Greedy Virtual Coordinates for Geographic Routing

Ben Leong
Department of Computer Science
National University of Singapore
benleong@comp.nus.edu.sg

Barbara Liskov and Robert Morris
MIT Computer Science and
Artificial Intelligence Laboratory
{liskov, rtm}@csail.mit.edu

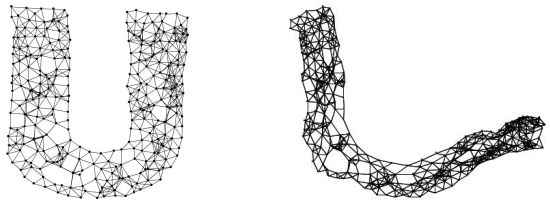
***Abstract*— We present a new approach for generating virtual coordinates that produces usable coordinates quickly and improves the routing performance of existing geographic routing algorithms. Starting from a set of initial coordinates derived from a set of elected perimeter nodes, *Greedy Embedding Spring Coordinates (GSpring)* detects possible dead ends and uses a modified spring relaxation algorithm to incrementally adjust virtual coordinates to increase the convexity of voids in the virtual routing topology. This reduces the probability that packets will end up in dead ends during greedy forwarding. The coordinates derived by GSpring achieve routing stretch that is up to 50% lower than that for NoGeo, the best existing algorithm for deriving virtual Euclidean coordinates for geographic routing. For realistic network topologies with obstacles, GSpring coordinates achieves from between 10 to 15% better routing stretch than actual physical coordinates.**

I. Introduction

Geographic routing algorithms [2, 11, 14, 17, 18] are an attractive alternative to traditional ad hoc routing algorithms [10, 27] for wireless networks because they scale better: the routing state maintained per node is dependent only on local network density and not network size [12].

Much of the work in recent years has been focused on improving routing performance with successively more complex and efficient algorithms [11, 14, 17, 18]. In this paper, we demonstrate that we can further improve routing performance by adjusting the routing coordinates. We present a new algorithm,*Greedy Embedding Spring Coordinates (GSpring)*, that is able to achieve routing performance comparable to actual physical coordinates for some classes of topologies and exceeds actual physical coordinates for realistic topologies with obstacles. Our approach is applicable in many settings because geographic location devices like GPS, Bat [8] and Cricket [28] are not yet cost effective for ubiquitous deployment on large wireless networks, and virtual coordinates are often employed when location information is not available [29].

Existing geographic routing algorithms work as follows: they first try to forward packets greedily, i.e., to the immediate neighbor that is closest in geographic distance to the destination; when a packet reaches a dead end, they then switch to a forwarding mode that guarantees packet delivery, e.g., face routing [11, 14, 18] or tree traversal [17]. Geographic routing algorithms tend to be most efficient when packets are forwarded greedily as much as possible [33], since greedy forwarding avoids switching to the costly guaranteed-delivery forwarding mode.



(a) Physical topology (b) GSpring topology

Fig. 1. Transformation of U-shaped physical topology to flat "smile" virtual topology by GSpring. The points represent the physical and virtual coordinates of nodes respectively and the lines indicate the connectivity between nodes.

Concave "voids" in the routing topology are bad for geographic routing since packets will tend to end up in the concave dead ends. GSpring starts from a set of initial coordinates, and uses a modified spring relaxation algorithm to incrementally adjust virtual coordinates. The key idea is for the nodes to detect situations that lead to dead ends during greedy forwarding, and to have them adjust their coordinates in such a way as to increase the convexity of existing voids in the routing topology. This reduces the probability that packets will end up in dead ends during greedy forwarding and thereby improves the routing performance of existing geographic routing algorithms.

GSpring uses a simple perimeter detection algorithm to identify nodes at the edge at the network and uses this information to assign initial coordinates. Subsequently, GSpring incrementally adjusts coordinates so that greedy forwarding succeeds more frequently. For example, for the U-shaped network shown in Fig. 1(a), packets forwarded greedily between nodes at the two ends of the U will mostly end up in a dead end and thereby have to be forwarded via face routing or tree traversal. The shape of the resulting virtual routing topology from running GSpring on the U-shaped network is a flat "smile" topology as shown in Fig. 1(b). With this new virtual routing topology, greedy forwarding will succeed almost all the time.

While GSpring typically requires about a thousand iterations for a 1,000-node network to converge, this is not an issue in a practical setting since GSpring quickly derives a set of coordinates that are relatively good and usable immedi-

ately once a network is initialized. Smaller networks will also converge in fewer iterations. GSpring coordinates are subsequently improved incrementally over time, and they can be used for routing even as they are improving.

The remainder of this paper is organized as follows: in Section II, we provide a review of existing and related work. In Section III, we describe how GSpring derives initial coordinates to bootstrap the algorithm. In Section IV, we describe how GSpring detects dead ends and incrementally adjusts coordinates so that greedy forwarding succeeds more frequently. Finally, we present our evaluation results for GSpring in Section V and conclude in Section VI.

II. Related Work

Rao et al. had earlier proposed the NoGeo family of coordinate assignment algorithms for ad hoc wireless networks [29]. In the most general version of their algorithm for systems where nodes have no location information, they designate two nodes as beacon nodes. Next, nodes determine if they are perimeter nodes from a heuristic based on their hop count from the beacons. Once the perimeter nodes are determined, $O(p^2)$ messages are exchanged, where p is the number of perimeter nodes, and the perimeter nodes use an error-minimization algorithm to compute their coordinates. Finally, the perimeter nodes are projected onto an imaginary circle and nodes determine their virtual coordinates using a relaxation algorithm that works by averaging the coordinates of neighboring nodes.

One major drawback for NoGeo is that it assumes that a network is static once the perimeter nodes are determined. Since the perimeter nodes are fixed, new nodes that join the system at physical locations outside the initial perimeter of the system will tend to get "flipped inward," causing the routing topology to "fold over" on itself. Such topologies are bad for geographic routing since geometric distance no longer corresponds to the routing distance, i.e., forwarding a packet greedily no longer guarantees that progress will be made. In contrast, because GSpring does not fix the coordinates of the perimeter nodes, it is a fully online algorithm that is amenable to the incremental addition and removal of nodes without any need for periodic system reset or global coordination. That said, it should be clarified that GSpring was developed for networks with non-mobile nodes. The assignment of coordinates in a highly mobile environment without geographic location devices is beyond the scope of this paper.

Arad and Shavitt have concurrently developed a algorithm called NEAR that attempts to predict dead ends and incrementally adjust routing coordinates to improve routing performance, by making adjustments based on the angle that each node makes with its neighbors [1]. Unlike GSpring, NEAR is a local algorithm so it can only detect local dead ends and is not effective for sparse networks with the average node degree below 10.

There is also a large body of work on the closely-related *node localization problem* for ad hoc wireless networks [5,21, 25, 30]. The goal is to assign coordinates to a set of non-location-aware wireless nodes in a distributed way so that they correspond as closely as possible to the actual physical coordinates. GSpring differs from these works in that the coordinates derived need not correspond to actual physical coordinates. In fact, it should be clear from the example of the U-shaped network in Fig. 1 that actual physical coordinates are often not optimal for geographic routing.

Also closely related to our work are some geographic routing algorithms based on non-Euclidean coordinate systems. Newsome and Song proposed a routing algorithm based on virtual polar coordinates called VPCR [23]. VPCR works relatively well, but it can incur significant overhead under node and network dynamics. Beacon Vector Routing (BVR) [7], HopID [34] and GLIDER [6] are routing algorithms that employ a set of landmark nodes (beacons). Coordinates are assigned to nodes based on their hop count distances to the beacons. Routing is done by minimizing a distance function to these coordinates. When a packet is trapped at a local minimum, they resort to scoped flooding. The major drawback of this approach is that it requires a large number of beacons (about 40) to achieve routing performance comparable to geographic routing algorithms. It is also somewhat cumbersome to have to specify a destination with a large set of distance vectors, and it may be costly to keep updating a node's coordinates when distance vectors change over time under network churn.

Caruso et al. proposed a variant of the landmark node scheme called VCap that finds three extremal-rooted landmark nodes to generate three dimensional coordinates [3]. Since VCap uses only 3 landmarks, it performs poorly in sparse networks. More recently, Mao et al. developed S4, a routing algorithm based on *compact routing* [32] that achieves good routing stretch with $O(\sqrt{N})$ state per node, where N is the network size [20]. S4 has been shown to be more efficient than BVR and has a theoretical worst-case bound of 3 for routing stretch.

Virtual coordinate assignment schemes (GNP [24], Big Bang [31] and Vivaldi [4]) have also previously been proposed for Internet applications, with a view to using the coordinates for estimating Internet round-trip times (RTTs) rather than for routing. GNP is a centralized system that uses a small number (5 to 20) of landmark nodes and coordinates are chosen based on the RTT measurements to these landmarks. Big Bang and Vivaldi are schemes that also derive coordinates by simulating physical systems. The former simulates particles in moving in a force field with friction, while the latter is similar to GSpring and simulates a physical system of springs.

III. Determining Initial Coordinates

GSpring works in two steps. First, each node assigns itself an initial coordinate. Subsequently, nodes adjust their coordinates by simulating a system of springs and repulsion

forces. The first step is only required when a network is first turned on and it quickly achieves a coordinate system that works approximately as well as NoGeo [29]. The second step takes a few hundred to about a thousand iterations (depending on topology and network size) and incrementally adjusts these initial coordinates to improve the greedy forwarding performance of the network.

In the rest of this section, we explain the algorithm for deriving initial coordinates. The algorithm for the subsequent incremental adjustment of coordinates is described in Section IV. More details can be found in [16].

A. Spring Rest Length

Each link between two neighboring nodes i and j that can communicate with each other is modeled with a spring of rest length l_{ij}. By scaling the spring rest length, the final coordinates obtained by GSpring will be scaled accordingly.

We found that it is preferable for nodes that share many common neighbors to be closer together in the virtual coordinate space than nodes that do not share any common neighbors. We thus define the *percentage of common neighbors*, r_{ij}, between two nodes i and j as follows: suppose i and j have a set of common neighbors $\mathcal{S}_{ij}$ and sets of neighbors that are not shared by the other, $\mathcal{S}_i$ and $\mathcal{S}_j$, respectively. Then,

$$r_{ij} = \begin{cases} 0, & \text{if } |\mathcal{S}_{ij}| + |\mathcal{S}_i| + |\mathcal{S}_j| = 0 \\ \frac{|\mathcal{S}_{ij}|}{|\mathcal{S}_{ij}|+|\mathcal{S}_i|+|\mathcal{S}_j|}, & \text{otherwise} \end{cases} \quad (1)$$

Hence, $0 \leq r_{ij} \leq 1$. The rest length of the spring between two nodes i and j, l_{ij}, is then given by:

$$l_{ij} = l_{max} - r_{ij}(l_{max} - l_{min}) \quad (2)$$

where l_{min} and l_{max} are constants such that $l_{min} < l_{max}$. In our implementation, we set $l_{max} = 100$ and $l_{min} = \frac{1}{10} l_{max}$.

B. Perimeter Detection and Initialization

When a network is first turned on, there is a need to bootstrap the network by assigning coordinates to a small set of nodes at the perimeter of the network. GSpring uses a simple hop-count-based algorithm to identify and assign coordinates to a small number of perimeter nodes and also to nodes that lie on the shortest paths between these nodes.

Perimeter Detection. We begin by detecting k nodes at the boundary of the network graph. We start by identifying a common reference node, r. This can be the node with the smallest identifier (*lowest-id node*). To achieve a consensus, each node will record and broadcast the identity of the node that it thinks is the lowest-id node. When a node hears about a node that has a lower identifier than its current lowest-id node, it will update its record and broadcast the identity of this new node. Also, by recording the hop count to this node, all nodes will eventually agree on this reference node, and will also know their own hop count to that node. This process will take no longer than $O(D)$ time, where D is the diameter of the network and the messages broadcast are small and contain only the identify of the lowest-id node and the broadcasting node's hop count to that node.

Next, the network comes to a consensus on the perimeter node p_1 that is farthest from r in terms of hop count. As before, each node will broadcast the node that it thinks is farthest from r and also that node's hop count to r. Ties are broken consistently by comparing node identifiers. Once p_1 is determined, p_2 is determined in a similar way, as the node that is farthest from p_1 in terms of hop count. Subsequent nodes p_i, $i = 3, \cdots, n$ are the nodes that have the maximum sum of the square roots of the hop counts from the nodes p_j, where $j = 1, \cdots, i-1$. This is illustrated in Fig. 2.

We use the sum of square roots instead of the sum of hop counts for $i > 2$ because the latter does not differentiate between two configurations with the same sum. For example, a node that is four and six hops away from two other perimeter nodes is as good as one that is five and five hops away. Using the sum of the square roots as the metric will tend to spread perimeter nodes evenly on the boundary of the network.

As more perimeter nodes p_i's are defined, each node p_i is associated with a vector of its hop counts to each of the other perimeter nodes p_j, $j \neq i$. Overall, this perimeter detection scheme will stabilize in $O(D)$ time, and the constant is relatively small since the number of perimeter nodes that we need to elect is small. The storage cost is small since the total maximum amount of information exchanged between any pair of nodes is at most equal to a square matrix consisting of $\binom{k}{2} = \frac{k(k-1)}{2}$ hop counts and k is small. We experimented with some values of k and found that we only require a small number of such perimeter nodes, and we set $k = 8$ in our implementation since it seems to work well in practice.

It might perhaps be helpful to clarify that while this process involves flooding the network, it only requires $O(1)$ messages per node. At each stage, the "winning" node effectively sends out a "wave" that "drowns out" the other competing nodes.

Arranging Perimeter Nodes on a Circle. After the perimeter nodes are elected, the next step is to assign a set of reasonable starting coordinates to them and a natural approach is to arrange them on a virtual circle. Our algorithm for doing so is based on a very simple observation: given the full hop count matrix for the set of perimeter nodes, we can, in general, deduce the adjacency relationship between the nodes along the perimeter of the network by identifying the pair of nodes with the minimal hops between them.

Coordinates for Perimeter Nodes. Once we have the cyclical ordering of the nodes $\{n_1, n_2, \cdots, n_k\}$, we determine the number of hops on the boundary of the network graph by summing the hop counts between adjacent nodes. Let the hop counts between adjacent nodes n_i and $n_{i+1 (\text{mod } k)}$ be h_i, and $H = \sum_{j=1}^{k} h_j$, then the radius of the virtual circle, C, is given by:

$$C = \frac{H \times l_{max}}{2\pi} \quad (3)$$

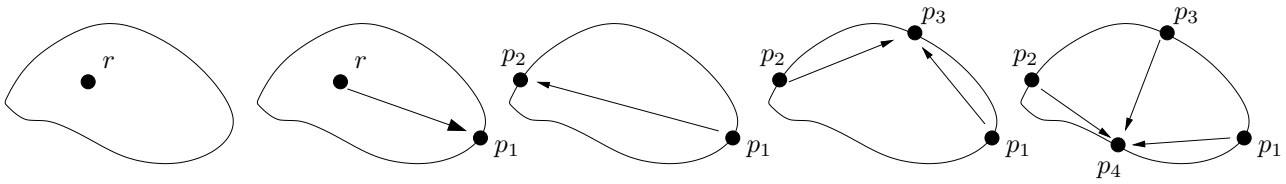


Fig. 2. Illustration of boundary detection algorithm.

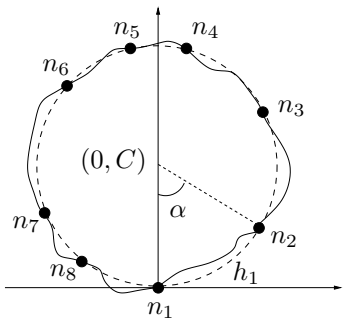


Fig. 3. Determination of starting coordinates for perimeter nodes.

The intuition here is to use $H \times l_{max}$ to approximate the circumference of the virtual circle.

Without loss of generality, we set the coordinates of n_1 as the origin, $(0, 0)$. The coordinates of the remaining nodes are spread out along a circle of radius C, centered at $(0, C)$ according to their relative hop distances. We illustrate this in Fig. 3. For example, the coordinates of n_2 is the point on the circle such that $\frac{\alpha}{2\pi} = \frac{h_1}{H}$, where h_1 is the hop count between n_1 and n_2. This process is analogous to hooking up and stretching out a trampoline.

Interpolation. The above procedure will derive the coordinates for k perimeter nodes. We observe that the hop matrix between all the perimeter nodes is known by all nodes since it is propagated by broadcast. Each node also knows its hop count from each perimeter node. A node x can thus determine that it is on the shortest path between one pair of perimeter nodes if $h_i = \bar{h}_i + \bar{h}_{i+1(\text{mod } k)}$, where h_i is the hop count between some pair of perimeter nodes n_i and $n_{i+1(\text{mod } k)}$ and $\bar{h}_i$ and $\bar{h}_{i+1(\text{mod } k)}$ are the hop counts from x to the two nodes, respectively. When a node satisfies this condition, it will derive its initial coordinates by interpolating accordingly between the coordinates of n_i and $n_{i+1(\text{mod } k)}$ and $\bar{h}_i$, which can be calculated from the hop matrix.

When this algorithm terminates, a small number of perimeter nodes at the boundary of the network and some nodes in the middle of the network will have derived a set of initial coordinates. The remaining nodes then derive their coordinates from these initialized nodes as described below.

C. Obtaining Initial Coordinates from Initialized Neighbors

Wireless nodes periodically broadcast *keepalive* messages to inform their neighbors of their presence. Nodes that know their coordinates will piggyback this information in their *keepalive* messages. Nodes without coordinates (i.e., a non-perimeter node that didn't receive its coordinates via interpolation, or a new node that has just joined the network) listen for broadcasts and derive their coordinate using one of the following rules:

- Case 1: If the node has only one initialized neighbor j, choose coordinates on the circle of radius l_{ij} centered at j that makes the greatest angle with a pair of the one-hop neighbors of j. If j has only one other neighbor, add i at the point on the circle directly opposite of that neighbor. If j has no other neighboring node, select a point on the circle at random.
- Case 2: If i has at least two initialized neighbors, find the two initialized neighbors with virtual coordinates that are farthest apart and pick the midpoint between these nodes as the initial coordinates.

If a node does not have an initialized neighbor, it will wait until at least one of its neighbors is initialized. Nodes that join the network after a network has stabilized will also derive their initial coordinates in a similar manner.

IV. Greedy Embedding Relaxation

After nodes obtain an initial assignment of coordinates, they will incrementally adjust their coordinates periodically by exchanging coordinate information with their initialized neighbors and collectively simulate a spring system with repulsion forces through a series of iterations. In this section, we describe the distributed algorithm for the adjustment of coordinates. The goal of this algorithm is to make concave voids in the routing topology more convex.

A. Preliminaries

A coordinate assignment is often referred to as an *embedding*. A *greedy embedding* is a graph that has the property that given any two distinct nodes s and t, there is a neighbor of s that is closer (in Euclidean distance) to t than s is [26]. In other words, we can pick any two nodes in the graph and successfully forward a packet between them using only greedy forwarding.

Since geographic routing works best when packets are forwarded greedily as much as possible [33], an important measure of "goodness" for a virtual coordinate assignment or embedding is the probability that a packet can be successfully forwarded between two randomly chosen nodes using only a simple greedy forwarding. We call this measure the *greedy forwarding success rate*.

We define the *region of ownership* of a node as the set of points that are closer to it than to its immediate neighbors in

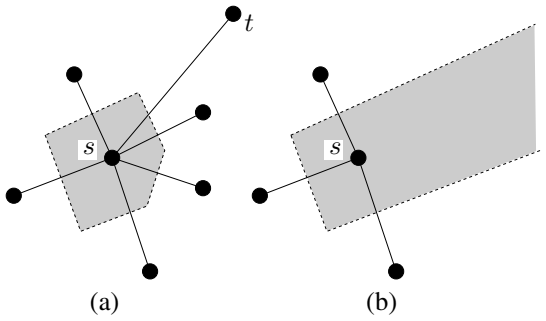


Fig. 4. Regions of ownership for the node s is shaded.

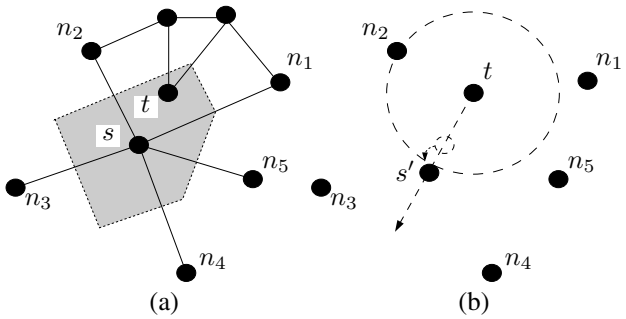


Fig. 5. Required adjustment to move node s so that node t is no longer in its region of ownership. Original region of ownership for s is shaded in gray.

the network connectivity graph. For example, in Fig. 4(a), the region of ownership for node s is a pentagon and independent of the position of node t since t is far from s. The region of ownership for a node is constructed by finding the intersection of all the half-planes formed by the bisectors of the edges to each neighboring node.

The region of ownership is often a closed polygon. The region of ownership can also be unbounded as illustrated in Fig. 4(b). The regions of ownership are often similar to the Voronoi diagram for a set of points. The key difference is that in the construction of the Voronoi diagram, we have global knowledge and hence it divides a plane into disjoint partitions. Region of ownerships are often not disjoint.

Theorem 1: *An embedding of a Euclidean graph is greedy if and only if the region of ownership of every vertex does not contain any other vertices of the graph.*

Proof: It is easy to see that a graph which has a vertex u with a region of ownership that contains another vertex v of the graph cannot be greedy. Consider a packet with destination v at vertex u. Clearly, the packet is not deliverable using just greedy forwarding since u is closer to the destination than all v's neighbors.

Next, suppose that there exists a *non-greedy* graph embedding where the region of ownership of every vertex does not contain any other vertices. Since the embedding is non-greedy, it means that there exists a source-destination vertex pair where greedy forwarding will cause a packet to reach a dead end at some intermediate vertex u, such that node u is not the destination node v. We know that v must be in the region of ownership for u. If not, the packet would be forwarded to one of u's neighbors. However, since the region of ownership of every vertex does not contain any other vertices of the graph, we have a contradiction. ■

The key insight of our work is that the region of ownership can be used to adjust coordinates to increase the greedy forwarding success rate of a virtual routing topology and thereby improve the performance of existing geographic routing algorithms. To understand how this is done, consider the example in Fig. 5(a), where we have a node s with another node t within its region of ownership. We observe that to ensure that t does not lie in the region of ownership for s, it is sufficient for us to shift s away from t to a point s' such that the distance between s' and t is greater than the distance between t and the neighbor of s that is nearest to t (i.e., n_2). A simple way to achieve this is to have s be repelled by t as long as s remains within the circle centered at t and a radius determined by the neighbor of s that is nearest to t, as illustrated in Fig. 5(b). While we use only local adjustments, such adjustments in aggregate have a net global effect of incrementally increasing the convexity of the voids in the routing topology.

We refer to all the nodes within a node's region of ownership as its *conflict set*. If there are multiple nodes within the conflict set, we can repeat the above process to find a point s' that satisfies the above condition for all nodes in the conflict set. There are, however, some configurations for which it is impossible to do so by simply shifting the position of s alone [16].

B. Basic Spring Relaxation Update Rule

From *Hooke's Law*, the force vector that the spring between two nodes i and j exerts on node i, F_{ij}, is given by:

$$F_{ij} = \kappa \times (l_{ij} - \parallel x_i - x_j \parallel) \times u(x_i - x_j) \tag{4}$$

where κ is the spring constant, x_i and x_j are the coordinates of nodes i and j, respectively, l_{ij} is the rest length of the spring, the scalar quantity $(l_{ij} - \parallel x_i - x_j \parallel)$ is the displacement of the spring from rest, and $u(x_i - x_j)$ is the unit vector from x_j to x_i.

The net force exerted on a node i, F_i, is the sum of the forces from the springs attached to all its immediate neighbors:

$$F_i = \sum_{j \neq i} F_{ij} \tag{5}$$

A node will periodically update its coordinates based on the virtual coordinates of its immediate neighbors using the following rule:

$$x_i = x_i + \frac{\min(|F_i|, \alpha_t)}{|F_i|} F_i \tag{6}$$

where α_t is a damping constant that decreases over time.

C. Greedy Embedding Update Rule

After a node has obtained initial coordinates and the update rule described in Equation (6) no longer yields any significant changes to its virtual coordinates, it will send a geocast[1] message to its region of ownership. Nodes in the region will respond with their current virtual coordinates. After a pre-determined interval, the node will have heard from all the nodes within its conflict set. Once the conflict set is determined, a node adjusts its coordinates so as to "move away" from the nodes in its conflict set.

If nodes are discovered within its region of ownership, a node will use a modified coordinate update rule. Each node k in the conflict set for node i will exert a force of repulsion R_{ik} on node i as follows:

$$R_{ik} = \delta \times u(x_i - x_k) \tag{7}$$

where δ is the repulsion constant. The total force acting on a node is now the sum of the spring forces and a capped total of the repulsion force as follows:

$$F_i = \overbrace{\sum_{j \neq i} F_{ij}}^{\text{spring forces}} + \overbrace{\frac{\min(|\sum_{k \neq i} R_{ik}|, R_{max})}{|\sum_{k \neq i} R_{ik}|} \sum_{k \neq i} R_{ik}}^{\text{capped conflict set repulsion forces}} \tag{8}$$

The repulsion force from the conflict set serves two purposes: (i) it tends to force nodes that are topologically separated from each other apart; and (ii) it makes concave voids more convex and hence improves the greedy forwarding success rate of the network. The reason why we need to cap the repulsion forces at some maximum R_{max} is that in a large network, a given node may find that it has a very large conflict set and we do not want the repulsion of the conflict set to overwhelm the spring forces. In our implementation, $\kappa = 0.5$, $\delta = 0.5$ and $R_{max} = 10$.

Analogy to Simulated Annealing. One issue that we have to deal with is that the nodes may sometimes end up in a local minimum analogous to a local minimum-energy state for a physical system: the physical analogy is a tangled mess of springs. To help the system break out of such minima, when a node s has at least one node in its conflict set, with a small probability p at each update step, instead of making an incremental adjustment according to Equation (6), it will set its coordinates as the point s' where it has no nodes in its conflict set, if such a point s' exists. While this process occasionally causes the system to end up in a somewhat unfavorable configuration, the basic spring relaxation algorithm will restore the configuration to a "good" state relatively quickly. In our implementation, we set $p = 0.1$.

Rendezvous. While GSpring as described uses geocast, what it requires is not geocast, but a *rendezvous* mechanism [16]. Because geographic routing uses coordinates and not node identifiers, there must be a way for nodes to discover the coordinates of destination nodes. This information is provided by a *location service*; possible implementations are discussed in GLS [19] and by Rao et al. [29].

[1]Geocast [9, 13, 16, 22] is a routing primitive that delivers a packet to all the nodes in a specified target spatial region instead of to an individual node.

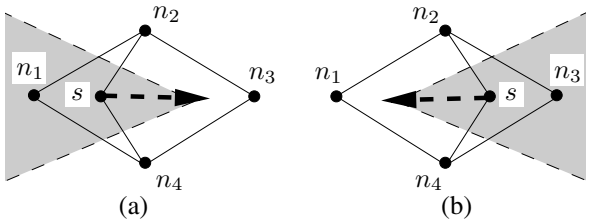


Fig. 6. Configuration where a node, s, will oscillate and is unable to obtain virtual coordinates that will keep other nodes out of its region of ownership (shaded in gray).

A location service can also be used as a rendezvous mechanism. Nodes update their coordinates with the location service when they join the network and when they move. GSpring can then query the location service to obtain the information. The relative costs of geocasting versus querying will depend on the design and implementation of the respective services.

It is also important to note that routing with geographic coordinates would be quite impractical if the destination coordinates are constantly changing. Hence, in a practical implementation of GSpring, each node will have maintain two sets of coordinates: a set of routing coordinates that correspond to a node's record at the location service and another set of GSpring coordinates that are adjusted periodically according to the GSpring algorithm. A node's routing coordinates are then updated with its GSpring coordinates and the location service updated accordingly at regular intervals.

D. Damping and Hysteresis

Since GSpring simulates a spring system, nodes can in principle oscillate forever. The introduction of repulsion forces can also give rise to an oscillating configuration. One example of such a configuration is shown in Fig. 6. In this example, s is first repelled by n_1, and subsequently by n_3 when it ends up as shown in Fig. 6(b), and hence the system returns to the configuration in Fig. 6(a). However, since it had heard from n_1 before, it will keep n_3 as the node in its conflict set and remain in the configuration in Fig. 6(a).

Hence, a node will keep track of the nodes that it hears from and adopt a new conflict set for computing the repulsion forces as described in Equation (8), only if it hears from new nodes in the new conflict set. Like others [4], we achieve stability by introducing *damping* and *hysteresis*.

The rate of progress for GSpring is controlled by the size of the damping constant α_t, which decreases with the progress of time. More specifically, since the nodes broadcast *keepalive* messages periodically to inform its neighbors of its location, we use the interval between broadcasts as the update interval and each node tracks the number of iterations it spends performing relaxation. Once the number of iterations exceeds a pre-determined threshold T, α_t is scaled by an exponentially

decreasing constant as follows:

$$\alpha_t = \begin{cases} \alpha_{max}, & \text{if } t < T \\ \alpha_{max} e^{-\frac{t}{T}}, & \text{otherwise} \end{cases} \quad (9)$$

where α_{max} and T are constants and t is the count of the number of iterations after a node starts updating its coordinates. If the magnitude of the displacement $\min(|F_i|, \alpha_t)$ falls below a minimum threshold α_{min}, a node will consider itself stabilized and no longer updates its coordinates. At this point, the node will also record the force vector F_i that is acting on it as F_{stop}.

The parameter α_{max} controls the *hysteresis* factor in the system. If the force vector F_i acting on a node changes sufficiently so that $|F_{stop} - F_i| > \alpha_{max}$ at some point, a node will reset its record of t to zero and start updating its coordinates again. In our implementation, $\alpha_{min} = 1$, $\alpha_{max} = 5$ and $T = 50$.

V. Performance Evaluation

In this section, we present the results of our evaluation of GSpring. We evaluated the routing performance of existing geographic routing algorithms with coordinates obtained with the GSpring algorithm, with actual physical coordinates, and with those obtained with NoGeo [29], which is the best existing algorithm for deriving virtual Euclidean coordinates.

In our simulations, we adopted a simple radio model: all nodes have a uniform radio range; two nodes can communicate if and only if they are within radio range of each other and if their line-of-sight does not intersect an obstacle. The simulations were performed using our own high-level event-driven simulator [15]. While the uniform radio model is relatively simple, by including obstacles, we generated a diverse range of topologies, which we believe is adequate for the purposes of understanding the performance of GSpring under common operational scenarios [16].

We measured routing performance in terms of *hop stretch*, where hop stretch is the ratio of the number of hops on the route between two nodes to the number of hops in the shortest path (in terms of hops). We also evaluated the scalability of GSpring with regards to network size and the cost overhead in terms of the number of iterations required for convergence to a set of stable coordinates and the number of geocasts messages sent and received.

A. Routing Performance

Geographic routing is known to be a relative easy problem for dense networks (where the average node degree is greater than 16) [14]. While many proposed algorithms have been shown to work well for dense networks, most perform relatively poorly for sparse networks [1,7,23,29]. Large networks are likely to be heterogeneous, with both dense and sparse regions, so our approach is to systematically evaluate networks over a range of network densities up to an average node degree of 16.

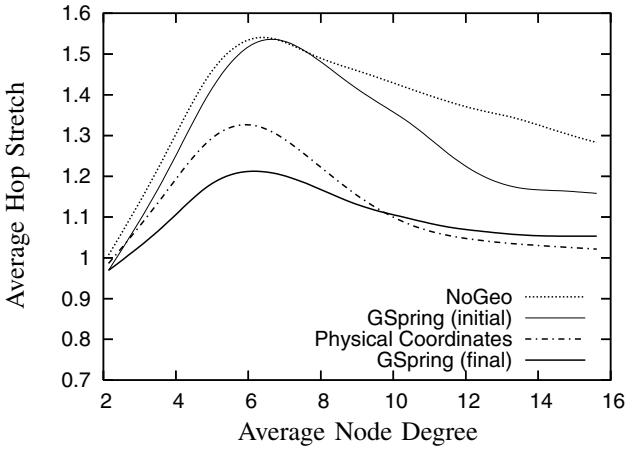


Fig. 7. Plot of GDSTR hop stretch with GSpring coordinates.

We evaluate GSpring by studying its effect on routing using existing geographic routing algorithms [16]. Because of space constraints, we will only present the results for GDSTR [17] here, since GDSTR has been shown to be generally more efficient and significantly cheaper to deploy than geographic face routing algorithms [11, 14, 18].

In Fig. 7, we plot the routing performance of GDSTR with GSpring coordinates (both initial assignment according to hop-count algorithm described in Section III and the final coordinates after the spring relaxation algorithm converges), NoGeo coordinates and for actual physical coordinates in small networks (with up to 500 nodes). These networks were generated by scattering an appropriate number of nodes at random over a 10×10 unit square.

Our results show that while the routing performance of the initial coordinates obtained by GSpring are comparable to that for NoGeo, the final coordinates yield significantly better routing performance. In particular, the final coordinates obtained by GSpring seem to achieve better routing performance than actual physical coordinates for sparse networks (average node degree below 8) and comparable performance for dense networks.

B. Greedy Forwarding Success Rate

We measured the greedy forwarding success rates for the various networks and found an inverse relation between routing stretch and greedy forwarding success rate. Our results are shown in Fig. 8. For relatively sparse networks with average node degrees between 5 and 8, GSpring achieves greedy forwarding success rates that are about 15% higher than that for the true physical coordinates. Given the significant improvement in the routing performance for GSpring coordinates in Fig. 7, it is surprising that GSpring only achieves a 15% improvement in the greedy forwarding success rate. It might be worth further study to understand why GSpring is only able to improve the greedy forwarding success rate by 15% and to see if modifications can be made to the algorithm to further improve the greedy forwarding success rate.

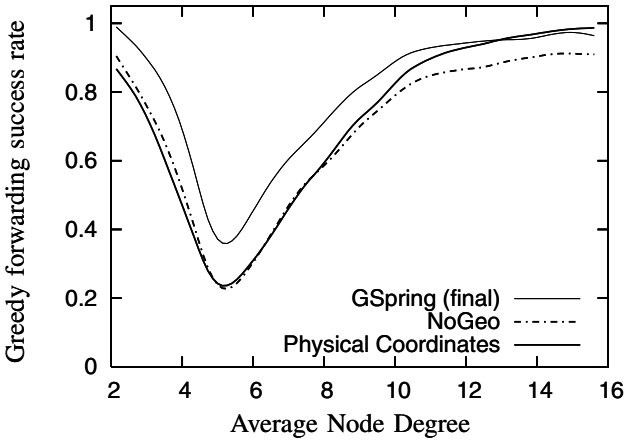


Fig. 8. Plot of greedy forwarding success rate with network density.

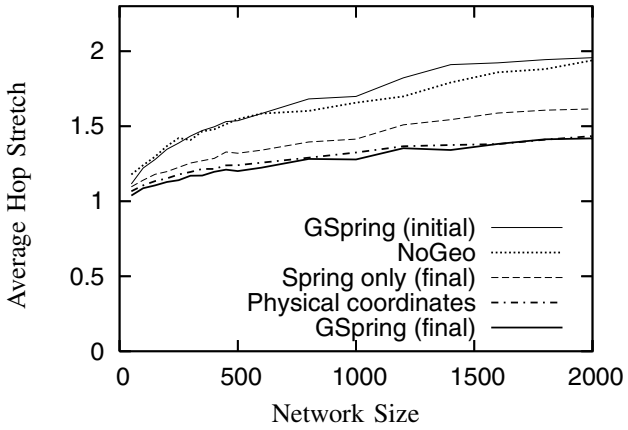


Fig. 9. Plot of GDSTR hop stretch for sparse networks (average node degree 6.5).

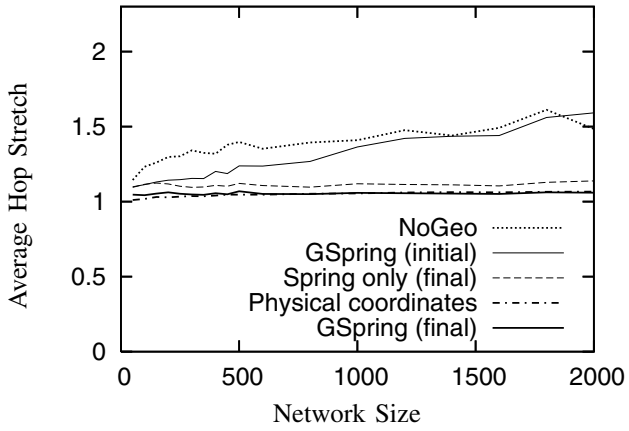


Fig. 10. Plot of GDSTR hop stretch for dense networks (average node degree 12).

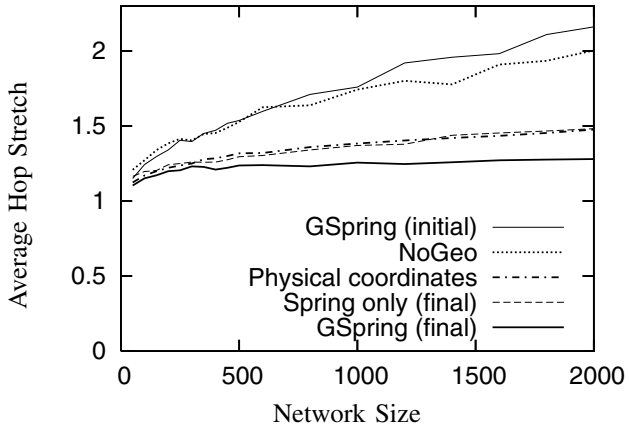


Fig. 11. Plot of GDSTR hop stretch for networks with obstacles (average node degree 7).

C. Scalability and Obstacles

To understand the scaling properties for GSpring and the effect of obstacles, we evaluated routing performance over a range of topologies of different densities and studied the effect of obstacles [16]. Due to space constraints, we present only the results for three sets of networks that represent the major classes of network topologies: sparse networks, dense networks, and networks with obstacles. These networks were generated by scattering nodes randomly over a $x \times x$ unit square, and scaling x by a factor of $\sqrt{n}$ for each network size n. A hundred networks were generated and evaluated for each network size for sizes that range from 50 to 2,000 nodes. To generate the networks with obstacles, we scattered some cross-shaped obstacles at random. The results are shown in Figs. 9, 10 and 11 respectively.

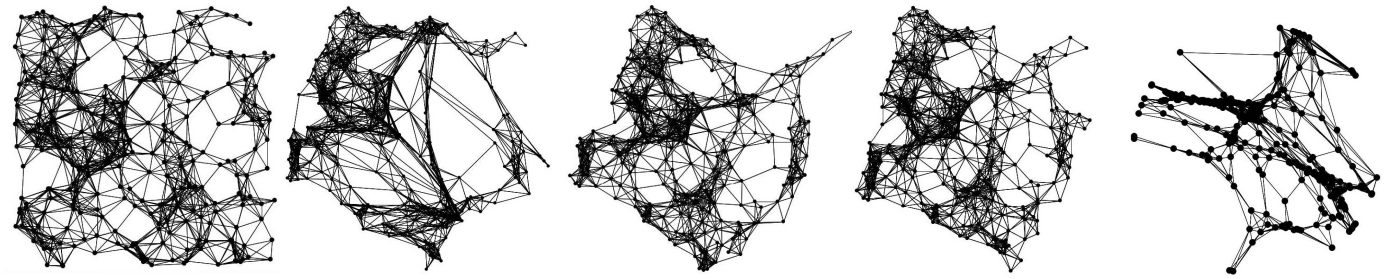
In all three cases, GSpring derives an initial set of coordinates that yields routing performance that is similar to NoGeo. The coordinates obtained after GSpring stabilizes are however able to achieve significantly better performance. In particular, as shown in Figs. 9 and 10, GSpring achieves hop stretch that is approximately equal to that achieved by actual physical coordinates, which is between 30% to 50% lower than that for NoGeo coordinates.

Since obstacles are common in real networks, it is important to understand the performance of GSpring in the presence of obstacles. As shown in Fig. 11, NoGeo performs poorly for networks with obstacles. The routing performance for NoGeo worsens progressively with increasing network size. For 2,000-node networks, GSpring achieves up to 50% lower hop stretch than NoGeo. GSpring on the other hand, achieves better routing performance than actual physical coordinates, and we found that its performance is somewhat independent of the obstacle density. In particular, for the networks we investigated, GSpring achieves from between 10 to 15% better routing stretch than actual physical coordinates.

Also shown in these figures is the routing performance of coordinates obtained with only the Spring Relaxation Update Rule and without simulating the conflict set repulsion forces ("Spring only"). We see that while conflict set repulsion is not very helpful for dense networks, it is critical for good performance in both sparse networks and networks with obstacles.

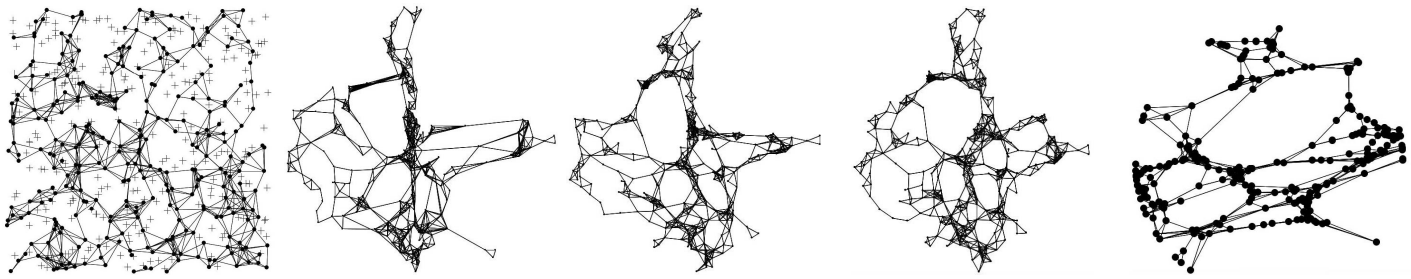
D. Example Topologies

To provide some physical intuition for the effect of GSpring, we plot the derived coordinates for two example networks with 300 nodes in Figs. 12 and 13. The virtual topologies generated by NoGeo are also provided for reference.



(a) Actual location (b) GSpring (initial) (c) Spring only (final) (d) GSpring (final) (e) NoGeo

Fig. 12. Derived coordinates for sample dense 300-node unit disk graph (UDG) network.



(a) Actual location (with obstacles) (b) GSpring (initial) (c) Spring only (final) (d) GSpring (final) (e) NoGeo

Fig. 13. Derived coordinates for sample 300-node network with cross-shaped obstacles.

These examples clearly illustrate that GSpring makes the voids in the routing topology more convex. Because the network in Fig. 12 is dense and does not have many convex voids, the final configuration of the GSpring coordinates is quite similar in shape to the actual physical configuration, On the other hand, because many of the voids for the network in Fig. 13 are concave, the final configuration is quite different from the actual physical layout of the network.

E. Convergence and Costs

We evaluated the costs of GSpring in terms of the number of iterations required for convergence and the number of geocast messages that have to be sent. In Fig. 14, we plot the hop stretch of random 1,000-node networks of various configurations against the number of iterations. These results demonstrates that GSpring converges relatively quickly and hop stretch falls sharply to within 10% of the final hop stretch within about 500 iterations.

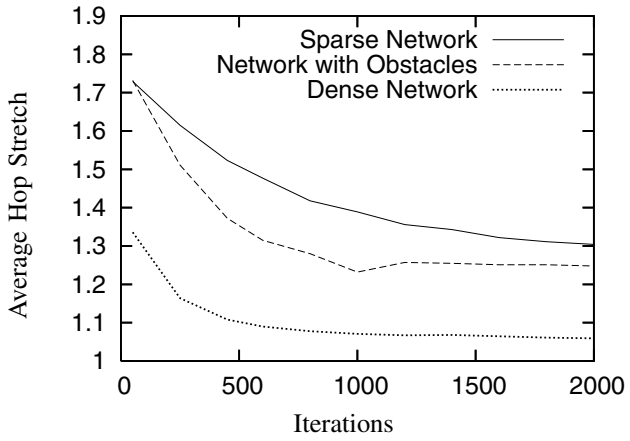


Fig. 14. Routing performance of random 1,000-node networks over time.

In Fig. 15, we plot the cumulative number of geocasts messages sent and received per node over time for random 1,000-node networks over time. We make two observations from these results: (i) GSpring requires a relatively small number of geocast messages, and (ii) networks with obstacles tend to generate more messages. The latter is a natural consequence of the fact that networks with obstacles will tend to have more concave voids and hence nodes have large conflict sets.

Since GSpring requires several simulation parameters (i.e., spring rest length, spring constant, repulsion constant, etc.) to be set, we systematically explored a range of values for each parameter. We found GSpring to be relatively robust to the parameter settings, i.e., GSpring seems to work relatively well over a wide range of parameter settings and it does not require much effort to optimize the algorithm [16]. Like other spring algorithms [4], GSpring will converge over a relatively wide range of "reasonable" increments.

We also found that we can trivially increase the rate of convergence by increasing the damping and hysteresis constants, α_{min} and α_{max}, with little effect on routing performance [16]. The key tradeoff is a slight increase in the number of geocast messages in most cases. This is because with increased damping or hysteresis, nodes are likely to stop adjusting their coordinates sooner, often even before they have completely eliminated all the nodes in their region of ownership.

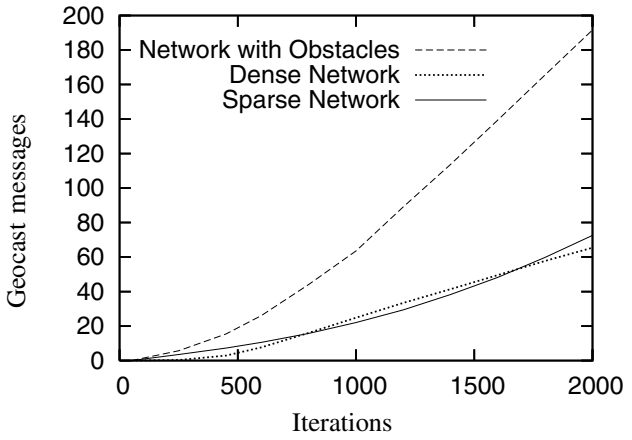


Fig. 15. Cumulative number of geocasts messages sent and received per node over time for random 1,000-node networks.

VI. Conclusion

In this work, we demonstrate that we can improve the routing efficiency of existing geographic routing algorithms with a good virtual coordinate assignment. GSpring quickly derives a set of coordinates that are relatively good and usable immediately, and subsequently adjusts the coordinates incrementally to increase the convexity of voids in the virtual routing topology. After it converges, GSpring achieves routing stretch that is up to 50% lower than that for NoGeo [29], and it often achieves up to 10 to 15% better stretch compared to routing over actual physical coordinates by converging to a virtual topology that has a higher greedy forwarding success rate than the actual physical topology.

GSpring was developed for networks with non-mobile nodes. While the assignment of virtual coordinates to quasi-static networks is practical and sometimes a necessity, it is not clear that it is feasible to use virtual coordinates for routing in a highly mobile environment. The deployment of GSpring in networks with a mixture of static and mobile nodes and in real radio networks remains as future work.

References

[1] N. Arad and Y. Shavitt. Minimizing recovery state in geographic ad hoc routing. In *Proceedings of MobiHoc 2006*, May 2006.

[2] P. Bose, P. Morin, I. Stojmenovic, and J. Urrutia. Routing with guaranteed delivery in ad hoc wireless networks. *Wireless Networks*, 7(6):609–616, 2001.

[3] A. Caruso, S. Chessa, S. De, and A. Urpi. GPS free coordinate assignment and routing in wireless sensor networks. In *Proceedings of IEEE Infocom '05*, pages 150–160, March 2005.

[4] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A decentralized network coordinate system. In *Proceedings of the ACM SIGCOMM '04 Conference*, Portland, Oregon, August 2004.

[5] L. Doherty, K. S. J. Pister, and L. E. Ghaoui. Convex position estimation in wireless sensor networks. In *Proceedings of IEEE Infocom '01*, pages 1655–1663, 2001.

[6] Q. Fang, J. Gao, L. J. Guibas, V. de Silva, and L. Zhang. GLIDER: Gradient landmark-based distributed routing for sensor networks. In *Proceedings of IEEE Infocom '05*, March 2005.

[7] R. Fonseca, S. Ratnasamy, J. Zhao, C. T. Ee, D. Culler, S. Shenker, and I. Stoica. Beacon vector routing: Scalable point-to-point routing in wireless sensornets. In *Proceedings of NSDI 2005*, May 2005.

[8] A. Harter, A. Hopper, P. Steggles, A. Ward, and P. Webster. The anatomy of a context-aware application. In *Proceedings of Mobicom 1999*, August 2001.

[9] Q. Huang, C. Lu, and G.-C. Roman. Spatiotemporal multicast in sensor networks. In *Proceedings of SenSys 2003*, pages 205–217, New York, NY, USA, 2003. ACM Press.

[10] D. B. Johnson and D. A. Maltz. Dynamic source routing in ad hoc wireless networks. In *Mobile Computing*, volume 353. 1996.

[11] B. Karp and H. T. Kung. GPSR: greedy perimeter stateless routing for wireless networks. In *Proceedings of Mobicom 2000*, pages 243–254, Boston, MA, August 2000.

[12] Y.-J. Kim, R. Govindan, B. Karp, and S. Shenker. Geographic routing made practical. In *Proceedings of NSDI 2005*, May 2005.

[13] Y.-B. Ko and N. H. Vaidya. Geocasting in mobile ad hoc networks: Location-based multicast algorithms. Technical Report TR-98-018, Texas A&M, September 1998.

[14] F. Kuhn, R. Wattenhofer, Y. Zhang, and A. Zollinger. Geometric ad-hoc routing: Of theory and practice. In *Proceedings of PODC 2003*, July 2003.

[15] B. Leong. Geographic routing network simulator, 2004. http://www.comp.nus.edu.sg/ bleong/geographic/simulator.htm.

[16] B. Leong. *New Techniques for Geographic Routing*. PhD thesis, June 2006.

[17] B. Leong, B. Liskov, and R. Morris. Geographic routing without planarization. In *Proceedings of NSDI 2006*, May 2006.

[18] B. Leong, S. Mitra, and B. Liskov. Path vector face routing: Geographic routing with local face information. In *Proceedings of ICNP 2005*, November 2005.

[19] J. Li, J. Jannotti, D. S. J. D. Couto, D. R. Karger, and R. Morris. A scalable location service for geographic ad hoc routing. In *Proceedings of MobiCom '00*, pages 120–130, 2000.

[20] Y. Mao, F. Wang, L. Qiu, S. S. Lam, and J. M. Smith. S4: Small state and small stretch routing protocol for large wireless sensor networks. In *Proceedings of NSDI 2007*, April 2007.

[21] D. Moore, J. Leonard, D. Rus, and S. Teller. Robust distributed network localization with noisy range measurements. In *Proceedings of the ACM SenSys '04*, pages 50–61, November 2004.

[22] J. C. Navas and T. Imielinski. Geocast - geographic addressing and routing. In *Proceedings of MobiCom '97*, pages 66–76, 1997.

[23] J. Newsome and D. Song. GEM: Graph EMbedding for routing and data-centric storage in sensor networks without geographic information. In *Proceedings of SenSys 2003*, November 2003.

[24] T. Ng and H. Zhang. Towards global network positioning. In *Proceedings of IEEE Infocom '02*, June 2002.

[25] D. Niculescu and B. Nath. Ad hoc positioning system (APS) using angle of arrival (AoA). In *Proceedings of IEEE Infocom '03*, March 2003.

[26] C. H. Papadimitriou and D. Ratajczak. On a conjecture related to geometric routing. In *Proceedings of ALGOSENSORS 2004*, pages 9–17, July 2004.

[27] C. Perkins and P. Bhagwat. Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. In *Proceedings of ACM SIGCOMM'94 Conference*, pages 234–244, August 1994.

[28] N. B. Priyantha, A. Miu, H. Balakrishnan, and S. Teller. The cricket compass for context-aware mobile applications. In *Proceedings of Mobicom 2001*, July 2001.

[29] A. Rao, C. H. Papadimitriou, S. Shenker, and I. Stoica. Geographic routing without location information. In *Proceedings of Mobicom 2003*, pages 96–108, San Diego, CA, September 2003.

[30] C. Savarese, J. Rabay, and K. Langendoen. Robust positioning algorithms for distributed ad-hoc wireless sensor networks. In *Proceedings of the USENIX Technical Annual Conference*, June 2002.

[31] Y. Shavitt and T. Tankel. Big-bang simulation for embedding network distances in euclidean space. In *Proceedings of the IEEE Infocomm*, April 2003.

[32] M. Thorup and U. Zwick. Compact routing schemes. In *Proceedings of SPAA '01*, pages 1–10, New York, NY, USA, 2001. ACM Press.

[33] G. Xing, C. Lu, R. Pless, and Q. Huang. On greedy geographic routing algorithms in sensing-covered networks. In *Proceedings of MobiHoc '04*, pages 31–42, 2004.

[34] Y. Zhao, B. Li, Q. Zhang, Y. Chen, and W. Zhu. Hop ID based routing in mobile ad hoc networks. In *Proceedings of ICNP 2005*, November 2005.