Dynamic Coverage in Ad-hoc Sensor Networks

Hai Huang* Andréa W. Richa*,† Michael Segal‡

Abstract

Ad-hoc networks of sensor nodes are in general semi-permanently deployed. However, the topology of such networks continuously changes over time, due to the power of some sensors wearing out, to new sensors being inserted into the network, or even due to designers moving sensors around during a network re-design phase (for example, in response to a change in the requirements of the network). In this paper, we address the problem of how to dynamically maintain two important measures on the quality of the coverage of a sensor network: the best-case coverage and worst-case coverage distances. We assume that the ratio between upper and lower transmission power of sensors is bounded by a polynomial of n, where n is the number of sensors, and that the motion of mobile sensors can be described as a low-degree polynomial function of time. We maintain a $(1+\epsilon)$-approximation on the best-case coverage distance and a $(\sqrt{2}+\epsilon)$-approximation on the worst-case coverage distance of the network, for any fixed $\epsilon > 0$. Our algorithms have amortized or worst-case poly-logarithmic update costs. We are able to efficiently maintain the connectivity of the regions on the plane with respect to the sensor network, by extending the concatenable queue data structure to also serve as a priority queue. In addition, we present an algorithm that finds the shortest maximum support path in time $O(n \log n)$.

*Department of Computer Science and Engineering, Arizona State University, Tempe, AZ 85287-5406, USA. This work was supported in part by NSF CAREER Award CCR–9985284. Tel: 1-480-965-7555 Fax: 1-480-965-2751 Email: {hai, aricha}@asu.edu

†Contact author.

‡Communication Systems Engineering Department, Ben-Gurion University of the Negev, Beer-Sheva 84105, Israel. Tel: 972-8-6477234 Fax: 972-8-6472883 Email: segal@cse.bgu.ac.il

1 Introduction

Ad-hoc sensor networks are emerging as a new sensing paradigm and have thus received massive research interest recently. Usually sensor nodes are semi-permanently deployed, since the sensors themselves barely have any moving capacity. However, the topology of such networks continuously changes over time due to a variety of reasons: For example, a sensor node may wear out due to its very limited battery power; a new sensor node may be inserted into the network; or the layout of a sensor network may need to be changed in order to improve the quality of the network coverage in response to a change in the network requirements, which is accomplished by changing the placement of current (or inserting, deleting) sensors in network.

In this paper, we address the problem of how to dynamically maintain two important measures on the quality of the coverage of a sensor network: the best-case coverage distance and the worst-case coverage distance of the network. We also address a closely related problem, namely that of finding a shortest maximum support path.

In a sensor network, each sensor bears the ability to detect objects around it. The coverage of a sensor is limited by its energy level. Assuming that a sensor's detecting ability is omnidirectional, we can model the coverage of a sensor as a disk (under 2-norm on the Euclidean plane[1]) centered at the sensor. The radii of such disks are determined by the energy level of the sensors. The coverage area (or simply coverage) of the sensor network is the union of all such disks.

A sensor network is often used to detect intruders. An intruder may start at a point S, follow an arbitrary trajectory (path) on the plane, and stop at some other point T on the plane. In some applications, a sensor network may need to keep track of the intruder at all times, as the intruder follows its trajectory; in some other applications, the network's function may be simply to detect the presence of an intruder, in which case the network only needs to cover some part of the trajectory. Thus, given two points S and T, two relevant types of trajectories on the plane are proposed [10]: the *maximum breach path* and the *maximum support path* (In [10], these paths are called *maximal* breach path and *maximal* support path, respectively.).

The maximum breach path measures the vulnerability of a sensor network by, as the name suggests, completely avoiding the coverage area of the sensor network: It is a trajectory between the start point

[1] A *disk* of radius r centered at (x, y) under 2-norm in $\mathbb{R}^2$ is the set of points (p, q) such that $\sqrt{(p-x)^2 + (q-y)^2} \leq r$.

S and the stop point T that stays "as far away" from the sensors as possible. On the other hand, the maximum support path measures the efficiency of the network coverage: This path is a trajectory between S and T which stays "as close to the sensors" as possible. The *distance of a point P to the sensor network* is defined as the smallest Euclidean distance from P to one of the sensor nodes. A *maximum breach path* from S to T is a path from S to T such that the minimum distance from a point P in the path to the sensor network is maximized: This distance is called the *worst-case coverage distance* of the network. Similarly, a *maximum support path* from S to T is a path such that the maximum distance of a point P in the path to the sensor network is minimized: This distance is called the *best-case coverage distance* of the network.

When the topology of a sensor network changes, the quality of its coverage most probably will be affected. We would like to maintain an assessment on the quality of the network coverage — which, as explained above, can be done by maintaining the worst-case and best-case coverage distances — efficiently at all times. This would give a clear indication on how effective the network coverage is at any given point in time, possibly calling for the insertion of new nodes in the network (e.g., when the coverage deteriorates due to node failures) or to a network re-design phase. Whenever necessary, the actual paths which give the best-case and worst-case coverage distances can be retrieved. As we will see later, in Sections 4 and 5, our algorithms for maintaining the worst-case and best-case coverage distances have poly-logarithmic update and query costs, as defined later. To the best of our knowledge, this is the first work which formalizes and addresses this problem in a dynamic scenario.

For a moment, let's assume that all sensors have the same energy power and thus that all disks have the same radius r. We call such a sensor network a *uniform sensor network* with *coverage radius r*. In a uniform sensor network, all of the paths whose minimum distance of a point in the path to a sensor is larger than the coverage radius are equivalent, in the sense that the sensors in the network will not be able to detect an intruder using any such path. Similarly, all of the paths whose maximum distance of a point in the path to a sensor is smaller than the coverage radius are equivalent, in the sense that any such path is entirely contained in the coverage area of the network. The *worst coverage radius* (see [10]) is defined to be the maximum coverage radius r such that there exists a trajectory P between given points S and T which does not intersect the interior region of the area covered by the uniform sensor network (i.e., P may "touch" the coverage area, intersecting it at a discrete number of points only). We

can think of the worst-coverage radius as being the maximum energy that can be assigned to the sensor nodes which still would not prevent an intruder from escaping from S to T without being detected (for simplicity, we assume that a sensor will not be able to detect an intruder who only touches its coverage area). Correspondingly, the *best coverage radius* (see [10]) is defined to be the minimum coverage radius r such that there exists a trajectory between S and T that is totally covered by the uniform sensor network.

We introduce uniform sensor networks as a merely conceptual tool in order to facilitate the presentation of our approximation algorithms and their analyses, following a similar approach as Li et al. [9] (The actual sensor network in consideration has nodes with arbitrary energy levels and therefore is *not* assumed to be uniform.). In fact, if we think of a uniform sensor network built on top of the placement of the sensor nodes currently deployed in the general sensor network in consideration, the *worst-coverage radius* of the uniform network is indeed *equal to the worst-case coverage distance* of the general sensor network, and the *best-coverage radius* is indeed *equal to the best-case coverage distance*.

In order to dynamically maintain the best- and worst-case coverage distance efficiently, we need to maintain some information on the current topology of the sensor network; when the network topology changes, we need to update this information. We also perform queries for the current best-case and worst-case coverage distances, based on the information maintained. Hence, the cost (or running time) of our algorithms are measured in terms of their respective *update cost* — i.e., the cost to update the topology information, which is charged per "relevant" topology change in the network — and the *query cost*, which is the cost incurred when answering a query for the current best-case or worst-case coverage distance.

In Sections 4 and 5, we formally define a "relevant topology change" — which will henceforth be called an *event* — for the problems of maintaining the best-case and worst-case coverage distances, respectively.

The remainder of the paper is organized as follows. Section 1.1 states our results. In Section 2, we present some related work in the literature. Section 3 covers some preliminaries and sketches the basic framework of our solutions. We present the low constant approximation algorithms for the best- and worst-case coverage distance in Sections 4 and 5 respectively. In Section 6 we address the closely related problem of efficiently finding a shortest maximum support path. Section 7 concludes the paper with some possible lines for future work.

1.1 Our results

In this section, we summarize the main results in this paper. One of the main contributions of this work is to take into account the dynamic nature of sensor networks, and to propose a framework which can be used to continuously monitor the quality of the network coverage. Let n denote the current number of sensors in the network.

In the following sections, we present two algorithms to maintain low constant approximations on the best-case and worst-case coverage distances. Both algorithms have low update and query costs. Namely, our algorithms achieve a $(1+\epsilon)$-approximation on the best-case coverage distance, and a $(\sqrt{2}+\epsilon)$-approximation on the worst-case coverage distance, for any fixed $\epsilon > 0$. The amortized update cost per event of the best-case coverage distance algorithm is $O(\log^3 n)$, and the respective query cost is worst-case $O(\log n)$. For the worst-case coverage algorithm, the update cost per event is worst-case $O(\log^2 n)$ and the query cost is worst-case $O(1)$. A formal definition of an event for each of the problems considered follows in Sections 4 and 5, respectively.

As a byproduct of our algorithm for maintaining the worst-case coverage distance, we extend the concatenable queue data structure to also serve as a priority queue. All the operations on this extended data structure have worst-case $O(\log n)$ running time.

We also present an $O(n \log n)$ algorithm for computing an *exact shortest maximum support path* between two given points S and T, improving on the best-known previous results by Li et al. [9]. A shortest maximum support path from S to T is a maximum support path from S to T such that the Euclidean length of the trajectory followed in this path is minimum. In [9], two algorithms are presented for computing the maximum support path: One algorithm computes an *exact* shortest maximum support path in $O(n^2 \log n)$ time; the other algorithm provides a *2.5-approximation* on the shortest maximum support path in $O(n \log n)$ time. One should note that the algorithms presented by Li et al. can be implemented in a distributed fashion (we use the communication complexity as the time bound for the sequential versions of their algorithms), whereas the algorithms presented in this paper are all centralized.

The update costs of our algorithms for approximately maintaining the best- and worst-case coverage distances are much cheaper than maintaining the best- or worst-case coverage distances using the best-known algorithms in the literature prior to this work. In fact, the best previously known algorithm for maintaining the best-case (resp., worst-case) coverage distance maintains the exact distance by repeatedly

re-computing the maximum support path (resp., maximum breach path) using the $O(n \log n)$ algorithm by Li et al. [9] (resp., the $O(n^2 \log n)$ algorithm by Meguerdichian et al. [10]) each time an event occurs. To the best of our knowledge, this is the first work that explicitly addresses the problems of *dynamically* maintaining (*approximations* of) these two distances.

2 Related Work

Meguerdichian et al. [10] considered the problems of finding the maximum breach path and the maximum support path on a sensor network. They [10] present an $O(n^2 \log \Delta)$ runtime algorithm for the maximum breach path problem, where n is the number of sensors in the sensor network, and Δ is the difference between the highest and the lowest weight of an edge in the Voronoi Diagram of the sensor network. Their algorithm for computing the maximum support path has the same running time as their maximum breach path algorithm. The $O(\log \Delta)$ factor can be easily converted into $O(\log n)$ in the algorithm that solves the maximum breach path problem if we perform a binary search over a sorted list of the radii of sensors instead of using a linear search as in [10]. The algorithms presented in [10] heavily rely on geometric structures such as the Voronoi Diagram and Delaunay triangulation of the network, which cannot be constructed efficiently in a distributed manner.

Li et al. [9] prove the correctness of the algorithms given in [10]. They also show how to find a maximum support path in $O(n \log n)$ time using a centralized algorithm, or with $O(n \log n)$ communication complexity bits in a distributed fashion. In addition, Li et al. [9] present two algorithms for computing a shortest (with respect to the Euclidean length of the trajectory followed in this path) maximum support path: an algorithm that computes an exact shortest maximum support path with $O(n^2 \log n)$ worst-case communication complexity, and an algorithm that computes a 2.5-approximation of a shortest maximum support path (i.e. the total length of the obtained path is at most 2.5 times the length of a shortest maximum support path) with $O(n \log n)$ communication complexity.

Meguerdichian et al. [11] proposed an exposure-based formulation for analyzing the coverage of paths taken by polygonal objects: They define a path-dependent "integral", which consists of the trajectories of all the points of a polygonal object (the polygonal object is able to rotate), and not only of the trajectory of the object's center point.

Recently, Zhang and Hou [13] proved that if the communication rage of a sensor is at least twice its sensing range, a complete coverage of a convex area implies connectivity among the working set of nodes

and derive optimality conditions under which a subset of working sensor nodes can be chosen for full coverage. Wang et al. [12] designed a Coverage Configuration Protocol (CCP) that can provide different degrees of connected coverage and present a geometric analysis of the relationship between coverage and connectivity. Huang and Tseng [8] present an algorithm with runtime of $O(n^2 \log n)$ that decides whether every point in a given service area is covered by at least one sensor.

3 Preliminaries

Before heading into the technical details of our algorithms, we introduce some basic concepts which will be used in both Sections 4 and 5. The first concept we introduce is that of *growing disks*, which will help us translate our problems into graph connectivity problems.

The growing disks concept was previously proposed in [9]. We restate it in terms of the coverage radius of a uniform sensor network as defined in Section 1. (In Section 1, we saw how the coverage radius of a virtual uniform overlay sensor network directly relates to the worst-case and best-case coverage distances of the actual network.) Assume we have a uniform sensor network with coverage disks centered at the sensors. Define $U(r)$ to be the region on the plane composed of the union of all of the coverage disks when the coverage radius is r. Let $\overline{U(r)}$ be the complement of the region $U(r)$. At the very beginning, we set the coverage radius to be equal to 0. Then $U(r)$ is the union of discrete singletons. As the coverage radius grows, the disks centered at the sensors become larger and might get connected into larger regions. Therefore, $\overline{U(r)}$ might get disconnected into separate regions. For any two given points S and T, the best coverage radius is the minimum r such that S and T are in the same connected region of $U(r)$, while the worst coverage radius is the minimum r such that S and T belong to two disconnected regions in $\overline{U(r)}$. Hence, the best and worst coverage radius problems translate to connectivity problems on $U(r)$ and $\overline{U(r)}$, respectively. Figure 1 illustrates these ideas. We will further translate the best and worst coverage radius problems into graph connectivity problems.

We first show how to translate the best coverage radius problem into a graph connectivity problem. A *uniform disk graph* is the intersection graph of disks with uniform radius r (see [4]). In this graph, disks are vertices and there is an edge between two vertices if and only if the corresponding disks intersect.[2] The connectivity of $U(r)$ is naturally modeled by that of a uniform disk graph of radius r, denoted by $G(U(r))$. The best coverage radius is the minimum r such that the vertex corresponding to the disk

[2]If we rescale one unit to be $2r$, then a uniform disk graph is a unit-disk graph.

containing S is connected to that corresponding to the disk containing T in $G(U(r))$.

We also translate the worst coverage radius problem into a graph connectivity problem. However this case is rather more involved and we delay its presentation to Section 5.

When r is fixed, suppose that we have a poly-logarithmic running time query to check whether the region in either $U(r)$ or $\overline{U(r)}$ containing S is connected to that containing T. Then we can build an α-approximation algorithm, $\alpha > 1$, for either the best or the worst coverage radius problem, as we show in the next paragraph.

For the best coverage radius, consider the sequence of $U(r_i)$, such that $r_i = \alpha r_{i-1}$. Let i be such that S and T are connected in $U(r_i)$ but not in $U(r_{i-1})$. Since the best coverage radius falls in the interval $[r_{i-1}, r_i]$ and since r_i is at most αr_{i-1}, we know that r_i is an α-approximation on the best coverage radius. A similar argument on the sequence of $\overline{U(r_i)}$'s gives an α-approximation of the worst coverage radius.

Assume sensors occupy some space and cannot overlap. Then there is a constant lower bound on the coverage radius, denoted by $r_{\min}$. Due to the limited battery power, we assume that there is a constant upper bound on the coverage radius, denoted by $r_{\max}$. Let $R = r_{\max}/r_{\min}$. We need to maintain $\log_\alpha(R)$ copies of $U(r_i)$ or $\overline{U(r_i)}$. If updating the relevant connectivity information for each $U(r_i)$ or $\overline{U(r_i)}$ takes time $g(n)$, the overall update time is $\log_\alpha(R) \cdot g(n)$. The update time is poly-logarithmic on n provided that $g(n)$ is poly-logarithmic on n, and that R is bounded by a polynomial on n.

4 Dynamic Best-Case Coverage Distance

In this section, we present our $(1+\epsilon)$-approximation algorithm to maintain the best-case coverage distance following the framework presented in Section 3. Recall that, as shown in Section 3, finding the best-case coverage distance for given points S and T is equivalent to finding the minimum r such that S and T are connected in $G(U(r))$. Thus our main goal is to devise an approach to maintain the connectivity of the uniform disk graph $G(U(r))$ such that both the update cost and the query cost are poly-logarithmic on n, where n is the number of sensors in the network.

Holm, Lichtenberg and Thorup [7] showed that the connectivity of a graph can be maintained in amortized poly-logarithmic update cost, whereas each query takes worst-case $O(\log n / \log\log n)$ time. Guibas et al. [5] used Holm et al.'s algorithm to maintain connectivity on a unit-disk graph. The update cost in [5, 7] is charged per edge insertion or deletion. In order to be able to detect when uniform disks meet or separate on the plane (corresponding to an edge insertion or deletion on a unit-disk graph,

respectively), Guibas et al. [5] introduced a kinetic data structure specially tailored to handle this scenario.

The kinetic data structure framework was first proposed by Basch et al. [2, 3] to deal with dynamics. Their main contribution is a method to maintain an invariant of a set of moving objects in a discrete manner. They introduce the idea of keeping *certificates* as triggers for updates. When an object moves and a certificate fails, the consistency of the kinetic data structure is invalidated and an update is mandatory. Each failure of a certificate incurs a setup of up to a constant number of new certificates. Hence we are allowed to monitor the dynamics of a set of objects discretely and efficiently. The kinetic data structure requires that we know the flight plan (a specification of the future motion) [2, 5] of all disks, and that the trajectory of each disk can be described by some low-degree algebraic curve. We have the freedom to change the flight plan of a disk at any time. Basch [2] shows that kinetic data structures can efficiently support the dynamic operations of inserting and deleting objects into the system, provided those operations do not occur too often. The details of kinetic data structures are beyond the scope of this paper. Please refer to [3, 2, 5] for more information.

The kinetic data structure utilized in [5] can be viewed as a discrete event monitor. The events we need to monitor in order to maintain accurate connectivity information on $G(U(r))$ are when two disks meet or separate. In [5], two types of certificates are set up and the data structure allows us to determine a priori the time when an event will occur. When an event occurs, the topology of the uniform disk graph $G(U(r))$ changes and an update on the connectivity information is triggered. Hence the update cost is the cost to update the connectivity information of $G(U(r))$ per event. When a certificate fails and an event occurs, it takes constant time for the kinetic data structure to process the failure (due to the setup of at most a constant number of new certificates). We do not explicitly take this cost into account when computing the update cost of the maintenance of the connectivity information of $G(U(r))$, since it would not change the asymptotic bound on the update cost.

We adapt the main theorem in [5], Theorem 5.4, to better serve our purposes. The uniform disk graph $G(U(r))$ corresponds to a unit-disk graph if we rescale one unit to be equal to $2r$.

Lemma 1 (Adapted from Theorem 5.4 in [5]) *In [5], an algorithm to dynamically maintain the connectivity of $G(U(r))$ is presented. The update cost is amortized $O(\log^2 n)$ per event. The query cost is worst-case $O(\log n / \log\log n)$.*

We still need to show how to determine which disks contain the given points S and T, at any given

time. We sort all sensors according to their distances to the fixed point S. We maintain a binary heap on this ordering. Once the ordering changes, we update the heap in $O(\log n)$ time. This introduces a new type of event — namely when a sensor changes its location and needs to be re-inserted in this ordering — besides the other two events defined earlier. The update cost for this event is $O(\log n)$. To check which disk contains S, we find the closest sensor p to S. We check if the distance from p to S is larger than the coverage radius. If so, then S is not contained in any disk. Otherwise, we know that the disk centered at p contains the point S. This query takes constant time. We maintain the ordering of the sensors with respect to T in a similar way.

Combining the result in this section with the algorithmic framework presented in Section 3, we have our $(1+\epsilon)$-approximation algorithm (for any $\epsilon > 0$) for the best-case coverage distance by maintaining $\log_{1+\epsilon} R$ copies of $G(U(r))$, for $r = 1, (1+\epsilon), (1+\epsilon)^2, \ldots$. We perform a query operation by doing a binary search on the $\log_{1+\epsilon} R$ copies of $G(U(r))$.

Theorem 1 *Our algorithm dynamically maintains a $(1+\epsilon)$-approximation, for any $\epsilon > 0$, of the best-case coverage distance. The update cost of this algorithm is amortized $O(\log^2 n \cdot \log_{1+\epsilon} R)$ per event and the query cost is worst-case $O((\log n / \log\log n) \cdot \log\log_{1+\epsilon} R)$.*

Corollary 1 *If $\epsilon > 0$ is fixed, then our algorithm has amortized $O(\log^3 n)$ update cost per event, and worst-case $O(\log n)$ query cost.*

5 Dynamic Worst-Case Coverage Distance

In this section, we present our $(\sqrt{2}+\epsilon)$-approximation algorithm, for any $\epsilon > 0$, to dynamically maintain the worst-case coverage distance. We first present a $(1+\epsilon)$-approximation algorithm (for any $\epsilon > 0$) for a simplified sensor network model, where the coverage disks are considered under infinity-norm. Since there is only a $\sqrt{2}$ gap between infinity-norm and 2-norm, a $(1+\epsilon/\sqrt{2})$-approximation factor for infinity-norm dilates into a $(\sqrt{2}+\epsilon)$-approximation factor when applied to the 2-norm scenario, for any fixed $\epsilon > 0$. The infinity-norm of a vector $v = (x_1, \cdots, x_d)$ in a d-dimensional space is defined as $||v||_\infty = \max(|x_1|, \cdots, |x_d|)$. Under infinity-norm, the distance between two points on the plane is the maximum of the difference of their x coordinates and the difference of their y coordinates. Hence the coverage region of a sensor is square shaped and its boundary is composed of four line segments. As we will see later, this simple boundary shape allows for an efficient maintenance scheme.

Recall the solution framework presented in Section 3. The core of our algorithm is to check, for any two given points S and T, whether the region in $\overline{U(r)}$ containing S is connected to that containing T. If we can maintain some information such that each query on connectivity of regions takes only poly-logarithmic time, the cost of update against mobility is also poly-logarithmic.

In our algorithm, regions in $\overline{U(r)}$ are represented by their boundaries. Only one region in $\overline{U(r)}$ may be infinite in area. We call such an unbounded region the *outer face*. All of the other (bounded) regions are called *inner faces*. Since we consider the infinity-norm, each disk is represented by a square on the plane. Thus the boundary of any inner face is a simple cycle composed of a sequence of line segments, while the boundary of the outer face comprises several simple cycles. To differentiate these cycles, we call a cycle that is the boundary of an inner face an *inner cycle*, and a cycle on the boundary of the outer face an *outer cycle*. Figure 3 illustrates some of these concepts. The shaded areas in the figure define $U(r)$, and the unshaded areas define $\overline{U(r)}$. In (b), $\overline{U(r)}$ is divided into two regions, the unbounded region is the outer face, the bounded region is the inner face. The boundary of the inner face is an inner cycle and that of the outer face is an outer cycle. In (c), the boundary of the outer face consists of two disjoint outer cycles.

Below we describe a method which translates the connectivity of regions in $\overline{U(r)}$ into a graph connectivity problem. The first step is to represent outer cycles and inner cycles by a graph. There are only vertical line segments and horizontal line segments in both outer and inner cycles, and those line segments only meet at their endpoints. Hence we can draw a graph such that the vertices are the endpoints and the edges are the line segments. We call this graph the *connectivity graph* $G(\overline{U(r)})$. (For convenience, the connectivity graph will actually be implemented in a slightly different way, as we explain in Section 5.1.)

Every outer or inner cycle is a cycle in the graph and any two of them are disjoint, i.e., disconnected in the graph. This coincides with the fact that any two distinct inner faces are disconnected, and that any inner face is disconnected from the outer face.

The connectivity of $G(\overline{U(r)})$ is thus analogous to that of $\overline{U(r)}$: Two regions are connected in $\overline{U(r)}$ if and only if their boundary cycles are connected in the graph, or they are both part of the outer face boundary. Thus we could apply the algorithm proposed by Holm et al. [7], which dynamically maintains graph connectivity, to maintain the connectivity of the regions in $\overline{U(r)}$. The update cost per edge insertion or deletion for each $G(\overline{U(r)})$ in [7] is amortized $O(\log^2 n)$ and the query cost is $O(\log n / \log\log n)$,

implying an overall amortized update cost of $O(\log^3 n)$ and worst-case query cost of $O(\log n)$, with an approximation factor of $(1+\epsilon)$, for any fixed $\epsilon > 0$. However, $G(\overline{U(r)})$ is the union of simple disjoint cycles, each uniquely defining a region in $\overline{U(r)}$ and thus it allows for a more efficient update and query scheme, at the expense of a small degradation in the approximation factor. As we will show later, we can maintain the connectivity of all $G(\overline{U(r)})$ in overall worst-case update cost of $O(\log^2 n)$, with worst-case query cost of $O(1)$, while maintaining a $(\sqrt{2}+\epsilon)$-approximation on the worst-case coverage distance, for any fixed $\epsilon > 0$.

In the remainder of this section, we first describe the dynamics of the connectivity graph. Then we define three types of events which mandate updates. Our update cost is charged per event. Following that, we present a data structure, which is an extension of concatenable queues [1], to maintain the connectivity of the graph efficiently. Finally we present our major result on the worst-case coverage distance.

5.1 Dynamics of Cycles

In this section, we first formally define the representation we use for the connectivity graph $G(\overline{U(r)})$. Second, we address the dynamics of the connectivity graph. And finally, we present an algorithm for maintaining the connectivity information on the regions of $\overline{U(r)}$.

The boundary of a standalone square is the simplest cycle in $G(\overline{U(r)})$. We represent a square by eight vertices and eight edges as shown in Figure 2. For every corner X of a square, we introduce two vertices X and X'. Hence we have $O(n)$ vertices and edges in $G(\overline{U(r)})$, where n always denotes the current number of sensors in the network. The extra vertices help us to efficiently maintain the graph when squares start to move and overlap on the plane (including when sensors are added or removed from the network). In the following, we will show that the dynamics of sensors will not change the $O(n)$ bound on the number of vertices and edges.

When two squares meet, at most two pairs of line segments of their boundaries intersect. Without loss of generality, suppose a vertical edge $B'C$ intersects with a horizontal edge $E'F$ at a point Z, and the new boundary comprises edges $B'Z$ and ZF. Then we simply relocate vertices C and E' to Z, insert an edge CE' and remove edges CC' and EE' from $G(\overline{U(r)})$. Figure 2 illustrates this operation. Note that we do not introduce any new vertex or remove any old vertex. In fact, since $G(\overline{U(r)})$ contains no information of the vertex's location, we do not need to perform any "relocation" of a vertex when we

operate on $G(\overline{U(r)})$. The cases of a vertical edge intersecting with a vertical edge, and of a horizontal edge intersecting with a horizontal edge are analogous, and can thus be also handled by at most two edge insertions and at most two edge deletions. Since we never change the number of vertices in the graph, and since each vertex has degree at most 2, the $O(n)$ upper bound on number of vertices and edges in $G(\overline{U(r)})$ always hold. The following fact follows:

Fact 1 *When two squares meet or separate, up to four edge insertions and deletions are needed to update the connectivity graph* $G(\overline{U(r)})$.

When the topology of the network changes, cycles in $G(\overline{U(r)})$ may also undergo changes. A cycle may break into two smaller cycles; or two cycles may merge into a longer cycle. Both these operations impose changes on the connectivity of $G(\overline{U(r)})$. Cycles break or merge only when two sensors' coverage disks meet or separate. Hence we need to detect the time when those happen in order to trigger an update.

When a cycle breaks, it could break into an outer cycle and an inner cycle (as shown in Figure 3). We need to differentiate outer cycles from inner cycles since all outer cycles define the same region, namely the outer face. In order to determine whether a cycle is an outer cycle, one only needs to identify the topmost edge of the cycle: If the topmost edge of the cycle is the top boundary of a square, then the cycle is an outer cycle; otherwise, the topmost edge of a cycle is the bottom boundary of a square, and the cycle is an inner cycle. Hence we need to maintain the topmost edge of each cycle as sensors move. The topmost edge of a cycle may change only when two horizontal line segments swap their y position. Therefore we also need to monitor these line segment swaps.

Recall that the original problem we aim to solve is to check whether the region containing a given point S is connected to that containing T. We need to determine which region contains a given point and also to update this information as sensors move. As described in Section 4, we sort all sensors according to the distance from the fixed point S and maintain a binary heap on this ordering, with update cost $O(\log n)$ on the heap. In order to check which region S belongs to, we need to find the cycle representing the region. Again we find the closest sensor p to S and check if the distance is smaller than the radius of the coverage disk of p. If so, then the point S does not belong to any region of $\overline{U(r)}$. Otherwise, we check the eight vertices of the square representing the closest sensor to S, find the closest one of these vertices to S, and the cycle containing this closest vertex represents the region containing S. This query takes constant time. We maintain a similar data structure for T. Thus we also need to monitor and detect the

time when two sensors swap their relative position in these orderings.

We summarize all of the above in the following three types of events, which we need to monitor in order to trigger mandatory updates, as sensors move on the plane:

(I) Two vertical line segments swap their x position,

(II) Two horizontal line segments swap their y position, and

(III) Two sensor swap their position in the orderings of the sensor's distance to the given points S and T.

When events (I) or (II) occurs, we can check in constant time whether two coverage disks meet or separate. If they do, we check whether the event leads to a cycle break or merge, and update the data structure accordingly. When event (II) occurs, we can check whether the two horizontal line segments belong to the same cycle. If so, we may also need to update the topmost edge of the cycle. When event (III) occurs, we update the orderings with respect to distances to S and T.

We use the kinetic data structure as defined in [3] as our event monitor (unlike $G(U(r))$, $G(\overline{U(r)})$ is not a unit-disk graph and therefore the results in [5] do not apply). Each event can be detected and processed in constant time.

In the following, we present our update scheme. We will also show that the update cost per event is $O(\log n)$. We store a cycle as a sequence of consecutive edges. In Section 5.2 we introduce a data structure which supports the following operations on sequences of edges:

INSERT - insert an edge into a sequence

DELETE - delete an edge from a sequence

CONCATENATE - concatenate a sequence to the end of another sequence

SPLIT - split a sequence into two sequences

SWAP - swap the y position of two edges

MAX - return the topmost edge of a sequence

MEMBER - return the representative edge of a sequence

Each of these operations can be executed in worst-case running time $O(\log n)$, as stated in Lemma 4.

The update per type (I) or (II) event is as follows. When squares move and the shape of a cycle changes, up to a constant number of **INSERT** and **DELETE** operations are needed to update the cycle per event. When two edges in a cycle exchanges their y position, we execute **SWAP** to update the y position per event. We can execute **MAX** to know whether a cycle is an outer cycle or not. Recall that a cycle is an outer cycle if and only if the topmost edge of the cycle is the top boundary line segment of a square. Cycle merges or breaks can be carried out by a constant number of **CONCATENATE** and **SPLIT** operations. Since only a constant number of **INSERT**, **DELETE**, **CONCATENATE**, **SPLIT**, **SWAP** and **MAX** operations are executed per event, the update cost per event is worst-case $O(\log n)$. As we have explained earlier, the update cost per type (III) event is also $O(\log n)$.

A data structure that supports the operations above can also be used for efficiently performing a connectivity check. Assume that S and T are not covered by any sensor in $U(r)$ and therefore both belong to $\overline{U(r)}$. We can find the closest vertices u and v to points S and T respectively, in constant time. Then we check if u and v belong to the same cycle by performing two **MEMBER** operations. If so, then S and T belong to the same region in $\overline{U(r)}$. Otherwise, we need to check whether the closest cycles to S and T are both outer cycles by two executions of the **MAX** operation. If both of them are outer cycles, then both S and T belong to the outer face, and hence are in the same region. Otherwise, S and T belong to two disconnected regions. This procedure can be implemented in $O(\log n)$ time.

We summarize all of the above in Lemma 2.

Lemma 2 *For any two given points S and T, we maintain a data structure with $O(\log n)$ update cost per event such that the query to check whether the region in $\overline{U(r)}$ containing S is connected to that containing T takes $O(\log n)$ time.*

Combining Lemma 2 with the algorithmic framework presented in Section 3, we have our $(1+\epsilon)$-approximation algorithm, for any $\epsilon > 0$, for the worst-case coverage distance under infinity-norm, as stated in the lemma below. If every time we perform an update operation, we keep track of the smallest r_i such that S and T are disconnected in $G(\overline{U(r)})$, then each query operation can be performed in $O(1)$ time.

Lemma 3 *Under infinity-norm, our algorithm dynamically maintains a $(1+\epsilon)$-approximation of the*

worst-case coverage distance for any $\epsilon > 0$. The update cost is worst-case $O(\log n \cdot \log_{1+\epsilon} R)$ per event, and the query cost is worst-case $O(1)$.

Hence the $(\sqrt{2}+\epsilon)$-approximation algorithm for the worst-case coverage distance under 2-norm follows:

Theorem 2 *Our algorithm dynamically maintains a $(\sqrt{2}+\epsilon)$-approximation of the worst-case coverage distance, for any $\epsilon > 0$. The update cost is worst-case $O(\log n \cdot \log_{1+\frac{\epsilon}{\sqrt{2}}} R)$ per event, and the query cost is worst-case $O(1)$.*

Corollary 2 *If $\epsilon > 0$ is fixed, then our algorithm has worst-case $O(\log^2 n)$ update cost per event, and worst-case $O(1)$ query cost.*

5.2 Extended Concatenable Queue

In this subsection we introduce a data structure that supports the operations **INSERT**, **DELETE**, **CONCATENATE**, **SPLIT**, **SWAP**, **MAX** and **MEMBER** efficiently. The data structure is an extension of the concatenable queue data structure [1]. In [1], a concatenable queue is implemented by a 2-3 tree (a Red-Black tree would also work, for example), and all the data is stored at the leaf nodes. A concatenable queue supports the operations **INSERT**, **DELETE**, **CONCATENATE**, **SPLIT** and **MEMBER**, and each operation takes time $O(\log n)$ in the worst case. In the following paragrpahs, we willn show how to also implement the **SWAP** and **MAX** operations on a concatenable queue in $O(\log n)$ time.

We associate each edge's y coordinate to the corresponding leaf node in the 2-3 tree. To each internal node t, we associate the maximum y coordinate of a leaf node in the subtree rooted at t. This is done by comparing all the y coordinates associated to t's children in the tree, taking constant time per internal node. When the y coordinate of an edge changes, and a **SWAP** operation is invoked, it takes at most $O(\log n)$ time to climb up the tree and update all the internal nodes on the way up. Starting from any given edge on a cycle, it takes $O(\log n)$ time to reach the root of the 2-3 tree where we can find the topmost edge of the cycle. Hence the $O(\log n)$ running time of **MAX** follows.

We need also to justify that the above modification does not increase the running time of all other operations. Per each **INSERT** or **DELETE**, it takes an additional $O(\log n)$ time to update the y coordinate of all internal nodes due to the edge insertion or deletion. Both **CONCATENATE** and **SPLIT** are implemented by up to $O(\log n)$ joins or breaks of trees at the root node. Since updating the

y coordinate at the root node takes constant time (by comparing all the children of the root), we incur at most an additional $O(\log n)$ time per **CONCATENATE** or **SPLIT**. Thus the asymptotic running time of **INSERT**, **DELETE**, **CONCATENATE**, and **SPLIT** remains unchanged. The running time of **MEMBER** is not affected by **SWAP** or **MAX** operations.

Lemma 4 *The extension of the concatenable queue data structure supports the operations of* **INSERT**, **DELETE**, **CONCATENATE**, **SPLIT**, **SWAP**, **MAX** *and* **MEMBER**. *Each operation has worst-case running time of* $O(\log n)$.

6 Exact Shortest Maximum Support Path

We consider the problem of finding a maximum support path between S and T such that the Euclidean length of the trajectory followed by this path is minimum. Below we present an $O(n \log n)$ runtime solution, thus improving on the best-known previous results by Li et al. [9]. One should note that the algorithms presented in [9] can be implemented in a distributed fashion, whereas the algorithm we present in this section is intrinsically centralized.

We proceed as follows. First we compute the best coverage radius r_{best} using the algorithm of Li et al. [9] in $O(n \log n)$ time. Next, we obtain a collection of uniform disks by setting the radius of each sensor to be r_{best}. Let U denote the union of all these uniform disks. Define the complement region of the union $C = \mathbb{R}^2 \setminus U$.

The problem of finding a shortest maximum support path is equivalent to the problem of finding a shortest S, T-path in $\mathbb{R}^2$ avoiding C, since we are seeking for a maximum support path and r_{best} is the best coverage radius (Since r_{best} is the best coverage radius, any maximum support path is contained in U; in fact any path from S to T in U is a maximum support path.). A shortest maximum support path can only contain straight line segments as edges, otherwise the path would not be shortest. Therefore, we can replace each arc in C by a straight line segment. In such fashion we obtain a new set of obstacles C' as a collection of polygonal objects with possible "holes" that have a total $O(n)$ number of vertices. We can remove these "holes" and obtain slightly larger number of disjoint polygonal objects by cutting the existing objects with segments that connect the vertices of the holes and the external boundary in an arbitrary fashion. Note that the number of total vertices of the disjoint polygonal objects has not changed, i.e., it is still $O(n)$. Thus, our problem translates to that of finding a shortest path in $\mathbb{R}^2$ that avoids the polygonal obstacles in C'. The idea now is to use an algorithm by Hershberger and Suri [6],

which finds a shortest path between S and T on the Euclidean plane that avoids polygonal obstacles in $O(n \log n)$ time. Hence the total running time of our algorithm is $O(n \log n)$.

As described in Section 2, two algorithms are presented for computing the maximum support path by Li et al. in [9]: One algorithm computes an *exact* shortest maximum support path in $O(n^2 \log n)$ time; the other algorithm provides a *2.5-approximation* on the shortest maximum support path in $O(n \log n)$ time. Our algorithm improves on the running time of the former and on the approximation factor of the latter algorithm. One should note, however, that the algorithms presented by Li et al. can be implemented in a distributed fashion (we use the communication complexity as the time bound for the sequential versions of their algorithms), whereas our algorithm is centralized in nature.

7 Future Work

In this paper, we present poly-logarithmic dynamic algorithms to maintain approximations of two relevant measures — namely, the best- and worst-case coverage distances — of the quality of the network coverage in wireless sensor networks. An interesting open question is whether we can maintain exact best-case and worst-case coverage distances for Euclidean metric with poly-logarithmic update time.

Acknowledgment

We express our thanks to Micha Sharir for his comments.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley Pub. Co., 1974.

[2] J. Basch. Kinetic data structures. In *Ph.D. Dissertation, Stanford University*, 1999.

[3] J. Basch, L. J. Guibas, and J. Hershberger. Data structures for mobile data. In *Proc. 8th ACM-SIAM Sympos. on Discrete Algorithms*, pages 747–756, 1997.

[4] B. N. Clark and C. J. Colbourn. Unit disk graphs. *Discrete Mathematics*, 86:165–177, 1990.

[5] L. J. Guibas, J. Hershberger, S. Suri, and L. Zhang. Kinetic connectivity for unit disks. *Discrete Comput. Geom.*, 25:591–610, 2001.

[6] J. Hershberger and S. Suri. An optimal algorithm for euclidean shortest paths in the plane. *SIAM Journal on Computing*, 28(6):2215–2256, 1999.

[7] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic graph algorithms i: connectivity and minimum spanning tree. Technical Report DIKU-TR-97/17, Dept. of Computer Science, Univ. of Copenhagen, 1997.

[8] C.-F. Huang and Y.-C. Tseng. The coverage problem in a wireless sensor networks. In *Proc. 2nd ACM International Conference on Wireless Sensor Networks and Applications*, pages 115–121, 2003.

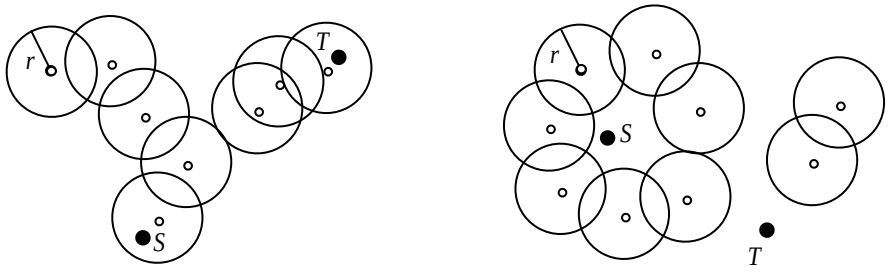
[9] X.-Y. Li, P.-J. Wan, and O. Frieder. Coverage in wireless ad-hoc sensor networks. *IEEE Transactions on Computers*, 52:1–11, 2003.

[10] S. Meguerdichian, F. Koushanfar, M. Potkonjak, and M. B. Srivastava. Coverage problems in wireless ad-hoc sensor networks. In *Proc. 20th IEEE INFOCOM*, pages 1380–1387, 2001.

[11] S. Meguerdichian, F. Koushanfar, Gang Qu, and M. Potkonjak. Exposure in wireless ad-hoc sensor networks. In *Proc. 7th ACM MOBICOM*, pages 139–150, 2001.

[12] X. Wang, G. Xing, Y. Zhang, C. Lu, R. Pless, and C. D. Gill. Integrated coverage and connectivity configuration in wireless sensor networks. In *Proc. 1st ACM Conference on Embedded Networked Sensor Systems*, 2003.

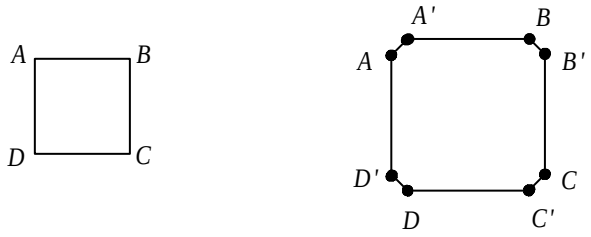
[13] H. Zhang and J. C. Hou. Maintaining sensing coverage and connectivity in large sensor networks. Technical Report UIUCDCS-R-2003-2351, UIUC, 2003.



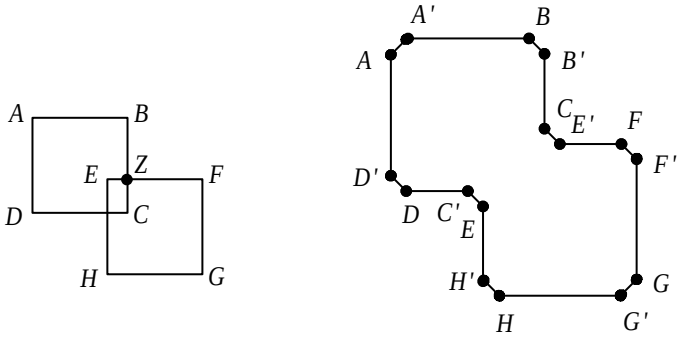
(*a*) Best-coverage radius: minimum r such that S and T are connected in $U(r)$

(*b*) Worst-coverage radius: minimum r such that S and T are disconected in $\overline{U(r)}$

Figure 1: Best and worst coverage radii.

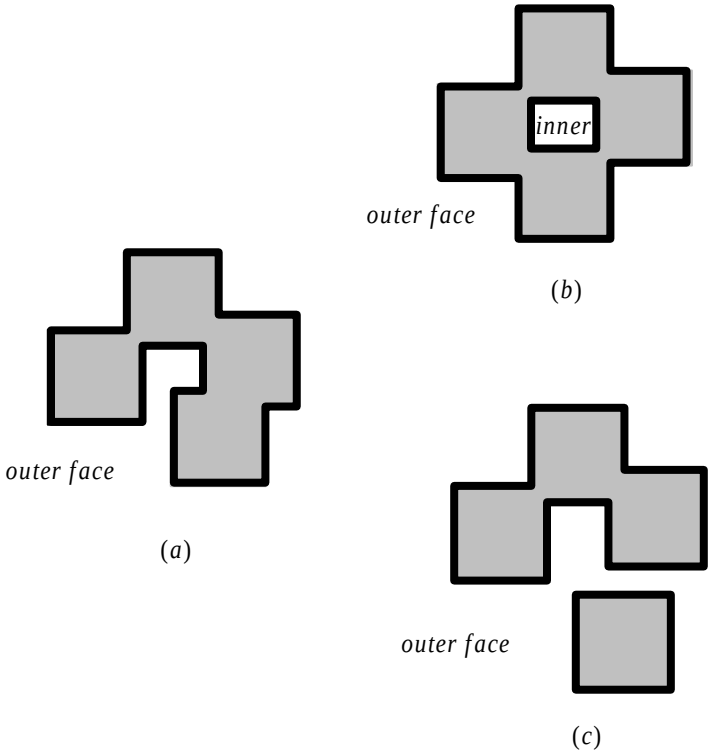


A square is represented by 8 vertices



Dynamics of vertices and edges when squares overlap. Vertices C, C', E and E' are relocated. Edges (C,C') and (E,E') are removed, and edges (C,E') and (E,C') are inserted.

Figure 2: Representation in $G(\overline{U(r)})$.



When the outer cycle in (*a*) breaks into two cycles, it can either break into an outer and an inner cycle, as shown in (*b*); or it can break into two outer cycles, as shown in (*c*).

Figure 3: Outer- and inner-cycles.