Efficient Algorithms for Answering Geo-Range Query

by

Xi Zhang
B.Eng., Beijing University of Chemical Technology, 2005

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Efficient Algorithms for Answering Geo-Range Query

by

Xi Zhang
B.Eng., Beijing University of Chemical Technology, 2005

Supervisory Committee

Dr. Kui Wu, Supervisor
(Department of Computer Science)

Dr. Yong Gao, Co-Supervisor
(Department of Computer Science, UBC)

Dr. Jianping Pan, Departmental Member
(Department of Computer Science)

Supervisory Committee

Dr. Kui Wu, Supervisor
(Department of Computer Science)

Dr. Yong Gao, Co-Supervisor
(Department of Computer Science, UBC)

Dr. Jianping Pan, Departmental Member
(Department of Computer Science)

ABSTRACT

In wireless sensor network, we usually need to combine the information gathered from multiple sensors to detect an event. To answer this question we present a new type of query, Geo-Range query. This query reports the geographic points where the average value of nearby sensors are greater than certain threshold. To perform this query, we developed two fast, efficient algorithms. The Brute-Force algorithm use exhaustive method to enumerate all possible values, which takes $O(n^3)$ running time. The Sweep-Line algorithm applies a conceptual line sweeping through the plane. The sweep-line moves through the plane and keeps tracking all the sensor points encountered. The algorithm takes $O(n^2 \log n)$ running time, while it still gives exact solution to the problem. We implement and simulate our algorithms in Visual Basic.Net.

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

I would like to thank:

Dr. Kui Wu for mentoring, support, encouragement, and patience.

family and friends for supporting me in the low moments.

I would like to thank all people who have helped and inspired me during my Master study. I would like to express my deep and sincere gratitude to my supervisor, Dr. Kui Wu, who has supported me thoughout my thesis with his patience and knowledge. Without his encouragement and effort, this thesis would not have been completed. One could not wish for a better or friendlier supervisor. I am also highly grateful to my co-supervisor Dr. Yong Gao for his guidance and suggestion on my thesis writing. I would like to acknowledge my office and lab mates for their company. I would also like to professors from Computer Science department and all the other people who helped my study during my time in UVic. Finally, I owe my most sincere gratitude to all my family members and dear friends. They have always supported and encouraged me to do my best in all matters of life. To them I dedicate this thesis.

Xi Zhang

Chapter 1

Introduction

1.1 What is Wireless Sensor Networks

With the fast progress of Micro-Electro-Mechanical Systems (MEMS) technology, it becomes feasible to integrate sensors, processors, wireless communications, and energy supply in a very tiny node. Such tiny nodes are normally called sensor motes (or sensor nodes) and can be as small as several cubic centimeters, as shown in Figure 1.1. When a large quantity of sensor motes are inter-connected with wireless communications, they can work together to carry out some very complex tasks. The network system formed by such collaborative sensors is called wireless sensor networks.

Nowadays, wireless sensor networks are widely used in many monitoring and control systems. They have found applications in various fields, such as coal mining [1], sea depth measuring [1], bird migration detection [2], and forest fire control [3, 4]. The ease of deployment and the continuously lowered cost allow the wireless sensors to be deployed in a very large scale. As a result, even if the energy and computational capacity of each sensor node is very limited, the aggregate of many sensor nodes actually possess substantial computational power, rendering large-scale information collection and processing possible.

1.2 Why Geo-Range Query?

After deployment, a wireless sensor network will provide useful information of interests. For instance, in an environmental monitoring system, we may be interested in the temperature, humidity, and air quality in a monitored area. How to effectively ob-

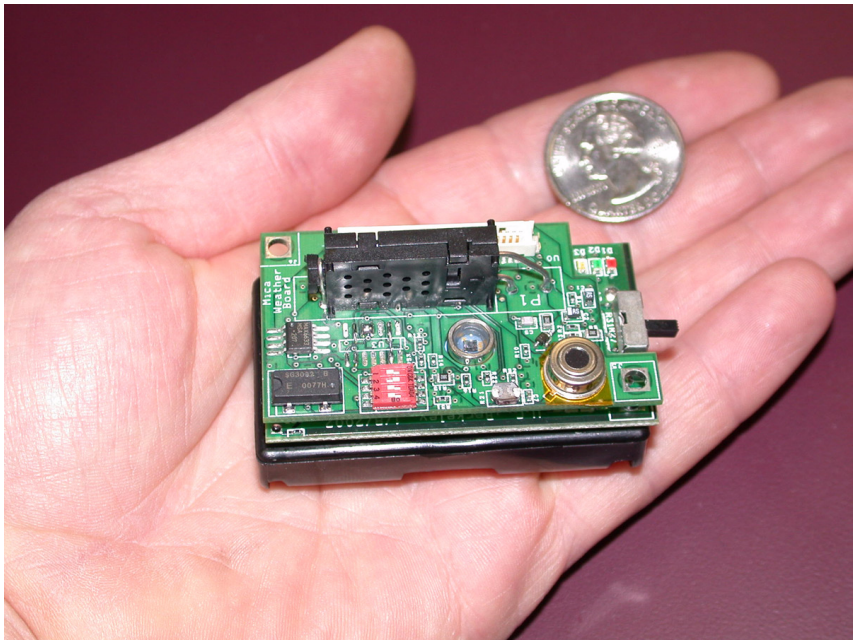

Figure 1.1: A Mica2 sensor mote from Crossbow Inc.

tain the desired information from the low-bandwidth, low-energy, distributed sensor nodes is thus of particular importance.

Recently, various kinds of methods and algorithms have been developed to solve various information query problems in wireless sensor networks, such as top-k query [5, 6], searching for extreme values [7] or median value. There is a common feature in these approaches: they all focus on finding the individual sensors whose sensing readings meet the query condition (e.g., top-k, max or median in the whole network). However, in many cases, queries based on the individual sensors may not be very useful or not very efficient, especially when sensor readings are noisy [8]. For example, when we try to detect a possible forest fire by querying the maximum value in the monitored area, the location of the sensor having the largest sensing reading is considered as the most possible place to catch a forest fire. However, this information may not be reliable and may cause a false alarm, because the individual sensing reading is often noisy and may be an outlier.

In such applications, 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. In the above fire detection example, we need to know the areas where the average temperature is larger than a threshold value because such areas may have a large chance to catch fire. Given the size of the area, we want the monitoring system to return the area that has the highest averaging temperature. As another example, in the ocean monitoring system, if oil leak is detected, we should track the areas and the spread of pollution. In both examples, we need to query the system for areas that meet a given query criterion (e.g., the average value of sensor readings within the area is larger than a value).

The Geo-Range query has two advantages. First, it can effectively capture the event of interests. Some phenomenon, e.g., oil and gas leak, usually spreads over an area, so it is more precise to return the search result in the term of areas. Second, it can effectively limit the impact of some malfunctional nodes or the noisy readings. The conclusion drawn from the all readings in areas is much more accurate and reliable than that from individual sensors.

1.3 Challenges of Geo-Range Query

In most cases, such as the detections of forest fire and ocean pollution, the range of an event starts from one point and expands gradually in all directions. The shape of

the area can thus be approximated as a circle. If we know the center and the radius of the circle, Geo-Range query can be easily calculated with the sensing data falling within the area. Nevertheless, we usually only know the selection criterion (e.g., the average value of an area is larger than a value) but the queried areas are uncertain or unknown, so we need the system to return the centers and the radii of the areas that meet the query condition. We call such areas *Fitting Areas*. To this end, the problem becomes hard to solve 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 may be answered with the results from contour map. There are many research efforts on building contour lines over wireless sensor networks [9, 10, 11, 12]. A contour line is a line along which the sensing data are equal or very close. Nevertheless, we find that 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. For example, the drift of gas leak may make building a contour map nearly impossible. We need to a new approach to solving the Geo-Range query problem.

1.4 Contribution of The Thesis

In this thesis, we focus on effective solutions to the Geo-Range query problem. We develop two polynomial time algorithms for executing Geo-Range query. Particularly, we make the following contributions:

- We propose and analyze a Brute-Force search algorithm.
- We design a Sweep-Line algorithm to speed up the searching procedure during Geo-Range query.
- We test our methods with synthetic data created with different network parameters. The experimental results demonstrate that our methods are effective and efficient.
- We briefly discuss the possible extension of our centralized algorithms to distributed algorithms.

Chapter 2

Related Work

In recent years, data processing in wireless sensor networks has attracted many research interests. Various methods of achieving efficient information query have been proposed, such as boundary detection and contour map. Although they are targeting at different purposes, these algorithms may shed some light on the design of effective algorithms to the Geo-Range query problem. In this chapter, we review some of these algorithms. We also introduce distributed in-networking processing as the background knowledge for our future research towards solving the distributed Geo-Range query problem.

2.1 Boundary Detection Algorithms

The problem of boundary detection is to detect whether or not a sensor resides on the boundary of a network. A sensor node is considered to locate on the boundary of a network, if the node resides on the edges of a sensor network or on the edges of the holes inside the network [13]. Suppose we have a sensor network with all nodes deployed randomly. There can be two kinds of nodes: the nodes that reside on the boundary and the nodes that are in the interior. There may be some holes in the network as well. As illustrated in Figure 2.1, the black nodes are considered boundary nodes.

In applications of wireless sensor networks, boundary detection and hole detection are important problems [14], because they can be used to keep track of the coverage range and detect events entering or leaving the region. The following methods can be used to detect boundary or holes of a sensor network.

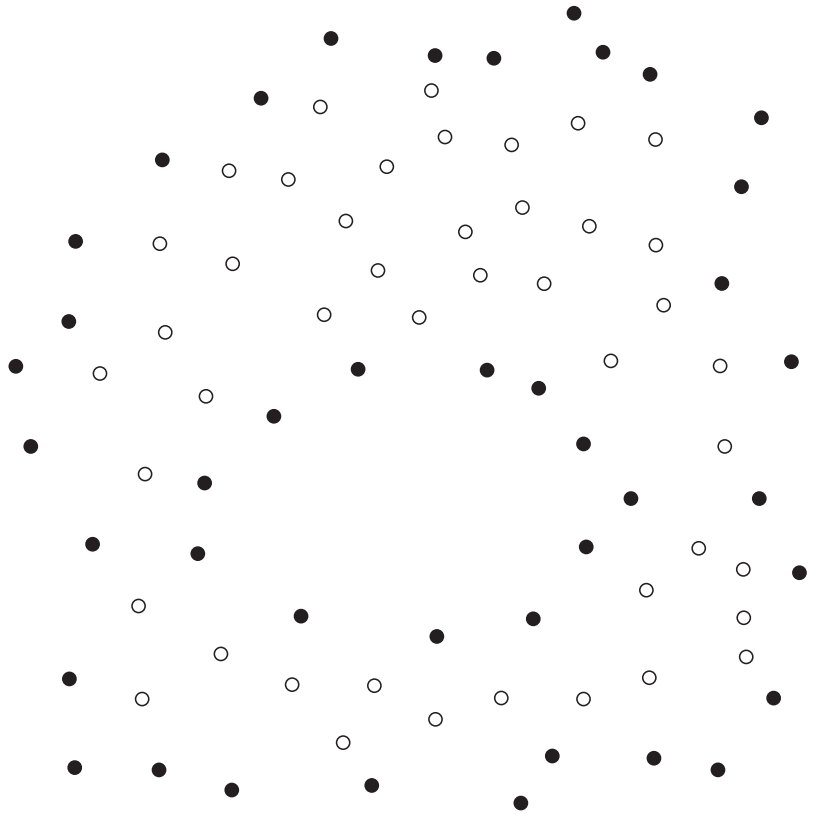

Figure 2.1: The black nodes are boundary nodes

One group of algorithms assumes that the sensor nodes know their exact locations within the network. This method is called Geometrical Approach [13]. For example, In [15], Fang et al. proposed an algorithm to build routes around holes in wireless sensor networks. The algorithm assumes that the nodes know their geographical locations using Global Positioning System (GPS). In addition, the communication graph is assumed to be a unit-disk graph. The authors developed a solution for greedy multi-hop forwarding to solve the problem of stuck packets, i.e., packets that cannot be forwarded to another node based on some given forwarding rules. For example, a forwarding rule could be "forwarding a packet to a neighboring node that is closer to the destination than the current node." If no such a neighboring node could be found, the packet is then stuck at the current node. Such phenomenon is called the local minimum problem. This could happen when the neighboring nodes of a node A all are further away from the destination than node A. In this case, the nodes that cause packet to get stuck are considered to form a hole of the network. By detecting holes, intelligent routing protocols can be design to let packets bypass the holes [13].

Another group of methods uses complex statistical and mathematical computations [13]. The algorithms proposed under this category identify the nodes as interior or boundary nodes by assuming that the node distribution in the network follows some statistical distribution. Fekete et al. [16] propose a straightforward, distributed protocol to allow nodes within a uniformly deployed sensor network to decide whether or not they are boundary nodes. The algorithm uses real-valued functions, called centrality indices, to determine the relative geographic locations among nodes. The basic idea is to assign higher values to more central nodes and lower values to the nodes close to boundaries. As such, the key issue is on the assignment of centrality indices. The algorithm in [16] makes use of the shortest paths containing a given vertex in the assignment of centrality indices. A technique called "restricted stress centrality" is then used to make decision. To this end, a threshold value is determined and distributed among all the nodes in the network. By comparing its "restricted stress centrality" to the threshold value, a node can estimate whether or not it is an interior or boundary node. Namely, a node having a "restricted stress centrality" higher than the threshold is considered an interior node. Otherwise, it is supposed to be a boundary node.

Another type of algorithms belongs to the "topological approach". The algorithms under this category make use of the topological information available to sensor nodes

in the network. Kroller et al. [17] search for combinatorial structures, called flowers and augmented cycles, in the network to detect boundary. The algorithm proposed by Wang et al. [18] uses global flooding (i.e., transmitting a message to the whole network) to build a shortest path tree. Holes are detected by making use of Nearest Common Ancestors. Funke [19] presents an algorithm that finds out holes inside the sensor network. It uses iso-contours to detect holes and boundaries. The basic idea is to build iso-contours based on hop count from a given root node. The end nodes of broken iso-contours are considered as boundary nodes.

2.2 Contour Map

Boundary detection is to find out whether or not a sensor resides on the boundary of a network. In some applications, we instead want to find the boundary of a phenomenon. For instance, in the detection of oil leak in ocean, we want to know the polluted area, which is described by the boundary (or edge) of the area. There are two types of methods for this purpose: one is binary detection; the other is to build contour line. The binary detection method makes a binary decision (i.e., whether or not a sensor is inside or outside the phenomenon area) [20, 9]. This type of decision may not very useful for some applications, because the underlying physical phenomena may be better described by a range of values rather than a binary decision. For instance, in the detection of oil leak, we may not be able to simply claim a sensor is inside or outside a polluted area. Instead, it may be more proper to build a map, called contour map, to describe the density of oil in different locations. Contour map consists of contour lines. A contour line is defined as a two-dimensional curve on which the measurement values of the phenomenon are a constant [10, 11, 12]. Figure 2.2 shows an example of the contour map in the detection of oil leak in ocean.

Chintalapudi et al. [20] designed distributed algorithms to detect the boundary of a phenomenon. They used binary decision and for this purpose they defined the notion of edges, a concept same as the boundary of phenomenon. They also developed performance metrics to evaluate localized edge detection algorithms. In the algorithms, each sensor gathers information from its local neighborhood and determines whether or not it is close to an edge.

Liao et al. [9] developed a distributed algorithm to build contour lines. After the contour lines are built, one can further locate the center and the average value of the observed phenomenon in interest. This algorithm improves the algorithms in [20] by

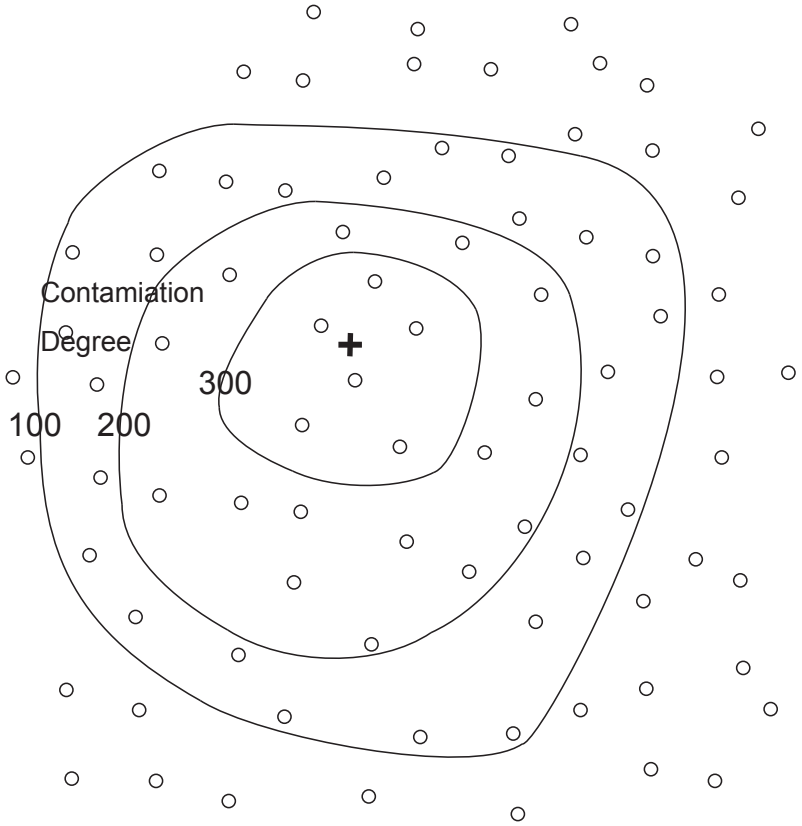

Figure 2.2: An example contour map

returning contour lines instead of binary decisions.

Liu et al. [21] introduced the concept of "isobath" into contour map building. Simply put, a contour map is built with a set of contour regions outlined by isolines of different isolevels. They designed an algorithm that intelligently selects a set of nodes as "isoline nodes." The contour lines are built based on information gathered from isoline nodes. This algorithm reduces the cost significantly while it still returns result with good quality.

In [22], Gandhi et al. use the same concept of "isobath." They considered the problem of approximating a family of isocontours in a field with a topologically-equivalent family of simple polygons. In other words, they use a polygon to approximate the contour lines. This approximation results in a much fast algorithm. In addition, the approximation error can be effectively controlled by tuning parameters of the polygons.

2.3 In-network Aggregation

In-network processing, particularly in-network aggregation, may be helpful in solving the Geo-Range query problem in a distributed fashion. In this thesis, however, we only focus on centralized solutions but leave detailed design of distributed algorithms open.

2.3.1 Aggregation Structure

According to the information-rich characteristics of wireless sensor networks, a network could be considered as a virtual database system [23]. Viewing the network as a virtual database system requires a friendly interface so that a user can extract the data of interest from the network. This is achieved by injecting queries to the network and asking sensor data to be sent back to the user via multi-hop communication. One such example is TinyDB, which maintains a routing tree structure and uses controlled-flooding to send the queries throughout the network [24].

Another data gathering approach is to use clustering methods, such as LEACH [25]. To save the energy of cluster heads, LEACH incorporates randomized rotation to select a cluster head within a cluster. In this way, different nodes would be chosen as cluster heads to balance the energy consumption in the whole sensor network. The cluster head is responsible for collecting data from the cluster and sending the

(aggregated) data to the processing center (some times also called the base station).

2.3.2 Aggregation Operators

Built upon an aggregation structure, how to calculate given aggregation function is another important problem in many sensor applications. For instance, the aggregation function could be a top-k query (i.e., finding the set of k sensor nodes that have the highest (or lowest) readings). How to answer continuous top-k queries in an energy-efficient way is a great challenge [26]. A simple implementation of answering the top-k query can use a centralized approach where all sensor readings are collected by the base station and then are processed at the base station. Clearly, this simply approach is not energy-efficient since it requires too many message transmissions and thus consume too much energy. To solve the problem, an in-network data aggregation technique, known as TAG, has been proposed in [27].

Other commonly used aggregation functions include AVERAGE, SUM, and COUNT. However, there are other complicated aggregations like MEDIAN. In [28], q-digest based MEDIAN operation is proposed. q-digest can be thought of as a histogram whose buckets contain the number of occurrences of each values. However, the overhead of sending entire histogram might be comparable to that of sending entire values if the size of the buckets is not wisely managed. To solve this problem, q-digest takes advantage of information redundancy among sensor nodes, and as such it is unnecessary to extract all sensor readings. With the above observation, q-digest merges two buckets if the number of occurrences in each bucket is smaller than a predefined threshold.

Chapter 3

Problem Formulation

In this chapter, we formulate a Geo-Range query problem, which this thesis mainly focuses on. The answer to this query will provide a basic building block to solve more complex Geo-Range query.

Let $S = \{s_1, s_2, \ldots, s_n\}$ be a set of n sensors deployed in a monitoring field. Let R be a predefined diameter of a circular area, the centre 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 c_i 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 $r \equiv R$

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 $\Psi_r(A_i) > T$. Note that Ψ_r is an aggregate function over a set of sensor nodes and is application dependent. For example, $\Psi_r(A_i) = \frac{\Sigma_{i \in A_i} Temp(s_i)}{\|A_i\|}$ 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). Fig. 3.1 illustrates the relationship between sensors and the range of query, where $A_1 = \{s_1\}$ and $A_2 = \{s_1, s_2\}$.

Due to spatial correlation among sensor readings, we assume that the difference of two sensor readings is upper-bounded by a value proportional to their distance, i.e., we assume that $|\delta(v_i, v_j)| \propto d(s_i, s_j)$ where v_i, v_j are sensor readings from sensors s_i, s_j, respectively, $|\delta(v_i, v_j)|$ is the absolute difference between v_i and v_j, $d(s_i, s_j)$ is

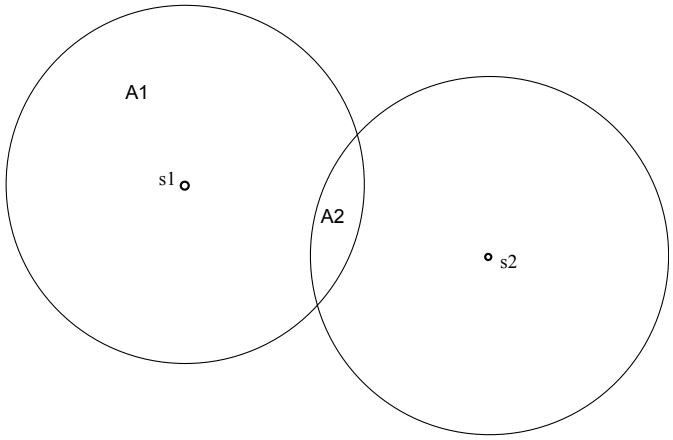

Figure 3.1: Illustration of areas and sensors.

the distance between the two sensors. Usually, wireless sensor network applications require spatially dense sensor deployment in order to achieve satisfactory coverage [29]. Such a spatial correlation is an important characteristic, which we will study in our later performance evaluation of proposed algorithms.

The rest of the thesis is organized as follows. A brutal-force search algorithm is introduced in Chapter 4. In Chapter 5, we introduce a Sweep-Line algorithm to reduce the time complexity in searching. In Chapter 6 we test the algorithms with simulation. In Chapter 7, we conclude the thesis and introduce future work.

Chapter 4

The Brute Force Search

4.1 Notation

For the ease of reference, we list the notation used in the rest of chapters in Table 4.1. For illustration purpose, we assume that the aggregate function in the Geo-Range query is to calculate the average value. Using other aggregation function does not impact the efficiency and correctness of our algorithms.

Table 4.1: Symbol Table

Symbol	Meaning
s_i	a sensor node
v_i	the value of interest from sensor node s_i
$p(x, y)$	Geographic point with location (x, y)
I	the set of all intersections
S	the set of all sensors
IS_i	the set of all inner sensors w.r.t. the intersection i
BS_i	the set of all border sensors w.r.t. the intersection i
R	the radius of the queried range
T	the threshold value in the Geo-Range query

4.2 Problem Analysis- Listing Representative Centers

Our first solution is similar to the moving windows technique applied in the field of Digital Image Processing to smooth an image. Digital image is a pixel matrix, and

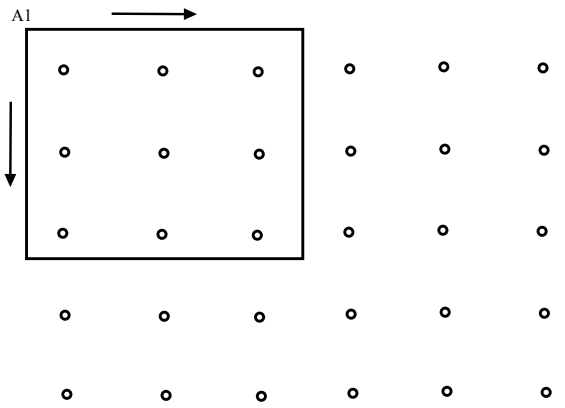

Figure 4.1: Moving window with a rectangular shape

the moving windows technique uses a rectangular fix-size window to calculate the average pixel value within the window, as shown in Fig 4.1. Nevertheless, in our case, the shape of the moving windows is a circle, as shown in Figure 4.2. Therefore, a new method should be developed to solve the problem when the shape of moving windows is circular.

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 all the 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. The above observation makes the design of a brute-force search algorithm computationally feasible.

Suppose that a set of sensors $S = \{s_1, s_2, ..., s_n\}$ are given. For each sensor s_i, we can draw a circle centered at 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 areas. 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. 4.3, for

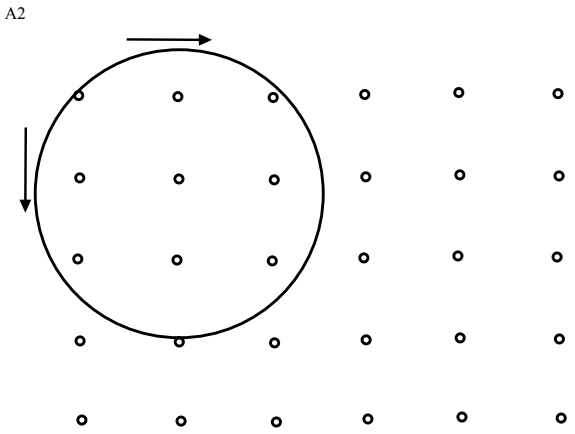

Figure 4.2: Moving window with a circular shape

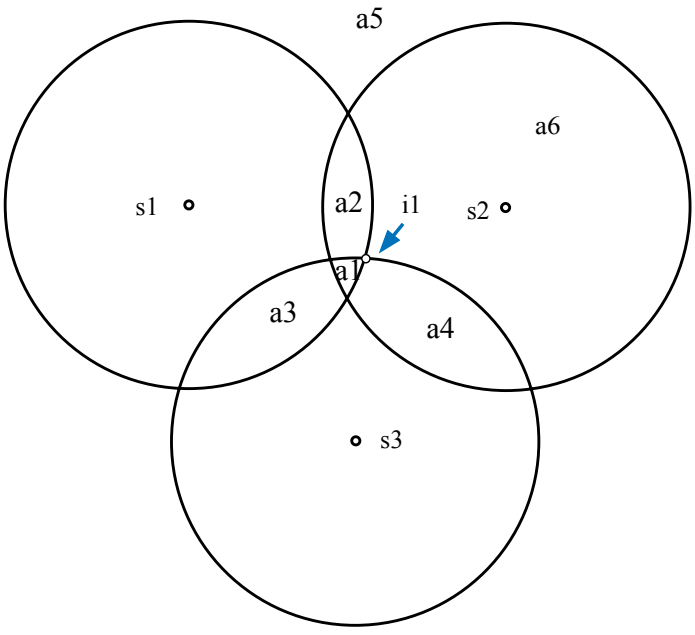

Figure 4.3: Dividing plane

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 Figure 4.3, a_1, a_2, a_4 and a_6 are adjacent areas of intersection i_1. Near 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 exactly 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 call the sensor a outer sensor, and we can simply ignore this sensor when the intersection point is in consideration.

Based on the above analysis, our brute-force search 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 query criterion. Algorithm 1 illustrates the steps. As an example, it uses the calculation of average value as the aggregate function in the Geo-Range query.

In Algorithm 1, the data structures I is to record intersection points. The function $GeoDistance(s_j, i)$ is to calculate the distance between sensor s_j and intersection point i. The function $AverageValue(IS_i, BS_i) > T$ is to calculate the average values from sensors in IS_i and in BS_i, i.e., it is to check the query condition.

Algorithm 1 Search for possible centers

INPUT: Sensors set S, threshold T, and searching range R.
OUTPUT: *Result*, the set of centers of circular areas that meet the query condition.

```
for each s_i, s_j ∈ S do
  I.Add(ComputeIntersection(s_i, s_j))
end for
for each i ∈ I do
  for each s_j ∈ S do
    if GeoDistance(s_j ,i) < R then
      IS_i.Add(s_j)
    end if
    if GeoDistance(s_j ,i) ≡ R then
      BS_i.Add(s_j)
    end if
  end for
  if AverageValue(IS_i, BS_i) > T then
    Result ← i
  end if
end for
return Result
```

The returned centers from Algorithm 1 are representative centers of circular areas meeting the query condition. It is impossible to find *all* centers because theoretically, an area includes infinite number of points. Nevertheless, Algorithm 1 provides us with a comprehensive list of representative centers. Based on the result, we can further find all possible areas, any point within which can be used as the center of a circular region meeting the query condition. We call such areas **fitting areas** and we need to list all fitting areas. This is illustrated in the next section.

4.3 Listing All Fitting Areas

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 4.4. When $k(k \geq 2)$ circles intersect at the same point, the plane is divided into $2k$ sub-area around the intersection point. As shown in Fig 4.4, 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. to any point in the sub-areas around 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. 4.3, 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 fitting area.

To avoid another around of global search, we can re-use the existing result in the previous section. Based on the above analysis, when checking a sub-area surrounding an intersection point i, values from the sensors in IS_i should be considered, but values from some sensors in BS_i may need to remove. This task turns out to be easy, due to the joyful fact that two adjacent sub-areas surrounding an intersection can only have one different value. For example, in Fig. 4.3, 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 surrounding 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 surrounding the intersection point i_1, a_1, a_2, a_6, a_4, respectively. When checking these sub-areas in the clockwise direction, we need to consider sensors s_1, s_2, s_3 when checking a_1, sensors s_1, s_2 when checking a_2, sensors s_2 when checking a_6, and sensors s_2, s_3 when checking a_4.

As another example as shown in Fig4.4, four circles intersect in the same point and

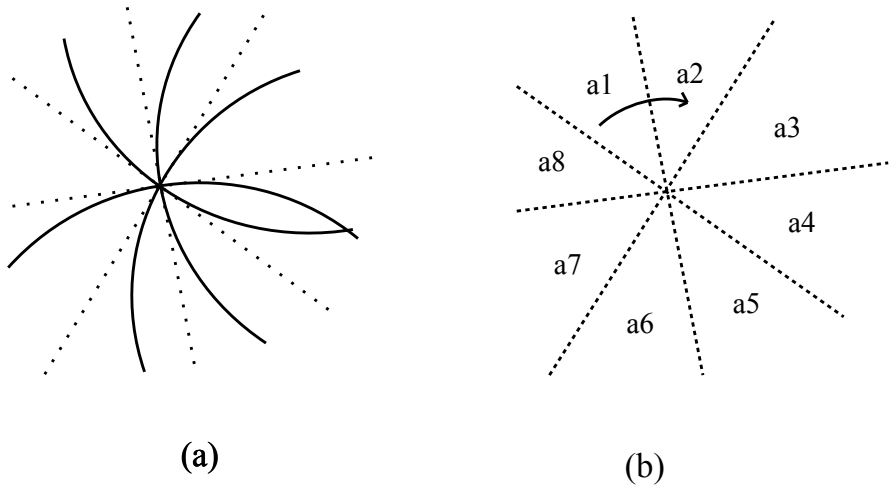

Figure 4.4: Multiple circles intersect at the same point

they cut the area into 8 sub-areas surrounding the intersection, a_1 to a_8, respectively. We can start from computing the average value of arbitrary area, say a_1. By moving through the sub-areas in the clockwise direction, as shown in Fig4.4(b), the average value of adjacent sub-areas can be easily computed.

4.4 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 mul-

tiple 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 $O(n^3)$.

Chapter 5

The Sweep-Line Method

The implementation of the Brute-Force search algorithm is straightforward. However, the algorithm is inefficient in some ways. For example, the overlapped area a_5 in Figure 4.3 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, a Sweep-Line algorithm with a binary tree data structure is designed in this chapter.

5.1 Sweep-Line Technique

The sweep-line technique makes use of the order of related geometric objects [30, 31]. Furthermore, algorithms for constructing an Euclidean Voronoi diagram of circles and an Euclidean Voronoi diagram of circles contained in a larger circle have been developed recently [32]. In paper [31] Kim et al. developed a Sweep-Line algorithm for hierarchy of the circles. Paper [30] gives an algorithm to compute the intersections of polygons. In this chapter, we will borrow the idea of Sweep-Line technique to compute circle intersections.

5.2 Sweepline 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 t_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 split into two equal length semicircle arcs, the left arc l_i and the right arc 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.

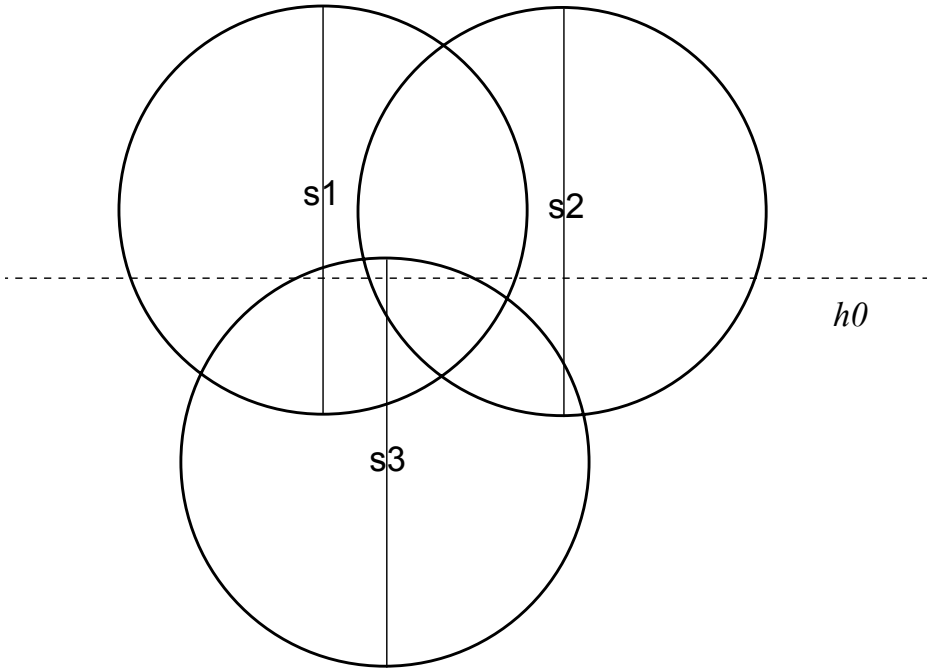

Figure 5.1: Sweep-line intersects with circle arcs

A sweep-line is defined as a horizontal line moving from the top to the bottom of the plane. As it moves downwards, the intersection between the sweep-line and the circles arcs changes. The key to the algorithm is to maintain the state of Sweep-Line, which is a list of arcs currently intersect with the Sweep-Line. The arc-list is sorted by the X-coordinate of the intersections. For example in Figure 5.1, suppose the left and right arcs of circle $s_i(i = 1, 2, 3)$ 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 and r_2. The purpose of the Sweep-line technique is 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.

5.3 Event and Event-list

Notice that the arc-list changes if and only if when the sweep line hits an event point: When it hits a top event, two new arcs (of the same circle) are added to the state of the sweep-line; when it hits a bottom event, two existent arcs (of the same circle) are removed; when it hits a cross event, the number of arcs in the arc-list remains unchanged, but two arcs should swap their positions. 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.

5.3.1 Enumerating Areas

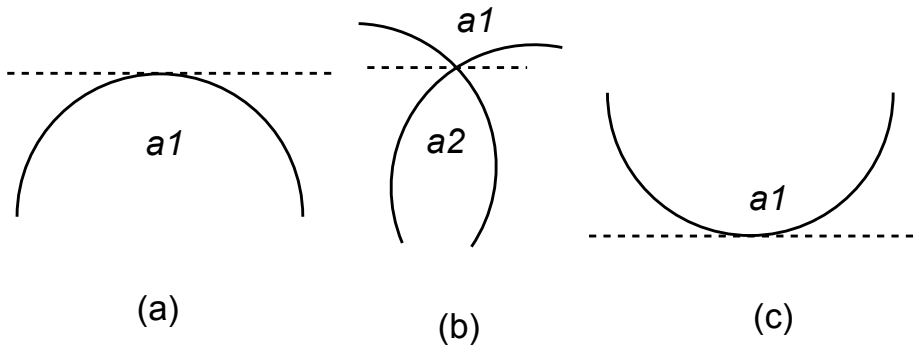

Figure 5.2: The alteration of areas at events

A list of areas is also maintained when the line sweeps through the plane. At a top event, a new overlapped area will emerge at this point. See Figure 5.2 (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 figure 5.2 (c), area a_1 will close and leave the sweep-line forever after it hits the bottom event. At a cross event as in figure 5.2 (b), however, an existent area a_1 will close at this point, as well as 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 subsection.

5.3.2 Data Structure of Sweep Line

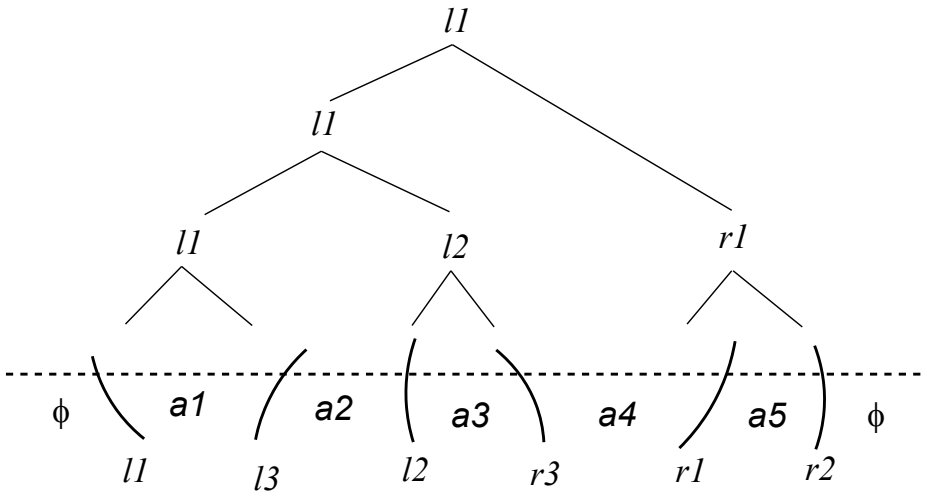

Figure 5.3: Data structure of the Sweep-line

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 inter-

secting with the sweep line. The arcs are sorted by the X-coordinates of their leftest point. Thus the arc list in Fig. 5.1 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. 5.3 is an example of a binary tree built from Fig. 5.1, 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 deleted 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.

5.3.3 An Example Illustration

To exemplify the procedure of the algorithm, we use an example with only two sensors. Fig. 5.4 shows two generator circle s_1 and s_2 intersecting with each other. Fig. 5.5 explains how the data structure is constructed and maintained during the process.

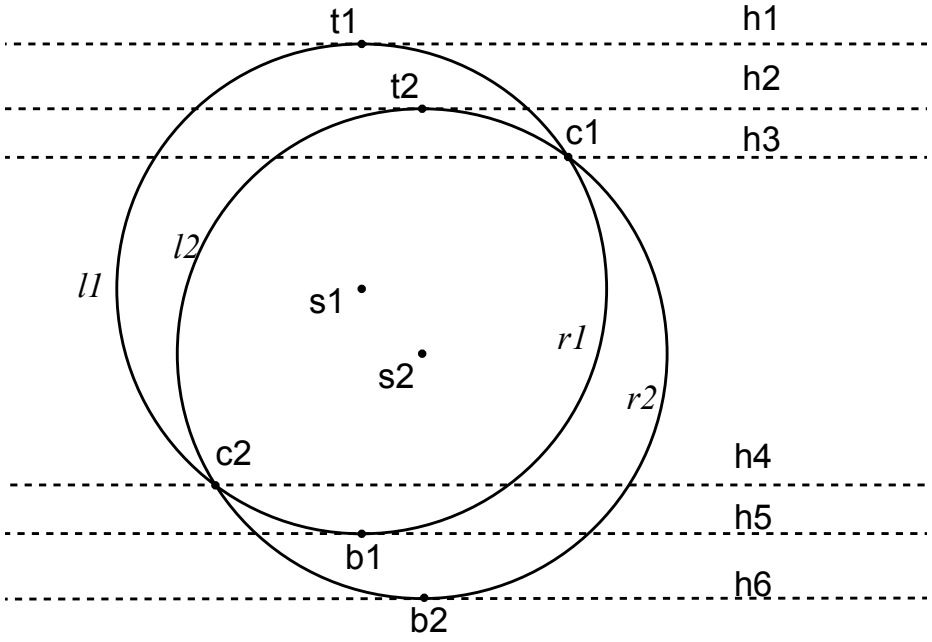

Figure 5.4: Illustration of Sweep-line moving through the plane

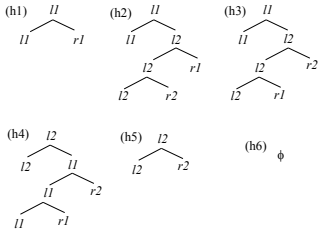

Figure 5.5: Illustration of tree changes

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:

- h_1**, top event** t_1**:** This is the first event, and the tree structure starts from empty. When the sweep line hits the top event t_1. The two associated arcs, l_1 and r_1, are inserted to the tree. Leaf node l_1's left adjacent area is empty, so

is r_1's right adjacent area. The associated sensor set of the area in-between l_1 and r_1 has the sensor s_1 only.

- h_2**, top event** t_2**:** 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 r_1. So the two associated arcs, l_2 and r_2, will be inserted as the children of the node r_1. The area in-between l_1 and r_1 is divided into 3 parts. The left and right unchanged. The middle part, due to the rule of top event, sensor s_2 is added to its associated area set. The set now is $\{s_1, s_2\}$.

- h_3**, cross event** c_1**:** The two related arcs of this event are r_1 and r_2. 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, s_1 is removed and s_2 is added to the associated sensor set. Before swapping, the algorithm outputs the middle area as result (if the associated sensor set, which is $\{s_1\}$, meets the query criterion).

- h_4**, cross event** c_2**:** The two related arcs of this event are l_1 and l_2. At this event, the algorithm swaps the location of the two arcs. The left and right areas remain unchanged. For the middle area, s_2 is removed and s_1 is added to the associated sensor set. Before swapping, the algorithm outputs the middle area as result (if the associated sensor set, which is $\{s_2\}$, meets the query criterion).

- h_5**, bottom event** b_1**:** The circle s_1 should be removed at this point. Thus, two related arcs l_1 and r_1 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 $\{s_1, s_2\}$, meets the query criterion).

- h_6**, cross event** b_2**:** The circle s_2 should be removed at this point. Two related arcs l_2 and r_2 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 $\{s_2\}$, meets the query criterion). The sweep line detects no further events in the queue and the algorithm ends.

Thus, the entire plane is scanned by the sweep-line. Four data set of associated sensors: $\{s_1\}$, $\{s_2\}$, $\{s_1, s_2\}$ and $\{s_2\}$ have been output. Although there is redundancy

Algorithm 2 Sweep-line Algorithm

INPUT: Sensors set S, the threshold T and searching range R.
OUTPUT: *Result*, the combinations of sensors from S which could be encircled by R.

```
for each s_i, s_j ∈ S do
  EventQueue.Add(ComputeIntersection(s_i, s_j))
end for
for each s_i ∈ S do
  EventQueue.Add(ComputeExtremePoints(s_i))
end for
EventQueue ← EventQueue.QuickSort()
while EventQueue.GoNextEvent() do
  if EventQueue.Event ≡ TopEvent then
    Insert arcs to the binary tree
  end if
  if EventQueue.Event ≡ CrossEvent then
    if the value of middle area is > T then
      Result ← coorespondingarea
    end if
    Swap the intersected arcs
  end if
  if EventQueue.Event ≡ BottomEvent then
    if the value of middle area is > T then
      Result ← coorespondingarea
    end if
    Remove arcs from tree structure
  end if
end while
return Result
```

in the output result, we can prove that the time complexity is still improved compared to the brute-force search algorithm, as analyzed in the next section.

5.4 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 used 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 $O(n^2 \log n)$.

Chapter 6

Further Discussion

In previous sections, we designed two fast algorithms to solve Geo-Query problem. Both algorithms are centralized which requires detailed data from WSN. Under certain circumstance, the procedure could be energy consuming.Different from centrolized algorithm, distributed algorithm usually provides fast approximated solution. The relationship between two type of algorithm could be seen as a trade off between efficiency and accuracy. In this chapter, we discuss future of applying distributed algorithm to Geo-Query problem.

One straightforward method is that each sensor computes for itself. A sensor can compute the average value locally if it has gathered all the data from related sensors. To this end, a limited flooding scheme could be used for sensors sending data to neighbours. Each sensor sends packet including geographic information and sensing value to nearby sensors. The radius of query R is considered as the flooding limitation. Packet will be automatically dropped when it reaches a location that is R away from its source. Once having all the information, sensor can decide if the average value is above threshold T. It is no denying that this method will be faster than the centralized algorithm developed in this paper. The disadvantage is that, the algorithm will only report fitting areas have sensors located within the area.

Although distributed algorithms improve the performance,they all require compromise of accuracy on some level. The problem of applying distributed algorithm to get exact Geo-query solution remains unsolved.

Chapter 7

Experimental Results

In this chapter, we evaluate the performance of our algorithms with numerical simulation. The simulation model includes two major components: sensor deployment and sensing value generation.

7.1 Sensor Deployment

We simulated two sensor deployment strategies on a square field: random deployment and grid deployment. Assume that the field is a square field with side length denoted by L. To remove the impact of field boundaries and to simplify simulation, we enforce that sensors nodes are located at least R away from the border of the field (i.e., the sensors are placed in the square area with side length of $L - R$, where R is the range value in the GeoRange query. We call this smaller square area *the area of interest*. We note that when L is much larger than R, the above constraint can be safely ignored.

For the random deployment, sensors are placed uniformly random in the area of interest. For grid replacement, sensors are placed on the intersection points of a rectangular grid. Note that we slightly rotate the grid (as shown in Figure 7.1) in the simulation to make output visually clearer.

7.2 Sensing Value Generation

We use three different ways to create sensing values as follows.

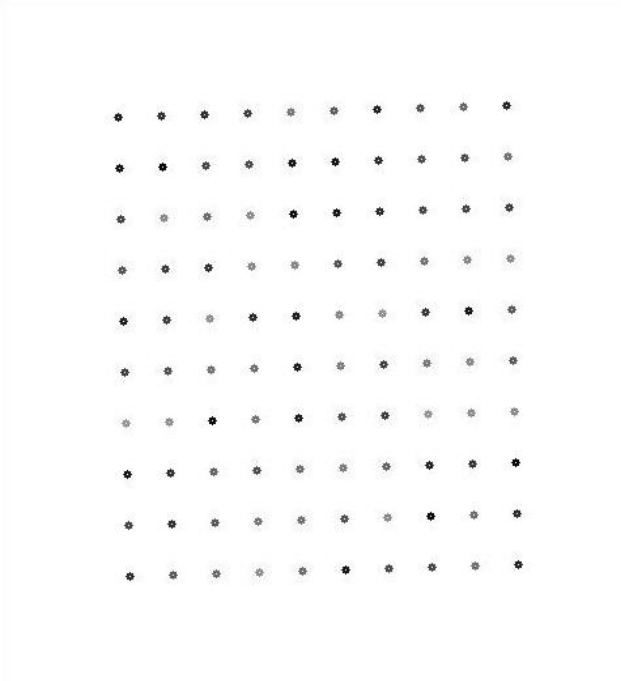

Figure 7.1: Grid deployment

7.2.1 Uniformly Random

In this method, each sensor selects a value chosen uniformly random between 0 and Max, where Max is the maximum sensing value in the whole area.

With this method, if we map all sensor values to the plane, we can get Figure 7.2, in which an area with a darker colour has a higher sensing value. This picture gives us the rough idea of the sensing value distribution in the field.

7.2.2 Spatial Correlation Model

In the spatial correlation model, we assume that the difference of two sensor readings is upper-bounded by a value proportional to their distance, i.e., we assume that $|\delta(v_i, v_j)| \leq \alpha \times d(s_i, s_j)$ where v_i, v_j are sensor readings from sensors s_i, s_j, respectively, $|\delta(v_i, v_j)|$ is the absolute difference between v_i and v_j, $d(s_i, s_j)$ is the distance between the two sensors, and α is a given coefficient value.

Similar to Figure 7.2, Figure7.3 shows the sensing value distribution within the field with values generated by the spatial correlated model. It can be seen that the spatial correlation model creates values that spread more smoothly over the field.

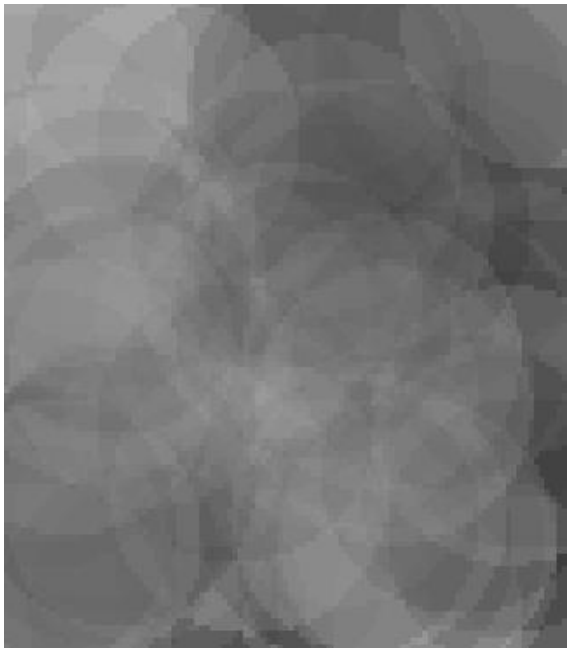

Figure 7.2: Sensing values with the uniformly random method

7.2.3 Spread Model

To make the sensing values represent a group of realistic applications, we adopt a simulation technique suitable for modelling the diffusion phenomena over the sea surface. The method uses Cellular Automata technique. Cellular Automata (CA) is a class of Automata defined for the simulated field, which is divided into discrete areas called "cells." The method was first developed by John von Neuman and Stanislaw Ulam in 1940s [33].

In this model, we borrow the idea from [34] to simulate the diffusion of oil in water. First, we divide the (water) field into cells. A group of cells includes the contamination material (oil) and density of the material in those cells may be different. Some cells represent unpenetratable areas, e.g., islands. When the process starts to run, the contamination will spread all over the water field. When the process stops, each sensor will obtain the sensing value of the nearest cell's contamination level (i.e., the density of the contamination material in the cell). Clearly, we need to define the diffusion process to model the density changes of contamination materials in the cells. A diffusion process is simulated with discrete time steps, with each step representing a round of propagation of contamination material. We use the same diffusion process

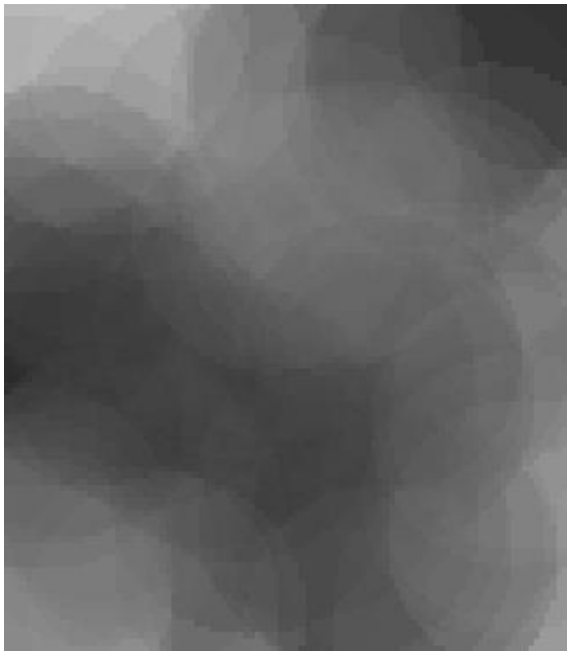

Figure 7.3: Sensing values with the spatial correlation model

as in [34]. Briefly speaking, the diffusion process without direction is described with a partial differential equation:

$u^n_{x,y} = \frac{1}{9}\sum_{i=-1}^{1}\sum_{j=-1}^{1} u^{n-1}_{x+i,y+j}$

Where $u^n_{x,y}$ is the contamination level of cell (x, y) at step n. The equation works as an average operator in Figure 7.4. Similarly, we can use the operator in Figure 7.5 to describe directional diffusion.

An example is shown in Figure 7.2.3. The field is divided into cells. The black cells represent unpenetratable areas, which are generated randomly; gray cells represent different level of contaminated material, and the darker the color, the higher the contamination level is. In the initialization of the simulation, the contaminated cells are placed on the same place on the field. These cells are high level of contamination, thus they appeas as gray cell in Figure 7.6. We then assume that the field has constant water current from the left toward the right. With the diffusion process used in [34], the contamination material spreads to the entire field (except the area denoted by gray cells).

Figure 7.7 and Figure 7.8 show the status on diffusion steps 70 and 140, respectively.In simulation, we use the values after the 70^{th} diffusion step.

Since the cells are divided from the plane, there are some cells will contain one or

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

Figure 7.4: Operator with no direction

0	1/9	2/9
0	1/9	2/9
0	1/9	2/9

Figure 7.5: Operator with direction

more sensors. With the assumption that sensor extrat sensing value locally, a sensor's value is set to be the contamination level of the cell where it resides in.

Figure 7.6: Sensing values at time 0

Figure 7.7: Values after 70 steps

Figure 7.8: Values after 140 steps

7.3 Simulation Parameters

The size of the plane is 500×500 square meters. The query range is set to 80 meters. To test the performance under different scales, the number of sensors is set to 20, 25, 30, 35, 40, 45, 50, respectively. Both the brute-force search algorithm and the sweep-line algorithm are tested. In all cases, the sensing values are bounded within the range of $[0, 100]$. When the average of sensing values within the circular area of

80 meters is larger than 70, the area is reported as a fitting area (i.e., an area meeting the query criterion). The simulation program was written in VB.NET, running on Windows XP system with Intel Core2 T7200 CPU and 1GB RAM.

The total running time is considered as the performance of the algorithm. For each set of parameters, we run the simulation 1000 times to get the averaged running time.

7.4 Testing Result

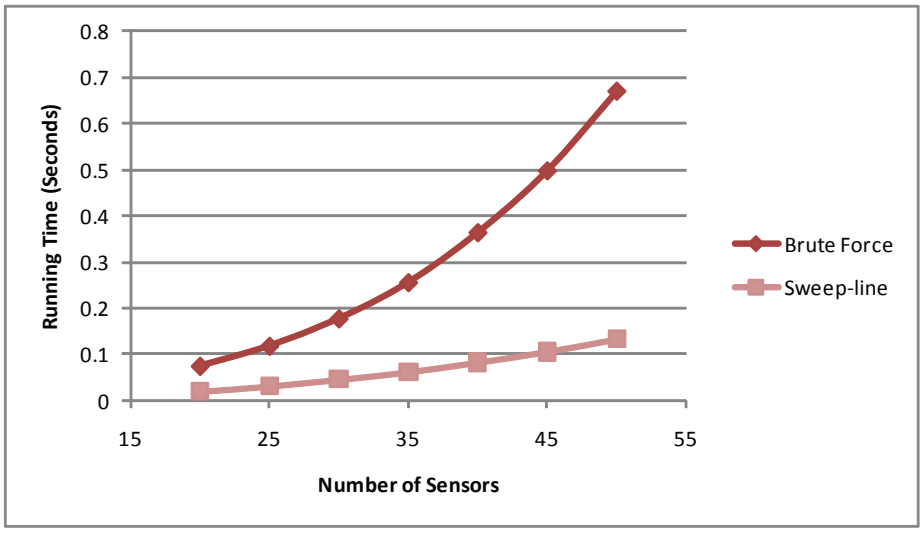

Figure 7.9: Sweep-line algithm improvemence

Figure 7.9 compares the average running time between the brute-force search algorithm and the sweep-line algorithm under random sensor placement. The result is combined from three different value generation methods. where X-axis shows the number of sensors, and Y-aixs shows the running time of the algorithms. From the figure we can see that that the sweep-line algorithm is significantly faster than brute-force algorithm. As we have analyzed before, the time complexity of the brute-force algorithm is $O(n^3)$, and the time complexity of the sweep-line algorithm is $O(n^2 \log n)$. Figure 7.9 shows that the running time of the brute-force algorithm goes up more quickly than the sweep-line algorithm as the number of sensors increases. This confirms our theoretical analysis on time complexity.

For the same set of simulation parameters, we use a metric *improvement ratio*, defined as the ratio of the running time of the brute-force algorithm over the running

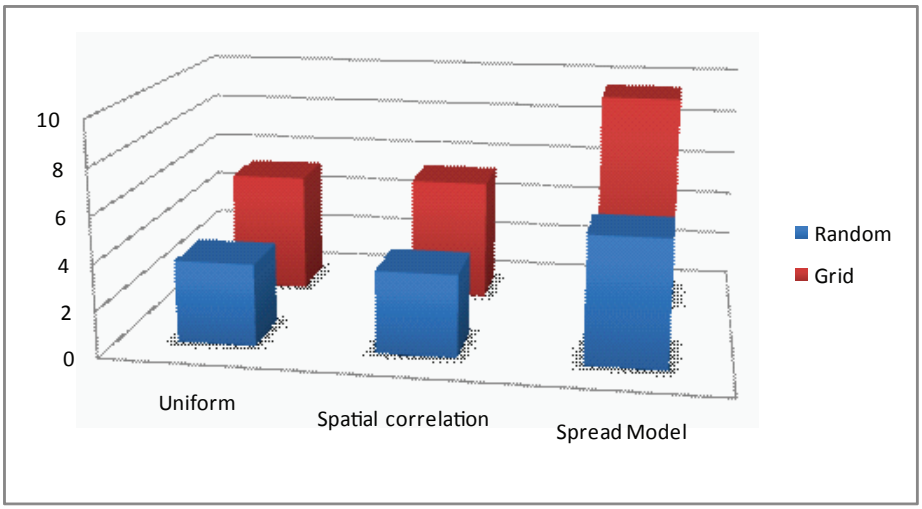

Figure 7.10: Improvement ratio

time of the sweep-line algorithm. Figure 7.10 shows the improvement ratio under different testing scenarios, with the number of sensors changing from 20 to 50. With the spread model and grid sensor placement, the sweep-line algorithm has the largest improvement over the brute-force search algorithm and is 9.53 times better. With the spatial correlation model and the random sensor placement, the sweep-line algorithm has the smallest improvement over the brute-force algorithm and is 3.57 times better. In any means, even the smallest improvement is significant.

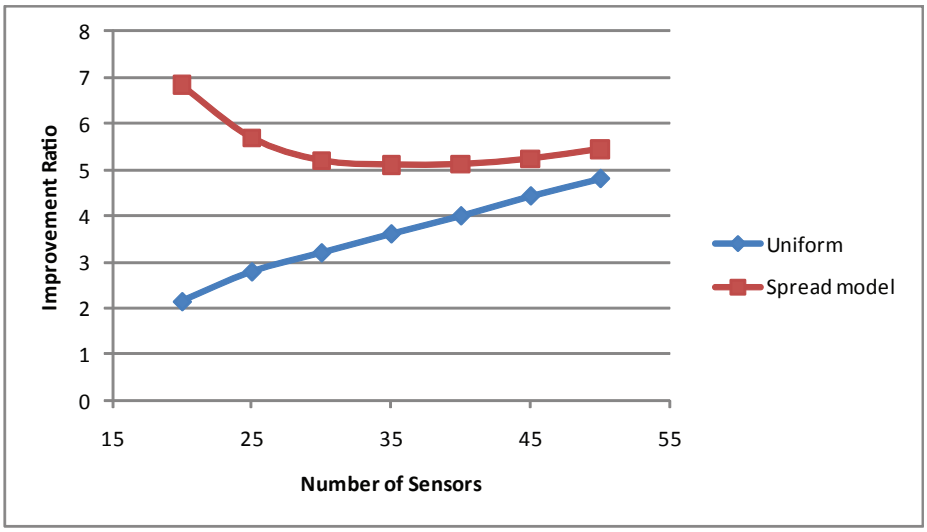

Figure 7.11: Improvement ratio based on different generation methods

Figure 7.11 shows the improvement ratio with various numbers of sensors under random placement. Because the test results using uniformly random data model and the test results using the spatial correlation data model are almost identical, we only show the uniformly random data model. From the figure, we can see that the improvement of the sweep-line algorithm over the brute-force search algorithm becomes larger when the number of senors increases.

Chapter 8

Conclusion

In this thesis, we design and implement algorithms to solve the Geo-Range query problem: In wireless sensor networks, given a set of sensor readings, we want to know all areas within which the average value of sensor readings is greater than a certain threshold.

We provide two fast, effective algorithms to answer Geo-Range query: the brute-force search algorithm and the sweep-line algorithm. The brute-force search algorithm uses exhaustive search to enumerate all possible sub-areas that meet the query criterion. The algorithm takes $O(n^3)$ running time, where n is the number of sensor nodes. We then use a virtual line, called sweep-line, to reduce the running time complexity. The virtual line sweeps top-down through the field and keeps track of all the sensors' locations. A binary tree is maintained during the procedure to reduce running time. The algorithm takes $O(n^2 \log n)$ running time, while it still outputs exact solution to the Geo-Range query problem.

Equipped with the insights gained in this study, we see a lot of opportunities of using our algorithms in real-world applications. For example, in the application of detecting forest fire or ocean oil leak, our algorithms can be used to return abnormal areas that are likely to be in danger.

Meanwhile, there are several interesting open challenges for future work. One interesting question is to answer a range query without a pre-defined radius, or answer a range query with different shapes of a queried area. Another interesting problem is to design distributed algorithms for Geo-Range query. Our algorithms are centralized and as such it assumed that data have been collected to the processing center. To save energy, in-network processing with distributed algorithms is more desirable. Unlike centralized algorithms, distributed algorithms usually provide fast approximate

solution. Obtaining exact solution with distributed algorithms is generally very hard. One possible solution is to apply our centralized algorithms in each sensor after it collects information from nearby sensors (specifically sensors within a distance of $2R$, where R is the query range). Nevertheless, such a solution is not really distributed in the sense that limited information flooding is still required. Designing a fully distributed algorithm to obtain exact solution to the Geo-Range query problem remains open.

Appendix A

Source Code

A.1 Graph Generation Algorithm(clsGraph.vb)

```
Public Class clsGraph
  Public Range As Decimal
  Public Threshold As Single ' the threshold of outputting data
  '================================

  Structure structArc
    Sub New(ByVal ss As Integer, ByVal isl As Boolean, ByVal aID As Integer)
      sensor = ss
      IsLeft = isl
      arcID = aID
    End Sub
    Dim sensor As Integer
    Dim IsLeft As Boolean
    Dim arcID As Integer
  End Structure

  Structure structSensor
    Sub New(ByVal xx As Decimal, ByVal yy As Decimal, ByVal sid As Integer)
      X = xx
      Y = yy
      value = Rnd() * 100
      ID = sid
      arcl = ID * 2
      arcr = ID * 2 + 1
    End Sub
    Public X As Decimal
    Public Y As Decimal
    Public value As Single
    Public ID As Integer
    Public arcl As Integer
    Public arcr As Integer
  End Structure
```

```
  Structure structIntersection
    Sub New(ByVal xx As Decimal, ByVal yy As Decimal, ByVal al As Integer, _
ByVal ar As Integer, ByVal IsT As Boolean)
      X = xx
      Y = yy
      arcl = al
      arcr = ar
      IsTop = IsT
    End Sub
    Public IsTop As Boolean
    Public X As Decimal
    Public Y As Decimal
    Public arcl As Integer
    Public arcr As Integer
  End Structure

  Structure structArea
    Dim pnt As PointF
    Dim value As Single
    Sub New(ByVal p As PointF, ByVal v As Single)
      pnt = p
      value = v
    End Sub
  End Structure

  Public sensors As New ArrayList
  Public arcs As New ArrayList
  Public intersections As New ArrayList

  Public areas As New ArrayList

  '========interface ====================

  Public Function GetSensorByArc(ByVal arc As Integer) As structSensor
    Return sensors(arcs(arc).sensor)
  End Function

  Public Sub OutputArea(ByVal p As PointF, ByVal v As Single)
    areas.Add(New structArea(p, v))
  End Sub

  Public Sub OutputResult(ByVal txt As TextBox, ByVal N As Integer)
    txt.Text = ""
    Dim i As Int16 = 0
    For Each S As structSensor In sensors
      i += 1
      txt.Text &= "s" & S.ID & ": " & GetRounded(S.value) & vbCrLf
      If i > N Then
        Exit For
      End If
    Next
```

```
    i = 0
    txt.Text += vbCrLf
    For Each A As structArea In areas
      i += 1
      txt.Text &= "(" & GetRounded(A.pnt.X) & ", " & GetRounded(A.pnt.Y) &_
"): " & GetRounded(A.value) & vbCrLf
      If i > N Then
        Exit For
      End If
    Next
  End Sub
  '============================

  Sub Initialize()
    sensors.Clear()
    intersections.Clear()
    arcs.Clear()
  End Sub

  Public Sub New(ByVal tmpP As PictureBox)
    P = tmpP
    B = New Bitmap(P.Width, P.Height)
    G = Graphics.FromImage(B)
  End Sub


  Public Sub GenerateRandomly(ByVal size As Decimal, ByVal r As Decimal_
, ByVal t As Single)
    Dim i As Integer
    Range = r
    Threshold = t
    Randomize()
    For i = 1 To size
      Dim sensorID As Integer = sensors.Count
      sensors.Add(New structSensor(Rnd() * (P.Width - 2 * (Range + 10))_
+ Range + 10, Rnd() * (P.Height - 2 * (Range + 10)) + Range + 10, sensorID))
      arcs.Add(New structArc(sensorID, True, arcs.Count))
      arcs.Add(New structArc(sensorID, False, arcs.Count))
    Next
  End Sub

  Public Sub GenerateGrid(ByVal size As Decimal, ByVal r As Decimal, ByVal t As Single)
    Dim i As Integer
    Range = r
    Threshold = t
    Randomize()
    Dim w, h As Integer
    w = Math.Sqrt(size)
    Dim wi As Integer = (P.Width - 2 * (Range + w)) / (w - 1)
    h = size / w
    Dim hi As Integer = (P.Height - 2 * (Range + h)) / (h - 1)
    For i = 0 To w - 1
```

```
    For j = 0 To h - 1
      Dim sensorID As Integer = sensors.Count
      sensors.Add(New structSensor(Range + 10 + i * wi + j, Range_
+ 10 + j * hi - i, sensorID))
      arcs.Add(New structArc(sensorID, True, arcs.Count))
      arcs.Add(New structArc(sensorID, False, arcs.Count))
    Next
  Next
End Sub


Public Sub RandomValue()
  Randomize()
  Dim si As structSensor
  For i = 1 To sensors.Count - 1
    si = sensors(i)
    si.value = Rnd() * 100
    sensors(i) = si
  Next
End Sub

Public Sub GradientValue(ByVal k As Single)
  ' k here is defined as how many radius two correlated sensors should be apart
  Randomize()
  Dim s As structSensor
  s = sensors(0)
  s.value = Rnd() * 100
  sensors(0) = s
  Dim i, j As Integer
  Dim lb, ub, lbj, ubj, d As Single
  Dim si, sj As structSensor
  For i = 1 To sensors.Count - 1
    lb = 0
    ub = 100
    si = sensors(i)
    For j = i - 1 To 0 Step -1
      sj = sensors(j)
      d = GeoD(si.X, si.Y, sj.X, sj.Y)
      lbj = sj.value - d / (Range * k) * 100
      ubj = sj.value + d / (Range * k) * 100
      lb = IIf(lbj > lb, lbj, lb)
      ub = IIf(ubj < ub, ubj, ub)
    Next
    si.value = Rnd() * (ub - lb) + lb
    sensors(i) = si
  Next
End Sub



Public Sub FieldValue(ByVal f As clsField)
  Dim s As structSensor
```

```
   Dim v As Single
   For i = 0 To sensors.Count - 1
     s = sensors(i)
     v = f.field(Math.Ceiling(s.X / f.gWidth), Math.Ceiling(s.Y /_
f.gHeight)).value * 1.5 + 30
     s.value = IIf(v > 99, 99, v)
     sensors(i) = s
   Next
 End Sub

 Public Function ComputeIntersections() As Integer
   Dim i, j As Integer
   Dim si, sj As structSensor
   Dim distance As Decimal
   Dim cx, cy, k, d1, d2, sind, cosd As Decimal
   Dim ix1, ix2, iy1, iy2 As Decimal
   For i = 0 To sensors.Count - 1
     For j = 0 To sensors.Count - 1
       si = sensors.Item(i)
       sj = sensors.Item(j)
       distance = GeoD(si.X, si.Y, sj.X, sj.Y)
       If j >= i + 1 And distance < Range * 2 Then

         cx = (si.X + sj.X) / 2
         cy = (si.Y + sj.Y) / 2

         k = Math.Abs((si.Y - sj.Y) / (si.X - sj.X))

         d1 = Math.Sqrt((si.X - cx) ^ 2 + (si.Y - cy) ^ 2)

         sind = Math.Abs((si.Y - sj.Y) / (2 * d1))
         cosd = Math.Abs((si.X - sj.X) / (2 * d1))

         d2 = Math.Sqrt(Range ^ 2 - d1 ^ 2)
         If (si.Y - sj.Y) / (si.X - sj.X) < 0 Then
           ix1 = cx - d2 * sind
           iy1 = cy - d2 * cosd
           ix2 = cx + d2 * sind
           iy2 = cy + d2 * cosd
         Else
           ix1 = cx + d2 * sind
           iy1 = cy - d2 * cosd
           ix2 = cx - d2 * sind
           iy2 = cy + d2 * cosd
         End If

         Dim arcl As Integer
         Dim arcr As Integer
         arcl = IIf(ix1 < si.X, si.arcl, si.arcr)
         arcr = IIf(ix1 < sj.X, sj.arcl, sj.arcr)

         ReorderArcs(arcl, arcr, iy1)
```

```
          intersections.Add(New structIntersection(ix1, iy1, arcl,_
 arcr, IIf(iy1 < iy2, True, False)))

          arcl = IIf(ix2 < si.X, si.arcl, si.arcr)
          arcr = IIf(ix2 < sj.X, sj.arcl, sj.arcr)

          ReorderArcs(arcl, arcr, iy2)
          intersections.Add(New structIntersection(ix2, iy2, arcl,_
 arcr, IIf(iy2 < iy1, True, False)))

        End If
        If j >= i + 1 And distance = Range * 2 Then
          intersections.Add(New PointF((si.X + sj.X) / 2, (si.Y + sj.Y) / 2))
        End If
      Next
    Next
    Return intersections.Count
  End Function

  Sub ReorderArcs(ByRef arcl As Integer, ByRef arcr As Integer,_
 ByVal y As Decimal)
    If GetSensorByArc(arcl).Y - Range > y - 0.001 Or GetSensorByArc_
(arcr).Y - Range > y - 0.001 Then
    End If
    If GetLineX(arcl, y - 0.001) > GetLineX(arcr, y) Then
      'swap
      Dim arctmp As Integer = arcl
      arcl = arcr
      arcr = arctmp
    End If
  End Sub

  Function GetLineX(ByVal a As Integer, ByVal y1 As Decimal)
    If arcs(a).IsLeft Then
      Return GetSensorByArc(a).X - Math.Sqrt(Range ^ 2 - (y1_
 - GetSensorByArc(a).Y ^ 2))
    Else
      Return GetSensorByArc(a).X + Math.Sqrt(Range ^ 2 - (y1_
 - GetSensorByArc(a).Y ^ 2))
    End If
  End Function



  '=========================

  Public P As New PictureBox
  Public G As Graphics
  Public B As Bitmap
  Public DrawWidth As Int16 = 1
```

```
  Public Sub DrawIntersections()
    For Each i As structIntersection In intersections
      Dim drawPen As Pen = New Pen(Color.Black)
      drawPen.Width = DrawWidth
      drawPen.Color = Color.Yellow
      G.DrawArc(drawPen, i.X - 2, i.Y - 2, 4, 4, 0, 360)
    Next
    P.Image = B
  End Sub

  Public Sub DrawSensors(ByVal ShowLabel As Boolean, ByVal ShowCircle As Boolean)
    Dim drawPen As Pen = New Pen(Color.Black)
    drawPen.Width = DrawWidth
    Dim tmpInt As Int16
    For i = 0 To sensors.Count - 1
      If ShowCircle Then
        drawPen.Color = Color.Blue
        G.DrawArc(drawPen, CSng(sensors(i).X - Range), _
CSng(sensors(i).Y - Range), CSng(Range * 2), CSng(Range * 2), 92, 176)
        drawPen.Color = Color.Red
        G.DrawArc(drawPen, CSng(sensors(i).X - Range), CSng(sensors(i).Y - Range)_
, CSng(Range * 2), CSng(Range * 2), 272, 176)
        G.DrawLine(Pens.Violet, CSng(sensors(i).X), CSng(sensors(i).Y - Range - 2)_
, CSng(sensors(i).X), CSng(sensors(i).Y - Range + 2))
        G.DrawLine(Pens.Green, CSng(sensors(i).X), CSng(sensors(i).Y + Range - 2)_
, CSng(sensors(i).X), CSng(sensors(i).Y + Range + 2))
      End If
      tmpInt = 100 + sensors(i).value * 1.5
      drawPen.Color = Color.FromArgb(tmpInt, tmpInt, tmpInt)
      drawPen.Width = 2
      G.DrawArc(drawPen, (sensors(i).X - 2), (sensors(i).Y - 2), 4, 4, 0, 360)
      If ShowLabel Then
        G.DrawString("s" & i, _frmALG2.Font, Brushes.White,_
 sensors(i).X + 5, sensors(i).Y - 5)
      End If
    Next
    P.Image = B
  End Sub

  Public Sub DrawClear()
    G.Clear(Color.Black)
    P.Image = B
  End Sub

  Public Sub DrawResult()
    Dim tmpInt As Int16
    Dim tmpP As New Pen(Color.White, 3)
    For Each A As structArea In areas
      If A.value > 100 Or A.value < 0 Then
        MsgBox("Result Value Greater Than 100")
        A.value = 100
      Else
```

```
        tmpInt = A.value * 2.5
        If tmpInt > 255 Or tmpInt < 0 Then tmpInt = 255
        tmpP.Color = Color.FromArgb(tmpInt, tmpInt, tmpInt)
        G.DrawArc(tmpP, (A.pnt.X - 5), (A.pnt.Y - 5), 10, 10, 0, 360)
      End If
    Next
    P.Image = B
  End Sub


End Class
```

A.2 Sweep-Line Algorithm(clsSweepLine.vb)

```
Public Class clsSweepLine

  Structure structEvent
    Enum EventType
      Intersection = 1
      Top = 2
      Bottom = 3
    End Enum
    Dim type As EventType
    Dim IsTop As Boolean
    Dim x As Decimal
    Dim y As Decimal
    Dim arcl As Integer
    Dim arcr As Integer
  End Structure

  Class clsTreeNode
    Public LeftChild As clsTreeNode
    Public RightChild As clsTreeNode
    Public arc As Integer
    Public Parent As clsTreeNode
    Public Brother As clsTreeNode
    Public LeftArea As clsArea
    Public RightArea As clsArea
    Public Leaves As Int16
  End Class



  Class clsArea

    Public value As Single = 0
    Public nodeNumber As Integer = 0
    Sub New()
```

```
  End Sub
End Class


Dim Graph As clsGraph
Dim events As New ArrayList

Sub New(ByVal tmpp1 As PictureBox, ByRef sensorGraph As clsGraph)
  Graph = sensorGraph
  P1 = tmpp1
  B1 = New Bitmap(P1.Width, P1.Height)
  G1 = Graphics.FromImage(B1)
End Sub

Sub Clear()
  Graph.sensors.Clear()
  Graph.intersections.Clear()
  events.Clear()
  Graph.arcs.Clear()
End Sub

Sub ReStart()
  InitializeSweepLine()
End Sub

Public Class myComparerClass
  Implements IComparer

  ' Calls CaseInsensitiveComparer.Compare with the parameters reversed.
  Public Function Compare(ByVal x As Object, ByVal y As Object)_
As Integer Implements IComparer.Compare
    Dim e1 As structEvent = x
    Dim e2 As structEvent = y
    If e1.y > e2.y Then
      Return 1
    ElseIf e1.y < e2.y Then
      Return -1
    Else
      Return 0
    End If
  End Function 'IComparer.Compare

End Class 'myReverserClass

Public Sub BuildEventQueue()



  For Each i As clsGraph.structIntersection In Graph.intersections
    Dim e As New structEvent
    e.type = structEvent.EventType.Intersection
    e.IsTop = i.IsTop
```

```
    e.x = i.X
    e.y = i.Y
    e.arcl = i.arcl
    e.arcr = i.arcr
    events.Add(e)
  Next

  For Each s As clsGraph.structSensor In Graph.sensors
    Dim e1 As New structEvent
    e1.type = structEvent.EventType.Top
    e1.arcl = s.arcl
    e1.arcr = s.arcr
    e1.x = s.X
    e1.y = s.Y - Graph.Range
    events.Add(e1)

    Dim e2 As New structEvent
    e2.type = structEvent.EventType.Bottom
    e2.arcl = s.arcl
    e2.arcr = s.arcr
    e2.x = s.X
    e2.y = s.Y + Graph.Range
    events.Add(e2)

  Next

  Dim myComparer = New myComparerClass()
  events.Sort(myComparer)

End Sub


Dim root As clsTreeNode

Dim eventp As Integer
'left child: equal or less. right child: greater



Public Function GoNextEvent() As Boolean
  eventp += 1
  If IsEnd() Then
    Return False
  End If
  Dim e As structEvent = events(eventp)
  If eventp = 8 Then
    eventp = eventp
  End If
  Select Case e.type
    Case structEvent.EventType.Top
' the event is the top point of a circle
' go find the leafnode that the top event x is right on the right to the node
```

```
        Dim leafnode As clsTreeNode = FindLeafNode(e.x, root, e.y)
        Dim lnode As clsTreeNode = AddNode(leafnode, e.arcl, e.y)
        Dim rnode As clsTreeNode = AddNode(lnode, e.arcr, e.y)
        lnode = rnode.Brother  ' here, lnode & rnode are 2 new added nodes
        ResetParentArcs(lnode.Parent, e.y)
        Dim larea As New clsArea()
        Dim marea As New clsArea()
        Dim rarea As New clsArea()
        If leafnode Is Nothing Then ' if is the root, start from scratch
          marea.value = Graph.GetSensorByArc(e.arcr).value
          marea.nodeNumber = 1
        Else ' if not,derivate from the parent
          Dim parentArea As clsArea
          If e.x > GetLineX(leafnode.arc, e.y) Then
            parentArea = leafnode.RightArea
          Else
            parentArea = leafnode.LeftArea
          End If
          larea.value = parentArea.value
          larea.nodeNumber = parentArea.nodeNumber
          rarea.value = parentArea.value
          rarea.nodeNumber = parentArea.nodeNumber
          marea.value = parentArea.value + Graph.GetSensorByArc(e.arcr).value
          marea.nodeNumber = parentArea.nodeNumber + 1
          'leafnode.RightArea = Nothing
        End If
        lnode.LeftArea = larea
        lnode.RightArea = marea
        rnode.LeftArea = marea
        rnode.RightArea = rarea
      Case structEvent.EventType.Intersection
        Dim node1 As clsTreeNode = Nothing
        Dim node2 As clsTreeNode = Nothing

        node1 = FindLeafNodeByArc(e.arcl, root, e.y)
        node2 = FindLeafNodeByArc(e.arcr, root, e.y)
        If node1 Is Nothing Or node2 Is Nothing Then
' this if statement is to prevent the float number error from system
          Dim eventpb As Integer = eventp - 1
          Dim eventpa As Integer = eventp + 1
          Do
            If eventpb > 0 Then
              Dim eb As structEvent = events(eventpb)
              If eb.type = structEvent.EventType.Bottom Then
                If eb.arcl = e.arcl Or eb.arcl = e.arcr Or eb.arcr =_
 e.arcl Or eb.arcr = e.arcr Then
                  ' if the previous event is bottom and has the same arc,
' which means the circle has been deleted wrongfully
                  Exit Function
                End If
              End If
              eventpb -= 1
```

```
            End If
            If eventpa < events.Count Then
              Dim ea As structEvent = events(eventpa)
              If ea.type = structEvent.EventType.Top Then
                If ea.arcl = e.arcl Or ea.arcl = e.arcr Or_
 ea.arcr = e.arcl Or ea.arcr = e.arcr Then
                  ' if the next event is top and has the same arc,
'which means the circle has not added yet
                  Dim tmpevent As structEvent = events(eventpa)
                  events(eventpa) = events(eventp)
                  events(eventp) = tmpevent
                  eventp -= 1
                  Exit Function
                End If
              End If
              eventpb += 1
            End If
          Loop
        End If

        If Not ReorderNodes(node1, node2) Then
          StopRunning()
          Return False
        End If


        Dim marea As New clsArea
        marea.value = node2.LeftArea.value
        marea.nodeNumber = node2.LeftArea.nodeNumber
        If marea.value / marea.nodeNumber > Graph.Threshold And_
 marea.nodeNumber > 0 Then
' output this result
          Graph.OutputArea(New PointF(e.x, e.y), marea.value / marea.nodeNumber)
        End If


 Dim s1 As clsGraph.structSensor = Graph.sensors(Graph.arcs(node1.arc).sensor)
Dim s2 As clsGraph.structSensor = Graph.sensors(Graph.arcs(node2.arc).sensor)

        If (e.x - s1.X) * (e.x - s2.X) >= 0 Then
'abstract whichever is lower
          marea.value -= IIf(s1.Y < s2.Y, s1.value, s2.value)
'add whichever is higher
          marea.value += IIf(s1.Y > s2.Y, s1.value, s2.value)

        ElseIf e.IsTop Then
          marea.nodeNumber += 2
          marea.value += s1.value + s2.value
        ElseIf Not e.IsTop Then
          marea.nodeNumber -= 2
          marea.value -= s1.value + s2.value
        End If
```

```
        node1.RightArea = marea
        node2.LeftArea = marea


        Dim arc1, arc2 As Integer
        arc1 = node1.arc
        arc2 = node2.arc
        node2.arc = arc1
        node1.arc = arc2
        ResetParentArcs(node2, e.y)
        ResetParentArcs(node1, e.y)
      Case structEvent.EventType.Bottom

        Dim node1 As clsTreeNode = FindLeafNodeByArc(e.arcl, root, e.y)
        Dim node2 As clsTreeNode = FindLeafNodeByArc(e.arcr, root, e.y)
        If GetLineX(node2.arc, e.y - 0.01) >_
GetLineX(node1.arc, e.y - 0.01) Then
          If node2.LeftArea.value / node2.LeftArea.nodeNumber_
> Graph.Threshold Then
            Graph.OutputArea(New PointF(e.x, e.y), node2.LeftArea.value_
/ node2.LeftArea.nodeNumber)
          End If
        Else
          If node1.LeftArea.value / node1.LeftArea.nodeNumber_
> Graph.Threshold Then
            Graph.OutputArea(New PointF(e.x, e.y), node1.LeftArea.value_
/ node1.LeftArea.nodeNumber)
          End If
        End If

        RemoveNode(node2, e.y)
        RemoveNode(node1, e.y)


    End Select
    Return True
  End Function


  Public Function ReorderNodes(ByRef node1 As clsTreeNode,_
ByRef node2 As clsTreeNode) As Boolean
' node2 will be on the right to node1 after this
    Dim node1right As clsTreeNode = FindRightNeighbour(node1, True)
    If Not node1right Is Nothing Then ' if correct,quit
      If node1right.arc = node2.arc Then
        Return True
      End If
    End If

    Dim node2right As clsTreeNode = FindRightNeighbour(node2, True)
    If Not node2right Is Nothing Then  ' if order is wrong, swap, quit
```

```
      If node2right.arc = node1.arc Then
        Dim tmpNode As New clsTreeNode
        tmpNode = node2
        node2 = node1
        node1 = tmpNode
        Return True
      End If
    End If

    Return False
  End Function

  Public Sub RemoveNode(ByRef node As clsTreeNode, ByVal y As Decimal)
' the tree is modified
    If node.Parent Is Nothing Then 'if it is the root
      root = Nothing
    Else
      If Not node.Parent.Parent Is Nothing Then
' if the node is not a child of root
        If node.Parent.Parent.LeftChild.arc = node.Parent.arc Then
          node.Parent.Parent.LeftChild = node.Brother
          node.Brother.Parent = node.Parent.Parent
          node.Parent.Parent.LeftChild.Brother = node.Parent.Parent.RightChild
          node.Parent.Parent.RightChild.Brother = node.Parent.Parent.LeftChild
        Else
          node.Parent.Parent.RightChild = node.Brother
          node.Brother.Parent = node.Parent.Parent
          node.Parent.Parent.LeftChild.Brother = node.Parent.Parent.RightChild
          node.Parent.Parent.RightChild.Brother = node.Parent.Parent.LeftChild
        End If
        ResetParentArcs(node.Parent, y)
      Else ' if the node is only a child of root
        node.Brother.Brother = Nothing
        node.Brother.Parent = Nothing
        root = node.Brother
      End If
    End If
  End Sub

  Function FindTheSameArc(ByVal node As clsTreeNode) As clsTreeNode
    If node.Parent Is Nothing Then
      Return node
    Else
      If node.Parent.arc <> node.arc Then
        Return node
      Else
        Return FindTheSameArc(node.Parent)
      End If
    End If
  End Function

  Public Function AddNode(ByRef parent As clsTreeNode, ByVal arc As Integer_
```

```
, ByVal y As Decimal) As clsTreeNode
    If parent Is Nothing Then ' if the tree is empty
      Dim nodenew As New clsTreeNode
      nodenew.arc = arc
      nodenew.Parent = Nothing
      nodenew.Brother = Nothing
      root = nodenew
      Return nodenew
    Else ' if the tree is not empty
      Dim nodea As New clsTreeNode
      nodea.arc = arc
      nodea.Parent = parent

      Dim nodep As New clsTreeNode
      nodep.arc = parent.arc
      nodep.Parent = parent
      nodep.LeftArea = parent.LeftArea
      nodep.RightArea = parent.RightArea

      nodea.Brother = nodep
      nodep.Brother = nodea

      If Graph.GetSensorByArc(parent.arc).ID = Graph.GetSensorByArc(arc).ID Then
' if 2 points are from the same top event
        If Graph.arcs(arc).IsLeft Then
          parent.RightChild = nodep
          parent.LeftChild = nodea
        Else
          parent.RightChild = nodea
          parent.LeftChild = nodep
        End If

      Else
        If GetLineX(arc, y) > GetLineX(parent.arc, y) Then
' if 2 points are not from the same top event
          parent.RightChild = nodea
          parent.LeftChild = nodep
        Else
          parent.RightChild = nodep
          parent.LeftChild = nodea
        End If
      End If

      Return nodea
    End If
  End Function


  '====Tree Searching==========================================

  Function FindLeafNode(ByVal x As Decimal, ByVal node As clsTreeNode,_
 ByVal y As Decimal) As clsTreeNode
```

```
  If node Is Nothing Then
    Return node
  End If
  If node.LeftChild Is Nothing Then
    Return node
  End If
  ' if statement should be on the 2 leaves

  If x >= GetLineX(node.RightChild.arc, y) Then
    Return FindLeafNode(x, node.RightChild, y)
  Else
    Return FindLeafNode(x, node.LeftChild, y)
  End If
 End Function

 Function FindLeafNodeByArc(ByVal arc As Integer, ByVal node As clsTreeNode,_
ByVal y As Decimal) As clsTreeNode
  Dim node1 As clsTreeNode = TryToFindLeafNodeByArc(arc, root, y)
  If node1.arc <> arc Then
    Dim node1l As clsTreeNode = node1
    Dim node1r As clsTreeNode = node1
    Do
      node1l = FindLeftNeighbour(node1l, True)
      If Not node1l Is Nothing Then
        If node1l.arc = arc Then
          node1 = node1l
          Exit Do
        Else
        End If
      End If
      node1r = FindRightNeighbour(node1r, True)
      If Not node1r Is Nothing Then
        If node1r.arc = arc Then
          node1 = node1r
          Exit Do
        Else
        End If
      End If
      If node1l Is Nothing And node1r Is Nothing Then
        Return Nothing
      End If
    Loop
    If node1.arc <> arc Then
      MsgBox("FindLeafNodeByArc: ERROR: nothing found")
    End If
  End If
  Return node1
 End Function

 Function TryToFindLeafNodeByArc(ByVal arc As Integer,_
ByVal node As clsTreeNode,ByVal y As Decimal) As clsTreeNode
  If node.LeftChild Is Nothing Then
```

```
' if no child, return the leaf node anyway
      Return node
    End If
    Dim IsArcFound As Boolean = False
    If node.LeftChild.arc = arc Then
      Return TryToFindLeafNodeByArc(arc, node.LeftChild, y)
    ElseIf node.RightChild.arc = arc Then
      Return TryToFindLeafNodeByArc(arc, node.RightChild, y)

    End If
    'otherwise, it is decided by y value
    If GetLineX(arc, y) > GetLineX(node.arc, y) Then
      Return TryToFindLeafNodeByArc(arc, node.RightChild, y)
    Else
      Return TryToFindLeafNodeByArc(arc, node.LeftChild, y)
    End If

  End Function

  Function FindRightNeighbour(ByVal node As clsTreeNode,_
 ByVal IsUp As Boolean) As clsTreeNode
    If IsUp Then
      If node Is Nothing Then
        Return Nothing
      End If
      If node.Parent Is Nothing Then
        Return Nothing
      End If
      If node.Parent.RightChild.arc <> node.arc Then
        Return FindRightNeighbour(node.Parent.RightChild, False)
      Else
        Return FindRightNeighbour(node.Parent, True)
      End If
    Else
      If Not node.LeftChild Is Nothing Then
        Return FindRightNeighbour(node.LeftChild, False)
      Else
        Return node
      End If
    End If
  End Function

  Function FindLeftNeighbour(ByVal node As clsTreeNode,_
 ByVal IsUp As Boolean) As clsTreeNode
    If IsUp Then ' when going up
      If node Is Nothing Then
        Return Nothing
      End If
      If node.Parent Is Nothing Then
        Return Nothing
      End If
      If node.Parent.LeftChild.arc <> node.arc Then
```

```
' if another child ain't itself
       Return FindLeftNeighbour(node.Parent.LeftChild, False)
 'return the child
     Else ' otherwise
       Return FindLeftNeighbour(node.Parent, True) ' going up the tree
     End If
   Else ' when going down
     If Not node.RightChild Is Nothing Then
       Return FindLeftNeighbour(node.RightChild, False)
     Else
       Return node
     End If
   End If
 End Function

 Sub ResetParentArcs(ByVal node As clsTreeNode, ByVal y As Decimal)

   If Not node.Parent Is Nothing Then

     If node.Parent.arc <> node.Parent.LeftChild.arc Then
       node.Parent.arc = node.Parent.LeftChild.arc
       ResetParentArcs(node.Parent, y)
     End If
   End If

 End Sub
 '=======================

 Function GetLineX(ByVal arc As Integer, ByVal y1 As Decimal)
   If Graph.arcs(arc).IsLeft Then
     Return Graph.GetSensorByArc(arc).X - Math.Sqrt(Graph.Range ^ 2 - (y1 - Graph.GetSensorByArc
   Else
     Return Graph.GetSensorByArc(arc).X + Math.Sqrt(Graph.Range ^ 2 - (y1 - Graph.GetSensorByArc
   End If
 End Function
 '=======debug============================================
 Private P1 As New PictureBox
 Private G1 As Graphics
 Private B1 As Bitmap

 Sub DrawTree()
   G1.Clear(Color.White)
   If eventp < events.Count And eventp > -1 Then
     Dim e As structEvent = events(eventp)
     For i = 0 To Graph.sensors.Count - 1
               Graph.G.DrawString(Graph.sensors(i).id, _frmALG2.Font,_
Brushes.Wheat, Graph.sensors(i).x + 2, Graph.sensors(i).y)
           Next
           Graph.G.DrawLine(Pens.Orange, 0, CSng(e.y), Graph.P.Width, CSng(e.y))
           Graph.P.Image = Graph.B
           If root Is Nothing Then
               G1.DrawString("DrawTree: The tree is empty", _frmALG2.Font,_
```

```
 Brushes.Black, P1.Width / 2 - 50, 10)
            Else
                DrawTreeIteration(root, 10, P1.Width - 15, 10)
            End If
        Else
            G1.DrawString("DrawTree: tree not available", _frmALG2.Font_
, Brushes.Black, P1.Width / 2 - 30, 10)
        End If


        P1.Image = B1
    End Sub


    Private Sub DrawTreeIteration(ByVal n As clsTreeNode, ByVal L As Single,_
 ByVal R As Single, ByVal Y As Decimal)
        G1.DrawString(GetNodeStrByArc(n.arc), _frmALG2.Font,_
 Brushes.Black, (L + R) / 2 - 10, Y)
        If Not n.LeftChild Is Nothing Then
            Dim M As Single
            If n.LeftChild.LeftChild Is Nothing And_
 (Not n.RightChild.LeftChild Is Nothing) Then
'if left child is leaf
                M = L + IIf((R - L) / 3 < 15, (R - L) / 3, 15)
            ElseIf (Not n.LeftChild.LeftChild Is Nothing) And_
 n.RightChild.LeftChild Is Nothing Then
                M = R - IIf((R - L) / 3 < 15, (R - L) / 3, 15)
            Else
                M = (L + R) / 2
            End If
            G1.DrawLine(Pens.Blue, CSng((L + R) / 2), CSng(Y + 11),_
 CSng((L + M) / 2), CSng(Y + 17))
            G1.DrawLine(Pens.Red, CSng((L + R) / 2), CSng(Y + 11),_
 CSng((M + R) / 2), CSng(Y + 17))

            DrawTreeIteration(n.LeftChild, L, M, Y + 17)
            DrawTreeIteration(n.RightChild, M, R, Y + 17)
        End If
    End Sub

  Sub ShowTreeView(ByRef tree As TreeView)
    tree.Nodes.Clear()
    If root Is Nothing Then
      tree.Nodes.Add("empty")
    Else
      ShowTreeViewInteration(root, tree.Nodes)
    End If

    tree.ExpandAll()
  End Sub

  Sub ShowTreeViewInteration(ByVal node As clsTreeNode, ByRef_
```

```
viewnodes As TreeNodeCollection)
   Dim tmpStr As String = GetNodeStrByArc(node.arc) & "(" & node.arc & ")" & " "_
& GetLineX(node.arc, events(eventp - 1).y)
   Dim tmpNodes As TreeNode = viewnodes.Add(tmpStr)

   If Not node.LeftChild Is Nothing Then
     ShowTreeViewInteration(node.LeftChild, tmpNodes.Nodes)
     ShowTreeViewInteration(node.RightChild, tmpNodes.Nodes)
   End If
 End Sub

 Sub ShowTreeText(ByVal txt As TextBox) ' this one only print the leaf nodes
   If Not root Is Nothing Then
     txt.Text = ""
     ShowTreeTextInteration(root, txt)
   End If
 End Sub

 Sub ShowTreeTextInteration(ByVal node As clsTreeNode, ByRef txt As TextBox)
   If node.LeftChild Is Nothing Then
     txt.Text &= GetNodeStrByArc(node.arc) & "(" & node.arc & ")" & " " &_
GetLineX(node.arc, events(eventp - 2).y) & vbCrLf
   Else
     ShowTreeTextInteration(node.LeftChild, txt)
     ShowTreeTextInteration(node.RightChild, txt)
   End If
 End Sub

 Function CheckValidity() As Boolean
   If Not root Is Nothing Then
     Return CheckValidityInteration(root)
   Else
     Return True
   End If
 End Function

 Function CheckValidityInteration(ByVal node As clsTreeNode) As Boolean 'FIXME
   If Not node.LeftChild Is Nothing Then
     If node.arc <> node.LeftChild.arc Then
       Return False
     End If
     Return CheckValidityInteration(node.LeftChild) And_
CheckValidityInteration(node.RightChild)
   Else
     Return True
   End If

 End Function

 Sub ShowEventQueue(ByRef txt As TextBox)
   txt.Text = ""
   For i = eventp - 5 To eventp + 5
```

```
      If i > 0 And i < events.Count - 1 Then
        Dim e As structEvent = events(i)

        txt.Text &= i & ":" & e.type.ToString & " (" & GetRounded(e.x) & "," &_
 GetRounded(e.y) & ")  {" & e.arcl & "," & e.arcr & "}"
        If i = eventp Then
          txt.Text &= " <--- " & vbCrLf
        Else
          txt.Text &= vbCrLf
        End If
      End If
    Next
  End Sub

  Private Function GetNodeStrByArc(ByVal arc As Integer)
        Return Graph.arcs(arc).sensor & IIf(Graph.arcs(arc).IsLeft, "L", "R")
  End Function

  Sub InitializeSweepLine()
    eventp = -1
    root = Nothing
    IsItShouldBeRunning = True
  End Sub


  Function IsEnd() As Boolean
    If eventp > events.Count - 1 Then
      Return True
    Else
      Return False
    End If

  End Function

  Dim IsItShouldBeRunning As Boolean

  Sub StopRunning()
    IsItShouldBeRunning = False
  End Sub

  Public Sub RunToTheEnd()
    While GoNextEvent() And IsItShouldBeRunning

    End While
    IsItShouldBeRunning = True
  End Sub
End Class
```

Bibliography

[1] M. Li and Y. Liu, "Underground structure monitoring with wireless sensor networks," *Proc. of IPSN*, pp. 69–78, 2007.

[2] R. Beckwith, T. Brooke, and J. Burrell, "Vineyard computing: sensor networks in agricultural production," *IEEE Pervasive Computing*, vol. 3, pp. 38–45, 2004.

[3] X. Meng, N. Wang, and L. Yu, "Real-time forest fire detection with wireless sensor networks," *Wireless Communications, Networking and Mobile Computing Conference*, vol. 2, pp. 1214–1217, 2005.

[4] M. Bagheri and M. Hefeeda, "Wireless sensor networks for early detection of forest fires," *IEEE Internatonal Conference on Mobile Adhoc and Sensor Systems*, pp. 1–6, 2007.

[5] W.-C. Lee, X. Tang, M. Wu, and J. Xu, "Monitoring top-k query in wireless sensor networks," *Proc. of ICDE*, p. 143, 2006.

[6] R. Braynard, C. Ellis, K. Munagala, A. Silberstein, and J. Yang, "A sampling-based approach to optimizing top-k queries in sensor networks," *Proc. of ICDE*, p. 68, 2006.

[7] K. Munagala, A. Silberstein, and J. Yang, "Energy-efficient monitoring of extreme values in sensor networks," *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pp. 169–180, 2006.

[8] E. Elnahrawy and B. Nath, "Cleaning and querying noisy sensors," *Proc. of WSNA*, pp. 78 – 87, 2003.

[9] M.-K. Chang, C.-C. Kuo, and P.-K. Liao, "Contour line extraction with wireless sensor networks," *2005 IEEE International Conference*, vol. 5, pp. 3202 – 3206, 2005.

[10] L. Chen, Y. Liu, and Q. Luo, "Contour map matching for event detection in sensor networks," *International Conference on Management of Data*, pp. 145 – 156, 2006.

[11] J. Gao, J. S. B. Mitchelly, R. Sarkar, and X. J. Zhu, "Light-weight contour tracking in wireless sensor networks," *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*, pp. 1175–1183, 2008.

[12] C. Buragohain, S. Gandhi, J. Hershberger, and S. Suri, "Contour approximation in sensor networks," *Distributed Computing in Sensor Systems*, vol. 4026, pp. 356–371, 2006.

[13] M. I. Khan, M. Merabti, and H. Mokhtar, "A survey of boundary detection algorithms for sensor networks," *Networking and Broadcasting (PGNET 2008)*, pp. 107–111, 2008.

[14] R. Nowak and U. Mitra, "Boundary estimation in sensor networks: theory and methods," *1st International Workshop on Information Processing in Sensor Networks*, pp. 80–95, 2003.

[15] Q. Fang, J. Gao, and L. J. Guibas, "Locating and bypassing routing holes in sensor networks," *Mobile networks and applications*, vol. 11, pp. 187–200, 2006.

[16] S. P. Fekete, M. Kaufmann, A. Kroeller, and K. Lehmann, "A new approach for boundary recognition in geometric sensor networks," *Canadian Conference on Computational Geometry*, 2005.

[17] S. P.Fekete, S. Fischer, A. Kroller, and D. Pfisterer, "Deterministic boundary recognition and topology extraction for large sensor networks," *SODA'06*, 2006.

[18] J. Gao, J. S. Mitchell, and Y. Wang, "Boundary recognition in sensor networks by topological methods," *International Conference on Mobile Computing and Networking*, pp. 122 – 133, 2006.

[19] S. Funke, "Topological hole detection in wireless sensor networks and its applications," *3rd ACM/SIGMOBILE international workshop on foundations of mobile computing (DIALM-POMC)*, pp. 44–53, 2005.

[20] K. K. Chintalapudi and R. Govindan, "Localized edge detection in sensor fields," *Sensor Network Protocols and Applications*, pp. 59– 70, 2003.

[21] M. Li and Y. Liu, "Iso-map: Energy-efficient contour mapping in wireless sensor networks," *Distributed Computing Systems*, pp. 36–36, 2007.

[22] S. Gandhi, J. Hershberger, and S. Suri, "Approximate isocontours and spatial summaries for sensor networks," *SESSION: Data representations and storage*, pp. 400–409, 2007.

[23] C. Jaikaeo, C.-C. Shen, and C. Srisathapornphat, "Sensor information networking architecture," *ICPPW'00*, pp. 23–30, 2000.

[24] P. Deng, Y. Hu, and F. Yu, "A novel data query method for wireless sensor networks," *Wireless Communications, Networking and Mobile Computing*, pp. 2519 – 2522, 2007.

[25] H. Balakrishnan, A. P. Chandrakasan, and W. B. Heinzelman, "An application specific protocol architecture for wireless microsensor networks," *IEEE TRANSACTIONS ON WIRELESS COMMUNICATIONS*, vol. 1, no. 4, pp. 660–670, 2002.

[26] W.-C. Lee, X. Tang, M. Wu, and J. Xu, "Monitoring top-k query in wireless sensor networks," *2009 Third International Conference on Sensor Technologies and Applications*, 2009.

[27] M. J. Franklin, J. Hellerstein, W. Hong, and S. Madden, "Tag: A tiny aggregation service for ad-hoc sensor networks," *In OSDI*, 2002.

[28] M. Seo, "A survey on in-network aggregation in sensor networks," *In Proc. of SenSys*, 2004.

[29] I. F. Akyildiz, E. Cayirci, Y. Sankarasubramaniam, and W. Su, "Wireless sensor networks: A survey," *Computer Networks*, vol. 38, pp. 393–422, 2002.

[30] J. Nievergelt and F. P. Preparata, "Plane-sweep algorithms for intersecting geometric figures," *Programming Techniques and Data Structures*, 1982.

[31] D.-S. Kim, B. Lee, and K. Sugihara, "A sweep-line algorithm for the inclusion hierarchy among circles," *Japan J. Indust. Appl. Math.*, vol. 23, pp. 127–138, 2006.

[32] S.-M. Hu, L. Jin, D.-S. Kim, D. Kim, and L. M. and, "A sweepline algorithm for euclidean voronoi diagram of circles," *Computer-Aided Design*, vol. 38, pp. 260–272, 2006.

[33] S. Wolfram, *A New Kind of Science*. Wolfram Media, January 2002.

[34] J.Kasegawa, S.Morishita, and T. Nakano, "Coastal oil pollution prediction by a tanker using cellular automata," *OCEANS '98 Conference Proceedings*, vol. 3, pp. 1324–1328 vol.3, Sep-1 Oct 1998.