and skip a fixed number of characters for each input line to find the coordinate strings, based on the training data format. This is at least an order of magnitude more efficient than a full-XML or GML compatible parser. For each coordinate string, we use our customized double parser instead of `atof` for faster conversion [8].

3. EVALUATION

We built our program in C++ with the boost library for R-tree index implementation. Our evaluation focuses on the computation time comparison and analysis since the accuracy is always 100% for the training dataset. The evaluation was performed on a 4-core 3.3 GHz Xeon E3-1230 processor, running 64-bit Windows 7.

3.1 Large Training Dataset

The provided training dataset is rather small for our program. Using only one thread, our program finishes both "INSIDE" and "WITHIN 1000" queries for the `points1000.txt` and `poly15.txt` dataset in less than a half second. Therefore we built our customized large dataset by cloning and offsetting the the training dataset multiple times. We chose cloning with offsetting instead of a custom-built random generator because we want to ensure the dataset has the same pattern such that the index performance would be similar. Since we are informed that the testing dataset will not exceed one million points or 500 polygons, we clone `1000points.txt` (with 69,619 point instances) 15 times and `15poly.txt` (with 40 polygon instances) 13 times, resulting in a dataset with 1.04 million points and 520 polygons.

3.2 Index Performance

Table 2 shows the program execution time with different indices, using both the provided training dataset of `points 1000.txt` and `poly15.txt`, and our large dataset. We use only a single thread and turn off other optimizations in order to measure the difference more accurately.

From this table, we can see the largest performance gain is from the polygon index, reducing computation time by more than 60% for "INSIDE" queries and more than 90% for "WITHIN 1000" queries. We found that at most five or seven polygons are returned by "INSIDE" and "WITHIN" queries, respectively, which is a significant reduction from the total number of polygons that need to be examined otherwise. The edge index is used by "WITHIN" queries only and it reduces the computation time by 10% on the provided training dataset. The interval index reduces the computation time by another 50% on the provided training dataset. Using our cloned large dataset, the effect of the polygon index is more prominent because the polygons are naturally divided by different clone copies, which the R-tree index can exploit for better node splitting.

With our parsing optimization, the computation time using a single thread for the "WITHIN 1000" query reduces to 4.8 second for our large dataset. We obtained the measurement by running the program multiple times so the input data is fully cached in memory to eliminate the disk read variance. Table 3 shows the scaling efficiency of our program using multiple threads. There are various serialized execution sections of our program which prevent linear scaling, including input reading and output writing, polygon index building, and work queue construction and consumption. In addition, there are four physical cores and four hyperthreading cores, so the performance gain beyond four threads is marginal.

dataset	indices	INSIDE	WITHIN 1000
poly15 + points1000	none	2.97	8.53
	polygon	0.66	0.80
	polygon + edge	0.66	0.71
	all three	0.30	0.34
poly520 + points1M	none	347.50	1,085.00
	polygon	13.89	37.56
	polygon + edge	14.04	20.45
	all three	8.25	13.75

Table 2: Total execution time (sec) with different indices using both the provided and our large datasets (single thread, without parsing optimization).

threads	1	2	3	4	5	6	7	8
time (s)	4.80	2.71	1.93	1.64	1.39	1.25	1.09	1.03

Table 3: Multithreading for "WITHIN 1000" query using our large dataset (with parsing optimization)

4. CONCLUSION

By analyzing the patterns of the training dataset and the performance of different indices, we found that indices are the key to top performance, and it is worthwhile to build a hierarchy of indices in exchange for the optimal query performance. We first filter out distant polygons using an R*-tree based polygon index, and then filter out distant or irrelevant edges using both an R-tree index and an interval tree index. Together with parsing optimizations and multithreading, our program computes the "WITHIN 1000" query for the provided dataset in less than 100ms and for our customized large dataset with one million points and 520 polygons in just one second, on a 4-core 3.3GHz Xeon Processor.

5. REFERENCES

[1] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. *The R*-tree: an efficient and robust access method for points and rectangles*, volume 19. 1990.

[2] M. De Berg, M. Van Kreveld, M. Overmars, and O. C. Schwarzkopf. *Computational geometry*. Springer, 2000.

[3] A. Guttman. *R-trees: A dynamic index structure for spatial searching*, volume 14. ACM, 1984.

[4] Y. Hu, S. Ravada, and R. Anderson. Geodetic point-in-polygon query processing in oracle spatial. In *International Conference on Advances in Spatial and Temporal Databases*. 2011.

[5] Y. Hu, S. Ravada, R. Anderson, and B. Bamba. Topological relationship query processing for complex regions in oracle spatial. In *SIGSPATIAL GIS*, 2012.

[6] A. Nanopoulos, A. N. Papadopoulos, and Y. Theodoridis. *R-trees: Theory and Applications*. Springer, 2006.

[7] I. E. Sutherland, R. F. Sproull, and R. A. Schumacker. A characterization of ten hidden-surface algorithms. *ACM Comput. Surv.*, 6(1):1–55, Mar. 1974.

[8] H. Wei, Y. Wang, G. Forman, Y. Zhu, and H. Guan. Fast Viterbi map matching with tunable weight functions. In *GIS*, 2012. (SIGSPATIAL Cup).