Edge-Based Locality Sensitive Hashing for Efficient Geo-Fencing Application

Yi Yu
School of Computing, National University of Singapore
Singapore 117417
yuy@comp.nus.edu.sg

Suhua Tang *
ATR Adaptive Communications Research Laboratories
Kyoto 6190288 Japan
shtang@atr.jp

Roger Zimmermann
School of Computing, National University of Singapore
Singapore 117417
rogerz@comp.nus.edu.sg

ABSTRACT

Geo-fencing is a promising technique for emerging location-based services. Its two basic spatial predicates, INSIDE and WITHIN pairings between points and polygons, can be addressed by state-of-the-art methods such as the crossing number algorithm. In the era of big-data, however, geo-fencing has to process millions of points and hundreds of polygons or even more in real-time. In this paper, we propose an efficient algorithm to improve the scalability of geo-fencing, which consists of two main stages. At the first stage, an R-tree is used to quickly detect whether a point is inside the minimum bounding rectangle of a polygon. In the second stage, instead of an exhaustive search, we design an edge-based locality sensitive hashing scheme adapted to the crossing number algorithm. As for the case of WITHIN detection, a probing scheme is suggested to locate adjacent buckets so as to check all edges near to a target point. By further exploiting batch processing and multi-threading programming, our algorithm can achieve a fast speed while retaining 100% accuracy over all training datasets provided by the GIS Cup 2013 organizers.

Categories and Subject Descriptors

H.2.8 [**Database Applications**]: Spatial Databases and GIS; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval

General Terms

Algorithms, Performance, Theory

Keywords

Geographic information systems, Geo-fencing, Locality sensitive hashing, Multi-probing

*The first two authors have the same contribution.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author. Copyright is held by the author/owner(s).
SIGSPATIAL'13, Nov 05-08 2013, Orlando, FL, USA
ACM 978-1-4503-2521-9/13/11.
http://dx.doi.org/10.1145/2525314.2527264.

1. INTRODUCTION

Advancements in positioning techniques and mobile communications have enabled emerging Location Based Services (LBSs), which have been integrated into social networking platforms (e.g., Facebook, Twitter, Foursquare). The wide spread of smartphones also generates a growing need for user-centric geo-fencing applications [1] in LBSs. Various geo-fencing services (e.g., Placecast, Sensewhere, Zentracker) have been established to meet these needs.

The geo-fencing problem posed by the GIS Cup 2013 [2] 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. Hereafter, a point P consists of $P.ID$, $P.seq$, $P.x$, $P.y$, and a polygon $Poly$ consists of $Poly.ID$, $Poly.seq$ and multiple rings composed of edges. Although state-of-the-art methods such as the crossing number algorithm [3] can address the two key spatial predicates of geo-fencing, it is difficult to realize a real-time geo-fencing service with millions of points and hundreds of polygons.

This work improves the scalability of geo-fencing by exploiting approximate and indexing techniques. At the time of extending the ideas of locality sensitive hashing (LSH) [4] to quickly locate potential geo-edges over a geographic dataset, we were not aware of any related work, which specifically relates to utilizing LSH in geo-fencing research. However, the interested readers may refer to [5][6] for some LSH-based data access methods. In particular, we first apply an R-tree to quickly check whether a point is inside the minimum bounding rectangle (MBR) of a polygon. Then, points are matched against polygons using the crossing number algorithm, which is accelerated by using edge-based locality sensitive hashing. We divide the whole x coordinate range of polygon vertices into non-overlapping sub-ranges, each associated with a bucket. An edge belongs to a bucket if its x coordinate overlaps the x sub-range of the bucket. A probing scheme is also proposed to implement WITHIN detection for finding all edges close to a target point. Our experimental results over the training datasets provided by the GIS Cup 2013 organizers verify the efficiency of the proposed algorithm with a 100% pairing accuracy. Related codes for geo-fencing can be downloaded from the following URL `http://eiger.ddns.comp.nus.edu.sg/~yiyu/GISCup13.html`.

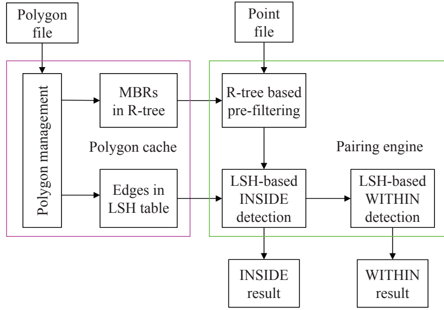

Figure 1: The proposed framework for geo-fencing.

2. BASIS OF PROPOSED ALGORITHM

Detecting whether a point is inside a polygon is not a new issue and can be realized by exploiting the crossing number algorithm [3]. The challenge, in the era of big-data, is how to do this efficiently when there are millions of points and hundreds of polygons. An investigation shows that each polygon in the contest on average contains around 200 edges. Performing the crossing number algorithm by checking each edge of each polygon is time-consuming.

Basic idea. We solve this problem by leveraging different approximate and indexing techniques, as shown in Fig. 1. An R-tree [7] is used to quickly detect whether a point is inside the MBR of any polygon. When a point is outside the MBR of a polygon, it is true that this point is outside the polygon as well. But a further examination is necessary when a point is inside the MBR of a polygon. In this step, only some of edges of a candidate polygon are examined by applying LSH.

R-tree [7]. The key idea of applying an R-tree is to use different levels of approximation of polygons. At the lowest level (leaf node), a polygon is represented by its own MBR. Then, nearby polygons are grouped together and represented by their MBR at the next higher level of the tree. In times of searching, a query (a point for INSIDE, a rectangle for WITHIN) that does not intersect with the MBR will not intersect with any of the polygons inside the MBR. This method helps to quickly remove non-relevant polygons.

LSH for INSIDE detection. The crossing number algorithm [3] states that a point is inside a ring if a ray from this point towards infinity crosses an odd number of edges of the ring. In our design, the ray is along the vertical line, from a point towards infinity. In the ideal case, only edges crossing the ray need to be examined. This is approximated by LSH in our design, and its basic idea is shown in Fig. 2. The x-coordinate of a polygon spans a range, e.g., $[x_0, x_N)$, which is equally divided into N sub-ranges, $[x_0, x_1), [x_1, x_2), \cdots, [x_{N-1}, x_N)$. Each sub-range is associated with a bucket storing edges and $[x_i, x_{i+1})$ corresponds to the ith bucket B_i. An edge whose x-range overlaps a sub-range is added to the associated bucket. In this way, an edge can appear in multiple buckets.

A well-known problem of LSH is the bias of samples in the buckets. This problem degrades LSH efficiency and occurs because overall samples are not uniformly distributed. Instead of managing all edges of all polygons in one hash table, we decide to use one hash table for each polygon. The number of buckets inside each hash table is fixed to N, so that the number edges examined inside a polygon will be reduced by an order N. In this way, the LSH performance does not rely on the distribution of polygons. In addition,

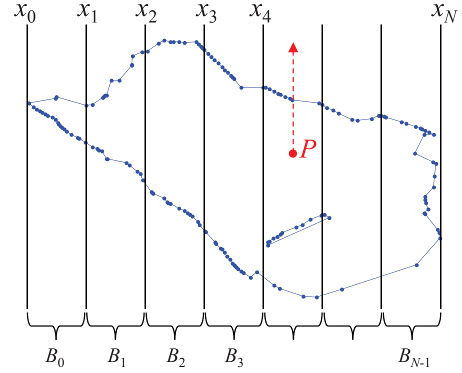

Figure 2: Hash table for organizing edges.

per-polygon LSH works well with the R-tree method.

LSH for WITHIN detection. The above LSH scheme for INSIDE detection need further improvement for WITHIN detection with a distance threshold d_{th}. In the R-tree detection, instead of a point P, the rectangle $Rect = (P.x - d_{th}, P.y - d_{th}, P.x + d_{th}, P.y + d_{th})$ should be used as a query. Then, multi-probing [6] is adopted in order to ensure the accuracy. More specifically, instead of only checking the bucket containing P, all buckets covered by the sub-range $[P.x - d_{th}, P.x + d_{th})$ are examined.

A point inside a polygon is also within a distance d_{th} of the polygon. In the following, we consider the scenario where a point P is outside the polygon, with two cases. (i) P is inside an internal ring of the polygon. Then, only edges of the specific internal ring should be further examined. (ii) P is outside the external ring of the polygon, but within a distance d_{th} of the MBR. Let the left-, top-, right- and bottom-most vertices of the external ring of a polygon be P_L, P_T, P_R and P_B respectively. Rays starting from these points and pointing outside divide the gray areas (where P potentially exists) into 4 areas (I, II, III, IV), as shown in Fig. 3, which is similar to 4 quadrants in a 2D space. It is easy to know that the vertex on the polygon nearest to a point P must be located on the edge of the range P belongs to. For example, in Fig. 3, P is in the range I, and only the edges in the top-right range need to be checked. This process can be combined together with the previous multi-probing procedure.

3. DETAILED ALGORITHM DESIGN

The framework in Fig. 1 is realized by three algorithms. The first algorithm describes how to organize polygons, as shown Algorithm 1. Basically, polygons are organized in such a way that their MBRs are stored in an R-tree (Lines 2-3) and their edges are saved in a hash table. After determining the basic parameters of a hash table (Lines 4-6), each edge is put into buckets (Lines 7-12) according to the hash key computed from the x coordinate (Lines 14-19).

In the INSIDE detection in Algorithm 2, for each point $P \in \mathbf{S}$, its candidate polygons are found, and each polygon has a separate buffer C_j holding points matched via R-tree (Lines 2-6). Then, for each candidate pair $(P, Poly_j)$, LSH-based searching is performed, and a matched pair is exported (Lines 7-12). In the LSH-based crossing number algorithm (IsInside), for each edge in bucket B_n associated with $P.x$, only those edges right on P are counted (Lines 18-21). P is regarded as inside the polygon if it is inside the outermost ring ($\#cross[1]$ is odd) and outside all inner rings

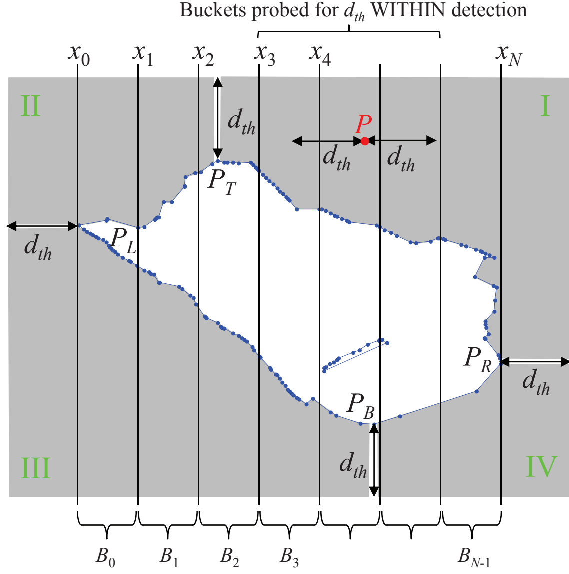

Figure 3: WITHIN detection.

Algorithm 1 Update $Poly_j$, the jth polygon.

1: **procedure** UPDATEPOLYGON($Poly_j$)
2: Find left-, top-, right-, bottom-most vertex of $Poly_j$.
3: Get the MBR of $Poly_j$, add it to R-tree.
4: $X_{min,j} = x$ coordinate of left-most vertex.
5: $X_{max,j} = x$ coordinate of right-most vertex.
6: $\Delta_j = (X_{max,j} - X_{min,j})/N$. ▷ Bucket width
7: **for** each edge (V_1, V_2) in $Poly_j$ **do**
8: Get $ring$ and $range$ (I, II, III, IV) of the edge.
9: $n_1 =$ GetHashKey$_j(V_1.x)$.
10: $n_2 =$ GetHashKey$_j(V_2.x)$.
11: Add $(V_1, V_2, ring, range)$ to $B_n, n = n_1, \cdots, n_2$.
12: **end for**
13: **end procedure**
14: **procedure** GETHASHKEY$_j(x)$
15: $x \leftarrow X_{min,j}$ if $x < X_{min,j}$.
16: $x \leftarrow X_{max,j}$ if $x > X_{max,j}$.
17: $n = int((x - X_{min,j})/\Delta_j)$.
18: Return n.
19: **end procedure**

($\#cross[n](n \neq 1)$ is even).

WITHIN detection is shown in Algorithm 3. Its basic flowchart is similar to that of the INSIDE detection except three points: (i) A rectangle decided by the distance threshold d_{th} is used instead of a point in the R-tree-based pre-filtering (Lines 4-6). (ii) Probing is performed on multiple buckets (Lines 13-15). (iii) Pairing between a point and the external ring is further optimized based on the range of a point (Lines 16-23).

Besides these basic algorithms, we optimize the program from different aspects, as follows: (i) Reducing the file I/O cost by reading/writing multiple data items at a time and using binary file I/O to read the text-format GML file. (ii) Batch processing and cache usage. Polygon instances do not change as often as points. When a set of points are to be compared with the same set of polygons, they can be processed in batch. An LSH table of a polygon is stored in a continuous buffer. Points matched to the same polygon via R-tree are separately stored in each buffer, and later processed together in the LSH-based matching. (iii) Parallel computation on multiple cores via multi-threading programming.

Algorithm 2 INSIDE detection.

1: **procedure** INSIDE(Point set **S**)
2: Clear candidate point set $C_j, j = 1, \cdots, M$.
3: **for** each point P in **S do**
4: Perform R-tree detection.
5: Add P to C_j if P is in the MBR of $Poly_j$.
6: **end for**
7: **for** each point P in $C_j, j = 1, \cdots, M$ **do**
8: inPoly = IsInside(P, $Poly_j$).
9: **if** inPoly is true **then**
10: Export (P.ID, P.seq, Poly$_j$.ID, Poly$_j$.seq).
11: **end if**
12: **end for**
13: **end procedure**
14: **procedure** ISINSIDE(P,$Poly_i$)
15: $n =$ GetHashKey$_j(P.x)$.
16: Clear $\#cross[n], n = 1, \cdots, \#rings$.
17: **for** each edge $(V_1, V_2, ring, range)$ in bucket B_n **do**
18: **if** $V1.x \leq P.x \leq V2.x$ **then**
19: $k = (V_2.y - V_1.y)/(V_2.x - V_1.x)$.
20: $dy = (P.y - V_1.y) - k \cdot (P.x - V_1.x)$.
21: **if** $(dy < 0)$ **then** $\#cross[ring] + +$.
22: **end if**
23: **if** $(|dy| < d_{min})$ **then**
24: $ringIdx = ring$; $rangeIdx = range$.
25: $d_{min} = |dy|$.
26: **end if**
27: **end if**
28: **end for**
29: $inPoly$=$\#cross$[1] is odd&&$\#cross[n](n{\neq}1)$ is even.
30: Return $inPoly, ringIdx, rangeIdx$.
31: **end procedure**

4. EXPERIMENTAL RESULTS

The ACM GIS Cup 2013 provides a training dataset which includes two point files (Point500 with 39,289 instances, Point1000 with 69,619 instances), two polygon files (Poly10 with 30 instances, Poly15 with 40 instances), and the ground truth of INSIDE/WITHIN detection each with 4 files under different combinations of inputs and predicates. We locally conduct studies in a laptop PC with Intel Core i5 CPU and 64-bit Windows 7. In the evaluation, we compare the average time of four methods: kNN, R-tree, LSH, and R-tree+LSH. kNN is an exhaustive search method, R-tree is for the pre-filtering stage, LSH is for pairing a point with a polygon when the point is inside the MBR of the polygon and $N = 100$ buckets are used for each polygon. R-tree+LSH includes all functions of the proposed scheme. We conducted two experiments, each predicate of which is generated by different combinations of point and polygon files. In the first experiment, we focus on the time taken to match points and polygons, neglecting the overhead such as file I/O and data conversion. In the second experiment, overall execution time is evaluated. In all experiments, 100% accuracy is achieved. The execution time contains the running time for all points and polygons in a test. It is measured by the function QueryPerformanceCounter in the Windows

Algorithm 3 WITHIN detection.

```
1: procedure WITHIN(Point set S, threshold d_th)
2:     Clear candidate point set C_j, j = 1, ··· , M.
3:     for each point P in S do
4:         Comp. Rect=[P.x−d_th,P.y−d_th,P.x+d_th,P.y+d_th].
5:         Perform R-tree detection with Rect.
6:         Add P to C_j if Rect overlaps the MBR of Poly_j.
7:     end for
8:     for each point P in C_j, j = 1, ··· , M do
9:         (inPoly,ringIdx,rangIdx) = IsInside(P, Poly_j).
10:        if inPoly is true then
11:            Export (P.ID, P.seq, Poly_j.ID, Poly_j.seq).
12:        else
13:            n_1 = GetHashKey_j(P.x − d_th).
14:            n_2 = GetHashKey_j(P.x + d_th).
15:            for (V_1,V_2,ring,range)∈B_n, n=n_1,···,n_2 do
16:                search=(ringIdx>1 || range==rangeIdx).
17:                if ring == ringIdx && search then
18:                    Comp. dist of P and the edge V_1,V_2.
19:                    if dist ≤ d_th then
20:                        Export(P.ID,P.seq,Poly_j.ID,Poly_j.seq).
21:                        Cont. to process next point in C_j.
22:                    end if
23:                end if
24:            end for
25:        end if
26:    end for
27: end procedure
```

environment and averaged over 100 runs.

Results of both experiments are shown in Figs. 4-5. Of the four schemes, the execution time of kNN is the largest. Because the number of polygons in the training set is small (10 or 15), the effect of R-tree is limited. LSH can greatly reduce the execution time compared with kNN, and R-tree+LSH achieves the least execution time in all cases.

Without considering the overhead, in Fig. 4, LSH improves the execution speed by 970% (computed as (kNN time)/(LSH time) -1) for INSIDE detection, and by 3.7 for WITHIN 1000 detection, both averaged over 4 scenarios. A large d_{th} requires to probe many buckets and affects the execution speed. As d_{th} decreases to 100 and 10, LSH improves the execution speed by 770% and 850%, respectively, compared with kNN. It is expected that the performance of the WITHIN detection will approach that of the INSIDE detection as d_{th} approaches 0.

The overall time (including file I/O and GML format conversion) in Fig. 5 increases in all schemes. An investigation also shows that the overhead time is almost the same in all schemes. In our implementation, the overhead time is already reduced by using block read to improve the efficiency of file I/O and using a separate thread to handle file I/O.

5. CONCLUSION

The idea behind geo-fencing is very intuitive: users 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. We proposed an efficient edge-based locality sensitive hashing algorithm for solving the geo-fencing problem, which is implemented in Visual C++. Our work is very novel: i) The idea of LSH is first applied to large-scale pairing between points and polygons over geographic datasets. ii) A probing method is suggested to look up all geo-edges close to a target point. Evaluation results confirm that the proposed algorithm has good efficiency and retains 100% accuracy.

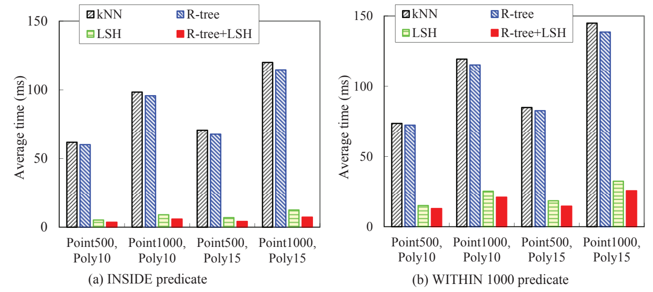

Figure 4: Average time taken to pair points and polygons (Without file I/O and data conversion).

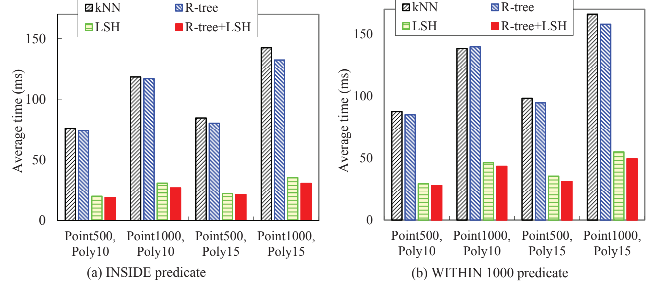

Figure 5: Average time taken to pair points and polygons (With file I/O and data conversion).

Acknowledgment

We are grateful to the organizers for their considerable efforts in the ACM SIG Cup 2013. This research is supported by the Singapore National Research Foundation under its International Research Centre @ Singapore Funding Initiative and administered by the IDM Programme Office.

6. REFERENCES

[1] Axel Kupper, Ulrich Bareth, and Behrend Freese. Geofencing and background tracking - the next features in LBS. In *INFORMATIK'11*, 2011.

[2] ACM GIS Cup 2013. `http://dmlab.cs.umn.edu/GISCUP2013/index.php`.

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

[4] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proc. 30th ACM STOC*, 1998.

[5] Yi Yu, Michel Crucianu, Vincent Oria, and Lei Chen. Local summarization and multi-level LSH for retrieving multi-variant audio tracks. In *ACM Multimedia*, pages 341–350, 2009.

[6] Yi Yu, Michel Crucianu, Vincent Oria, and Ernesto Damiani. Combing multi-probe histogram and order-statistics based LSH for scalable audio content retrieval. In *ACM Multimedia*, pages 381–390, 2010.

[7] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *ACM SIGMOD*, pages 47–57, 1984.