Algorithms for Answering Geo-Range Query

Xi Zhang, Kui Wu
Computer Science Dept.
University of Victoria
BC, Canada V8W 3P6

Yong Gao
Computer Science Dept.
University of British Columbia, Okanagan
BC, Canada V1V 1V5

Abstract—In wireless sensor networks, we usually need to detect interesting events based on the information gathered from multiple sensors. One useful detection is to test whether or not the average sensory value within an area is larger than a given threshold. Such type of query is called geo-range query. It should report the geographic centers where the average value of nearby sensors is greater than a certain threshold. Answering geo-range query is nontrivial because we do not know in advance the satisfying geographic centers, which may not be necessarily the same as the locations of sensors. We develop two efficient algorithms: the brute-force search algorithm and the sweep-line algorithm. The brute-force search algorithm uses exhaustive search to enumerate all possible satisfying sub-regions. Its time complexity is \( O(n^3) \), where \( n \) is the number of sensor nodes. The sweep-line algorithm uses a virtual line sweeping top-down through the plane. The algorithm takes \( O(n^2 \log n) \) running time, and still obtains exact solution to the problem.

I. INTRODUCTION

Wireless sensor networks (WSNs) have been widely used in many monitoring and control systems, in which methods and algorithms need to be developed to solve various information query problems, such as top-k query [1] and query for extreme values [2]. In many cases we are more interested in finding collective phenomenon instead of individual sensor readings. In this case, we need to execute a new type of query, called geo-range query, over a region. For example, in the ocean monitoring system, if oil leak is detected, we should track the areas and the spread of pollution. We thus need to query the system for areas that meet a given query criterion.

Nevertheless, answering geo-range query is not easy. In many cases, such as the detection of forest fire or ocean pollution, the range of an event starts from one point and expands gradually in all directions. The shape of the area can usually be approximated as a circle. If we know the center and the radius of the circle, geo-range query can be easily answered with the sensing data falling within the area. The problem, however, is that we do not have such information in advance, and we need to find the centers and the radii of the areas that meet the query condition. We call such areas fitting areas. Finding all fitting areas is hard, because the center of a fitting area may not necessarily be the location of a sensor. Calculation of different combinations may easily end up with an exponential increase in the number of possible fitting areas. In other words, the uncertainty in the possible centers could make the exhaustive search very costly.

Intuitively, geo-range query might be answered with the results from contour map. There are many research efforts on building contour lines over wireless sensor networks [3]–[6]. A contour line is a line along which the sensing data are equal or very close. Nevertheless, contour map cannot be used directly for answering geo-range query because it is not very helpful in pinning down the centers of fitting areas. In addition, building contour lines is a non-trivial task, especially when the monitored phenomenon changes frequently.

In this paper, we focus on effective solutions to the geo-range query problem. We develop two fast algorithms for executing geo-range query. Particularly, we propose and analyze a polynomial time brute-force search algorithm, and a sweep-line algorithm to speed up the searching procedure during geo-range query.

II. PROBLEM FORMULATION

Let \( S = \{s_1, s_2, \ldots, s_n\} \) be a set of \( n \) sensors deployed in a monitoring field. We assume that the locations of sensors are known. Let \( R \) be a predefined diameter of a circular area, the center of which could be any point within the field. Let \( A_i \) be a subset of sensors (i.e. \( A_i \subseteq S \)), covered by a circular area with the radius of \( R \). We are interested in answering the following query described in SQL language:

SELECT \( A_i \) 
FROM \( S \) 
WHERE \( \Psi_r(A_i) > T \) AND \( r = R \)

The above query is to find all the subsets of sensors such that a returned subset includes sensor nodes, which fall within a circular area of radius \( R \) and have sensor readings meeting the condition \( \Psi_r(A_i) > T \), where \( \Psi_r \) is an aggregate function over a set of sensor nodes and is application dependent. For example, \( \Psi_r(A_i) = \frac{\sum_{s \in A_i} Temp(s_i)}{|A_i|} \) computes the average temperature value of all the sensors located within the area, if \( Temp(s_i) \) is to get the temperature value of sensor \( s_i \).

III. BRUTE-FORCE SEARCH ALGORITHM

A. Introduction of the Algorithm

Our first solution is to use a circular moving window, as shown in Fig. 1 to help the exhaustive search for possible fitting areas. We observe that, although the number of combinations of sensor values is exponential, the number of combination that meets our query criterion is in fact quadratic. For any given point \( p(x, y) \) on the plane, a sensor falling within the circle centered at \( p(x, y) \) with radius \( R \) is called a related sensor with respect to (w.r.t.) \( p(x, y) \). If the average value of readings from all related sensors w.r.t. \( p(x, y) \) is above
the threshold $T$, $p(x, y)$ should be reported as the center of a circular area meeting our query. Note that $p(x, y)$ is not necessarily the location of a sensor.

Given a set of sensors $S = \{s_1, s_2, ..., s_n\}$, we can draw a circle centered at each $s_i$ with radius of $R$. We call the circle the generator circle of $s_i$. The $n$ generator circles may intersect with each other and divide the plane into smaller sub-areas. It is easy to see that for any point $p(x, y)$ within an overlapped area $a_i$, the set of related sensors w.r.t. $p(x, y)$ is determined. For example, as shown in Fig. 2, for any point in the area $a_1$, the set of related sensors is $\{s_1, s_2, s_3\}$; For any point in the area $a_2$, the subset of related sensors is $\{s_1, s_2\}$. Hence, by enumerating all the overlapped areas or isolated generator circles, we can obtain all the possible results meeting the criterion in the geo-range query.

Since an overlapped area is formed by overlapping generator circles, an overlapped area must be encircled by arcs, and all arcs are connected by the intersections of generator circles. Thus all overlapped areas can be enumerated by checking the adjacent areas of each intersection. For example, in Fig. 2, $a_1$, $a_2$, $a_4$ and $a_6$ are adjacent areas of intersection $i_1$. Surrounding an intersection point, there are three types of sensors:

- **Inner sensor**: If the distance between the sensor and the intersection point is less than $R$, we call it an inner sensor w.r.t. the intersection, because the sensor is within the query range of the intersection. And the sensor is also a related sensor w.r.t. any point in each overlapped areas adjacent to this intersection point. We use $IS_i$ to denote the set of all inner sensors w.r.t. the intersection point $i$.

- **Border sensor**: If the distance between the sensor and the intersection point is equal to $R$, we call the sensor a border sensor w.r.t. the intersection point. Let $BS_i$ denote the set of all border sensors w.r.t. intersection point $i$.

- **Outer sensor**: If the distance between a sensor and the intersection point is greater than $R$, we can simply ignore this sensor when the intersection point is in consideration.

Based on the above analysis, our brute-force algorithm works as follows. It first computes the intersection points of all generator circles. Then for each intersection point, it performs an exhaustive search to find out inner sensors and border sensors w.r.t. this intersection point. After that, it is easy to check whether or not the values from the inner sensors and border sensors satisfy the given query criterion.

Note that it is impossible to find all centers because theoretically, an area includes infinite number of points (if we allow the coordinate of a point to be real numbers) and thus number of centers could be infinite. Nevertheless, the above method provides us with a comprehensive list of representative centers. Based on the result, we can further find all possible areas, within which any point can be used as the center of a circular region meeting the query condition. We call such areas satisfying areas and the problem is solved if we can list all satisfying areas. In the following, we illustrate the method to decide whether or not a sub-area is a satisfying area.

It is not hard to see that there are at least two border sensors w.r.t. an intersection point. It is possible that multiple circles may intersect at the same point, as shown in Fig 3. When $k(k \geq 2)$ circles intersect at the same point, the plane is divided into $2k$ sub-areas surrounding the intersection point. As shown in Fig 3, four arcs cut the area into 8 sub-areas, $a_1$ to $a_8$, respectively. We need to decide whether or not each sub-area is a satisfying area.

To this end, we first note that an inner sensor w.r.t. an intersection point is also an inner sensor w.r.t. any point in the sub-areas adjacent to the intersection point. A border sensor w.r.t. the intersection point, however, may become an outer sensor w.r.t. points in some sub-areas. As shown in Fig. 2, border sensors $s_1$ and $s_2$ become outer sensors w.r.t. points in sub-area $a_5$, and as such sub-area $a_5$ may not be a satisfying area.

Based on this observation, we can define the following proposition.

- **Proposition**: If a sub-area $a_i$ is a satisfying area, then all inner sensors w.r.t. $a_i$ and a subset of border sensors w.r.t. $a_i$ are satisfying areas.

### Diagrams

#### Fig. 1. Moving window with a circular shape

![Fig. 1. Moving window with a circular shape](image1)

#### Fig. 2. Plane divided by arcs

![Fig. 2. Plane divided by arcs](image2)

#### Fig. 3. Multiple circles intersect at the same point

![Fig. 3. Multiple circles intersect at the same point](image3)
Therefore, when checking a sub-area surrounding an intersection point $i$, values from the sensors in $IS_i$ should be considered, but values from sensors in $BS_i$ may need to remove. This tasks turns out to be easy, due to the joyful fact that two adjacent sub-areas can only have one different value. For example, in Fig. 2, w.r.t. the intersection point $i_1$, $s_2$ is the inner sensor and its value should be always considered when checking the sub-areas around $i_1$. $s_1$ and $s_3$ are the border sensors, and their values may or may not be included in an sub-area. There are 4 sub-areas around the intersection point $i_1$, $s_1$, $s_2$, $s_3$ when we check $a_1$, sensors $s_1$, $s_2$ when we check $a_2$, sensors $s_2$ when we check $a_6$, and sensors $s_2$, $s_3$ when we check $a_4$.

B. Complexity Analysis

Given $n$ sensors in the plane, the time complexity of the Brute-Force search algorithm is $O(n^3 \log n)$. In first step, to compute intersections, we need pairwise comparison between sensors, resulting in $O(n^2)$ operations. For each pair of sensors, there are at most two intersections, so the number of intersections is also $O(n^2)$. For each intersection, we need to search all $n$ sensors to find inner sensors, which increases the running time to $O(n^3)$. Also for each intersection, a sorting algorithm is needed for border sensors, which is required in the last step when we check the sub-areas around the intersection point. This step takes $O(n \log n)$ time in the worst case (i.e., all $n$ circles intersect at the same point). To sum up, the over all time complexity is $O(n^3 \log n)$.

Nevertheless, the above worst case is unlikely to happen, since the case that multiple circles intersect at the same point is extremely rare. If we make the assumption that no three circles intersection at the same point, the time complexity of brute-force search algorithm will be $O(n^3)$.

IV. A SWEEP-LINE ALGORITHM

The brute force search algorithm can be further improved. For example, the overlapping area $a_5$ in Fig. 2 is computed repetitively because it has multiple intersection points on its circumference. To improve the performance by taking the advantage of the geographic information of sensors, we design a sweep-line algorithm with a binary tree data structure.

A. The Sweep-Line Technique

The sweep-line technique is efficient by making use of the order of related geometric objects [7], [8]. Furthermore, algorithms for constructing a Euclidean Voronoi diagram of circles and a Euclidean Voronoi diagram of circles contained in a larger circle have been developed recently [9]. In paper [8], Kim et al. developed a sweep-line algorithm for hierarchy of the circles. Paper [7] gives an algorithm to compute the intersections of polygons. In this section, we will borrow the idea of sweep-line technique to compute the intersections of circles.

B. Sweep Line and Arc List

Suppose that two extreme points are assigned to a circle $s_i$, the top most and the bottom most of the circle denoted by $l_i$ and $b_i$, respectively. Extreme points and intersections are called event points. All event points are sorted by their $Y$-coordinates in an event queue.

Each generator circle $s_i$ is divided by a vertical line passing through the center of the circle, and is split into two semi-circles, the left semi-circle $l_i$ and the right semi-circle $r_i$. Having noticed that all the overlapped areas are surrounded by arcs, by tracing down all the arcs, we can enumerate all the overlapped areas.

![Fig. 4. A sweep line intersects with circles](image)

A sweep line is defined as a horizontal line moving from the top to the bottom of the plane. As it moves downwards, the intersections between the sweep line and the circles change. The key is to maintain the state of the sweep line, which is denoted as a list of arcs currently intersecting with the sweep line. The arc-list is sorted by the X-coordinate of the intersections. For example, in Fig. 4, suppose the left and right semi-circles of $s_i (i = 1, 2, 3)$ are $l_i$ and $r_i (i = 1, 2, 3)$, respectively. The sorted arc-list of the sweep line $h_0$ is as following: $\{l_1, l_3, l_2, r_3, r_1, r_2\}$. We need to make sure that all objects above the sweep-line are represented by the state of the sweep-line, and all objects beneath it do not affect the state of the current sweep line. The algorithm sweeps the plane by moving the sweep line downwards from event point to event point without turning back.

C. Event and Event List

Notice that the arc list changes if and only if when the sweep line hits an event point. The specific changes made by the events are as follows:

- **Top event:** This event occurs when the sweep line $h$ hits the top most point of a generator circle. When the sweep line hits such a point, two arcs of the generator circle (i.e., the left and the right semi-circles) will be inserted to the sorted arc list.

- **Cross event:** This event occurs when the sweep line $h$ hits an intersection point between two generator circles. When the sweep line hits such a point, two existing arcs in the arc list will swap their position.

- **Bottom event:** This event occurs when the sweep line $h$ leaves a generator circle (i.e., hits the bottom most point...
of the generator circle). When the sweep-line hits such a point, the two arcs of the generator circle will be removed from the sorted arc list.

D. Enumerating Areas

\[ a_1 \quad a_2 \quad a_3 \quad a_4 \quad a_5 \]

Fig. 5. The alteration of areas at events

A list of areas is also maintained when the line sweeps through the plane. At a top event, a new overlapping area will emerge at this point. See Fig. 5 (a), area \( a_1 \) will emerge right after the sweep line hits the top event. At a bottom event, an existent area will close at this point. In Fig. 5 (c), area \( a_1 \) will close and leave the sweep-line forever after it hits the bottom event. At a cross event as in Fig. 5 (b), however, an existent area \( a_1 \) will close at this point, and a new overlapping area \( a_2 \) will emerge at this point. The specific operations of inserting and removing areas will be described in the following.

E. Data Structure of Sweep Line

An appropriate data structure is essential for the design of the algorithm. The time complexity of operating the sorted arc list could be high when done naively. The maximum number of arcs that could intersect with the horizontal line is \( O(n) \). If the arc list is stored in linked list, each operation of the arc list will be \( O(n) \) time. To reduce the running time, a binary tree is implemented to represent the data structure of arc list. As described above, the state of the sweep line is denoted as a list of arcs intersecting with the sweep line. The arcs are sorted by the X-coordinates of their leftest point. Thus the arc list in Fig. 4 should be as following: \( \{l_1, l_3, l_2, r_3, r_1, r_2\} \). Once sorted, it is easy to build up a binary tree to speed up the operation of the list. Fig. 6 is an example of a binary tree built from Fig. 4, and \( a_1, a_2, a_3, a_4, a_5 \) and \( a_6 \) are the areas in-between them.

Generated overlapping areas could be stored along this binary tree, since there is only one area between each pair of neighbouring arcs. There are always two arcs related to a event. For an event, two arc either emerge, close or swap their location. For the top and bottom events, the two arcs may be generated or deleted accordingly at the event point; for the cross event, existing two arc are swapping their position. It means that the two arcs related to the event must be neighbouring arcs. So relevant to each event are three sub-areas. The operation is as follows:

- In a top event, an existent area will be divided into three new areas. The left area and the right area should have the same set of associated sensors as the original one; the middle area will have one more sensor (the one related to the top event) in its set.
- In a bottom event, three existent neighbouring areas will merge as one. The middle area will be divided and the associated sensor set will be output as the result (if the sensor set meets the query criterion). The left and right areas, which should have exactly the same attribute at this time, will be merged as one.
- In a cross event, the number of areas remains the same during the operation. The left and right area do not change. For the middle area, however, the set of related sensors should be changed according to different situations (as further illustrated in the next sub-section). If any areas are detected closing at a cross event, the corresponding result will also be output (if the sensor set meets the query criterion). For the simplicity of discussion, we assume that no three circles intersect at the same point.

F. An Example Illustration

\[ s_1 \quad s_2 \]

Fig. 7. Illustration of a sweep line moving through the plane

To exemplify the procedure of the algorithm, we use an example with only two sensors. Fig. 7 shows two generator circle \( s_1 \) and \( s_2 \) intersecting with each other. Fig. 8 explains how the data structure is constructed and maintained during the process.

The algorithm starts with an empty tree. As the sweep line moves down, nodes are gradually added to the tree, and removed later. Meanwhile, an area-list is also maintained by the algorithm. Every leaf node has two pointers pointing to the left and right adjacent areas, respectively. So every time an arc is inserted, deleted or modified, the operation to the areas could be done by the pointers, which will not increase
the running complexity of the algorithm. From top to down, the operations are as follows:

- **h₁, top event** _t₁_: This is the first event, and the tree structure starts from empty. When the sweep line hits the top event _t₁_. The two associated arcs, _l₁_ and _r₁_, are inserted to the tree. Leaf node _l₁_’s left adjacent area is empty, so is _r₁_’s right adjacent area. The associated sensor set of the area _in-between_ _l₁_ and _r₁_ has the sensor _s₁_ only.

- **h₂, top event** _t₂_: Another top event is detected. Now the tree has two leaf nodes. The algorithm searches downwards the tree and finds out the _x_-coordinates of the new event is to the right of arc _r₁_. So the two associated arcs, _l₂_ and _r₂_, will be inserted as the children of the nodes _r₁_. The area _in-between_ _l₁_ and _r₁_ is divided into 3 parts. The left and right unchanged. The middle part, due to the rule of top event, sensor _s₂_ is added to its associated area set. The set now is { _s₁, s₂_ }.

- **h₃, cross event** _c₁_: The two related arcs of this event are _r₁_ and _r₂_. Two arcs of the cross event should swap their location. For the three area affected by this event, the left and right area remains unchanged. For the middle area, _s₁_ is removed and _s₂_ is added to the associated sensor set. Before swapping, the algorithm outputs the middle area as result (if the associated sensor set, which is { _s₁_ }, meets the query criterion).

- **h₄, cross event** _c₂_: The two related arcs of this event are _l₁_ and _l₂_. At this event, the algorithm swaps the location of the two arcs. The left and right areas remain unchanged. For the middle area, _s₂_ is removed and _s₁_ is added to the associated sensor set. Before swapping, the algorithm outputs the middle area as result (if the associated sensor set, which is { _s₂_ }, meets the query criterion).

- **h₅, bottom event** _b₁_: The circle _s₁_ should be removed at this point. Thus, two related arcs _l₁_ and _r₁_ are removed from the tree structure. The middle area is removed as well. The associated sensor set of the middle area is output as result (if the associated sensor set, which is { _s₁, s₂_ }, meets the query criterion).

- **h₆, cross event** _b₂_: The circle _s₂_ should be removed at this point. Two related arcs _l₂_ and _r₂_ are removed from the tree structure. The middle area is removed as well. The set of the middle area is output as result (if the associated sensor set, which is { _s₂_ }, meets the query criterion). The sweep line detects no further events in the queue and the algorithm ends.

### G. Complexity Analysis

The sweep-line technique can be implemented to have an _O_(_n^2 log n_) running time if more complicated data structures such as the red-black tree are used to maintain the sorted list of the intersecting arcs. First step, since the number of sensors is _n_, hence the number of extreme points is 2n. The number of intersections between circumferences is at most _n^2_. The number of events is _O_(_n^2_). Second, the number of leaf nodes in the binary tree is _O_(_n_), since there could be as many as 2n arcs intersect with the sweep line. In our implementation, we instead use a binary-tree based data structure. This binary-tree based data structure is similar in spirit to the skip-list data structure which has an average _O_(_log n_) operation time. Thus for every event, the time on inserting or deleting a node is _O_(_log n_). To sum up, the time complexity of the proposed algorithm is _O_(_n^2 log n_).

### V. Conclusion

We design two effective algorithms to answering range query in wireless sensor networks, where the centers of possible fitting areas are unknown and may not necessarily be the locations of some sensors. The first algorithm uses brute force search and has a running time of _O_(_n^3_), where _n_ is the total number of sensor nodes. The second algorithm adopts a sweep line scanning top-down through the plane to speed up the search process. It has an average running time of _O_(_n^2 log n_). We implemented and tested both algorithms over various application scenarios. Due to space limit, we omit the experimental results in this paper.

### REFERENCES


