Proceedings of the International Conference on Inventive Computation Technologies (ICICT-2018)
IEEE Xplore Part Number: CFP18F70-ART; ISBN:978-1-5386-4985-5

Low computation in-device geofencing algorithm using hierarchy-based searching for offline usage

Bhavin Jawade
Deptt. Of Information Tech.
Shri G.S. Institute
Of Technology and Science
Indore, India
e-mail: bhavinjawade@gmail.com

Khushbu Goyal
Deptt. Of Information Tech.
Shri G.S. Institute
Of Technology and Science
Indore, India
e-mail: khushbug014@gmail.com

***Abstract*—Most applications use external services and APIs to implement geofencing. This has a major drawback that the user location data is accessible to the external service provider. Another important drawback is the continuous requirement of network connection for geofencing. Typical implementation of geofencing cannot be done within the mobile device as they require high computation for repetitive searching. In this research paper we propose new geofencing architecture based on arranging geofences in a tree like structure (geo-tree). Due to the low computation cost of our parsing algorithm, it is fast and can be used directly within mobile devices reducing network cost and more importantly keeping user location data secure. This research paper also talks about the tested efficiency of the architecture and about the probable future scopes where the efficiency can be further increased.**

***Keywords*- Algorithm, Geofencing, Locations, Data Structure.**

I. INTRODUCTION

Geofencing is the use of GPS technology to create a virtual geographic boundary, enabling software to trigger a response when a mobile device enters or leaves a particular area [1].A self implementation of geofencing using brute force technique to iterate over all geofences to search for correct geofence for current location would be very inefficient and such a computation cannot be performed inside the mobile device itself. Other implementations might use web-based services (APIs) provided by various organizations to resolve the issue of efficiency of searching. But the use of external web-based services has its own limitations and drawbacks.

1) *Addition network cost required for API calls.*
2) *Cannot work efficiently in weak network areas*
3) *Your location data is not safe. Your location data is accessed by organizations providing the web-based service.*

To resolve this issue, we have designed a new architecture for efficient low computation geofencing. This architecture resolves the above drawbacks. It can work offline and requires no excessive network cost during runtime and therefore resolves the first and second, above mentioned drawbacks. The algorithm can work on mobile devices themselves and therefore there is no data sharing (breach) with external organizations. This research paper primarily describes our own implementation of geofencing and its applications.
This architecture is divided into 2 parts. The first part consists of a parsing algorithm that parses a geo-tree. The second part corresponds to creation and updating of this geo-tree.

II. GEO-TREE AND ITS STRUCTURE

A. What is a geo-tree.

The location of a mobile device could be taken via many ways. Apart from GPS location service other methods like "*Localization using Wifi and GSM*" [3] or "*Multiple sensing mechanisms*" [4] or *"by listening for the cell IDs of fixed radio beacons"* [2] could also be used, especially in places where GPS is not available like Indoors. Throughout this paper where ever we mention the word "offline" we refer to the situation in which wifi and GSM are not available.

A geo-tree is a hierarchical tree like structure that is used to efficiently store the geofences and checkpoints. Here we would like to mention that we define geofences and checkpoints differently. Checkpoints are the indivisible areas, they are areas entry and exit around which we want to monitor. Geofences are simply geographical areas with definite boundaries. Geofences could be subdivided into more geofences, ie a geofence may contain more geofences or checkpoints. This also means that all checkpoints are geofences but not all geofences are checkpoints. For an example, city Delhi could be considered as a geofence divided into more geofences like Karol Bagh, Vasant Vihar etc. Then finally we could have the India Gate or any other government building as a checkpoint.

This geo-tree is sent to the mobile device before a geofencing application starts. Creation and updating of geo-tree takes place on the server database [8].

So, the structure of a geo-tree consists of nodes which are necessarily a geofence. Each node of a geo-tree contains the information about the geofence, most importantly the

coordinates (latitude and longitude) of the center of the geofence and the radius that defines the area and boundary of that geofence. The root node of a geo-tree is the outer most geofence (consider it like a city). This has some children, we call these children as sets. Each set can then have multiple children called regions. And finally, each region one or more checkpoints. This could be extended further like regions can have children that are themselves regions, as per the scenario as in Figure 2. For now, we are considering a 4-level tree, with a root node (city) and then sets, regions and checkpoints as inFigure1.

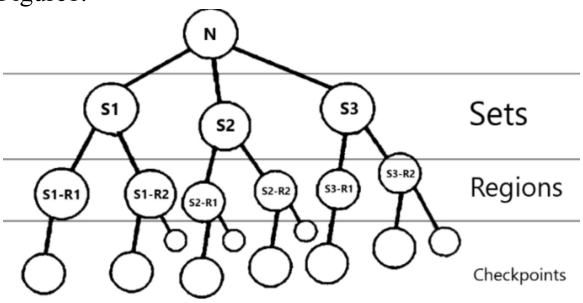

Fig 1: Structure of 4 level geo-tree

A json based representation of the geo-tree is shown in Figure 3. After implementation, geographical plotting of the geo-tree looks as shown in figure 4.


```
"root" : {
 "sets" : [ {
  "center" : {
   "info" : "someinfo",
   "latitude" : "22.730689",
   "longitude" : "75.870965",
   "radius" : "0.9"
  },
  "regions" : [ {
   "center" : {
    "info" : "someinfo",
    "latitude" : "22.725298",
    "longitude" : "75.8716349",
    "radius" : "0.3"
   },
   "checkpoints" : [ {
    "info" : "CheckpointInfo",
    "latitude" : "22.72568866",
    "longitude" : "75.87001899",
    "radius" : "0.1"
   }, {
    "info" : "CheckpointInfo",
    "latitude" : "22.72558071952263",
    "longitude" : "75.87289363030243",
    "radius" : "0.1"
   }]
  }]
 }]
}
```


Fig 2: Json based representation of geo-tree

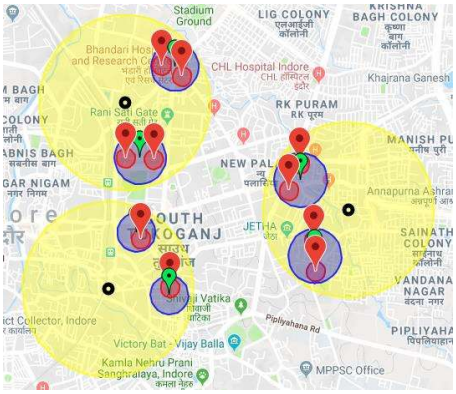

Fig 3: Geographical plotting of geo-tree, Color code: Yellow represents sets, blue represents regions and red represents checkpoints

III. Storage and Retrieval of Geo-Tree

Storing the whole geo-tree in the main memory is not an efficient solution. Therefore, the geo-tree should be stored in the file in secondary memory. In our implementation we stored the geo-tree in form of a JSON on the server and the copy of same is stored in secondary memory of the mobile device. Initially as the device starts moving we retrieve the data of sets (LEVEL 1 of geo-tree: without the regions and checkpoints) and stored in a list(L_{set}). Now we apply the find nearest set algorithm on L_{set} and as soon as the user enters the set retrieve the data of regions of that set and store it in list (L_{region}). Now apply the find nearest region algorithm on L_{region}. As soon as the device enters a region retrieve the data of checkpoints in that region and store it in a list ($L_{checkpoint}$). Now you apply the binary search algorithm on $L_{checkpoint}$.

IV. Parsing A Geo-Tree

The parsing of geo-tree takes place within the mobile device. The parsing uses two different algorithms at various levels of the parse tree. Both the algorithms depend on the harvesian distance formula used to calculate the great circle distance [10][5] between any two points on earth using coordinates in degrees.

$$a = sin^2(\frac{\Delta\phi}{2}) + cos(\phi_1) \cdot cos(\phi_2) \cdot sin^2(\frac{\Delta\lambda}{2})$$

$$c = 2 \cdot arctan2(\sqrt{a}, \sqrt{1-a})$$

$$d = R \cdot c$$

Where:
ϕ = latitude
λ = longitude
R = 6371 (Earth's mean radius in km)

Harvesian formulae assumes that earth as spherical and hence bears an error of about 0.5%. Vincenty's formulae for calculating distance on a spheroid can be used to get more accurate distance.

A. *Algorithm 1- Nearest Geofence.*

The first algorithm is called find Nearest Geofence algorithm. This algorithm finds the nearest geofence (within current level of the geo-tree) from the current location. This is done efficiently by calculating the distance between the current location and the center of each geofence at that level of tree and storing the minimum distance (say S_m), ie distance from the nearest geofence. Now no computation, ie recalculation for nearest geofence takes place until the device has covered this distance. This is done by keeping in mind the fact that the device won't enter any geofence until it has travelled the distance from its nearest geofence (S_m). Because all the checkpoints are stored in regions and regions are stored in sets, the relative number of sets with respect to checkpoints is very less. And therefore, the find nearest geofence algorithm needs tocalculate distance from very less number of nodes. And the recalculation takes place after a certain distance (S_m) is covered making the algorithm more efficient. The distance between the center of a geofence to current location can be calculated using Haversine formula.

Function findNearestGeofence (C_l, L_s):

Input: L_s – List of all children at a level, of the entered geofence, and C_l – Current Location and inArea.
Output: Nearest Geofence Index or the geofence Index in which you are present.

```
begin:
   initialize
      fmin = infinity;
      index = 0;
   for each node in Gs:
         centerdist = getHarvesianDistance (Cl, node);
         circumdist = cdist – node. radius;
         fmin = fmin < circumdist? fmin: circumdist;
         indexofnearest = fmin < circumdist?
index:            indexofnearest;
         if circumdist <= 0 then:
            if inArea == OUTSIDE_SET then:
               inArea = INSIDE_SET;
               indexOfSet = index;
               Lr = getList ("SET", indexOfSet);
            else if inArea == INSIDE_SET then:
               inArea = INSIDE_REGION;
               indexOfRegion = index;
         Lc = getList ("SET", indexOfRegion);
            isEnteredSetFlag = true;
         index++;
   currentfmin = fmin;
end;
```

The *findNearestGeofence* function is responsible for searching Nearest geofence at the current level in the geo-tree. It takes as input the list of all nodes that are children of the current geofence you have entered in. Then iterates over the children to find the nearest geofence.

Here in the algorithm you can see that *centerdist* is the distance between current position and the center of various nodes. We get *circumdist* by reducing the radius of the node from the *circumdist* which the shortest distance of the current position from the circumference of the circle is created around the geofence. Now it is checked whether this distance is shortest among the distances calculated from the various nodes, if yes store it in *fmin* along with the index of the shortest distance.
Check if the current location is already inside any set if yes then the get all the sets inside that region and store them into a list for further calculations. At last store the minimum min distance found in a variable for further use.

findNearestGeofence is called by another function *isInside*. *isInside* is responsible for handling the distance check.

This algorithm has a diminishing nature. That means as we move away from the concentrated area of geofences, the number of times recalculation takes place decreases. Consider that the user has just exited a set. Now as he moves away from the all the sets the recalculation will take place less frequently. This is a desired effect because if geofences are present in only a small part of the city, recalculation should not take place when you are far away from the geofences. And the frequency should increase as one moves towards the geofences. We plotted the frequency of recalculation with the distance from the nearest geofence with our testing data (Figure 4). This plot shows the diminishing nature, as the distance from nearest set increases, frequency of recalculation tends to become constant.

Function isInside (C_l):

Input: L_s – List of all children at a level, of the entered geofence. And C_l – Current Location and inArea
Output: Nearest Geofence Index or the geofence Index in which you are present.

```
begin:
      tempDis = totaldistance - distanceTillExit;
         if inArea == OUTSIDE_SET then
            if tempDis >= currentfmin then
               if inArea == OUTSIDE_SET then
                  findNearestSet (currentpoint, geolistSet);
               if inArea == INSIDE_SET then
                  findNearestSet (currentpoint, geolistRegion);
         else if inArea == INSIDE_SET then
            node = geolistSet[indexOfSet];
            centerdist = getHarvesianDistance (Cl, node);
            fdist = cdist – node. radius;
            if fdist <= 0 then
               inArea = INSIDE_SET;
               if tempDis >= currentfmin then
                  findNearestSet                    (currentpoint,
geolistRegion);
               else
                  distanceTillExit = totaldistance;
                  inArea = OUTSIDE_SET;
         else if inArea == INSIDE_REGION then
            node = geolistRegion[indexOfRegion];
            centerdist = getHarvesianDistance (Cl, node);
            fdist = cdist – node. radius;
         if fdist <= 0 then
                  inArea = INSIDE_REGION;
                  checkpointindex_temp=
```

```
            binarysearch (Cl, geolistGeofences);
            if checkpointindex_temp == -1 then
              /* not inside a checkpoint */
            else
               inArea = INSIDE_CHECKPOINT;
               indexOfCheckpoint = checkpointindex_temp;
              /* Act: Inside a Checkpoint Now */
          else
               distanceTillExit= totaldistance;
              inArea = INSIDE_SET;
      else if inArea == INSIDE_CHECKPOINT then
        node = geolistCheckpoints[indexOfCheckpoints];
        centerdist = getHarvesianDistance (Cl, node);
```

```
          circumdist = cdist – node. radius;
        if fdist <= 0 then
              inArea = INSIDE_CHECKPOINT;
           else
            /* Act: Exited a Checkpoint */
          inArea = INSIDE_REGION;
  end;
  = INSIDE_REGION;
  end;
```

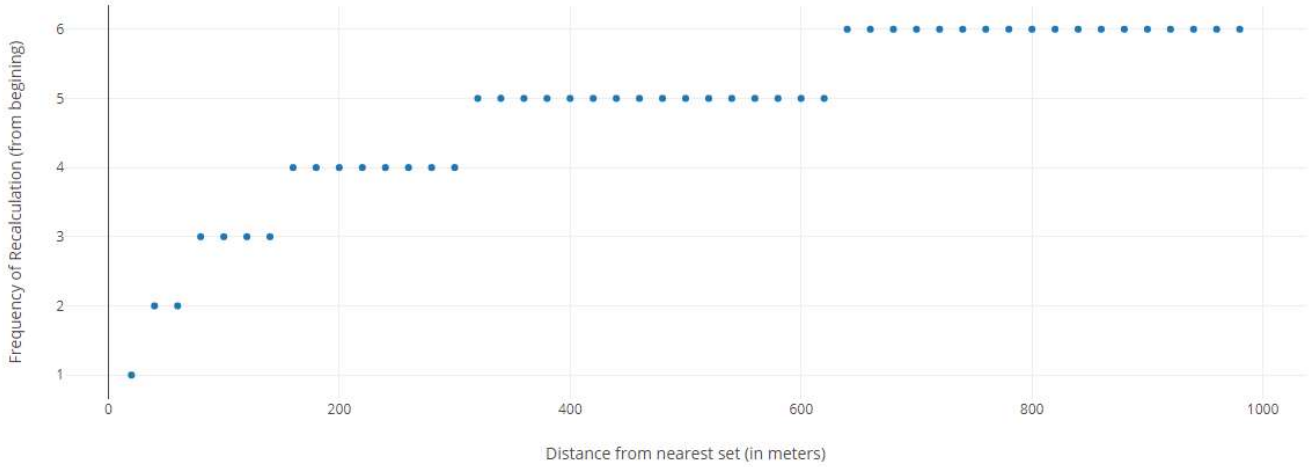

Fig 4: Plot between frequency and distance from nearest geofence

B. Algorithm 2.

The second algorithm used by the system is any good functional searching algorithm, in this research we have worked with binary search. We first sort the geofences at a same level based on their latitudes and store them in a list (L_s). Now we apply binary search to check if the current location lies within (*isInside*) any of the geofences of the L_s. The check for *isInside* is done by calculating the distance using the Haversine formula and then checking if the distance is less than the radius of the geofence. As we know that Binary search has time complexity of Log_2N, where N is the number of points. Therefore, if we have let's say we have 100 geofences in the list, the number of times *isInside* is computed is approximately 7, as Log_2100 = 6.6438. The list of the geofences on which binary search is to be performed is stored in sorted order beforehand, so no runtime sorting required.

In this algorithm *isInside* () first find if the difference between the total distance travelled and the distance from the last exit and call it *tempDis*.

If you are initially not inside any of the sets then you take following steps:

1) If tempDis is greater than current fmin then check again if you are still outside the set if yes, then call function findNearestSet on list of sets else if it is inside the
2) outermost set then call findNearestSet on list of regions inside that set.

If initially you are inside a set, then take the following steps:

3) Calculate the distance of current point from the node's radius and reduce the length of radius from it and call it fdist. If fdist is less than or equal to zero, then we can say that current point is still inside the set and call findNearestSet on list of regions inside that set.

If initially you are inside a region then take the following steps:

4) Calculate the distance of current point from the node's radius and reduce the length of radius from it and call it fdist. If fdist is less than or equal to zero, then we can say that current point is still inside the region then call binarysearch function on list of geofences in the region. If it is found that the point is in the inside a geofence then return that you are inside a checkpoint.

If initially you are inside a checkpoint, then take the following steps:

5) Calculate the distance of current point from the node's radius and reduce the length of radius from it and call it fdist. If fdist is less than or equal to zero, then we can say that current point is still inside the checkpoint. If fdist is

less than or equal to zero, then the point is still in the checkpoint else it has exited the checkpoint.

The above explained steps are followed in the algorithm for the up comming locations of the user.

C. *Using both the algorithm working together.*

Both the algorithms have their limitations and strengths. The first algorithm adapts as the distance from the nearest set increases and reduces the number of times find Nearest geofence is called but has a drawback that whenever find

Nearest geofence is called distance calculation and comparison must be done for all the geofences at that level of the geo-tree. Whereas the second algorithm being binary search requires that even for a larger list of geofences the calculation and comparison for distance be done only Log_2N times which is very less but has a drawback that it must be done for every changing location of the user. It is a continuous device sensing application [7]. Therefore, we used these two algorithms together. We have cited our observation in the form of following rules:

1) Algorithm 1: should be used when the geofences are far away from each other, and probability of entering them is low.
2) Algorithm 2: should be used when the geofences are close to each other, and probability of entering them is high.

Function BinarySearch (C_l, L_c):

Input: L_c – List of all checkpoints in the current branch of the tree, and C_l – Current Location.
Output: Index of checkpoint in which you are present, -1 otherwise.

```
begin:
        found = -1;
        startIndex = 0;
        endIndex = checkpointlist. size ()-1;
        midIndex;
        while startIndex <= endIndex:
            midIndex = (endIndex + startIndex)/2;
   if keyElement is inside this checkpoint at midIndex then
                found = midIndex;
                return midIndex;
   else if keyElement has greater latitude:
                startIndex = midIndex + 1;
   else
                endIndex = midIndex - 1;
return found;
end;
```

Based on above 2 rules, we realize that the first algorithm should be used for level 1, ie. the level of sets and for level 2, ie. the level of regions. Whereas the second algorithm should be used for level 3, ie. level of checkpoints.
These two algorithms can be used for various levels of the geo-tree depending upon the geographical arrangement of the geofences.

V. Results and Analysis

Use-case Analysis of the above system: Let us consider an example of a city which is divided into 4 sets each of radius 5 km. Now each set is further divided into 5 regions of radius 1km and each region has 20 – 30 checkpoints. The above example represents the arrangement for any typical densely populated city.

For such an arrangement the Algorithm 1 is being used on sets and regions, and the number of sets in the city is only 4, therefore whenever *findnearestgeofence* is called upon sets, the calculation for distance will be done only for 4 geofences, which is efficient. Same holds true for regions. As we enter a region the Algorithm 2 is implemented on all the checkpoints of that region. If that region has 15 to 30 checkpoints, then the complexity of the binary search will be from $Log_2 15 = 3.90$ (approx. 4) to $Log_2 30 = 4.90$ (approx. 5). As the checkpoints are closer to each other call for binary search on each location increases accuracy and reduces delay of entrance within checkpoints. Plot in Fig 4 shows how frequency of recalculation (no of time algorithm runs) varies with the distance from nearest geofence.

TABLE I. EXPERIMENTAL RECORD

Parameter	**Count**
No of locations in path	**3458**
Total distance covered	**50.3 Km**
Total No of times algorithm-1 recomputed	**1032**
Total No of times algorithm-2 recomputed	**44**
Total No of times Exit computation takes place	**85**

VI. Exit from Geofences

Entry into the geofences is already discussed in the parsing of the geo-tree using algorithm 1 and algorithm 2. The exit condition is quite simpler because during exit we already know the exact geofence we must exit. Therefore, no searching algorithm is required for that. But instead of continuously checking for exit condition for every location, we should use the corollary of Algorithm 1. That is once you entered inside a geofence compute the distance of your location from the center of the geofence and subtract it from the radius and store it(E_{min}). Now you don't have to check for exit condition until the user has travelled the distance of $E_{min.}$

VII. Construction of Geo-Tree

The below proposed Geo-Tree construction and updating algorithm is optional and for simpler implementations, geo-tree could be made hardcoded. For a larger implementation with

more frequent updates we hereby describe in brief a dynamic geo-tree construction and updating algorithm.

Take as input the coordinates of all the checkpoints. These are the fixed and static and won't change with addition or deletion of other checkpoints. Now start group checkpoints together to form regions. The reader is suggested to write algorithm that tries to balance these regions in such a way that nearby checkpoints are in the same region but at the same time no region is over-dense. Similar principle must be followed for creating sets from regions.

On addition or deletion of a checkpoints, the centers and radii of the regions must adjust and then the radii and centers of sets. Reader must note that as the geo-tree is not created on the mobile device and is instead created and maintained on a remote server, the computation required for balancing of geo-tree does not affect the performance of geofencing on the mobile device.

VIII. Conclusion and Future Scope

We implemented and tested the above algorithm as an android app. We can conclude that the above architecture for in-device implementation of a geofencing service works well without causing computational overhead. Apart from cell phone devices, our algorithm could be used for other kinds of Mobile computing applications [6], embedded systems and IoT devices. The algorithm can be especially used for creating applications that are supposed to provide geo-spatial services to its users (like a location-based reminder service) as there is no need to use external service APIs or other network based Geofencing techniques like "Network Proximity" [9] method.

Future Scope: This architecture can still be improved. Additionally, creation of the geo-tree could be done using a density-based clustering algorithm like DBSCAN. For extremely large number of geofences creation of geo-tree could be optimally done by using a hybrid agglomerative hierarchical Clustering algorithm (like BIRCH) with a density-based clustering algorithm.

Acknowledgment

The authors gratefully acknowledge the support and guidance of their mentors Mr. Pawan Makhija, Mrs. Megha Khuliya and Dr. Apporv Gaiwak.

References

[1] F. Reclus and K. Drouard Geofencing for fleet & freight management Intelligent Transport Systems Telecommunications, (ITST),2009 9th International Conference on pp. 353-356

[2] A. LaMarca et al. Place lab: Device positioning using radio beacons in the wild. In Lecture Notes in Computer Science, volume 3468, pp. 116–133, Munich, 2005.

[3] I. Constandache, S. Gaonkar, M. Sayler, R. R. Choudhury, and L. Cox. Enloc: Energy-efficient localization for mobile phones. In Proceedings of IEEE INFOCOM Mini Conference '09, Rio de Janeiro, Brazil, 2009.

[4] Z. Zhuang, K.-H. Kim, and J. Singh Improving Energy Efficiency of Location Sensing on Smartphones MobiSys'10, June 15–18, 2010, San Francisco, California, USA, Copyright 2010 ACM 978-1-60558-985-5/10/06

[5] By Mr. Reid "Shortest distance between two points on earth" http://wordpress.mrreid.org/haversine-formula/ This is an electronic document. Date of publishing 20/12/2011

[6] Want, R., Schilit, B., Adams, N., Gold, R., petersen, K., goldberg, D., Ellis, J. & Weiser, m. The Parctab Ubiquitous Computing Experiment, In: Imielinski, T. (Ed.) Mobile Computing, Pp. 45-101 (Kluwer Publishing). (1997)

[7] H. Lu et al. The jigsaw continuous sensing engine for mobile phone applications. In Proc. 8th ACM Conf. Embedded netw. Sens. Syst., SenSys'10, pp. 71–84. ACM, 2010.

[8] Cheng Zhi. The Web Database Application System Optimization Research. Published in Seventh International Conference on Measuring Technology and Mechatronics Automation, Nanchang, China, 2015.

[9] Namiot D., Sneps-Sneppe M. (2013) Geofence and Network Proximity. In: Balandin S., Andreev S., Koucheryavy Y. (eds) Internet of Things, Smart Spaces, and Next Generation Networking. ruSMART 2013, NEW2AN 2013. Lecture Notes in Computer Science, vol 8121. Springer, Berlin, Heidelberg.

[10] Javin J. Mwemzi, Youfang Huang," Optimal Facility location on spherical surfaces", New york science Journal, April 2011.