New Techniques for Geographic Routing

by

Ben Wing Lup Leong

M.Eng., Massachusetts Institute of Technology (1997)
S.B., Massachusetts Institute of Technology (1997)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2006

Author . . . . . .
Department of Electrical Engineering and Computer Science
May 30, 2006

Certified by . . . . . .
Barbara Liskov
Ford Professor of Engineering
Thesis Supervisor

Accepted by . . . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

New Techniques for Geographic Routing

by

Ben Wing Lup Leong

Submitted to the Department of Electrical Engineering and Computer Science
on May 30, 2006, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

As wireless sensor networks continue to grow in size, we are faced with the prospect of emerging wireless networks with hundreds or thousands of nodes. Geographic routing algorithms are a promising alternative to tradition ad hoc routing algorithms in this new domain for point-to-point routing, but deployments of such algorithms are currently uncommon because of some practical difficulties.

This dissertation explores techniques that address two major issues in the deployment of geographic routing algorithms: (i) the costs associated with distributed planarization and (ii) the unavailability of location information. We present and evaluate two new algorithms for geographic routing: *Greedy Distributed Spanning Tree Routing (GDSTR)* and *Greedy Embedding Spring Coordinates (GSpring)*.

Unlike previous geographic routing algorithms which require the planarization of the network connectivity graph, GDSTR switches to routing on a spanning tree instead of a planar graph when packets end up at dead ends during greedy forwarding. To choose a direction on the tree that is most likely to make progress towards the destination, each GDSTR node maintains a summary of the area covered by the subtree below each of its tree neighbors using convex hulls. This distributed data structure is called a *hull tree*. GDSTR not only requires an order of magnitude less bandwidth to maintain these hull trees than CLDP, the only distributed planarization algorithm that is known to work with practical radio networks, it often achieves better routing performance than previous planarization-based geographic routing algorithms.

GSpring is a new virtual coordinate assignment algorithm that derives good coordinates for geographic routing when location information is not available. Starting from a set of initial coordinates for a set of elected perimeter nodes, GSpring 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, and improves the routing performance of existing geographic routing algorithms.

The coordinates derived by GSpring yield comparable routing performance to that for actual physical coordinates and significantly better performance than that for NoGeo, the best existing algorithm for deriving virtual coordinates for geographic routing. Furthermore, GSpring is the first known algorithm that is able to derive coordinates that achieve better geographic routing performance than actual physical coordinates for networks with obstacles.

Thesis Supervisor: Barbara Liskov
Title: Ford Professor of Engineering

Acknowledgments

Words cannot express my gratitude to Barbara Liskov for her patience and guidance. She has been infinitely patient and extremely supportive. I am blessed to have had her as my adviser.

I would also like to express my appreciation to Robert Morris and Erik Demaine for agreeing to be my thesis readers. I benefited tremendously from their wise counsel and indispensable advice.

I would like to acknowledge the wonderful people at PMG who have contributed tremendously towards making my life in graduate school a wonderful experience: Ben Vandiver, Daniel Myers, David Schultz, Dorothy Curtis, James Cowling, Liuba Shrira, Kathryn Chen, Rodrigo Rodrigues, Sameer Ajmani, and Winnie Cheng. Special mention goes to Roger Moh for being a wonderful housemate as well.

I spent my first two years of graduate school with ANA and I am also grateful for the friendship and support I received from the members of ANA: Arthur Berger, David Clark, Dina Katabi, George Lee, Indraneel Chakraborty, Ji Li, John Wroclawski, Karen Sollins, Steven Bauer, and Xiaowei Yang.

I was also grateful to have worked for Hari Balakrishnan as his TA for one semester. Hari is an outstanding teacher and a source of inspiration for me. I was also blessed with two wonderful fellow TAs, Jaeyeon Jung and Alex Yip.

I am indebted to Tracey Ho and Sayan Mitra for being wonderful collaborators. I am especially grateful to Sayan Mitra and Rui Fan for the many fruitful discussions over many random things. Without them, some of the ideas contained within this dissertation would not have come into existence!

The MIT Pistol Team has always been a part of my experience here at MIT. I would like to take this opportunity to remember Pat Melaragno, my late pistol coach, and to thank him for his encouragement and support. Pat was a tremendous source of encouragement during my undergraduate days at MIT. He was a friend, a mentor and a father to me. I am also grateful to Will Hart, the current pistol coach for his friendship and support these last few years that I've been back in graduate school. It's also been a blessing to get to know the young shooters and to see MIT win the Collegiate Pistol Championships last year.

I probably would not have made it through grad school without the fellowship and support of my friends from the GCF Thursday night bible study: Aaron and Michelle Kirtley, Ali Hadiashar,

Allan Fong, Andrew McDonnell, Damien Jourdan, Esther Chen, Esther Lee, Gary and Karlene Maskaly, Hyung Jun Kim, Johanna Goth, Jun Sun, Karen Zee, Livia King, Mobolaji Olurinde, Shandon Hart, and Wouter Waalewijn.

Park Street Church has been my family away from home for these last five years. I am so blessed to have met many wonderful friends at the Park Street Church Sunday International Fellowship (SIF): Amy Liu, Babayel Anne, Betty Nelson, Cindy Lu, Dave and Janet Harkness, Gaby and Jose, Ilo and Irena Mitre, Isabella and Herman, Jie Hui Zhu, Lily Law, Macia Santos, Milda, Minjung Cho, Myrna Peterson, Stuart and Hsuan Delorme, Tom and Terri Baskett, and Tracy Vail. I am especially grateful for the love and support from my fellow Singaporeans at SIF: Jijon Sit, Felicity Chan, Kaloh Yeh, Sumin Koo, Karen Hong, and Pamela Yoong.

Finally, I would like to acknowledge my parents and family for their relentless support and love. Special mention goes to my beloved wife, Waiping, for just being the most wonderful wife in the world. I thank her for taking care of everything at home so that I have no worries and can concentrate on my studies.

Blessed am I indeed. The fact that this dissertation was completely on time is a miracle. I praise the Lord for His loving goodness. I pray that I may continue to walk in His ways as I return to Singapore and embark on a new phase of life.

Contents

List of Figures

Chapter 1

Introduction

As wireless sensor networks continue to grow in size, we are faced with the prospect of emerging wireless networks with hundreds or thousands of nodes. While earlier generations of such networks employed routing protocols that did not require a point-to-point routing primitive [32, 33], geographic routing algorithms have recently been proposed as a new routing primitive for data-centric storage [74] and for running more complex queries [55] over such networks. It has also been proposed that geographic routing be used for reduced state routing over the wired Internet [24]. We can expect an increasing demand for point-to-point routing in sensor networks in the future as data-centric applications and networks containing actuators become more common.

Although geographic routing algorithms have been available for several years, there are few known deployments of such algorithms in practice. To our knowledge, there are two main obstacles that stand in the way of deployment: (i) practical difficulties and costs associated with distributed planarization and (ii) the unavailability of location information. The goal of this research is to develop new techniques and algorithms that can address these two shortcomings and make deployments more practical and feasible than they are today. In this dissertation, we present two new algorithms that are specifically designed to address these difficulties: *Greedy Distributed Spanning Tree Routing (GDSTR)* [51] and *Greedy Embedding Spring Coordinates (GSpring)*.

Existing geographic routing algorithms [7, 39, 47, 52] 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 switch to a forwarding mode that guarantees packet delivery. The idea is to route the packet around "voids" in the routing topology by forwarding it along a face of the planarized network graph. This technique, called *face routing*, was first proposed by Kranakis et al. [46] and is the current default approach to geographic routing. We henceforth refer to these algorithms collectively as *geographic face routing* algorithms.

GDSTR is a geographic routing algorithm that uses a spanning tree as the backup routing topology instead of a planar graph like the geographic face routing algorithms. In order to choose a direction on the tree that is most likely to make progress towards the destination, each GDSTR

node maintains a summary of the area covered by the subtree below each of its tree neighbors. While GDSTR requires only one tree for correctness, it uses two for robustness and to give it an additional forwarding choice.

GSpring is a new virtual coordinate assignment algorithm that derives good coordinates for geographic routing of non-location-aware wireless nodes. Starting from a set of initial coordinates derived from a set of elected perimeter nodes, GSpring uses a modified spring relaxation algorithm that incrementally adjusts virtual coordinates to increase the convexity of voids in the virtual routing topology. This reduces the probability of packets ending up in dead ends during greedy forwarding and improves the routing performance of existing geographic routing algorithms.

More specifically, these two algorithms represent our attempt to solve the following problems:

- Given a network of nodes with assigned coordinates, route packets efficiently between connected nodes, without planarizing the network graph and without requiring more than $O(1)$ state to be stored at each node; and
- Given a network of nodes with no location information, assign coordinates to the nodes to maximize the greedy forwarding success rate of the network.

Our work makes very few assumptions. First, because geographic routing uses coordinates and not node identifiers, there must exist a mechanism for nodes to discover the coordinates of destination nodes. For this purpose, geographic routing must be augmented by a *location service*, which we assume exists and hence, we shall not attempt to address the design of a geographic location service in this dissertation. Existing solutions like GLS [53] or a Distributed Hash Table (DHT) [73] can be employed.

Second, we make few assumptions about radio behavior. The only requirement is that nodes must agree on whether or not they are neighbors. GDSTR is robust against location errors, unlike previously proposed algorithms [79]. While GSpring assumes the availability of geographic routing algorithm that supports geocast[1], a requirement that is satisfied by GDSTR, this requirement can be replaced by a different centralized or distributed location-service-like mechanism, as described in Section 5.5.

Third, our work assumes that the nodes in the system are *quasi-static*. Our algorithms can cope with intermittent node failures and the occasional node joining the system, but are not designed to work with large scale changes in the connectivity of the network over a short interval.

The rest of this chapter explains why geographic routing is a challenge in practical systems, our reasons for believing that virtual coordinates can achieve better geographic routing performance than true physical coordinates, our contributions and the organization of this dissertation.

[1] A geocast algorithm is a routing algorithm for which a target region can be specified and a message will then be routed to all nodes in the specified region.

1.1 Challenges of Geographic Routing

It was originally believed that geographic face routing could provide guaranteed packet delivery with only a minimal amount of state stored at each node [37]. Unfortunately, it turns out that face routing is critically dependent on the planarization of the network topology graph for correctness, and the distributed planarization of network graphs for practical radio networks is an extremely challenging problem [38]. The distributed planarization algorithms that were initially proposed [2, 20, 21, 54, 84–86] depended on a rather strict condition, called the Unit Disk Graph (UDG) assumption, for correctness. This assumption is often violated in practice because of obstacles and the physical characteristics of real radios [41].

A related and somewhat more subtle source of difficulty was that these algorithms also assumed that nodes knew their radio ranges and locations accurately. A recent empirical study has found that the communication ranges of wireless networks are highly dependent on the environment and may be highly irregular [88]. Errors in the localization of the nodes can also cause planarization to fail [40, 79].

A major breakthrough was made by Kim et al. in developing the Cross-Link Detection Protocol (CLDP) [42], which produces a subgraph on which face-routing-based algorithms are guaranteed to work correctly. Their key insight is that starting from a connected graph, nodes can independently probe each of their links using a *right-hand rule* to determine if the link crosses another link in the network. CLDP uses a two-phase locking protocol to ensure that no more than one link is removed at any time from any given face; in this way it guarantees that the removal of a crossed link will not disconnect the network. While CLDP is able to planarize an arbitrary graph, every single link in the network has to be probed multiple times, and has a high cost.

While distributed planarization is now a solved problem, the high maintenance costs and complexities associated with the deployment of face routing algorithms (with CLDP) make it worthwhile to consider an alternative approach to face routing. In particular, since we know that geographic routing tends to work best when packets are forwarded greedily [87], a natural approach would be to consider a backup routing mode that does not require the use of a planar graph and hence allows us to avoid the planarization problem altogether [51]. Our hypothesis is that a geographic routing algorithm that does not require planarization would be able to avoid the associated costs of planarization and hence make geographic routing less costly and more practical.

1.2 Case for Virtual Coordinates in Geographic Routing

While geographic location devices like GPS, Bat [27] and Cricket [71] are relatively mature technologies, these technologies are not yet cost effective for ubiquitous deployment on large wireless networks. A natural question would be the following: suppose we have a small number of nodes

that are equipped with location devices, how do we derive good virtual coordinates for the remaining non-location-aware nodes efficiently so that we can achieve good routing performance for existing geographic routing algorithms?

The idea of assigning virtual coordinates to non-location-aware nodes is not new. In fact, one previous approach, NoGeo [73], can derive relatively good virtual coordinates when no location information is available.

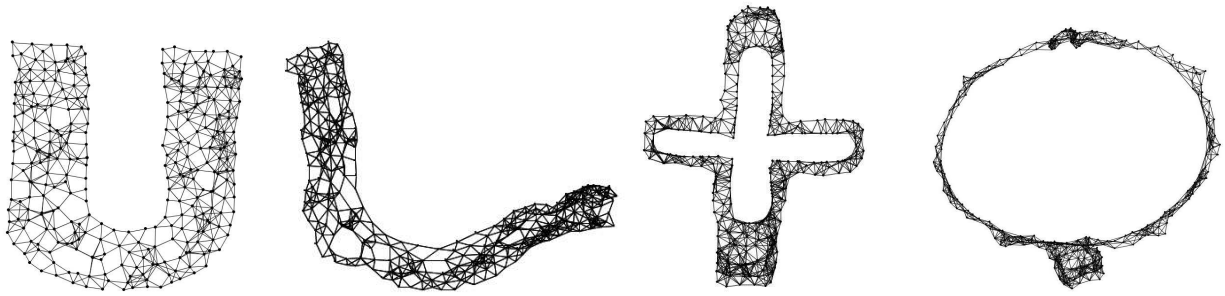
In this dissertation, we take the idea one step further: since virtual coordinates do not need to mimic actual physical coordinates for geographic routing to be relatively efficient, instead of simply deriving a set of virtual coordinates suitable for geographic routing, we should exploit the flexibility in the choice of coordinates to optimize the routing performance of existing geographic routing algorithms.

Geographic routing algorithms tend to be most efficient when packets are forwarded greedily as much as possible [87], since greedy forwarding avoids switching to the costly guaranteed-delivery forwarding mode. Also, as the density of nodes in the network increases, the shortest path between pairs of nodes corresponds increasingly to the Euclidean straight line between them. Another observation is that concave "voids" in the routing topology are bad for geographic routing since packets will tend to end up in the concave dead ends. This observation prompted us to explore the hypothesis that we can improve routing performance by choosing virtual coordinates that increase the success rate of greedy forwarding.

This idea is perhaps best illustrated with examples: for the U-shaped network shown in Figure 1-1(a), packets routed greedily between the two ends of the U often end up in a dead end. If the topology of the network is deformed slightly into a "smile" as shown in Figure 1-1(b), then greedy forwarding will work almost all the time. Another example is the cross-shaped network shown in Figure 1-1(c). Again, because of the large hole in the middle of the network, packets forwarded greedily between a random pair of nodes will again often end up in a dead end. While the network shown in Figure 1-1(d) may look quite different in shape from the network in Figure 1-1(c), they are isomorphic, i.e., there is a bijective mapping from the nodes of one network to the other that preserves the connectivity between corresponding nodes. Again, greedy forwarding will work almost all the time for the network in Figure 1-1(d).

Finally, even where we can derive virtual coordinates that allow greedy forwarding to succeed more often, it is not entirely obvious that geographic routing performance will necessarily improve because the increase in the probability of greedy forwarding might have been achieved at the cost of more routing hops. One of the key hypotheses that this dissertation explores is that *virtual coordinates that allow greedy forwarding to succeed more often will indeed improve the routing performance of existing geographic routing algorithms.*

Another argument for using virtual coordinates instead of true physical coordinates is that a change in location does not always have an effect on geographic routing. A geographic routing algorithm needs to react only when there is a change in network connectivity. Nodes that maintain the same connectivity need not change their virtual coordinates simply because their physical locations



(a) U-shaped network (b) "Greedy" isomorph of U-shaped network (c) cross network (d) "Greedy" isomorph of cross network

Figure 1-1: Examples of routing topologies and their "greedy" isomorphs. The points represent the physical and virtual coordinates of nodes, and the lines indicate the connectivity between nodes.

change. As an example, suppose we have a static sensor network with GPS-equipped nodes on board a large ship and the ship sails around the world. While the physical locations of the nodes will change over time according to their GPS readings, it is likely to be more efficient for the nodes to adopt virtual coordinates that are independent of the GPS readings.

1.3 Contributions

The key contributions of this thesis are two algorithms: Greedy Distributed Spanning Tree Routing (GDSTR) and Greedy Embedding Spring Coordinates (GSpring).

GDSTR is a new routing approach that avoids the need for planarization by using trees to route around voids. To aggregate geographic information, we use a new kind of spanning tree, called a hull tree, where each node maintains information about the points that can be reached below it in the tree. Hull trees are much cheaper to build initially and to maintain in a distributed environment than a planar graph. Nevertheless GDSTR performs well and generally outperforms the best existing face routing algorithms.

In addition, we describe two natural extensions to GDSTR – approximate routing and geocast. Approximate routing is a primitive that routes a packet to the node that is closest to a specified destination point and geocast is a primitive that delivers a packet to all the nodes in a specified target region. We show that these extensions are supported in a very natural fashion by hull trees.

GDSTR works well for sparse networks with large voids; face routing algorithms are however able to achieve marginally better routing performance for dense networks with small voids. We address this scenario with GDSTR+, a variant of GDSTR that uses local hull trees to improve routing when the voids are small.

GSpring is a new virtual coordinate assignment algorithm that derives good coordinates for geographic routing of non-location-aware wireless nodes. GSpring incrementally adjusts the routing

coordinates to increase the convexity of voids in the virtual routing topology. GSpring not only allows us to exploit existing geographic routing algorithms when location information is not widely available, it often allows GDSTR to achieve superior stretch even when 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. To the best of our knowledge, GSpring is the first algorithm to derive coordinates that can achieve better geographic routing performance than actual physical coordinates.

1.4 Organization of this Dissertation

The rest of this dissertation is organized as follows.

In Chapter 2, we provide an overview of the related and previous literature on geographic routing, planarization, geocast and virtual coordinates.

In Chapter 3, we describe Greedy Distributed Spanning Tree Routing (GDSTR). We describe hull trees, explain how they are used for routing, and how they are built and maintained. We also describe how hull trees can be used to implement geocast and approximate routing, and GDSTR+, a variant of GDSTR that achieves superior routing performance for dense networks with small voids.

In Chapter 4, we compare the performance of GDSTR routing to existing geographic face routing algorithms and present the experimental results for the costs of GDSTR in terms of both storage and bandwidth. We also evaluate the performance of GDSTR+ and our new hull-tree-based geocast algorithm.

In Chapter 5, we describe Greedy Embedding Spring Coordinates (GSpring), an online virtual coordinate assignment algorithm that incrementally adjust virtual coordinates to increase the convexity of voids in the virtual routing topology.

In Chapter 6, we evaluate the performance of GSpring, by comparing the routing performance of existing geographic face routing algorithms with coordinates obtained with the GSpring algorithm to that with actual physical coordinates and those obtained with the NoGeo algorithm [73].

Chapter 7 summarizes the work in this research and describes open questions and possible future direction for future research that builds upon the work in this dissertation.

In Appendix A, we describe a new convex hull tree maintenance and tree traversal algorithm that presents a node with a view of the locations accessible via each neighboring node. We also present the results of some simulations that support our design choices.

Chapter 2

Related Work

In this chapter, we provide an overview of the related literature on geographic routing. We provide a survey of previous geographic routing and planarization algorithms. Finally, we conclude with a survey of related routing algorithms and also a survey of previous work on geocast.

2.1 Geographic Routing

In this section, we provide an overview of previous location-based routing schemes and describe existing geographic face routing algorithms.

The early proposals for geographic routing, suggested over a decade ago, were simple greedy forwarding schemes that did not guarantee packet delivery in a connected network [17, 28, 82], since packets are not delivered when greedy forwarding causes them to end up at a local minimum; instead, they are dropped at this point.

The first geographic (or geometric) routing algorithm to provide guaranteed delivery was *face routing* [46] (originally called Compass Routing II). Several practical algorithms that are variations of face routing have since been developed, including GFG [7], GPSR [39] and the GOAFR+ family of algorithms [47, 48]. The latest addition to the family is GPVFR, which improves routing efficiency by exploiting local face information [52]. While GOAFR+ is asymptotically optimal and bounds worst-case performance with an expanding ellipse search, GPVFR generally achieves the best average case stretch performance among existing geographic face routing algorithms. There are also other proposals for routing schemes [3, 21, 31, 44, 45] that are loosely based on location but not directly related to these. A survey can be found in [22].

De Couto and Morris explored techniques for dealing with problems with geographic forwarding [12]. In particular, they highlighted that nodes often do not know their locations and proposed *location proxies* as a solution. Location proxies are nodes that know their geographic location and

can hence participate in geographic forwarding. A non-proxy node that does not know its location uses a route discovery mechanism to find a nearby location proxy, and uses that as a forwarding point for its packets. In addition, De Couto and Morris proposed *intermediate node forwarding* as a probabilistic solution for routing around voids. The idea is that when a packet gets trapped in a local minimum, a node will pick random intermediate points through which to forward the packet.

2.1.1 Geographic Face Routing

Here, we present an overview of face routing, which is the basis for all previous work on geographic routing with guaranteed delivery. All of these algorithms require graph planarization, which we describe in Section 2.2. The key insight in face routing is the observation that a planar graph is composed of a set of well defined faces, each of which has nodes as its vertices. Suppose we want to route a packet from a source node s to a destination node t. The imaginary line st will then intersects a fixed number of faces as shown in Figure 2-1. By traversing the network along these faces, one will eventually reach the destination.

Since faces are well-defined at each node, a node needs only to know about its immediate neighbors, and a packet can traverse a face by using a simple *right-hand* rule. What this means is that when a node receives a packet on a given link, the packet is forwarded on the first link that is counterclockwise of the ingress link.

We observe that the line st intersects with a number of edges and it is these edges that are the crossover points from one face to the next. Hence, to detect that a packet has reached the boundary of one face and should cross over to traverse a new face, a node only needs to check if the line st intersects with any of its outgoing edges. When a face change is detected, a packet is forwarded using the right-hand rule with respect to the line st instead of the ingress link. Packets will naturally already contain information about their source and destination, and they only need to store additional information about the first node they traversed on the current face in order to detect cycles and non-delivery.

The difference among existing face routing algorithms lies in the manner in which they route around the planar faces. GPSR uses a deterministic right hand rule when forwarding a packet along a face, i.e., it will always forward a packet clockwise or anti-clockwise around a void.

GOAFR+ picks a random forwarding direction to start with, but instead of forwarding continuously along a face, it keeps track of how far the packet has gone along the face and if a packet seems to have wandered far enough along a face and not made any apparent progress toward the destination, GOAFR+ will backtrack and try the other forwarding direction. By expanding the area of the search incrementally, GOAFR+ ensures that the length of the final path traversed is no longer than a constant multiple of the optimal path.

GPVFR tries to pick the optimal forwarding direction when it switches from greedy forwarding to face traversal by having each node maintain several hops worth of information about its adjacent planar faces.

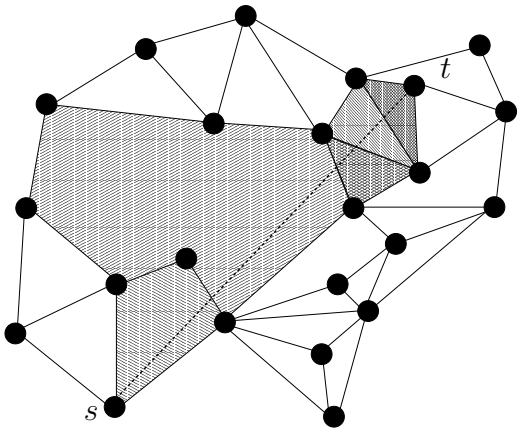


Figure 2-1: Routing from node s to node t with face routing.

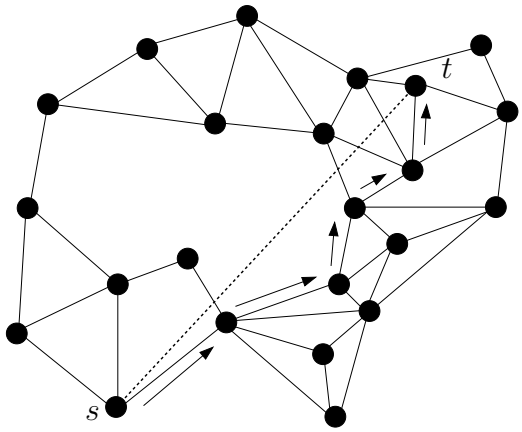


Figure 2-2: An example of the path taken between s and t.

2.1.2 Theoretical Results on Geographic Face Routing

There has been quite a lot of theoretical work on online routing in triangulations [4], convex subdivisions [6], competitive bounds [5] and quasi-planar subdivisions [10].

Bose et al. showed in [4] that a deterministic oblivious (memoryless) routing algorithm exists for arbitrary triangulations, and in [6] that no deterministic oblivious routing algorithm can guarantee packet delivery for convex subdivisions. They also showed that there is no competitive online routing algorithm that is competitive[1] under the Euclidean metric for arbitrary triangulations, and that no competitive online routing algorithm exists under the link distance metric even when the graph is restricted to a Delaunay, greedy or minimum-weight triangulation. Bose et al. later showed [5] that there exists an $O(1)$-memory c-competitive routing strategy for a select class of triangles.

Chávez et al. showed that face routing algorithms will work correctly not only for planar graphs, but also for graphs that are quasi-planar [10].

We had earlier generalized Bose et al.'s result (Theorem 2 of [6]) and showed that no deterministic oblivious routing algorithm exists for planar graphs with only local information [52]. The following theorem is a reformulation of this result:

> **Theorem 1** *For any given non-negative integer h, there does not exist a deterministic oblivious routing algorithm that guarantees packet delivery for all graphs if nodes are limited to knowing only those nodes that are up to h hops away.*

Proof: We construct a set of graphs such that no oblivious algorithm can route correctly in all the graphs in the set, assuming that the nodes have complete local information only up to h hops. In the graphs in Figure 2-3 every box $\Box$ represents an identical chain of h nodes, the other

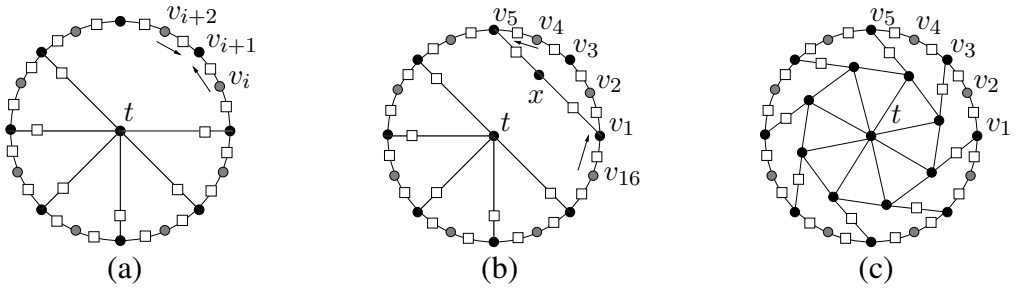


Figure 2-3: Counter-examples showing non-existence of oblivious algorithm with limited information.

16 nodes are located at the vertices of a regular decahexagon, and the destination node t is located at its center.

Suppose, for the sake of contradiction, that there exists an oblivious algorithm A that routes correctly in all graphs, where the nodes have correct local information up to h hops. We claim that, according to algorithm A, the gray nodes in all the 3 graphs in Figure 2-3 behave identically.

If not, then in graph 2-3(a) v_i forwards clockwise and v_{i+2} forwards counter-clockwise and the packet gets trapped in $\{v_i, v_{i+1}, v_{i+2}\}$, since there are no routes from the nodes in the $\square$ nodes to t. Note that all the gray nodes have identical local information up to h hops and are symmetric with respect to the location of t.

Let us assume that all the gray nodes forward packets counter-clockwise. Now, node v_5 cannot forward a packet towards node x in graph 2-3(b), because then, the packet gets caught in the $\{x, v_1, v_{16}, v_1, v_2, \ldots, v_5\}$ circuit. From the point of view of node v_5 the graphs 2-3(b) and 2-3(c) are identical because it has the same local hop information up to h hops. In graph 2-3(c), a packet from node v_5 would never enter the inner octagon, and therefore would never reach t. Similar arguments can be made for the other black nodes by rotating the construction in 2-3(b) and 2-3(c). ■

We conjecture that if nodes have no more than h hops of information, and packets have bounded (finite) memory, then if there are no constraints on the network, it is not possible to guarantee packet delivery with a deterministic algorithm. Planar graphs and trees are highly constrained graphs. It is well known that trees with labeled nodes can be traversed with deterministic termination with a bounded amount of memory.

There is likely to be a relationship between how memory is divided between the nodes and the packets. For example, if all the nodes have $O(n)$ memory, where n is the number of nodes in the network, we can store a full routing table at each node. On the other hand, if a packet is allowed to carry $O(n)$ information, there is a trivial algorithm to fully traverse the network even if there is no memory at the nodes. Existing geographic routing algorithms represent a situation where both the nodes and the packets have memory and hence neither needs $O(n)$ memory to achieve correctness.

[1] A routing algorithm is c-competitive if the path it finds is less than a factor of c times the shortest path.

2.2 Planarization

When Kranakis et al. first proposed Face Routing [46], they only described a routing mechanism and did not propose a method for constructing planar subgraphs from an existing network connectivity graph. Bose et al. proposed using the Gabriel Graph (GG) [7] as the planar subgraph, while Karp and Kung suggested using the Relative Neighborhood Graph (RNG) as well [39]. There are well-established algorithms that allow computing both the Gabriel Graph [20] and Relative Neighborhood Graph [84] in a distributed way when nodes have only local information. The main drawback of these algorithms is that they depend on the Unit Disk Graph assumption, which unfortunately has been shown not to hold practical radio networks. Other distributed algorithms that produced planar spanners on UDG graphs have also been proposed. These include the *Localized Delaunay Graph* [54] and the *Restricted Delaunay Graph* [21].

Kuhn et al. first investigated the planarization of topologies that do not obey the UDG-assumption in [49], and showed that for a class of graphs known as *Quasi-Unit Disk Graphs*, it was possible to augment the graph with *virtual links* to ensure that GG planarization is successful and correct. In their proposed approach, the nominal radio range is normalized to one and links must exist between nodes that are less than distance d apart, $d < 1$. Where links do not exist between nodes that are between distance d and 1 apart, their algorithm replaces the "missing links" with *virtual links*. Their analysis shows that this technique is scalable when $d \geq 1/\sqrt{2}$.

A major breakthrough was made by Kim et al. in developing the Cross-Link Detection Protocol (CLDP) [42], which produces a subgraph on which face-routing-based algorithms are guaranteed to work correctly. Their key insight is that starting from a connected graph, nodes can independently probe each of their links using a *right-hand rule* to determine if the link crosses some other link in the network. CLDP uses a two-phase locking protocol to ensure that no more than one link is removed at any time from any given face; in this way it guarantees that the removal of a crossed link will not disconnect the network. While CLDP is able to planarize an arbitrary graph, every single link in the network has to be probed multiple times and it has a high cost.

Fang et al. proposed an algorithm called *BOUNDHOLE* that sends probe packets to detect voids in the routing topology [15]. Related to this algorithm is Brad Karp's Greedy Perimeter Probing (GPP) [37] which also uses probe packets to explicitly map the planar faces. These algorithms are only tangentially related to CLDP and not particularly useful since they assume that a planar graph exists.

2.3 Other Routing Algorithms for Wireless Networks

In this section, we describe non-location-based routing schemes that have been proposed for wireless networks.

The use of spanning trees for routing in ad hoc networks is not a new idea. Several previous schemes that use trees for routing have previously been proposed, though none of them exploit location information like existing geographic routing algorithms.

There are previous routing algorithms for ad hoc networks that use spanning trees, though none of them leverages location information like GDSTR. Radhakrishnan et al. first proposed the use of a set of distributed spanning trees for routing in ad hoc wireless networks [72]. Their algorithm constructs the spanning trees in an ad hoc manner and messages are delivered using a flooding-based algorithm.

Newsome and Song proposed an approach for routing in sensor networks which embeds a labeled graph in the network topology [61]. They proposed Virtual Polar Coordinate Routing (VPCR), which routes packets on what they refer to as an embedded ringed tree graph. VPCR was evaluated in a regime where the average node degree is about 15, and was found to achieve a stretch (which the authors refer to as *dilation*) of about 1.2. This does not compare favorably with geographic routing algorithms, since in the same regime, geographic routing algorithms are able to achieve unit stretch almost all the time. However, considering that VPCR does not require nodes to have access to location information because it assigns its own virtual polar coordinates, the achieved performance is reasonably good.

Beacon Vector Routing (BVR) [19], GLIDER [16], and HopID [89] 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 virtual coordinate algorithm called VCap that finds three extremal-rooted landmark nodes to generate three dimensional coordinates [9]. This is another variant of the landmark node scheme. Since it uses only 3 landmarks, it performs extremely poorly in sparse networks.

A common application of the spanning tree in the wired domain is the Ethernet spanning tree. The Ethernet spanning tree is not efficient for large networks because packets often have to be routed through the root of the tree. GDSTR does not suffer from the same problem, for several reasons. First it usually forwards packets greedily; the spanning tree is used only to route around voids and GDSTR reverts to greedy forwarding as soon as it is safe to do so. Second, the location information in the tree allows it to route efficiently. Finally, the location information also allows it to avoid routing through the root.

Existing ad hoc routing algorithms that are not position-based (geographic) can be classified as either *proactive*, *reactive* or *hybrid* [57]. Proactive algorithms employ classical routing strategies such as distance-vector routing (e.g. DSDV [69]) or link-state routing (e.g. OLSR [34], TBRPF [64]) and maintain routing information about available paths in the network even if such paths are not currently used. Reactive algorithms on the other hand, maintain only routes that are in

use, and thereby reduce the overhead of maintaining and exchanging routing information in the network (e.g. DSR [35], AODV [68], TORA [66, 67]). Reactive protocols will typically require route discovery and therefore incur some delay before the first packets in a session are exchanged.

Shortest-path algorithms like Distance Vector (DV) [69] and Link State (LS) [34, 64] require each node to have $O(N)$ memory; where N is the number of reachable destinations. On-demand algorithms [35, 66, 68] require a node to have memory at least proportional to the number of its destinations, and often more if aggressive caching is employed [35].

The Zone Routing Protocol (ZRP) [25] is a hybrid algorithm that expands the amount of state stored at a node to a local neighborhood up to a fixed number of hops away. ZRP requires both a route discovery mechanism and query control protocol to work efficiently [26]. Other similar protocols include the limited-radius variant of DSDV [69] and a modified k-hop DSDV variant proposed by De Couto and Morris [12].

2.4 Geocast in wireless ad hoc networks

Geocast was first suggested by Navas and Imielinski [60]. In the geocast model, nodes are each assigned geographic coordinates. Packets have geographic destination addresses represented by a closed polygon and are delivered to all the nodes with coordinates that lie within the polygon. In their work, Navas and Imielinski developed a geocast system that works on Internet hosts equipped with GPS devices.

Ko and Vaidya first investigated the problem of geocast in mobile ad hoc networks [43] and proposed the use of a "forwarding zone" to decrease the delivery overhead of geocast packets. Like GDSTR+, GeoGrid [56] partitions the geographic area into a logical 2D grid.

Huang et al. proposed a spatiotemporal version of geocast, called mobicast, for sensor networks as a new communication abstraction [29]. These approaches are similar in that they are flooding-based approaches. Flooding is likely to be costly in dense networks. Huang et al. subsequently proposed a variant of their mobicast algorithm that is based on face routing [30]. It exploits the properties of planar graphs to provide delivery guarantees and reduce overhead. Since distributed planarization is costly [42, 51], their approach is likely to be costly even though the message overhead of the geocast protocol may be low. Mobicast is not directly comparable to geocast because it involves a temporal component.

2.5 Virtual Coordinates

Rao et al. had earlier proposed the NoGeo family of coordinate assignment algorithms for ad hoc wireless networks [73]. 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.

Although NoGeo has been shown to work well for dense networks, it does not work as well for sparse networks and for networks with many obstacles. Also, NoGeo requires global coordination to determine and compute coordinates for the perimeter nodes. This makes the algorithm complicated to deploy and debug for large systems: for example, care must be taken to ensure that each perimeter node obtains the full set of vectors to allow them to compute mutually consistent starting positions. New nodes that join the system at physical locations outside the initial perimeter of the system will also tend to get "flipped in". In contrast, GSpring is a fully online algorithm and amenable to the incremental addition of nodes without any need for periodic system reset or any global coordination.

Papadimitriou and Ratajczak conjectured that any planar 3-connected graph can be embedded in the plane in such a way that for any nodes s and t, there is a path from s to t such that the Euclidean distance to t decreases monotonically along the path. A consequence of this conjecture is that in any ad hoc network containing such a graph as a spanning subgraph, two-dimensional virtual coordinates for the nodes can be found for which the method of purely greedy geographic routing is guaranteed to work [65]. While only tangentially related to GSpring, Theorem 2 of [65] inspired the early ideas for GSpring.

Fang et al. proposed a local rule called the *TENT* rule to detect whether a node can be a dead-end for a packet in unit disk graph networks [15]. In the TENT rule, all the 1-hop neighbors of a node n are ordered counter-clockwise. For each pair of adjacent nodes u and v, a pair of bisectors are drawn of nu and nv, which intersect at a point p. The claim is that if p lies beyond the radius of communication for n, then n is a node at which a packet could possibly get *stuck*. n is referred to as a *strong stuck node*.

2.5.1 Node Localization

There is a large body of work on the closely-related *node localization problem* for ad hoc wireless networks [8, 13, 58, 63, 70, 76–78]. In this problem, 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. Although the node localization problem is significantly more stringent than the problem that GSpring attempts to solve, solutions to this problem are of interest because the methods and algorithms employed are also applicable for deriving virtual coordinates.

Solutions to this problem can be characterized according to (i) whether some location-aware anchor nodes are available, and (ii) whether the schemes are incremental or concurrent, as follows:

- **Anchor-based algorithms**. These algorithms require the availability of a minimum number or fraction of location-aware nodes in the network. It turns out that a large number of anchor nodes are usually required for such algorithms to derive coordinates with acceptable errors [8, 13, 63, 76–78, 78].

- **Anchor-free algorithms**. These algorithms use only local distance information to derive coordinates when nodes have no pre-configured location information [58, 70, 76].

- **Incremental algorithms**. These algorithms start with a small number of nodes that have assigned coordinates, and nodes are incrementally added to this set by computing their coordinates from the previously computed coordinates, e.g., ABC [76].

- **Concurrent algorithms**. These algorithms calculate and compute the coordinates for all nodes in parallel [8], e.g., Terrain [76] and Hop-Terrain [77].

In general, node localization has been found to be difficult under three scenarios: (i) anchor nodes are closely co-located or too few in number (for anchor-based algorithms. This is also known as the *sparse anchor node problem*), (ii) sparseness in the network (i.e., nodes are relatively far apart and have few neighboring nodes), and (iii) errors in the locations of the anchor nodes or measurement of inter-nodal distances. These scenarios also adversely affect the performance of GSpring to some extent. The following is a brief survey of some existing node localization algorithms.

Nagpal et al. proposed an anchor-based algorithm for localization that works in a dense network with average node degree 15 or greater [59]. A number of *seed* nodes that know their locations are scattered in the network and the non-location-aware nodes attempt to estimate their locations by minimizing the errors based on their hop counts to these seed nodes using gradient descent.

Doherty et al. proposed an anchor-based algorithm for localization using only connectivity constraints among beacons [13]. Their approach achieves good error rates only when there is a relatively large number of anchor nodes (more than 40) and when the network density is high.

Bulusu et al. proposed an anchor-based scheme that uses the radio connectivity of a node to a square grid of location-aware anchor nodes to determine its coordinates [8]. The coordinates of non-anchor nodes are calculated as the centroid of all the anchor nodes that are within radio range. Their algorithm is a concurrent algorithm that can achieve about a 12% localization error with 12 anchor nodes per non-anchor node, which is a significant number and practically infeasible.

Savarese et al. proposed an incremental anchor-free algorithm called ABC [76]. ABC selects three in-range nodes and assigns them coordinates to satisfy the inter-node distances (which are measured a priori). The algorithm then incrementally calculates the coordinates of the remaining nodes with the already calculated coordinates. The authors report that with a range error of 5% that they are able to obtain an average position error of 60%.

Savarese et al. also proposed an anchor-based algorithm called Terrain [76]. The Terrain algorithm starts with the ABC algorithm, but instead of performing incremental computation, their algorithm

performs a concurrent optimization using the distances to the anchors and also the anchors' coordinates. The authors report that they were able to achieve 25% average position error with a range error of 5%.

Savarese et al. subsequently proposed a two-phase Variant of their Terrain algorithm in [77]. In the first stage, the algorithm uses a variant of the Terrain algorithm called Terrain-hop that is robust against ranging errors. In the second phase, the algorithm performs a simulated-annealing-based simulation and they are able to achieve a 12% average position error with a range error of 5%.

Savvides et al. developed a system called AHLoS (Ad-Hoc Localization System) that uses an two-phase anchor-based localization algorithm [78]. During the first estimation phase, nodes without location information will use ranging information and known beacon node locations in their neighborhood to estimate their locations. Node distances are estimated from the received signal strength and coordinates are obtained by solving a set of over-constrained equations. Once a node has initialized its location, it will assist other nodes by propagating its location estimate through the network. Their algorithm is able to produce position errors within 20 cm of the actual position when the ranging error is small (about 2 cm). However, about 10% of the nodes in the network are required to be anchor nodes.

Niculescu et al. proposed an anchor-based distributed algorithm that uses angle-of-arrival for localization [63]. In their algorithm, nodes iteratively obtain position and orientation information starting from the anchor (landmark) nodes. A potential problem with their approach is that the angle of arrival tends to be difficult to measure in practical networks.

Priyantha et al. proposed an anchor-free algorithm where nodes start from a random initial coordinate assignment and converge to a consistent solution using only local node interactions [70]. They estimate the network's global layout using communication hops and subsequently a force-based relaxation to optimize this layout.

Moore et al. proposed an anchor-free localization algorithm that uses the notion of *robust quadrilaterals* to avoid flip ambiguities that may corrupt localization computations [58]. Like AFL, their algorithm requires that nodes are able to estimate their distances from their neighbors.

2.5.2 Other Virtual Coordinate Algorithms

Virtual coordinate assignment schemes have also previously been proposed for Internet applications, with a view to using the coordinates for estimating Internet roundtrip times (RTTs). GNP [62] 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 [80] and Vivaldi [11] are two schemes that also derive coordinates by simulating physical systems. The former simulates particles moving in a force field with friction, while the latter is similar to GSpring and simulates a physical system of springs.

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 [61]. VPCR works relatively well, but it can incur significant overheads under node and network dynamics. Others include Beacon Vector Routing (BVR) [19], GLIDER [16], HopID [89] and VCap [9], as described above.

Chapter 3

Greedy Distributed Spanning Tree Routing (GDSTR)

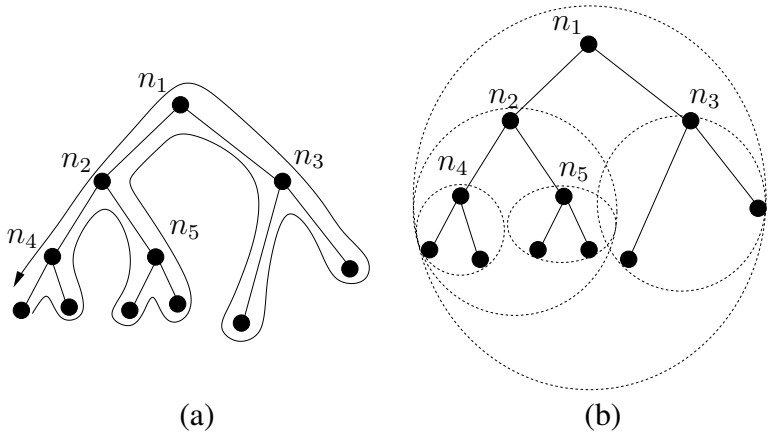
In this chapter, we describe Greedy Distributed Spanning Tree Routing (GDSTR). We describe hull trees, explain how they are used for routing, and how they are built and maintained. We also describe how hull trees can be used to implement geocast and approximate routing and GDSTR+, a variant of GDSTR that achieves superior routing performance for dense networks with small voids.

3.1 Overview

Like previous geographic face routing algorithms, GDSTR forwards packets using simple greedy forwarding whenever possible. It switches to forwarding on a spanning tree only to route packets around "voids," and escape from a local minimum. It switches back to greedy forwarding as soon as it is feasible to do so.

The reason GDSTR is able to guarantee the delivery of packets in a connected network is that the tree traversal forwarding mode is guaranteed to deliver the packet to any node in the network without greedy forwarding. In other words, even though greedy forwarding tends to be the more common forwarding mode in practice, we can think of the tree traversal forwarding mode as the basic routing algorithm and greedy forwarding as a best effort first try because it tends to be more efficient, if it works [87].

It is well-known that, given a spanning tree that contains all n nodes in a network, we can successfully deliver a packet to any node in the network by traversing the tree in a manner similar to a depth-first search as shown in Figure 3-1(a). This traversal requires no state to be stored in the packet and guarantees that a packet will be delivered in no more than $2n - 3$ hops. If the specified



(a) (b)

Figure 3-1: Example of a spanning tree and a *hull tree*. Although convex hulls are polygons, for simplicity they are represented with ellipses in Figure 3-1(b).

destination is not found in the tree, then we can terminate the traversal in exactly $2n - 2$ hops if we store information about the starting node in the packet.

A major contribution of our work is the definition of a new distributed data structure, an augmented spanning tree that we call a *hull tree*, that allows us to restrict the above search problem to a small subtree of the full spanning tree for a given destination, thereby guaranteeing packet delivery in much fewer than $2n - 3$ hops.

3.1.1 Hull Trees

A hull tree is a spanning tree where each node has an associated *convex hull* that contains the locations of all its descendant nodes. Hull trees provide a way of aggregating location information and they are built by aggregating convex hull information up the tree. This information is used in routing to avoid paths that are not productive; instead we traverse a significantly reduced subtree, consisting of only the nodes with convex hulls containing the destination point. An example of a hull tree corresponding to the spanning tree shown in Figure 3-1(a) is illustrated in Figure 3-1(b).

Each node in a basic hull tree stores information about the convex hulls that contain the coordinates of all the nodes in subtrees associated with each of its child nodes. The convex hull information is aggregated up the tree. Each node computes its convex hull from the union of its coordinate and the points on the convex hulls of all its child nodes, and this information is communicated to the parent node. Consequently, the convex hull associated with the root node is the convex hull of the entire network and contains all the nodes in the network.

The convex hull for a set of points is the minimal convex polygon that contains all the points; it is minimal because the convex hull will be contained in any convex polygon that contains the given points. The hull is represented as a set of points (its vertices), and this set could be arbitrarily

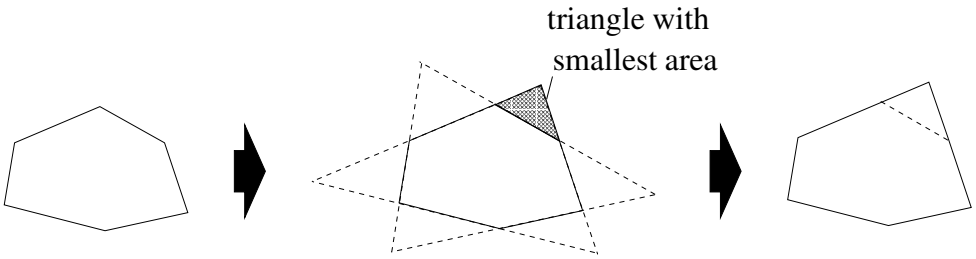


Figure 3-2: Procedure to reduce the size of a convex hull.

large. To ensure that the convex hulls use only $O(1)$ storage instead of $O(n)$ storage, where n is the network size, we can limit the number of vertices for a convex hull to a maximum of r points. To reduce a convex hull with s vertices to a smaller one with $s - 1$ points, we can project the boundary lines to form an adjacent triangle at every face. We pick the smallest triangle in this set of s triangles and add that triangle to the hull as illustrated in Figure 3-2.

Limiting the number of points on the convex hulls allows us to save storage, but the resulting hulls will be larger and this increases the probability that the hulls of two siblings nodes in a tree will intersect. Intersections between convex hulls are undesirable because they introduce ambiguity in the routing process and make it less efficient. However, our experiments (described in Section 4.2.1) show that routing behavior is not affected by using as few as 5 points to represent a hull.

3.1.2 Routing with Hull Trees

To route packets on a hull tree, we forward a packet to child nodes that have a convex hull containing the destination. If none of the child nodes have convex hulls containing the destination, we know that the destination is not reachable down the tree, so we forward the packet up the tree. Figure 3-3(a) shows what happens when node n_3 sends a packet to node n_5: since n_5 is not in n_3's convex hull, the packet will be forwarded up the tree to n_1, and from there to n_2, since its convex hull contains the destination.

One minor complication that can arise is that the destination may lie in the intersection of the convex hulls of two child nodes. In this case, we can still guarantee packet delivery by systematically searching all the subtrees that have convex hulls containing the destination. This situation is illustrated in Figure 3-3(b). In this example, node n_6 sends a packet to n_5. The packet is first forwarded up the tree to the root n_1 since the convex hull of n_5 and n_3 do not contain n_5. At n_1, there are two child nodes n_2 and n_4 that have convex hulls containing n_5. Node n_1 does not have sufficient information to decide which child node is the better choice, so it sorts n_2 and n_4 in some arbitrary order, and forwards the packet to the first node in the sequence (which turns out to be n_4 in this example). When n_4 receives the packet, it realizes that the convex hulls of its child nodes do not contain n_5, and so the packet is returned to n_1. Since n_1 receives the packet from n_4, n_1 concludes that the destination cannot lie in the subtree for n_4 and tries the next possible option and forwards the packet to n_2. This time, the packet is successfully forwarded down the tree to n_5.

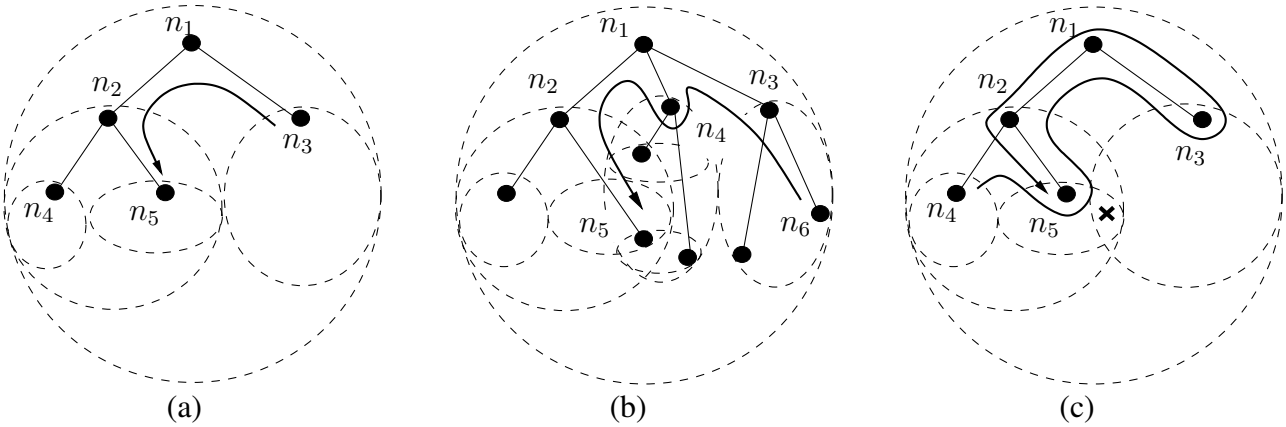


Figure 3-3: Routing over sample hull trees. Some child nodes are omitted to avoid clutter. The destination of the forwarded packet is marked with a cross (applicable to (c) only).

If it turns out that the destination specified by a packet does not correspond to a node in the network, the above traversal process would not terminate. However, by recording the node at which we start the tree traversal in the packet, we can conclude that a packet is undeliverable when we come back to that node. This termination condition is analogous to that used to terminate traversal of planar faces by existing geographic face routing algorithms.

Figure 3-3(c) illustrates an example involving an undeliverable packet. Suppose node n_4 sends a message to an unreachable destination x, and initially this packet is routed greedily to n_2, and then to n_5, which is a local minimum. At this point, n_5 records itself in the packet and switches to routing in tree forwarding mode. The packet is forwarded on the subtree consisting of the nodes with hulls that contain the destination (which in our example are the nodes n_1, n_2, n_3 and n_5). The packet is first sent to the parent node n_2 and from there to n_1. The destination is contained in the convex hulls of both of n_1's child nodes, but since the packet was received from n_2, it is forwarded to n_3. After forwarding over subtrees of n_3 (not shown on the diagram), the packet is returned to n_1, which forwards it to n_2, its first child whose convex hull contains the packet. n_2 forwards the packet to n_5. At this point, n_5 sees that it is the originator of the tree traversal and hence concludes that the packet is undeliverable.

In the above discussion, it should be clear that the convexity of the hulls is not a requirement for correctness. In fact, the key purpose of the hulls in the hull tree is to maintain a summary of the area that contains the points in each subtree. The reason we choose to use convex hulls is that it turns out that the convex hull is a relatively compact representation for a set of points and it is relatively easy to perform mathematical computations like the checking of containment of points and finding intersections. It is plausible to use non-convex hulls or other shapes like circles and ellipses to summarize the regions covered by each subtree.

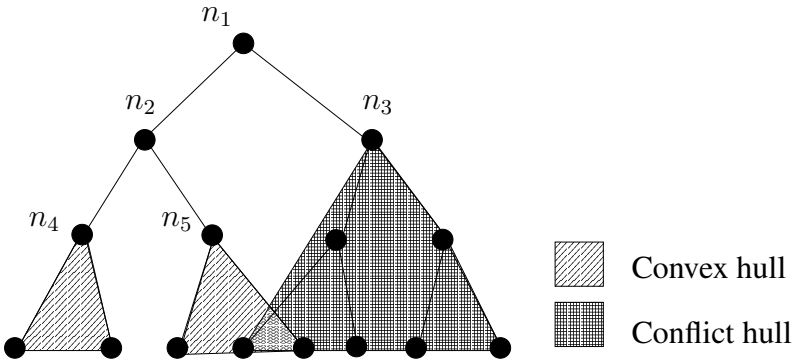


Figure 3-4: An example showing the child convex hulls and conflict hull for node n_2.

3.2 Conflict Hulls: Optimization for Undeliverable Packets

When performing tree traversal on the basic hull tree described in Section 3.1.1, the search subtree always includes the root node. Even after a node has exhaustively searched all its child subtrees, it still does not know whether there exists a convex hull in a distant branch of the tree that contains the destination. It therefore has no choice but to forward the packet to its parent node. This process can only stop at the root because the root node is the only node in the system that has sufficient (global) information to make an *authoritative* decision.

An obvious way to remedy this situation is to propagate information from the root down the tree so that each node can make *authoritative* decisions. We achieve this by having each node maintain information about the set of convex hulls $\mathcal{H}$ that intersect with its own convex hull. We refer to these hulls as *conflict hulls*.

More precisely, each node stores conflict hulls for nodes with which it shares a common ancestor, where that node is immediately below the common ancestor, and has a convex hull that intersects with its hull. We illustrate this with an example in Figure 3-4. In addition to the convex hulls of its child nodes n_4 and n_5, node n_2 will record the convex hull of n_3 as its conflict hull. Similarly, node n_5 will also record the convex hull of n_3 as its conflict hull.

The conflict hull information is propagated down the tree as follows: an intermediate node (or node that is not a leaf node) broadcasts the convex hulls of its child nodes and its own set of conflict hulls. The child nodes will record the set of convex hulls of its siblings and the conflict hulls of its parent that intersect with its own convex hull as its set of conflict hulls. The conflict hulls of its parent that do not intersect with its own convex hull are ignored. This is a somewhat coarse-grained scheme; a more specific scheme would be to check if the hulls broadcast by the parent intersect with any of the convex hulls of the child nodes and store only the ones that intersect. It turns out that just considering the associated hull for a node works well in practice and we can avoid additional computations.

These conflict hulls are helpful for the following reason: in the basic hull tree, nodes are not authoritative over their convex hulls because it is possible for a point within its convex hull to be

reachable through a distant branch of the tree. With conflict hulls, nodes are now able to determine the set of points that are possibly reachable through a distant branch of the tree at a finer granularity. In particular, a node is now completely authoritative over the set of points in its convex hull that do not lie in the intersections between its convex hull and its conflict hulls. Ambiguity remains only for the points that lie within the intersections between its convex hull and the conflict hulls.

With this additional information, a node that receives a packet from its last child during tree traversal will check if any of its conflict hulls contain the destination. If not, it will forward the packet to its first child instead of the parent. Effectively, convex hulls allow us to prune search paths down the routing subtree during tree traversal and the conflict hulls allow us to prune some paths up the tree.

Impact on Storage. With the basic hull tree, we can ensure that the each node requires only $O(1)$ storage, relative to total network size. The number of convex hulls stored at each node is no more than the number of neighbors, and the size of each individual hull can be kept bounded as described in Section 3.1.1. By augmenting the hull tree with conflict hulls, we can no longer bound the total storage requirement on each node. In the worst case, it is possible that each node may have to maintain $O(N)$ state for the conflict hulls.

In the same way that the storage requirement for convex hulls can be limited by using convex hulls with fewer points, the storage requirement for conflict hulls can be reduced by storing a conflict hull that is the union of all the conflict hulls, when there is more than one conflict hull. As before, there is a possible tradeoff in routing efficiency. With a larger conflict hull, the intersection between the conflict hull and a node's convex hull will likely be larger and hence the set of a node's authoritative set is reduced accordingly. Another possibility is to store only the intersection between the conflict hull and the local convex hull, if it saves storage. This will however require additional computation.

The number of conflict hulls is dependent on the topology of the network and on the chosen tree building algorithm. In practice, the number of conflict hulls generated by our chosen tree building algorithm is usually very small (no more than 2 or 3) and the additional storage requirement is modest. This is because of the way that we build the tree.

3.3 Multiple Hull Trees: Offering More Routing Choices

Kuhn et al. have shown that in evaluating the performance of geographic routing algorithms, it is critical to study their routing performance for random topologies with average node degrees in the region between 4 and 8 [47] (also called the *critical region*) because geographic routing algorithms are almost always uniformly good for both very sparse and very dense random networks.

The main factor that affects routing performance for topologies in the critical region is the voids, and the key difference between existing geographic routing algorithms is the manner in which

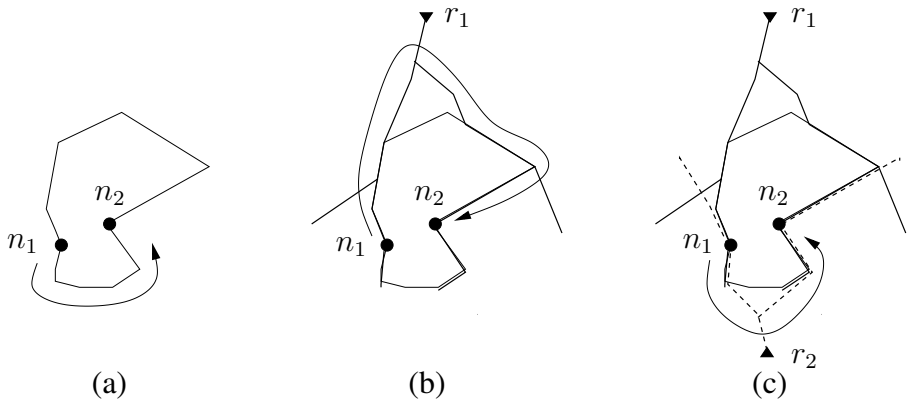


Figure 3-5: An example illustrating the benefits of two trees when traversing a void.

they route around these voids. There are often two forwarding choices around a void (clockwise and counterclockwise), both correct. The key difference is that one path is longer than the other. Choosing the longer path will hurt routing performance.

Figure 3-5 illustrates the last point. Suppose that node n_1 in Figure 3-5(a) needs to route a packet around a void to n_2. A face routing algorithm is likely to have a face exactly corresponding to the void, and must choose between routing clockwise or counterclockwise. While the forwarding direction does not affect correctness, it can be costly when a bad choice is made. In this example, the optimal choice is counterclockwise. While having some local face information (information about a bounded number of hops along each face) allows a face routing algorithm to pick the optimal direction fairly often [52], it is not possible to guarantee that the optimal routing direction will be picked when the void is large and a node only has access to the topology information of its immediate vicinity in the network.

Figure 3-5(b) shows that if we only have one hull tree rooted at r_1, n_1 would be forced to route clockwise. However, if we have two hull trees as shown in Figure 3-5(c), and the other tree is rooted at r_2, at the opposite end of the network, n_1 is presented with the other choice as well. This example demonstrates how two trees rooted at opposite ends of a network can effectively "approximate" a planar face and offer approximately the same choices as that available to some face routing algorithms.

While it is true that having more hull trees provides more options and will not hurt routing performance as long as we are able to choose wisely, there is a cost associated with maintaining more trees. It turns out that with two extremal-rooted trees, GDSTR is usually presented with both routing choices around voids in the network. Hence, GDSTR maintains two hull trees. This decision is validated with our simulations that demonstrate that we can achieve only a marginal improvement in routing performance with more than two hull trees.

If any of the convex hulls contain the destination point, a node is able to determine the correct forwarding direction from this hull. Otherwise, the more favorable forwarding direction around a void can be determined by choosing the tree with a root that is nearest to the destination. The

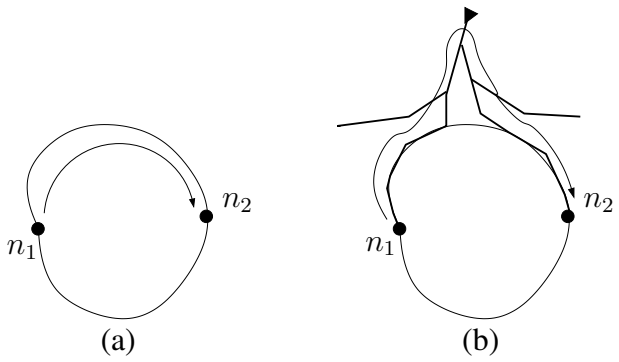


Figure 3-6: An example illustrating why GDSTR imposes an overhead compared to face routing.

reason this works is that when none of the trees have convex hulls that contain the destination, the packet will be forwarded up the tree. Choosing a tree with a root that is nearer to the destination will often cause the packet to be forwarded in the correct general direction of the destination most efficiently.

In contrast, face routing algorithms only maintain information about the nodes within a small localized vicinity and this is often insufficient to pick the correct forwarding direction in sparse networks with large voids. For this reason, GDSTR is expected to achieve superior routing performance in such networks.

For dense networks with small voids, it generally does not matter which forwarding direction is picked. However, it turns out that tree routing does not traverse a void quite as efficiently as face routing and incurs a slightly higher overhead. This is illustrated in Fig. 3-6. Suppose that node n_1 in Fig. 3-6(a) needs to route a packet around a void to n_2. A face routing algorithm is likely to have a face exactly corresponding to the void as shown in Fig. 3-6(a). On the other hand, because a tree is unable to "approximate" the void exactly, routing around the void using the tree will require the packet to take a short detour and incur some additional routing overhead as shown in Fig. 3-6(b). For this reason, GDSTR is expected to perform marginally worse than face routing in dense networks with small voids. We present a solution to address this problem in Section 3.8.

Finally, using a single tree as the basis of routing is inherently fragile. If the root node fails, the entire tree may collapse and have to be rebuilt, and while this is happening, routing will not work well. Hence, in addition to offering additional routing choices, maintaining multiple trees also provides some degree of resilience to such network changes.

3.4 Building & Maintaining Good Hull Trees

In previous sections, we discussed the key ideas and insights for GDSTR. In this section, we will discuss how to build hull trees that yield good routing performance. Hull trees are built in

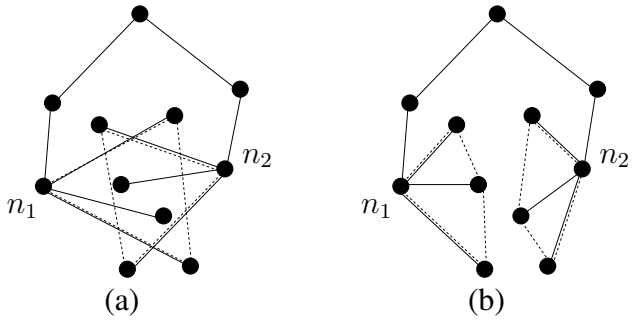


Figure 3-7: Examples of "bad" and "good" trees.

two steps. First, we construct a basic spanning tree and then, the convex and conflict hulls are determined from the topology of the spanning tree.

GDSTR will work correctly with any distributed spanning tree. However, routing performance will be best if there is minimal overlap among the convex hulls of different tree branches. Intuitively, we want trees that are geographically "compact" in order to avoid intersecting convex hulls. Figure 3-7 illustrates this idea. While the coordinates of the nodes in both Figures 3-7(a) and 3-7(b) are the same, the tree configuration in Figure 3-7(a) creates an undesirable intersection in the hulls for nodes n_1 and n_2. From these examples, it is clear that we want to build trees that cluster nearby nodes in the same subtree.

In addition, we want a hull tree that will conform closely to the shape of the voids in the network, i.e., "wrap" around the voids, so that we can route around voids in the smallest number of hops. Also, when we have two hull trees, we want the two hull trees to provide complementary paths around each void. We found that we can achieve the former by choosing the right spanning tree algorithm and the latter by rooting the hull trees at the extreme ends of the network.

3.4.1 Building the Spanning Tree

After evaluating a number of spanning tree algorithms, we found that the *minimal-depth tree* yields the best routing performance. The *minimal-depth tree* is constructed by having each node choose the neighbor with the minimal number of hops to the root as its parent. When a node has a choice between multiple neighboring nodes that are the same number of hops from the root, the nearest node is chosen preferentially. The details and simulation results for the other spanning tree algorithms are contained in Appendix A.

With hindsight, it is not surprising that the *minimal-depth tree* yields the best routing performance. Firstly, the minimal-depth tree tends to choose the shorter links preferentially. If we refer to the example in Figure 3-7, it should be clear that shorter links reduce the occurrences of "crossing" links. This results in trees with subtrees that are more clustered together, thereby reducing the probability of intersections between convex hulls, and generating fewer conflict hulls.

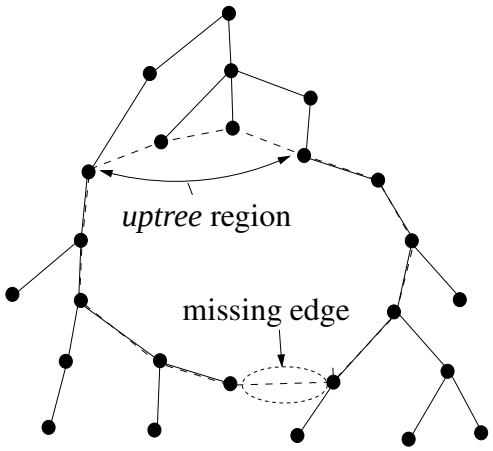


Figure 3-8: An example of the interaction between min-depth tree and routing void. The dashed lines indicate the shape of the void.

In Figure 3-8, we see an example of how the min-depth tree interacts with a void in the routing topology. We call the region of the void nearest of the root of the tree, the *uptree* region. In this region, the edges of the void are not edges in the hull tree and hence routing around this region involves routing up and down the tree. For the remaining parts of the void, all the edges (except one) are part of the min-depth tree. Because the min-depth tree yields the shortest paths between each node and all of its ancestor nodes, tree traversal is efficient along these regions.

Because spanning trees do not contain cycles and yet must contain all nodes in the network, we know that there is exactly one edge on the far side of the void from the root that is not an edge in the tree. We call this the *missing link*. This missing edge restricts packet forwarding along the void using this hull tree to only one direction. Given two hull trees, we observe that as long as the two trees do not share the same missing link, they will together allow the void to be traversed in both directions (clockwise and counterclockwise) around the void.

To increase the probability of generating hull trees that cover disjoint regions of the void, our approach is to set their roots at opposite ends of the network. In particular, the nodes with the minimal and maximal x coordinates are chosen as the roots of the two spanning trees. It is easy to visualize why two trees that are rooted at the extreme x coordinates will tend to meet each void in the network at different sides. For most topologies, doing so will cause the two trees to "approach" each void from opposite directions and it becomes highly improbable for them to share the same missing edge.

To summarize, the choice of the min-depth tree as the underlying spanning tree algorithm allows us to build hull trees that have fewer intersecting convex hull. The rooting of the spanning trees at opposite ends of the network ensures that generally the two trees will allow voids to be traversed in both directions.

The spanning tree algorithm makes few assumptions about radio behavior. The only requirement is that nodes must agree on whether they are neighbors. GDSTR is also robust against location

errors [79], because if a node has a wrong location, the hulls in its part of the hull trees will grow to include the node's wrong location. When greedy routing to that node hits a dead end, the tree traversal will eventually route to the tree branch that includes the node because of the large hull.

3.4.2 Building Hull Trees

Each node broadcasts a *keepalive* message periodically to inform its neighbors of its location. In each message, the node includes its view of the root of each tree and its distance in hop count from each root. Through these exchanges, all the nodes will eventually come to a consensus as to which nodes should be the roots; each node will also know its hop count from the root. The inter-beacon transmission interval is jittered by up to 20% to avoid synchronization between the transmissions of neighboring nodes [18]. Given a mean inter-beacon interval T, the actual inter-beacon transmission intervals are uniformly distributed in $[0.8T, 1.2T]$.

Once the tree has been formed, each node broadcasts its chosen parent node as well as its convex hull. To compute its convex hull, a node determines the minimal convex hull that contains the union of the convex hulls of its children in that tree and its own coordinate. The convex hull information is aggregated up the tree.

The convex hull for a set of points can be computed in $O(n \log n)$ operations using the Graham's Scan algorithm [23]. As the number of points in the set increases, it achieves the optimal asymptotic efficiency of $O(n \log n)$ time.

Algorithm 1 (Graham's Scan) *Given a set of points $\mathcal{S}$ and an empty stack $\mathcal{W}$.*

1. *Select the rightmost lowest point p_0 in $\mathcal{S}$. Set $p[0] := p_0$ and $i := 2$.*
2. *Sort $\mathcal{S}$ according to the angle that each point makes with the horizontal line through p_0, with p_0 as a center. For ties, discard the points that are closer to p_0.*
3. *Let $p[\cdot]$ be the sorted array of points. Push $p[0]$ and $p[1]$ onto stack $\mathcal{W}$.*

```
while i < N {
    Let p_1 be the first point on the top of stack W.
    Let p_2 be the second point on the top of stack W.
    if (p[i] is strictly left of the line p_2 to p_1) {
        Push p[i] onto W.
        increment i
    } else {
        Pop the top point p_1 off the stack W
    }
}
```

The points contained in the stack $\mathcal{W}$ form the convex hull of $\mathcal{S}$.

Once the root acquires hulls from all its children, each node determines the set of conflict hulls $\mathcal{H}$ and adds this information to its *keepalive* messages. Information about the conflict hulls is propagated down the tree starting at the root; each node in turn informs its children about intersections between their hulls and other known hulls. Once the information about the conflict hulls has propagated down to the leaves of the tree, the tree is fully built and consistent. This algorithm (like other tree building algorithms) takes at most $3D$ rounds of message exchanges to complete, where D is the diameter of the network graph.

3.4.3 Maintaining & Repairing Hull Trees

We use the same algorithm to repair a tree when nodes fail. If the mean inter-message interval is T seconds, even in the worst case where the root of a tree fails, a hull tree can be restored within $3TD$ seconds. To speed up tree repair and recovery, we can trigger immediate transmissions in place of regular messages when a node failure is detected.

A node concludes that a neighbor has failed when it does not hear from it after a pre-determined multiple of the *keepalive* message interval. If the failed node is a child, a node will reduce and update its convex hull; if the failed node is a parent, a node will choose a new parent. In either case, it sends the new information in its next *keepalive* message. When the (new or old) parent hears about the changes, it will update its state accordingly.

Hence, it is straightforward to update the routing state when anything changes in the system. When a node hears the *keepalive* message from a neighbor, it updates its own state and the information that it broadcasts in its subsequent *keepalive* message. If nothing changes, a node does not need to update anything.

In fact, a node only has to broadcast its hull tree information when there are changes to the state of its hull trees. If nothing changes after the same hull tree information has been sent for several rounds, subsequent *keepalive* messages will contain only the node's identifier and location. When there is a change in its hull tree information, a node resumes broadcasting its hull tree information for another few rounds.

Our approach is simple and robust. However, it is perhaps instructive to consider the minimal number of broadcasts that are required to build a hull tree if we could synchronize the communication of the nodes perfectly. To broadcast the identity of a root node and its associated hop count, each node requires one broadcast (if they do it in exactly the right order). Similarly, one broadcast is required to aggregate the convex hull information from the leaf nodes and for the root to broadcast conflict hull information down to the leaf nodes. The minimal total number of broadcasts is $3n$, where n is the total number of nodes and we would expect that a reasonable implementation of the hull tree building algorithm to require $O(n)$ messages.

3.5 GDSTR: Geographic Routing with Hull Trees

In this section, we describe the details of the GDSTR routing algorithm.

Since there are two hull trees, a tree must be chosen when a packet switches from greedy forwarding to tree forwarding mode. Also, there are different heuristics that can be employed with regard to the ordering of neighbors/child nodes during tree traversal and the choice of hull tree when some hull tree contains the destination and when none of them contains the destination. A study of how these heuristics affect routing performance is presented in Appendix A.

In summary, we found that the following simple heuristic works best for choosing a hull tree when we switch from greedy forwarding to tree traversal:

1. If some child node has a convex hull containing the destination node, pick any tree from the set of trees with such nodes.
2. Otherwise, if none of the child nodes (in any tree) have convex hulls that contain the destination node, pick the tree with the root that is nearest to the destination.

The following is a precise description of GDSTR that incorporates the use of multiple trees and the set of conflict hulls $\mathcal{H}$. A GDSTR data packet p is tagged with the following state components:

- $mode$: current forwarding mode (Greedy/Tree),
- n_{min}: node visited that is nearest to destination,
- $tree$: identifier for chosen forwarding tree,
- n_{anchor}: tree traversal anchor node.

n_{min} is the node at which a packet switches from greedy forwarding to tree traversal. It is used to determine when routing should revert to greedy forwarding. n_{anchor} is used to mark the termination point for tree traversal. It is set when a packet first changes to Tree Traversal mode and updated when a packet is forwarded down a subtree for which forwarding up to the parent would not be productive. While n_{min} is often the same as n_{anchor}, they are occasionally not the same node. n_{min} is used by a node to determine whether is it safe for a packet to revert to greedy forwarding mode and n_{anchor} is used to determine if the packet is undeliverable during tree traversal. In particular, when a packet is forwarded to a child node that does not have a conflict hull containing the destination, the child node is set as the new anchor node n_{anchor}.

Algorithm 2 (GDSTR) *When a node v receives packet p for destination node t from a neighboring node u, do:*

1. **Preliminary Checks:**
   (a) **Packet Delivery:** *If $v = t$, the packet has been delivered.*
   (b) **Check for switch to Greedy mode:** *If $p.mode \neq$ Greedy and there is at least one immediate neighbor w such that $|wt| < |(p.n_{min})t|$, then set $p.mode :=$ Greedy, $p.n_{min} := w$ and clear $p.n_{anchor}$ and $p.tree$ if they are set. Execute step 2 or 3 according to $p.mode$.*
2. **Greedy Mode:** *Find the node w in the set of immediate neighbors that is closest to the destination t.*
   (a) **Greedy Forwarding:** *If $|wt| < |vt|$, set $p.n_{min} := w$ and forward the packet to w.*
   (b) **Switch to Tree Traversal Mode:** *Choose one of the hull trees for forwarding and set $p.tree$ to the chosen tree's identifier, $p.mode :=$ Tree and $p.n_{anchor} := v$. Then, find the set of child nodes with convex hulls that contain the destination t.*
      - *If set is non-empty, arrange the child nodes (relative to $p.tree$) with convex hulls that contain the destination point in an ascending sequence according to the global ordering of node identifiers and forward the packet to the first such child node.*
      - *Else, forward the packet to the parent node in $p.tree$.*
3. **Tree Mode:** *If the root for $p.tree$ has changed, follow step 2(b), else follow step 3(a).*
   (a) **Check Termination Condition:** *Conclude that packet is* **not deliverable** *if $v = p.anchor$ and one of the following conditions holds:*
      (i) u is the last child and v is the root node for $p.tree$;
      (ii) u is the last child and the set of conflict hulls $\mathcal{H}$ does not contain destination node t; or
      (iii) u is the parent node.
   (b) **Tree Traversal:** *Execute step (i), (ii) or (iii) according to the node u from which packet was received.*
      (i) u **is a child node that does not have t in its convex hull***: This means that we are trying to forward the packet to the subtree containing the destination t.*
         - *Set $p.anchor := v$.*
         - *If v does not have a child node with a convex hull that contains the destination t, forward the packet to the parent node. Else, forward the packet to the first child node.*
      (ii) u **is a child node that contains t in its convex hull***: This means that we are searching a subtree.*
         - *If v has a greater child node with a convex hull that contains the destination t, forward the packet to it.*

- *If the set of conflict hulls $\mathcal{H}$ does not contain the destination t, forward the packet to the first child node with a convex hull that contains the destination t.*
- *Else, forward the packet to the parent node.*

(iii) u **is a parent node***: This means that we are searching a subtree.*

- *If the set of conflict hulls $\mathcal{H}$ does not contain the destination t, set $p.anchor := v$.*
- *If v does not have a child node with a convex hull that contains the destination t, forward the packet to the parent node. Else, forward the packet to the first child node.*

Note that there are two possible scenarios when a packet switches to Tree Traversal mode. It can either be in a subtree containing the destination t, or not. In the latter case, GDSTR will forward the packet progressively up a hull tree until it reaches a node that is a node in the subtree containing the destination t.

While there will be periodic changes in the membership of the network, we do not expect such changes to be common. It is possible for the root node of a hull tree to fail. This causes the associated hull tree to be re-built. Since we use a tree building algorithm that guarantees the uniqueness of a tree given a root node, the $tree$ identifier on a packet is the node identifier of the root node of the hull tree. When a hull tree is rebuilt with a new node, the identifier on the packet will no longer match that of the new tree and so we may have to revert to Step 2(b) in Step 3 to pick a new tree. Smaller, transient changes in the hull trees may cause some packet losses, but such losses are not a major concern since changes in network membership due to node failures and new nodes joining the network are expected to be rare.

The following is a statement of correctness for GDSTR.

Theorem 2 *Given a pair of nodes v and t in a connected graph G, GDSTR guarantees packet delivery from v to t.*

The following is the proof that if a packet is deliverable, it will be delivered to the destination node and if it is not deliverable, the routing algorithm will eventually terminate. By *deliverable*, we mean that the destination coordinate in the packet corresponds to a live node and there is a path from the source node to the destination node.

Proof: First, suppose a packet from v to t is deliverable, but GDSTR fails to deliver it. This failure can arise only as a result of one of two situations: (i) the packet is trapped in some loop; or (ii) the packet terminates at some node $u \neq t$.

It is not possible for a packet to be trapped in a loop because (i) if the packet is in Greedy mode, $|(p.n_{min})t|$ is strictly decreasing and (ii) if the packet is in Tree Traversal mode, it will eventually

return to the anchor node $p.n_{anchor}$ and GDSTR will terminate. Note that it is possible for a packet to oscillate between the Greedy and Tree Traversal modes. However, since $|(p.n_{min})t|$ is also strictly decreasing each time a packet switches to Greedy mode, a packet cannot be trapped in a loop indefinitely by switching between the two modes.

Termination can occur during greedy forwarding (i.e., Step 1(a)) or in Step 3(b). Termination during greedy forwarding occurs only if a packet is delivered to its destination. Since we assume that GDSTR fails to deliver the packet, it remains to show that it is not possible for the algorithm to terminate in Step 3(a).

For a packet to terminate in Step 3(a), it must have fully traversed the subtree that contains t. This means that the packet would have reached t.

Next, we show that a packet that is undeliverable will eventually terminate. Suppose for the sake of contradiction that this statement is false, i.e., we have a packet that is undeliverable and yet it loops indefinitely. Since $|(p.n_{min})t|$ strictly decreases whenever a packet is forwarded in Greedy mode and whenever it switches to Greedy mode, at some point, $|(p.n_{min})t|$ will reach a minimum value greater than zero. Since the number of nodes in the network is finite, it is not possible for $|(p.n_{min})t|$ to continue to decrease and not reach zero. Since $|(p.n_{min})t|$ is constant only in Tree Traversal mode and the packet cannot switch to Greedy mode, it has to loop indefinitely in Tree Traversal mode. This is impossible since the packet will eventually return to the anchor node $p.n_{anchor}$ and GDSTR will terminate. Note that $p.n_{anchor}$ is first set in step 2(b), and incrementally updated in step 3(b) to ensure that $p.n_{anchor}$ remains on the traversed subtree. ■

3.6 Approximate Routing

In data-centric sensor applications [55, 74], the goal is often to route a packet to the node that is closest in geographic distance to the specified destination, rather than to a specific node in the network. We refer to this routing primitive as *approximate routing*.

Existing face routing algorithms achieve this goal by forwarding a packet around a face (void) [14]. The same can be achieved with hull trees. GDSTR can be modified to achieve the same by considering the destination not as a simple point, but as a "fat" point centered at the destination and bounded by the node nearest to the destination that has thus been visited by a packet. Next, the search subtree is constructed by checking for intersections between this "fat" point and the convex and conflict hulls.

This is illustrated in Figure 3-9. In this example, suppose node n_5 sends a packet addressed to t. Since node n_6 is nearest to t, it is the intended recipient of the packet. When the packet is forwarded to n_2, n_2 has to forward the packet up to its parent even though its convex hull intersects with the circle $\mathcal{C}$, and it does not have any conflict hulls. $\mathcal{C}$ is centered at the destination t and with radius $|n_{min}t|$, where n_{min} is the node nearest to the destination that has been visited by a packet.

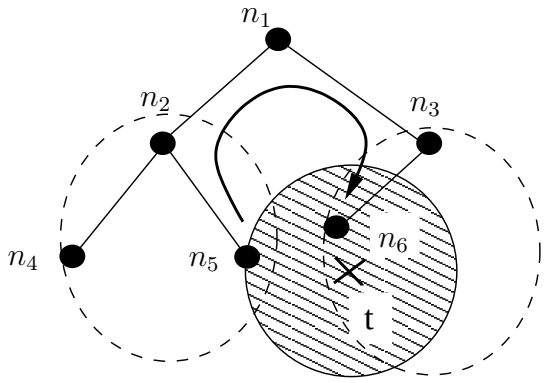


Figure 3-9: An example illustrating approximate routing.

The above process is equivalent to a systematic search for the node that is nearest to t on the hull tree. As n_{min} is updated in this process, the size of the search region shrinks in size. Since n_{anchor} is updated, the packet may terminate at $n_{anchor} \neq n_{min}$. If so, we redirect the packet to n_{min}. The modified algorithm is stated below as Algorithm 3.

Algorithm 3 (Approximate GDSTR) *When a node v receives packet p for a destination point t from a neighboring node u, do:*

1. **Preliminary Checks:**
   - *(a)* **Packet Delivery:** *If $v = t$, the packet has been delivered.*
   - *(b)* **Check for switch to Greedy mode:** *If $p.mode \neq$ Greedy and there is at least one immediate neighbor w such that $|wt| < |(p.n_{min})t|$, then set $p.mode :=$ Greedy, $p.n_{min} := w$ and clear $p.n_{anchor}$ and $p.tree$ if they are set. Execute step 2 or 3 according to $p.mode$.*
2. **Greedy Mode:** *Find the node w in the set of immediate neighbors that is closest to the destination t.*
   - *(a)* **Greedy Forwarding:** *If $|wt| < |vt|$, set $p.n_{min} := w$ and forward the packet to w.*
   - *(b)* **Switch to Tree Traversal Mode:** *Choose one of the hull trees for forwarding and set $p.tree$ to the chosen tree's identifier, $p.mode :=$ Tree and $p.n_{anchor} := v$. Then, find the set of child nodes with convex hulls that intersect the circle $\mathcal{C}$, centered at t with radius $|p.n_{min})t|$.*
     - *If set is non-empty, arrange the child nodes (relative to $p.tree$) with convex hulls that contain the destination point in an ascending sequence according to the global ordering of node identifiers and forward the packet to the first child node.*
     - *Else, forward the packet to the parent node in $p.tree$.*

3. **Tree Mode:** *If the root for $p.tree$ has changed, follow step 2(b), else follow step 3(a).*
   (a) **Check Termination Condition:** *If $v = p.anchor$ and one of the following conditions holds:*
      (i) *u is the last child and v is the root node for $p.tree$;*
      (ii) *u is the last child and v's convex hull fully contains the circle $\mathcal{C}$, and none of the conflict hulls in $\mathcal{H}$ intersects the circle $\mathcal{C}$; or*
      (iii) *u is the parent node.*

      do:
      - *If $v = p.min$, deliver the packet to v.*
      - *Else, set $t = v$ and $p.mode :=$ Greedy, clear $p.n_{anchor}$ and $p.tree$ if they are set and follow step 2(a).*

   (b) **Tree Traversal:** *Execute step (i), (ii) or (iii) according to the node u from which packet was received.*
      (i) *u* **is a child node that has a convex hull that does not intersect** *$\mathcal{C}$: This means that we are trying to forward the packet to the subtree containing $\mathcal{C}$.*
      - *Set $p.anchor := v$.*
      - *If v does not have a child node with a convex hull that intersects $\mathcal{C}$, forward the packet to the parent node. Else, forward the packet to the first child node.*

      (ii) *u* **is a child node that has a convex hull that intersects** *$\mathcal{C}$: This means that we are searching a subtree.*
      - *If v has a greater child node with a convex hull that intersects $\mathcal{C}$, forward the packet to it.*
      - *If there is a child node, excluding u, that has convex hull that intersects $\mathcal{C}$ and none of the none of the conflict hulls in $\mathcal{H}$ intersects the circle $\mathcal{C}$, forward the packet to the first child node.*
      - *Forward the packet to the parent node.*

      (iii) *u* **is a parent node***: This means that we are searching a subtree.*
      - *If v's convex hull fully contains the circle $\mathcal{C}$, and none of the conflict hulls in $\mathcal{H}$ intersects the circle $\mathcal{C}$, set $p.anchor := v$.*
      - *If v does not have a child node with a convex hull that intersects $\mathcal{C}$, forward the packet to the parent node. Else, forward the packet to the first such child node.*

This algorithm is a straightforward extension of Algorithm 2 with the two following modifications:

1. Instead of traversing only the child nodes with convex hulls that contain the destination coordinate t, a packet will traverse all child nodes with convex hulls that intersect the circle $\mathcal{C}$.

2. The conflict hulls can only prune nodes at the top of the search subtree if $\mathcal{C}$ is fully contained in a node's convex hull. The reason for this is that a node is only authoritative over the points within its convex hull that is not intersected by a conflict hull. If there are points in $\mathcal{C}$ that lie outside its convex hull, a node does not have sufficient information to decide that the destination is not reachable through a distant branch of the tree, and hence has to forward the packet to its parent.

The correctness of this algorithm follows from the following property of a hull tree:

Lemma 1 *Given a set of nodes on a hull tree $\mathcal{T}$, let $\mathcal{P}$ be the set of all nodes contained in the region $\mathcal{R}$ and $\mathcal{T}'$ be the set of all nodes that have convex hulls that intersect $\mathcal{R}$. Then, $\mathcal{P} \subset \mathcal{T}'$. Furthermore, the set of nodes $\mathcal{T}'$ is a subtree of $\mathcal{T}$.*

Proof: It is easy to see that $\mathcal{P} \subset \mathcal{T}'$, since any node $n \in \mathcal{P}$ is contained in $\mathcal{R}$ and therefore its convex hull must have at least one point in $\mathcal{R}$, which implies that it intersects $\mathcal{R}$.

To show that $\mathcal{T}'$ is a subtree of $\mathcal{T}$, we just have to show that $\mathcal{T}'$ is connected. A connected subset of the nodes in a tree cannot contain loops and is therefore a tree.

Assume $\mathcal{T}'$ is not connected, i.e., there exists two nodes $n_1, n_2 \in \mathcal{R}$ such that no path exists between them through the nodes in $\mathcal{T}'$.

Consider the root node r of $\mathcal{T}$. Clearly $r \in \mathcal{T}'$ since the convex hull for r contains all the nodes in the network. Also a path must exist between r and n_1, since there is a path from the root to every node in $\mathcal{T}$. We also know that the convex hulls of all the nodes on the path between r and n_1 must contain the coordinate of n_1 since this is a property of the convex hull aggregation algorithm. This means that these nodes are all in $\mathcal{T}'$. The same argument applies to n_1 and so we know that there is a path in $\mathcal{T}'$ between r and both n_1 and n_2. We can therefore construct a path in $\mathcal{T}'$ between n_1 and n_2 by concatenating these two paths and obtain a contradiction. ■

The correctness of this algorithm follows from Algorithm 3. The main difference between the two algorithms is in the definition of the tree to be traversed in Tree Traversal mode.

Proof: [of Algorithm 3] Suppose a packet from v to t is deliverable, but Algorithm 3 fails to deliver it to z, the node in the network that is nearest to t. This failure can arise only as a result of one of two situations: (i) the packet is trapped in some loop; or (ii) the packet terminates at some node u such that $u \neq z$.

As before, it is not possible for a packet to be trapped in a loop because (i) if the packet is in Greedy mode, $|(p.n_{min})t|$ is strictly decreasing and (ii) if the packet is in Tree Traversal mode, it will eventually return to the anchor node $p.n_{anchor}$ and the algorithm will terminate. Note that it is possible for a packet to oscillate between the Greedy and Tree Traversal modes. However, since $|(p.n_{min})t|$ is also strictly decreasing each time a packet switches to Greedy mode, a packet cannot be trapped in a loop indefinitely by switching between the two modes.

Termination can occur during greedy forwarding (i.e., Step 1(a)) or in Step 3(a). Termination during greedy forwarding occurs only if a packet is delivered to the specified destination t. It remains to show that it is not possible for the algorithm to terminate a node u is not the node in the network that is nearest to t.

Recall that $p.n_{min}$ is the node visited by the packet this is closest to t. This means that if $p.n_{min} \neq z$, z must lie within the circle $\mathcal{C}$ centered at the destination t and with radius $|p.n_{min}t|$. From Lemma 1, we conclude that for a packet to terminate in Step 3(a), it means that it has fully traversed all the nodes contained in $\mathcal{C}$. This means that the packet would have visited z. If $p.anchor \neq p.n_{min} = z$, setting the destination to $p.n_{min}$ will certainly succeed in routing a packet if we start routing afresh from $p.anchor$. In the worst case, a packet would traverse the subtree covering the circle $\mathcal{C}$ over again.

The proof that packet forwarding must eventually terminate is identical to the proof for Algorithm 2. It is based on the observation that $|p.n_{min}t|$ is monotonically decreasing whenever a packet is in Greedy mode and that Tree Traversal will eventually take a packet back to $p.anchor$. ■

3.7 Implementing Geocast with Hull Trees

Geocast [60] is a routing primitive that delivers a packet to all the nodes in a specified target region $\mathcal{R}$. In Section 3.6, we showed that given a hull tree $\mathcal{T}$ and a subtree $\mathcal{T}'$ consisting of all nodes with convex hulls that intersect $\mathcal{R}$, $\mathcal{T}'$ contains all the nodes in the target region (see Lemma 1). This property suggests the following straightforward approach to implement geocast with hull trees: forward a geocast packet to any node on $\mathcal{T}'$ and subsequently broadcast it to all the nodes in the subtree $\mathcal{T}'$. Because this process is analogous to fireworks, we call this algorithm the *Fireworks* algorithm.

To route a geocast packet to $\mathcal{T}'$, we route the packet with GDSTR like a regular GDSTR packet to a destination $t \in \mathcal{R}$. In principle, any point in the target region $\mathcal{R}$ would work. If $\mathcal{R}$ is convex, we can use the centroid of the region as initial destination t since the centroid will be contained within $\mathcal{R}$.

In forwarding the geocast message to t using GDSTR, one of two situations can occur: (i) the message reaches a node in $\mathcal{T}'$ before it reaches t; or (ii) t does not correspond to a node in network and routing terminates without visiting any node in $\mathcal{T}'$. In the latter, we simply forward the packet up the hull tree. At some point, the packet will eventually either reach a node with a convex hull that intersects $\mathcal{R}$, i.e., a node on $\mathcal{T}'$, or end up at the root of the tree because none of the nodes is contained in the target region $\mathcal{R}$.

Algorithm 4 (Fireworks (GDSTR Geocast)) *When a node v receives packet p for region $\mathcal{R}$ and effective destination t, $t \in \mathcal{R}$, from a neighboring node u, do:*

1. **Check for Geocast Mode:** *If $p.mode = $ Geocast, follow step 5.*
2. **Check Reached Broadcast Tree:** *If v has a child with a convex hull that intersects with $\mathcal{R}$, follow step 5. Otherwise, follow step 3.*
3. **FindTree Mode:** *If $p.mode = $ FindTree:*
   - *If v is the root node for $p.Tree$, algorithm terminates here.*
   - *Otherwise, forward p to the parent node in $p.Tree$.*
4. **GDSTR Routing:** *Route packet to destination t according to GDSTR (Algorithm 2). If packet is undeliverable, set $p.mode :=$ FindTree and follow step 3.*
5. **Tree Broadcast:** *If $p.Tree$ is not set, pick hull tree with root nearest to t and set $p.Tree$ accordingly. Determine target set for message broadcast with respect to $p.Tree$ according to the following rules:*
   - *If $p.mode =$ Geocast, node from which geocast message was originally received is not to be included in set of targets.*
   - *Each child node that has a convex hull that intersects $\mathcal{R}$ is added to the target set.*
   - *If the convex hull of associated hull tree $p.Tree$ fully contains $\mathcal{R}$ and none of the conflict hulls $\mathcal{H}$ intersects $\mathcal{R}$, do not add parent to target set. Otherwise, add the parent node to the target set.*

   If $p.mode \neq$ Geocast, set $p.mode :=$ Geocast. Broadcast p to all nodes in target set.

The proof of correctness for this algorithm is as follows.

Proof: Step 5 follows from the definition of $\mathcal{T}'$, i.e., if the packet p reaches a node in $\mathcal{T}'$, then Step 5 uniquely defines the broadcast tree $\mathcal{T}'$. From Lemma 1, we know that all the nodes in $\mathcal{R}$ are contained in $\mathcal{T}'$ and broadcasting the geocast message on $\mathcal{T}'$ will deliver the message to all nodes in the target region $\mathcal{R}$. It only remains to for us to show that a packet will eventually reach a node in $\mathcal{T}'$.

If the packet p switches to Geocast mode in Step 2, clearly $v \in \mathcal{T}'$ and we are done. Suppose p switches to FindTree mode in Step 3 because it is undeliverable according to GDSTR. Then, according to Step 3, the packet will be forwarded to the parent node and further up the tree as long as the convex hulls of the ancestor nodes do not intersect $\mathcal{R}$. Any ancestor node with a convex hull that intersects $\mathcal{R}$ is by definition in $\mathcal{T}'$.

The final node in the chain of ancestors is the root of the hull tree. The convex hull of the root contains the coordinates of all nodes in the network. If this convex hull does not intersect $\mathcal{R}$, then the target set is clearly an empty set; otherwise, then the root node is in $\mathcal{T}'$ and we are done. ■

Geocast is exactly analogous to approximate routing with the following two caveats:

1. Instead of considering the destination as an incrementally shrinking "fat" point, the destination is now an area.

2. Instead of traversing the search subtree systematically with one packet during the Tree Traversal, we effectively traverse the entire search subtree in parallel by having each node broadcast a packet to all its neighbors to the subtree, except for the node from which the packet was received.

3.8 GDSTR+: Using Local Information to Improve Routing and Geocast Performance

GDSTR works well for sparse networks with large voids. For dense networks, geographic face routing algorithms can achieve better performance since the voids tend to be small and it generally does not matter which forwarding direction is picked. The is because when there are many hops between the root of the hull trees and the leaves, the hull trees are not able to approximate voids quite as well as planar faces and GDSTR therefore incurs additional routing overhead. Also, geocast is expected to be less efficient when the hull tree is large.

One possible approach would be to use GDSTR in sparse networks and geographic face routing algorithms in moderately dense networks. However, such an approach does not address the high maintenance costs of planarizing the network graph [42, 51]. Another issue is that large networks are likely to be heterogeneous, with some dense regions and some sparse regions. Ideally, a good geographic routing algorithm should work well in networks of all densities.

Our earlier work with GPVFR [52], demonstrated that we can improve routing performance for face routing by storing a small amount of local face information at each node. The key reason for this improvement is not directly related to face routing; the improvement arises because with a few hops of extra face information, GPVFR can improve the probability of picking a good forwarding direction. When the network is dense and voids are small, a few hops of information is in fact sufficient to guarantee that the correct forwarding direction around a void is chosen all the time. Based on this observation, we developed GDSTR+, a variant of GDSTR that incorporates local information to improve routing and geocast performance in dense networks.

3.8.1 Overview

The key idea in GDSTR+ is to augment GDSTR with two forests of "local" trees and an additional *greedy-hull* forwarding mode. In GDSTR+, a node will first attempt to forward a packet greedily as before. If greedy forwarding fails, it will switch to the new greedy-hull forwarding mode by using the information contained in the convex hulls of a local hull tree. By local, we mean that the tree contains only the nodes in a limited locality. Since correctness cannot be guaranteed, forwarding

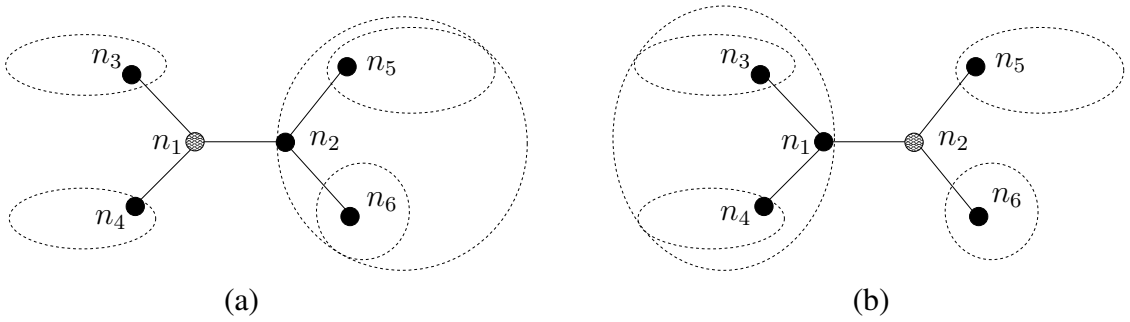


Figure 3-10: Example showing the aggregation of convex hulls for local trees in GDSTR+. Again, convex hulls are represented with ellipses for simplicity.

can sometimes fail using the local tree and in such a case, a node will switch to forwarding on one of the two original global hull trees, which is guaranteed to succeed.

In order to implement greedy-hull forwarding, the local hull trees employ a different convex hull aggregation algorithm than that employed by GDSTR to maintain global hull trees. This new aggregation mechanism presents a node with a view of the locations accessible via each neighboring node. To aggregate hulls this way, each node broadcasts the hulls of all its neighbors instead of its own hull.

We illustrate the local hull tree aggregation algorithm with the example shown in Figure 3-10. In particular, the convex hull associated with each neighbor contains the set of destination points that are reachable through that neighbor. In this example, both nodes n_1 and n_2 have three neighbors. In Fig. 3-10(a), we show the convex hulls from n_1's perspective. Under this new scheme, n_2's *keepalive* message will contain the hulls for n_1, n_5 and n_6 from its perspective. From n_1's perspective, its convex hull for n_2 is the convex hull that contains n_2 and the convex hulls for n_5 and n_6. Similarly, as shown in Fig. 3-10(b), From n_2's perspective, its convex hull for n_1 is the convex hull that contains n_1 and the convex hulls for n_3 and n_4. This provides each node with a view of the geographic coordinates accessible via each neighboring node in the tree.

We illustrate this process with the example network in Figure 3-11. In this example, node s receives a packet with destination t and has to decide how to forward the packet. Since node s is a dead end for greedy forwarding, it tries to forward the packet in greedy-hull forwarding mode. In this mode, node s considers the convex hulls of its neighboring nodes and finds the point on the hulls that is closest to the destination t. It turns out to be point r on the convex hull of node n_2. Hence, node s forwards the packet to n_2 instead of n_1.

Like GDSTR, the coordinate of the node at which a packet switches to greedy-hull forwarding mode (s in the above example) is recorded as n_{min} and a packet will revert to greedy forwarding mode as soon as a neighboring node that is closer to the destination than n_{min} is found. In addition, the coordinate of the point that is closest to the destination t that is found on any convex hull is also recorded in the packet, as a new component p_{hull}. p_{hull} will be updated at each step until a packet either ends up at p_{hull} and has no neighbor with a hull point closer to the destination than p_{hull} or if

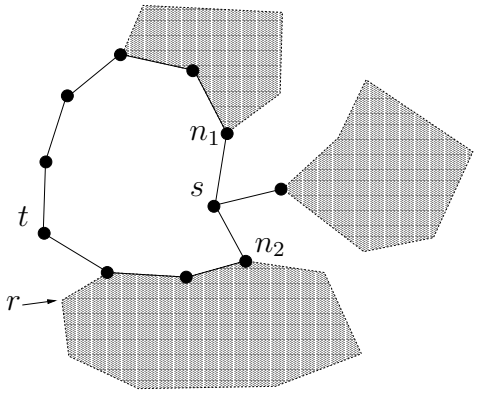


Figure 3-11: Example of *greedy-hull* forwarding. The shaded polygons are the convex hulls of neighboring nodes from the perspective of node s.

it switches to greedy mode. In the former case, a packet will switch to GDSTR tree traversal mode on one of the two available global trees. p_{hull} is only updated with a value that is monotonically nearer to the destination. If a packet in greedy forwarding mode ends up in a dead end and the associated node does not have a convex hull that has a point that is closer to the point p_{hull}, the packet will bypass the greedy-hull forwarding mode and switch directly to tree traversal on one of the two global trees. This is to prevent oscillations.

3.8.2 Building Local Trees

There are many possible schemes to construct the local hull trees. We adopt the following simple scheme: we divide the coordinate space into a square grid of fixed length, and all the nodes within each grid square will be members of the same local tree. The root of a local tree is the node with a coordinate that is closest to the center of a grid square. The process of building such a tree has two steps: first, a node attempts to route a packet to the center of its local grid square. Doing so will allow it to identify the root node of its local tree. Next, it determines its parent by routing a packet to the root node.

The second step is necessary because a given node may not be connected to the root of its hull tree through neighbors that are within the same grid square. This is illustrated in the example shown in Fig. 3-12(a). In this example, node r is the root of the local hull tree since it is nearest to the center of the square grid; node s is however connected to neighbors that are outside the square grid. The resulting local hull tree is shown in Fig. 3-12(b). This example also shows that it is possible for nodes near the edge of a grid square to be members of the local hull trees of neighboring grid squares. We impose the strict membership condition that all the nodes within each grid square belong to the same local hull tree to provide correctness guarantees for geocast (to be described in Section 3.8.4).

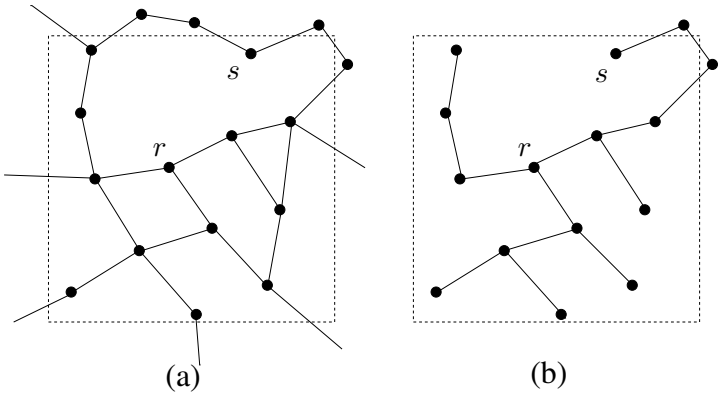


Figure 3-12: Example demonstrating the building of local hull trees. The connectivity of an example grid square is shown in (a). The resulting local hull tree is shown in (b).

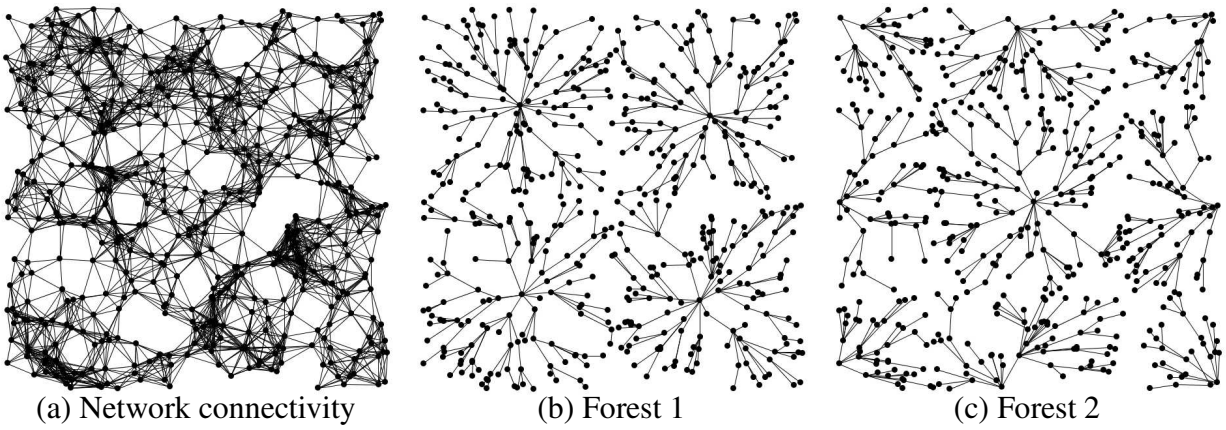


Figure 3-13: Example network with its two associated forests of local hull trees.

The above process will create a forest of local hull trees, where in general most nodes will be a member of only one local hull tree and only a small number near the boundaries of a grid square will be members of two or more trees. The purpose of these local hull trees is to provide nodes with a local view of the surrounding network. With only one forest of trees, the nodes near the centers of the grid squares will have a relatively balanced view of the network, while those near the edges of the grid squares will have a somewhat skewed perspective. To address this, we build a second forest of local hull trees using grid squares with an offset from the original grid such that the centers of the new grid squares are the corners of the original grid. The resulting forests are illustrated in Fig. 3-13.

In building the local hull trees, it is necessary to decide on the width of the grid squares. Prior work has shown that about 4 hops of information is sufficient to choose the optimal forwarding direction when the network is dense [52]. Hence, in our implementation, we use a grid width that is 5 times the radio range. For a random network, the resulting local trees are about 3 or 4 nodes deep.

3.8.3 Routing Algorithm

GDSTR+ is effectively a cross between GDSTR and GPVFR. From GDSTR, we borrow the idea of approximating voids with trees; from GPVFR, we borrow the idea of using local information in a greedy fashion first before resorting to routing with the more costly tree traversal routing mode. A GDSTR+ data packet p is tagged with the following state components:

- $mode$: current forwarding mode (Greedy/Greedy-Hull/Tree),
- n_{min}: node visited that is nearest to destination,
- p_{hull}: point on observed convex hull that is nearest to destination,
- $tree$: identifier for chosen forwarding tree,
- n_{anchor}: tree traversal anchor node.

Algorithm 5 (GDSTR+) *When a node v receives packet p for destination node t from a neighboring node u, do:*

1. **Preliminary Checks:**
   (a) **Packet Delivery:** *If $v = t$, the packet has been delivered, and we are done.*
   (b) **Check for switch to Greedy mode:** *If $p.mode \neq$ Greedy and there is at least one immediate neighbor w such that $|wt| < |(p.n_{min})t|$, then set $p.mode :=$ Greedy, $p.n_{min} := w$ and clear $p.n_{anchor}$ and $p.tree$ if they are set. Execute step 2, 3 or 4 according to $p.mode$.*
2. **Greedy Mode:** *Find the node w in the set of immediate neighbors that is closest to t.*
   (a) **Greedy Forwarding:** *If $|wt| < |vt|$, set $p.n_{min} := w$ and forward the packet to w.*
   (b) **Switch to Greedy-Hull Mode:** *Consider the convex hulls of each neighboring node with respect to $p.tree$ and find p_{min}, the point that is closest to the destination t. If $|p_{min}t| < |p.p_{hull}t|$, follow step 3; otherwise, follow step 4.*
3. **Greedy-Hull Mode:** *If $p.tree$ is not set or if the root for $p.tree$ has changed, follow step 3(a), else follow step 3(b).*
   (a) **Choose Hull Tree:** *Choose one of the local hull trees for forwarding and set $p.tree$ to the chosen tree's identifier. If a suitable tree is found, follow step 3(b). Otherwise, clear $p.tree$, set $p.mode :=$ Tree and follow step 3(c).*
   (b) **Check Hull Tree:** *Consider the convex hulls of each neighboring node with respect to $p.tree$ and find p_{min}, the point that is closest to the destination t. If $|p_{min}t| < |p.p_{hull}t|$ or $p_{min} = p.p_{hull}$, set $p.p_{hull} := p_{min}$, and forward p to the neighboring node that has the convex hull containing p_{min}.*

(c) **Switch to Tree Traversal Mode:** *Choose one of the hull trees for forwarding and set $p.tree$ to the chosen tree's identifier, $p.mode :=$ Tree and $p.n_{anchor} := v$. Then, find the set of child nodes with convex hulls that contain the destination t.*

- *If set is non-empty, arrange the child nodes (relative to $p.tree$) with convex hulls that contain the destination point in an ascending sequence according to the global ordering of node identifiers and forward the packet to the first child node.*
- *Else, forward the packet to the parent node in $p.tree$.*

4. **Tree Mode:** *If the root for $p.tree$ has changed, follow step 3(c), else follow step 4(a).*

(a) **Check Termination Condition:** *Conclude that packet is* **not deliverable** *if $v = p.anchor$ and one of the following conditions holds:*

(i) *u is the last child and v is the root node for $p.tree$;*

(ii) *u is the last child and the set of conflict hulls $\mathcal{H}$ does not contain destination node t; or*

(iii) *u is the parent node.*

(b) **Tree Traversal:** *Execute step (i), (ii) or (iii) according to the node u from which packet was received.*

(i) *u* **is a child node that does not have** *t* **in its convex hull***: This means that we are trying to forward the packet to the subtree containing the destination t.*

- *Set $p.anchor := v$.*
- *If v does not have a child node with a convex hull that contains the destination t, forward the packet to the parent node. Else, forward the packet to the first child node.*

(ii) *u* **is a child node that contains** *t* **in its convex hull***: This means that we are searching a subtree.*

- *If v has a greater child node with a convex hull that contains the destination t, forward the packet to it.*
- *If the set of conflict hulls $\mathcal{H}$ does not contain the destination t, forward the packet to the first child node with a convex hull that contains the destination t.*
- *Else, forward the packet to the parent node.*

(iii) *u* **is a parent node***: This means that we are searching a subtree.*

- *If the set of conflict hulls $\mathcal{H}$ does not contain the destination t, set $p.anchor := v$.*
- *If v does not have a child node with a convex hull that contains the destination t, forward the packet to the parent node. Else, forward the packet to the first child node.*

This algorithm is equivalent to GDSTR (Algorithm 2), with an additional Greedy-Hull Forwarding stage (Steps 3(a) and 3(b)).

In Step 3(a), the local hull tree is chosen by considering the convex hulls of all the neighboring nodes and identifying the convex hull with the point that is nearest to the destination in terms of geometric distance. The hull tree containing the said convex hull is chosen as the routing tree $p.tree$. In order to prevent loops, the resulting nearest point p_{min} must be strictly nearer to the destination t than p_{hull} if such a point is recorded in the packet. If such a point cannot be found, then the local hull tree selection in Step 2(b) will fail, and we revert to GDSTR tree traversal mode.

The correctness of GDSTR+ follows from the correctness of GDSTR. Given that GDSTR is correct, the correctness of GDSTR+ can be reasoned as follows: GDSTR+ is GDSTR with an additional greedy-hull forwarding mode. Greedy-hull forwarding does not have a termination condition and hence it can only affect correctness if it causes looping. We note that looping is impossible because in each forwarding step, either p_{hull} is monotonically decreasing or the packet is moving closer to p_{hull} along a fixed tree $p.tree$.

If a packet should end up in a dead end in Step 3 at a point where the grid square of a local tree contains the destination point, we can traverse a local tree instead of a global tree. However, we chose not to do so and decided to use the global trees to keep the algorithm simple. In any case, greedy-hull forwarding is almost always guaranteed to succeed in such circumstances.

GDSTR+ can also be modified to implement approximate routing. As described in Section 3.6, we adjust Algorithm 5 to search a circle $\mathcal{C}$ centered at t instead. In addition, since we know that all the points within each grid square are contained in the associated local hull tree, a local hull tree should be chosen for tree traversal once the circle $\mathcal{C}$ is small enough that it falls completely within a grid square.

3.8.4 Geocast

Geocast is most efficient over a minimal spanning tree that includes all the nodes in the target region. GDSTR does not attempt to build a minimal tree, it simply uses its existing hull trees. When the network is large, the global hull trees are large and hence while geocast over these trees achieves correctness, it is not an ideal technique. It would be preferable to geocast over smaller trees that cover an area that is just a little larger than the target area.

GDSTR+'s use of local hull trees presents us with an opportunity to improve geocast efficiency. In particular, because of the local tree building algorithm, we know that each local hull tree completely covers all the nodes in a given grid square. Hence, if the target region of a geocast message is completely contained in a grid square, we know that it will be broadcast correctly to all the required targets within the region. Because two forests are available, even if the target region of a geocast message is not completely contained within a grid square for one forest, it is likely to be contained within a grid square of the other. If indeed a suitable local tree cannot be found, correctness can be guaranteed by broadcasting on one of the global hull trees as before.

Algorithm 6 (Fireworks (GDSTR+ Geocast)) *When a node v receives packet p for region $\mathcal{R}$ and effective destination t, $t \in \mathcal{R}$, from a neighboring node u, do:*

1. **Check for Geocast Mode:** *If $p.mode =$ Geocast, follow step 6.*
2. **Check Reached Broadcast Tree:** *If v has a child with a convex hull that intersects with $\mathcal{R}$, follow step 5. Otherwise, follow step 3.*
3. **FindTree Mode:** *If $p.mode =$ FindTree:*
   - *If v is the root node for $p.Tree$, algorithm terminates here.*
   - *Otherwise, forward p to the parent node in $p.Tree$.*
4. **GDSTR+ Routing:** *Route packet to destination t according to Algorithm 5. If packet is undeliverable, set $p.mode :=$ FindTree and follow step 3.*
5. **Pick Hull Tree for Geocast:**
   - *If $\mathcal{R}$ is contained in either of the grid squares of the local hull trees, set $p.Tree$ as the local tree (in a grid square that contains $\mathcal{R}$) with a root that is closest to the t.*
   - *If the grid squares of the local hull trees do not completely contain $\mathcal{R}$, set $p.Tree$ as the global tree with a convex hull that contains $\mathcal{R}$; if such a global tree does not exist, pick the global tree with a root that is closest to t.*

   Follow step 6.
6. **Broadcast to Target Set:** *Determine target set $\mathcal{B}$ for message broadcast with respect to $p.Tree$ according to the following rules:*
   - *If $p.mode =$ Geocast, the node from which geocast message was originally received is not to be included in set of targets.*
   - *If $p.Tree$ is a local tree, each neighboring node that has an associated convex hull (from v's perspective) that intersects $\mathcal{R}$ is added to the target set.*
   - *If $p.Tree$ is a global tree, each child node that has a convex hull that intersects $\mathcal{R}$ is added to the target set. If the convex hull of associated hull tree $p.Tree$ fully contains $\mathcal{R}$ and none of the conflict hulls $\mathcal{H}$ intersects $\mathcal{R}$, do not add the parent node to target set. Otherwise, add the parent node to the target set.*

   If $p.mode \neq$ Geocast, set $p.mode :=$ Geocast. Broadcast p to all nodes in target set $\mathcal{B}$.

The correctness of this algorithm follows from the correctness of the GDSTR geocast algorithm (Algorithm 4).

Proof: The only difference between this algorithm and Algorithm 4 is that when a grid square contains the target region $\mathcal{R}$, we broadcast on the associated local hull tree instead of global tree.

We know from the proof of Algorithm 4 that if the message is broadcast on a global hull tree, all the points are in the target region $\mathcal{R}$.

Therefore, it suffices to show that local hull tree contains all the points in the target region $\mathcal{R}$. We know that this is true because the local hull tree contains all the points in the grid square and since the grid square contains the target region $\mathcal{R}$, all the points in the target region $\mathcal{R}$ must lie on the local hull tree. By construction, we know that the associated tree traversal algorithm will allow us to reach every single point in $\mathcal{R}$ on the local hull tree since during the broadcast step, the geocast message is forwarded to every neighbor with a convex hull that intersects $\mathcal{R}$. This means that if there is a node in $\mathcal{R}$ that is reachable, the message will eventually reach it.

■

3.9 Summary

The key insight of GDSTR is that for geographic routing, two hull trees can replace a planar graph as the backup routing topology when greedy forwarding fails, and it is significantly easier to build and maintain hull trees than a planar graph. GDSTR requires only one hull tree for correctness. However, we use a second tree because doing so provides better robustness in the event of node failures and an additional routing choice. In addition, we describe two natural extensions to the basic GDSTR routing algorithm – approximate routing and geocast (Fireworks algorithm).

We also describe GDSTR+, a variant of GDSTR that maintains two additional local hull trees. While GDSTR+ requires slightly more storage per node, the local hull trees are small and hence both cheap to build and maintain. With additional local information, GDSTR+ is expected to achieve better routing performance in dense networks with small voids.

Chapter 4

GDSTR Evaluation

In this chapter, we compare the performance of GDSTR routing to existing geographic face routing algorithms and present the experimental results for the costs of GDSTR in terms of both storage and bandwidth. We also evaluate the performance of GDSTR+ and show that it performs better than GPVFR [52], the best existing geographic face routing algorithm, in networks where the routing performance of GPVFR surpasses that of GDSTR. We also discuss the performance of our hull-tree-based geocast algorithm, but only for our own system and not in comparison to other algorithms.

Kuhn et al. showed that the routing performance of geographic routing algorithms is highly dependent on the network density [47]. In particular, geographic routing algorithms are almost always uniformly good for both very sparse and very dense random networks, and it is critical to study routing performance for topologies with average node degrees in the region between 4 and 8 (called the *critical region*). Moderately dense topologies with average node degrees between 8 and 14 are also of interest since there is a marginal difference in the routing performance of different algorithms. Topologies that are ultra-sparse or ultra-dense are not particularly interesting, since greedy forwarding will tend to work almost all the time and the routing performance of geographic routing algorithms tends to be indistinguishable.

To systematically explore the performance of GDSTR over different random topologies, we first evaluate GDSTR in small networks over a wide range of random topologies with average node degrees ranging from 0.7 to 14.4. Subsequently, we evaluate the routing performance and costs for GDSTR for a range of network sizes up to 5,000 for two network densities. In particular, we consider a set of sparse networks in the critical region (with an average node degree of 6.5) and another set of moderately dense networks with an average node degree of 12. We also investigated the effect of obstacles by generating two sets of routing topologies with a high and low density of obstacles, respectively.

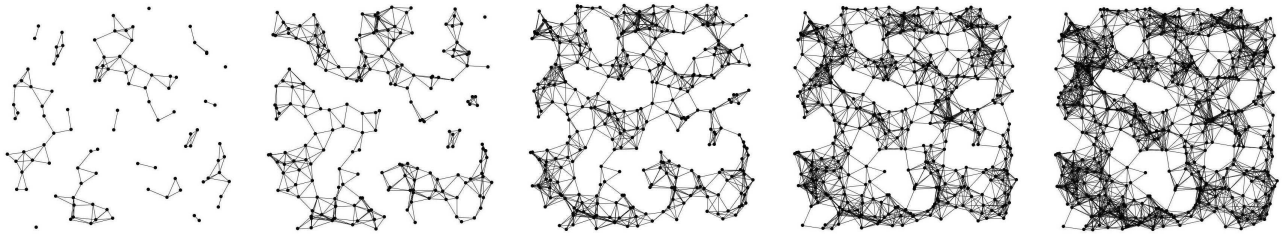


Figure 4-1: Sample networks of increasing average density and node degree.

4.1 Simulation Setup

This section describes our simulation setup for evaluating the performance of GDSTR. The simulations are performed using our own high-level, event-driven simulator [50].

For our simulations, we use a simple radio model: all nodes have unit 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 simulator supports linear, polygonal, and circular obstacles. Wireless losses are not simulated since our goal is to compare the basic algorithmic behavior of GDSTR to other geographic routing algorithms.

As discussed in Section 3.4.1, the underlying radio model does not matter for GDSTR. While the simulator is able to support non-uniform radio ranges, we consider only topologies with uniform unit radios since uni-directional links are not used by GDSTR, and topologies with non-uniform radio ranges can be replicated by adding obstacles. Even under this assumption, we are able to generate a diverse range of topologies, which we believe is adequate for the purposes of comparing GDSTR to existing geographic face routing algorithms.

We measure routing performance with respect to two metrics: (i) *path stretch* and (ii) *hop stretch*. Path stretch is the ratio of the total path length to the shortest path (in Euclidean distance) between two nodes; 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).

Effect of Network Density. To understand the effects of network density on routing performance and maintenance costs, we generate networks by randomly scattering between 25 to 500 nodes over a 10×10 unit square. This process generates networks with average node degrees between 0.7 and 14.4. For each density, we generate 200 networks, and then route 20,000 packets using each algorithm between randomly chosen pairs of source and destination nodes. The performance measurements presented are the average values over the 200 times 20,000 data points. We also use these topologies to evaluate the effects of parameters such as the number of hull trees and the value of r, the maximum size for the convex hulls.

Figure 4-1 shows some sample networks. The main point to notice here is that all these networks have large voids that will need to be routed around. The node degrees between 4 and 8 are critical

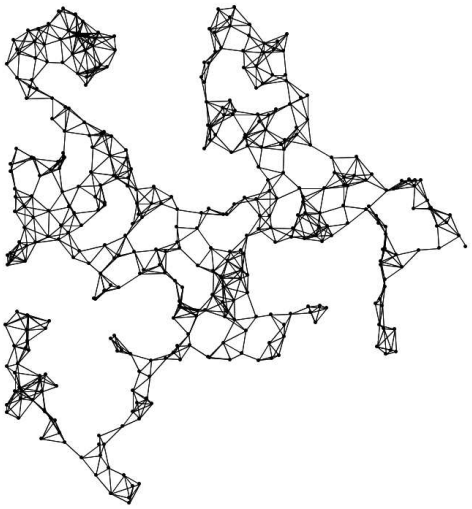


Figure 4-2: Sample 400-node sparse network.

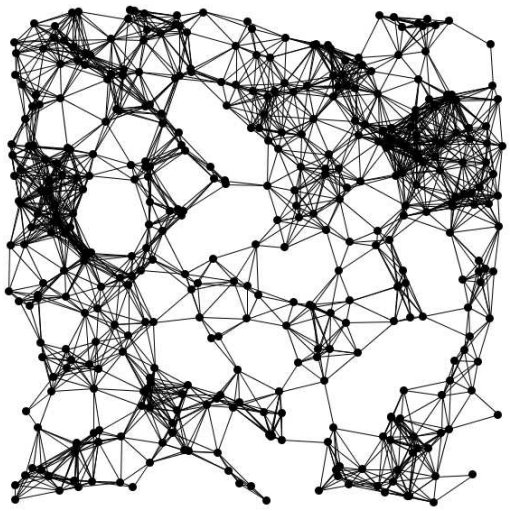


Figure 4-3: Sample 400-node dense network.

for all geographic routing algorithms, since greedy forwarding works well on denser networks, and sparser networks tend not to be well connected [47].

A density of 500 nodes in 100 square units is high enough that greedy forwarding almost always succeeds, and neither GDSTR nor face routing is needed. For this reason, we did not explore higher densities.

Effect of Network size. To evaluate the scaling of maintenance costs and performance, we generate networks with constant node density with sizes ranging from 50 to 5,000 nodes. As before, networks are generated for each size by scattering nodes randomly over a $x \times x$ unit square, scaling x by a factor of $\sqrt{n}$ for each network size n. This procedure sometimes generates networks that are not connected. We discard such networks and repeat the above until we have 200 connected networks for each network size.

We investigate scaling effects only with sparse and dense networks with average node degrees of approximately 6.5 and 12, respectively. We found that below an average node degree of 6.5, the random process described above often produces disconnected networks. Sample 400-node networks from the two sets are shown in Figures 4-2 and 4-3, respectively.

Effect of Obstacles. To evaluate the effect of obstacles, we generate some moderately dense networks and add a number of cross-shaped obstacles to them. Again, we generate a range of networks with constant node density from 50 to 5,000 nodes in size. The networks were generated for each size by scattering nodes randomly over an $x \times x$ unit square, where x was scaled by a factor of $\sqrt{n}$ for each network size n. Next, we add a number of cross-shaped obstacles (0.25 units across) proportional to the size of the area over which nodes are scattered.

This procedure sometimes generates networks that are not connected. Again, we discard such networks and repeat the above procedure until we have 200 connected networks for each network size. We found that it is difficult in practice to generate random connected networks for graphs

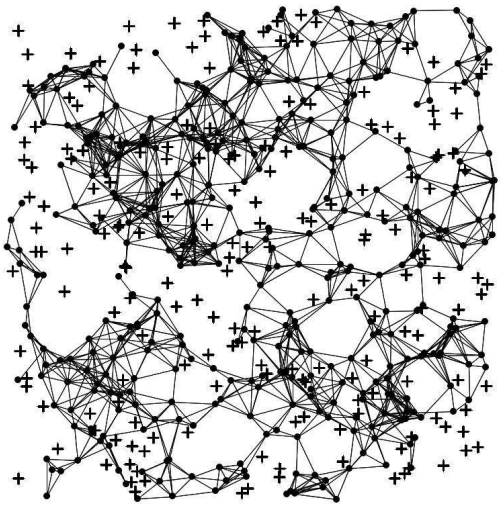


Figure 4-4: Sample 400-node network with low obstacle density.

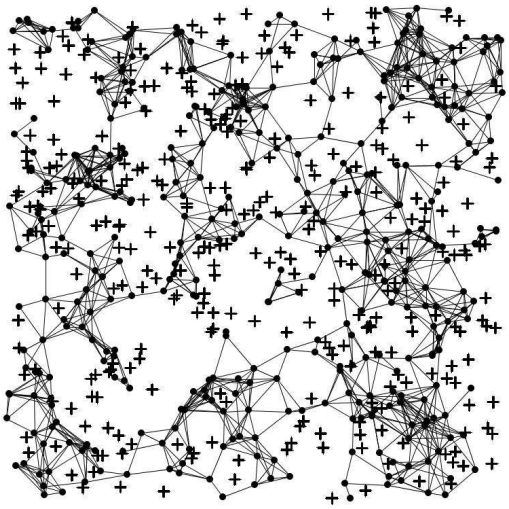


Figure 4-5: Sample 400-node network with high obstacle density.

with a density of obstacles above a given threshold. The resulting networks had an average node degree of 7; without the obstacles, the networks would have had an average node degree of 9.5. Sample 400-node networks from the two sets are shown in Figures 4-4 and 4-5, respectively.

Comparisons. We compare the routing performance of GDSTR to GPSR [39], GOAFR+ [47] and GPVFR [52]. We evaluate the geographic face routing algorithms (GPSR, GOAFR+ and GPVFR) with CLDP planarization [42] rather than *Gabriel Graph* (GG) [20] or *Relative Neighborhood Graph* (RNG) [84] planarization, since CLDP is currently the only algorithm that is known to work for practical networks. In any case, the networks with obstacles are not unit-disk graphs (UDGs) and hence GG and RNG would not planarize them correctly.

Our implementation of these routing techniques is based on the algorithms described in [39], [47], and [52], respectively. The configuration parameters for GOAFR+ are $\rho_0 = 1.4, \rho = \sqrt{2}$ and $\sigma = \frac{1}{100}$, as suggested [47]. For GPVFR, we limited the length of the propagated path vectors to 3. Unless stated otherwise, we used two hull trees for all experiments with GDSTR. Our implementation of CLDP follows the description in [42].

4.2 GDSTR Routing Performance

Figure 4-6 shows the hop stretch for deliverable packets for GDSTR, GPSR, GOAFR+ and GPVFR over a range of average node degrees.

GDSTR has the best performance for most of the range; that is, GDSTR routes packets along shorter paths than the other algorithms, and is thus likely to deliver packets faster and with less consumption of radio resources. The only exception is that the stretch of GPVFR is a few percent less than that of GDSTR for node degrees higher than 9.

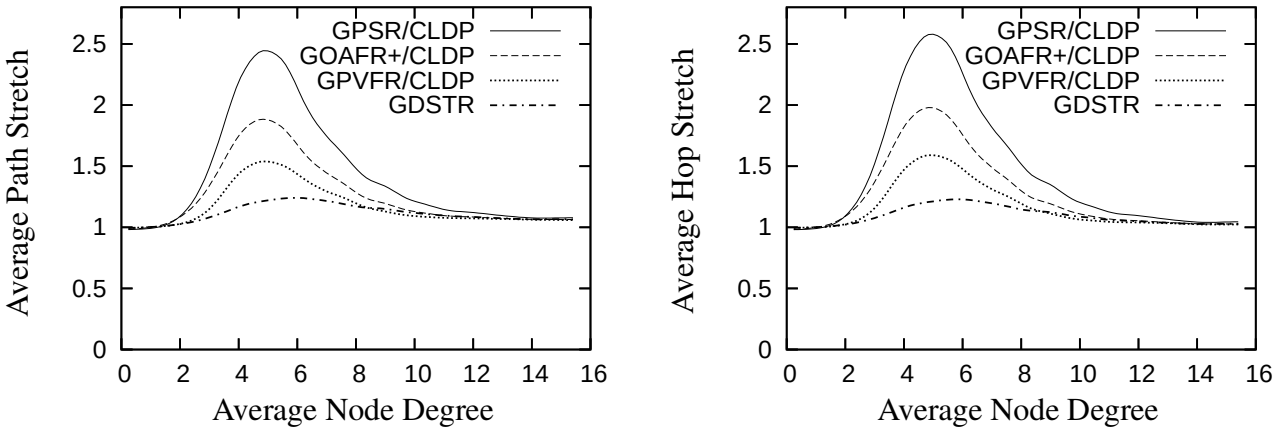


Figure 4-6: Plot comparing the stretch for GDSTR (two trees) to that for GPSR, GOAFR+ and GPVFR under CLDP planarization.

The differences in the performance of the various algorithms are most pronounced in the critical region with node degrees between 4 and 6. The reason for this is that networks in this region tend to have large outer perimeters and the voids that are generated are often concave. Packets tend to end up in a local minima fairly often for these topologies, and the routing algorithm must resort to forwarding the packet along a face (for the face routing algorithms) or on a tree (for GDSTR).

GPSR performs the worst because it uses a deterministic *right hand rule* when forwarding a packet along a face. It turns out that topologies in the critical region typically present nodes that need to switch to face traversal with one good forwarding direction and one terrible alternative. By choosing the same direction consistently, GPSR gets it wrong about half the time.

GOAFR+ is better than GPSR because it uses an expanding ellipse to bound the search radius. GOAFR+ picks a random forwarding direction to start with, but instead of forwarding continuously along a face, it keeps track of how far it has gone along the face. If a packet seems to have wandered far enough along a face and not made any apparent progress toward the destination, GOAFR+ will make the packet backtrack. By expanding the area of the search incrementally, GOAFR+ ensures that the length of the final path traversed is no longer than a constant multiple of the optimal path.

GPVFR tries to pick the optimal forwarding direction when it switches from greedy forwarding to face traversal. It does so by maintaining several hops worth of information about its adjacent planar faces. It turns out that in practice, by maintaining information about nodes that are up to 4 hops away along the planar faces, GPVFR will often make the correct decision when the network density is low. When the network density is relatively high (above an average node degree of 9), CLDP produces planar faces that are relatively small (usually with fewer than seven points). Thus, under such circumstances, GPVFR has enough information to guarantee that it chooses the correct forwarding direction almost all the time, which explains why it performs better than GDSTR for node degrees higher than 9.

When two forwarding directions are available, GDSTR's tree-choosing heuristic of picking the tree with a root that is closest to the destination allows us to choose a good forwarding direction

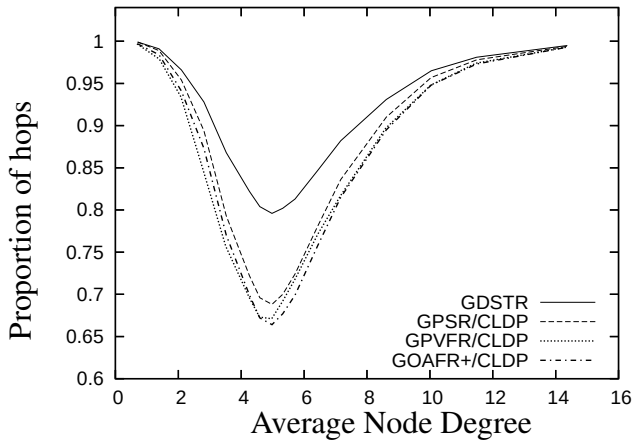


Figure 4-7: Proportion of hops taken in greedy forwarding mode.

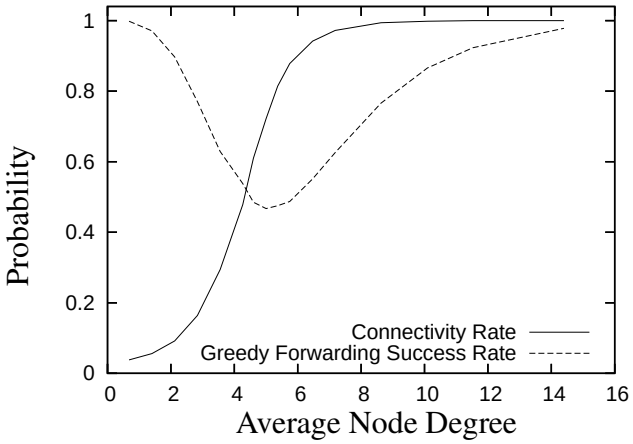


Figure 4-8: Connectivity probability and greedy forwarding success rate.

around a void most of the time. However, we believe that a more significant factor explaining why GDSTR outperforms the other algorithms in the critical region is that the convex hulls contain sufficient information to allow GDSTR to prune away many bad routing choices and route on a much reduced subtree that is often significantly smaller than large voids or the perimeter of the network (which often have one hundred or more nodes).

To further understand the performance of GDSTR, we also measured the distribution of the forwarding modes for GDSTR, GPVFR, GOAFR+, and GPSR. The distributions are plotted in Figure 4-7. These results suggest that a likely reason why GDSTR achieves better routing performance than GPSR, GOAFR+, and GPVFR is that it forwards packets in greedy mode 10% more frequently than these other algorithms in the critical region. For reference, the probability of two source-destination pairs being connected and the probability that greedy forwarding alone is sufficient to route between any random source-destination pair are shown in Figure 4-8.

4.2.1 How Many Trees are Useful?

GDSTR employs two hull trees rooted at the nodes with the maximal and minimal x coordinates. Since GDSTR can use more than two hull trees, it is important to understand how the number of hull trees will affect routing performance. Since our goal is to use hull trees to "approximate" voids and we want the roots of the k hull trees to be as far apart as possible, we can construct k rays rooted at the origin, and choose as roots the nodes whose projections onto these rays are farthest from the origin.

Figure 4-9 shows the effect of increasing the number of trees on the average hop stretch. Routing performance improves quite significantly when we increase the number of hull trees from one to two (achieving a peak improvement of approximately 10% in path and hop stretch); routing

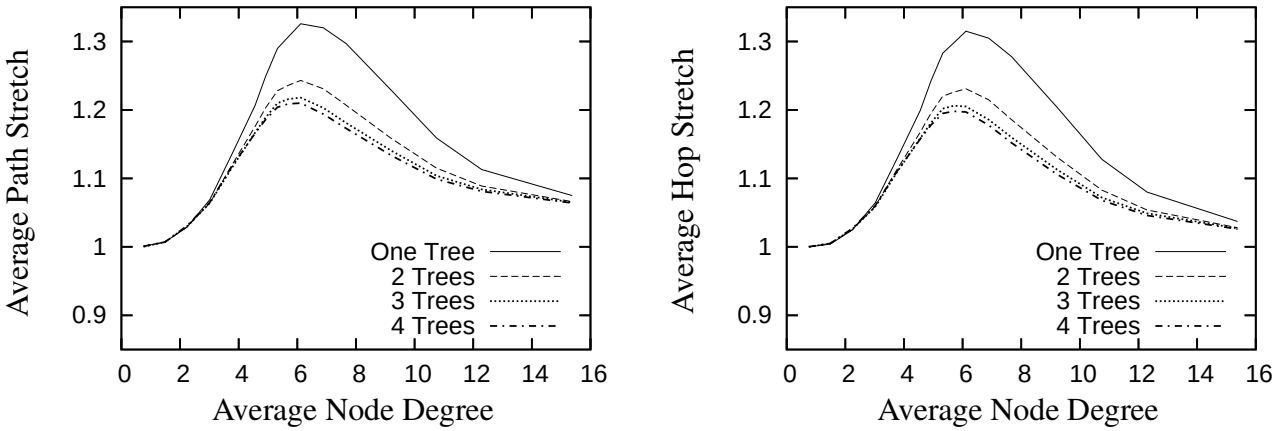


Figure 4-9: Routing stretch performance for GDSTR with different numbers of hull trees.

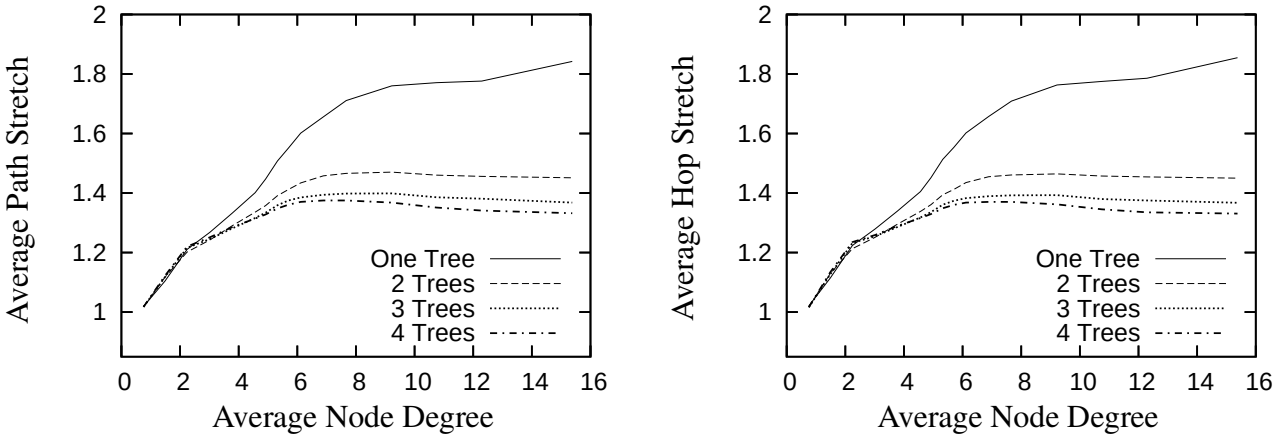


Figure 4-10: Routing stretch performance for GDSTR with different numbers of hull trees for packets that require tree traversal.

performance continues to improve with more trees. However, beyond two trees, the improvement is marginal.

Figure 4-9 shows aggregate routing performance averaged over a range of source-destination pairs. For dense networks, the probability that routing will succeed without the need for tree traversal is high. To better understand the effect of the hull trees (which is only used during tree traversal), in Figure 4-10, we plot the stretch for source-destination pairs for which greedy forwarding alone is sufficient and a packet has to be forwarded on the hull trees at some point.

We see from these results that indeed, increasing the number of hull trees from one to two has a very significant effect on routing stretch. This is especially so for denser networks. The difference in routing performance when more trees are employed is more apparent in Figure 4-10 when only the packets that require tree traversal are considered, because when greedy forwarding alone is sufficient to deliver a packet, it almost always achieves a stretch of one. Moreover, the high prob-

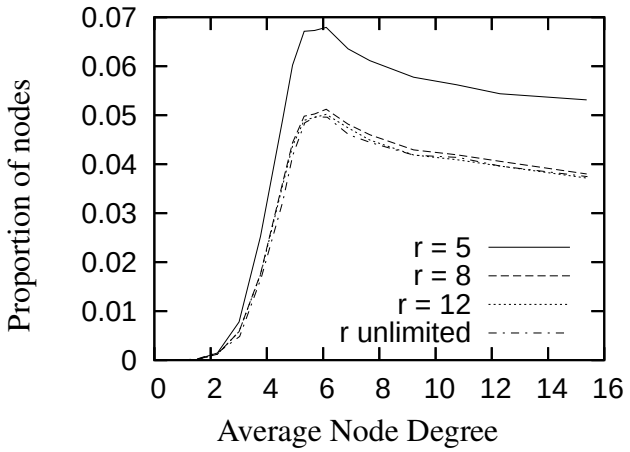


Figure 4-11: Proportion of nodes with conflict hulls.

ability of greedy forwarding success for dense networks brings down the average significantly in Figure 4-9.

4.2.2 Effect of Convex Hull Representation

In Sections 4.2 and 4.2.1, we did not limit r, the maximum size for the convex hulls. As shown in Figure 4-11, limiting the number of points used to represent the convex hulls will cause intersections between convex hulls to be more common, thereby generating more conflict hulls. In fact, in networks of average node degree 6, some 7% of the nodes will have conflict hulls.

When we repeated the measurements for routing stretch for different values of r, we found that surprisingly, *the value of r has a negligible effect on both path and hop stretch*, i.e., the stretch for $r = 5$ was virtually indistinguishable from the stretch when r is unlimited. We found that the reason for this is that although the hulls are bigger when r is limited and there are more intersections between the convex hulls of sibling nodes, intersections do not necessarily degrade routing performance as long as they are not particularly large or if they occur close to the leaves of a tree. In fact, intersections that do not contain any nodes do not affect routing performance.

It seems that even when r is reduced and the size of the hulls is increased, it is still relatively rare for nodes to fall into the intersections of hulls. Furthermore, intersections only matter when a packet is not forwarded in greedy mode. Since GDSTR forwards packets in greedy mode more than 75% of the time in our experimental setup and only occasionally switches to tree forwarding mode, it is not completely surprising that r does not seem to affect the aggregate routing performance.

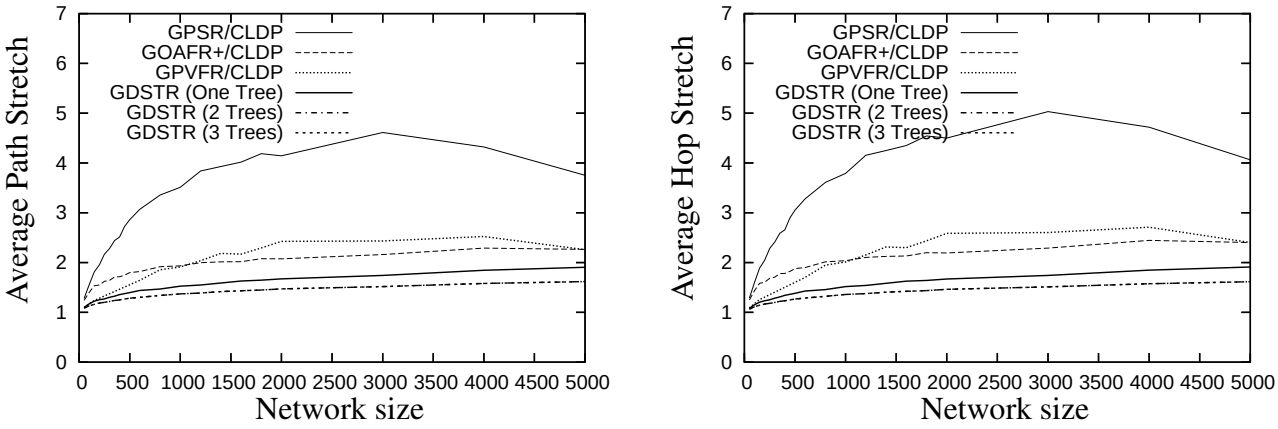


Figure 4-12: Plots of routing stretch for sparse UDG networks (average node degree 6.5).

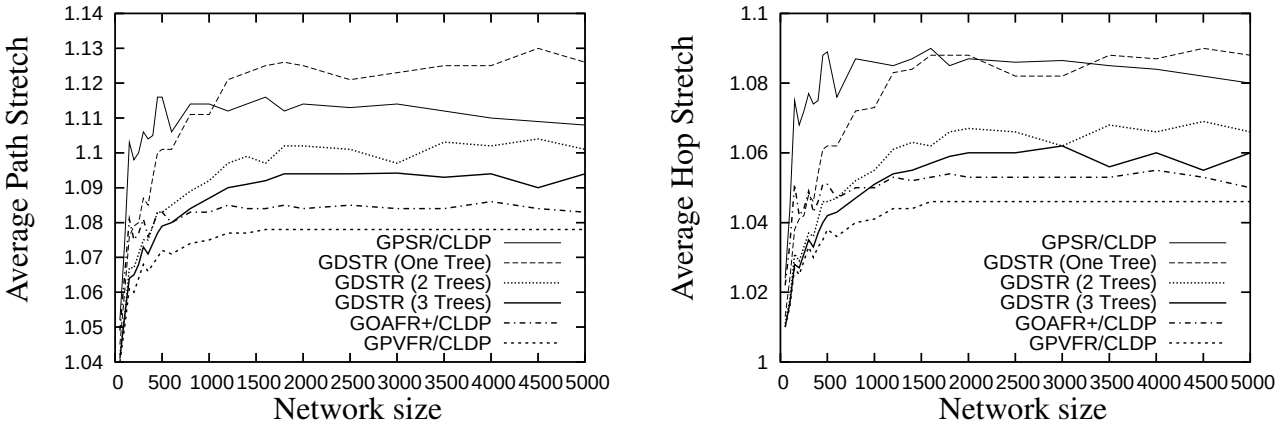


Figure 4-13: Plots of routing stretch for dense UDG networks (average node degree 12).

4.2.3 Scaling Up

In this section, we evaluate the performance of GDSTR and compare it to that for the face routing algorithms for networks of approximately constant density up to 5,000 nodes in size.

Effect of Network Density. The routing performance of sparse and dense networks is shown in Figures 4-12 and 4-13, respectively. As expected, the routing performance for dense networks is significantly better for all algorithms.

As shown in Figure 4-12, GDSTR performs significantly better than the geographic face routing algorithms in sparse networks. GPVFR performs slightly better than GOAFR+ when the networks are relatively small. For larger networks, GOAFR+ tends to perform better. The reason for this is that our implementation of GPVFR only maintains information about nodes that are up to 4 hops away along the planar faces. When the networks are small, it often able to choose a better forwarding direction using this information; when the networks are large, the voids tend to be

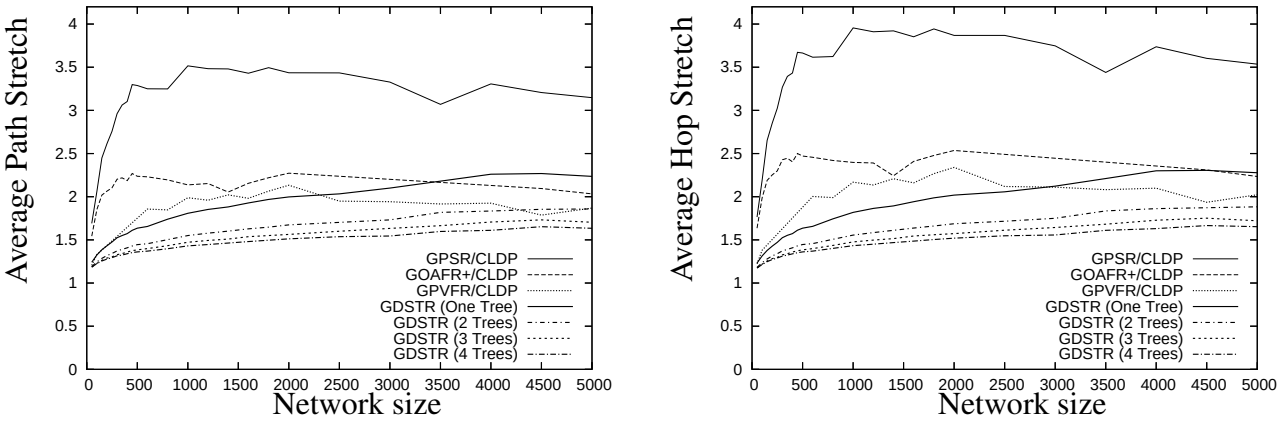


Figure 4-14: Plots of routing stretch for networks with high obstacle density (average node degree 6).

larger and the face information is less predictive. It is also interesting to note that while there is a noticeable improvement in performance when GDSTR uses two trees instead of one, the routing performance between GDSTR with 2 and 3 trees is almost indistinguishable.

In dense networks (see Figure 4-13), GOAFR+ and GPVFR achieve better routing performance than GDSTR since they are able to traverse voids almost perfectly, while tree traversal in GDSTR occasionally requires a short detour up the tree. It was somewhat surprising that GPSR did not perform quite as well as we expected. It turns out that using a fixed right-hand rule occasionally causes GPSR to traverse the perimeter of the network in the "wrong" direction. Such a mistake is very costly; hence, GPSR does not perform well on average. As the network size increases, the probability that a packet that is routed on the perimeter decreases. Thus, hence the routing performance of GPSR seems to improve slightly with increasing network size.

Note that the scale of the y axis for Figure 4-13 has been adjusted, since the stretch is very good (between 1.07 and 1.13) for all the algorithms and the difference between them is not significant.

Obstacles. The hop stretch for networks with high and low obstacle densities are shown in Figures 4-14 and 4-15, respectively. Sample networks from each set are shown in Figures 4-4 and 4-5, respectively. The key difference between the networks from the two sets lies in the sizes of the voids.

These results demonstrate that for sparse networks with large voids, the routing performance of GDSTR is consistently better than that for existing face routing algorithms, while for denser networks with small voids, existing face routing algorithms can sometimes achieve a slightly lower stretch. As mentioned, the reason for this is that extremal-rooted trees do not approximate voids quite as well when there are many hops between the leaf nodes and the root.

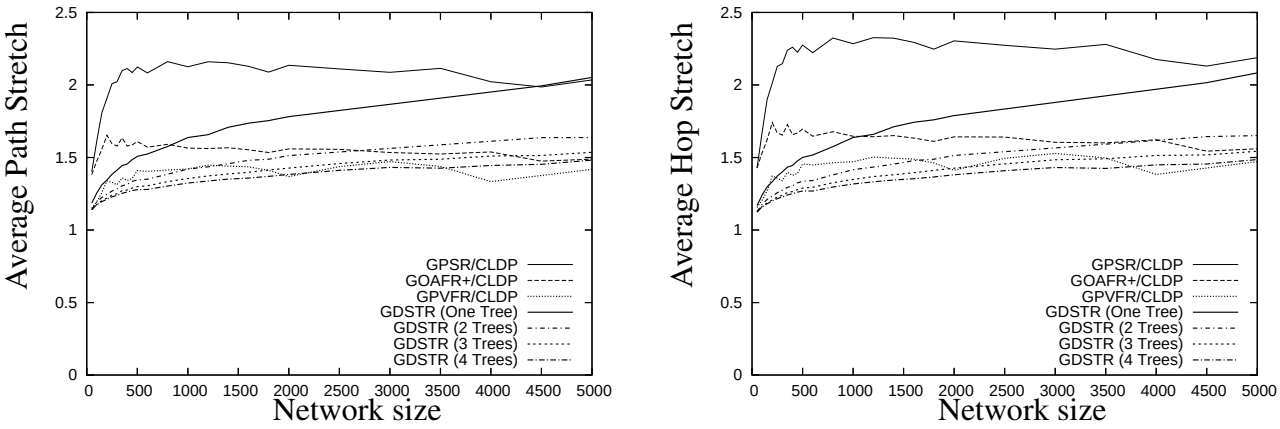


Figure 4-15: Plots of routing stretch for networks with low obstacle density (average node degree 7).

4.3 Costs

In this section, we present experimental results for the costs of GDSTR. Our main concern is with bandwidth, since it is likely to be a limiting factor in radio networks. However we begin by discussing the storage costs of our system, since storage concerns were once a primary motivation for geographic routing algorithms.

4.3.1 Small Networks

First, we evaluate how costs scale with network density (average node degree) by considering networks up to 500 nodes in size.

Storage Costs. Figures 4-16 and 4-17 show the average and maximal storage required by any nodes over the range of densities investigated, respectively. We assume that a set of coordinates and a node identifier are 8 and 12 bytes in size, respectively. We can see from the figure that the maximum is about 1,000 bytes. This amount of storage is hardly a concern for modern sensor devices like the Mica2 [83], which has 128K of program memory and 512K of flash RAM.

The figure shows the storage requirements when GDSTR uses two hull trees. In general, GDSTR with two hull trees requires more than twice as much storage on average as existing face routing algorithms at low network densities. However, as the network density increases, the storage requirement of the neighbor set becomes comparable to the storage requirement for the hull trees.

The figure also shows the effect of limiting the size of the convex hulls, r, on storage. These results show that by limiting r, there is negligible effect on the average storage requirement. When $r = 5$, we can reduce the maximum storage required by up to 30% at low network densities. Since the associated storage costs are small, we find that *in practice, we can set a large limit on the maximum size of the convex hulls*.

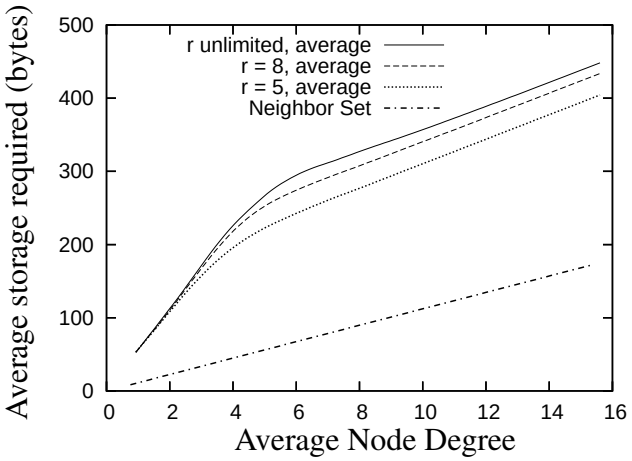


Figure 4-16: Average amount of routing state stored at each node for various values of r for GDSTR with two hull trees.

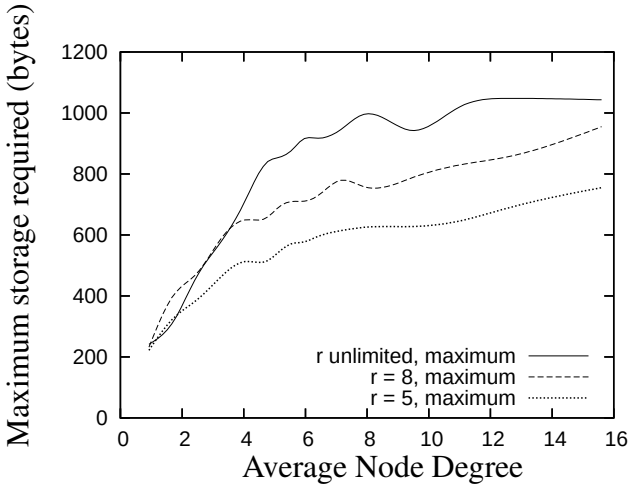


Figure 4-17: Maximum routing state stored at any node for various values of r for GDSTR with two hull trees.

Bandwidth Costs. In the following experiments measuring the costs of stabilization and maintenance, we compare the cost of GDSTR with the cost of building and maintaining a planar graph with CLDP. The reason for this is that the other associated costs of existing geographic face routing algorithms [39, 47, 52] are small relative to the cost of CLDP. The costs for GPSR [39] and GOAFR+ [47] are negligible; GPVFR [52] does impose some maintenance cost on the network to maintain its face information, but the cost is also small relative to CLDP.

We quantify the bandwidth costs for each algorithm in terms of the number of messages sent or forwarded by nodes during stabilization and repair. For GDSTR, we count the number of *keepalive* messages that contain new hull information. For CLDP, we count the probe messages.

The average size of these messages is shown in Figure 4-18. As shown, the relative sizes of the CLDP probes and GDSTR broadcast messages are comparable. CLDP probes are largest in the critical region (average node degree 4 to 8) because the probes contain the points on the faces and these networks tend to have the largest perimeters.

Startup Costs. To investigate the startup costs for a network, we start all the nodes in the network at approximately the same time and measure the average number of messages sent by each node before the network converges.

CLDP involves a locking mechanism, so a configuration involving binary backoff will likely be able to optimize its startup performance. We do not know the optimal parameters, so we used the following simple probe model: all nodes have the same probing period with a 20% jitter (to avoid synchronization), and at the start of each period, a node probes all the links that require probing. If a reply is received, it is acted upon immediately. If a probe message is dropped because it encounters a *locked* edge, the node will resend the probe during the next probe interval. A node is deemed to have converged when all its links are marked either *dormant* or *non-routable* and it does not have to initiate any more probes during the next probing interval. Similarly, a GDSTR node is

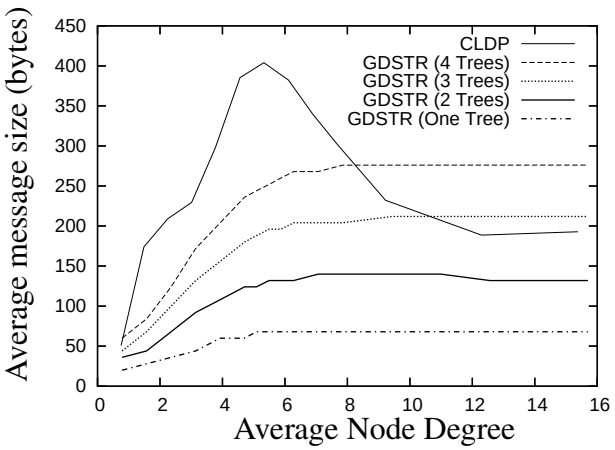


Figure 4-18: Comparing the sizes of CLDP probes and GDSTR broadcast messages.

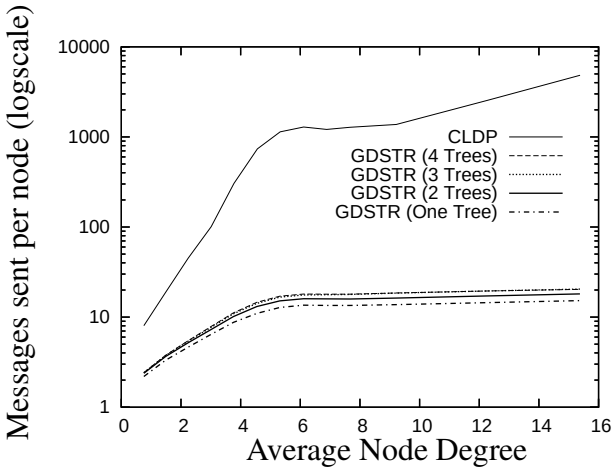


Figure 4-19: Packets sent or forwarded per node for stabilization.

deemed to have converged when it no longer needs to broadcast hull information in its *keepalive* messages.

In Figure 4-19, we plot the average number of messages that would have sent or forwarded by each node during stabilization, with all nodes starting up without any state. As shown, CLDP sends about two orders of magnitude more messages than GDSTR before the network stabilizes. For node degrees between 6 and 14, each CLDP node will send about 1,500 messages; for GDSTR, the corresponding number is slightly more than 10 messages.

We see in Figure 4-19 that the startup costs for CLDP increase rapidly until node degree 6 and taper off thereafter. This is because below node degree 6, the experimental topologies usually consist of several small, disjoint networks. As the node degree increases, the networks become larger and more tree-like, and they tend to have larger perimeters that are costly to probe. After a critical density of about node degree 6, the networks become more connected, and their perimeters are somewhat more convex. The probing costs do not increase much at this stage with increasing density, because the network perimeters either stay relatively constant or may even shrink slightly. However, the probing costs for CLDP are also proportional to the number of edges in the network graph, so when the network density increases beyond node degree 8, the increase in the number of edges (links) becomes the dominating factor, and we again see an increase in the CLDP probing cost.

Figure 4-19 also shows that the number of messages required per node by GDSTR plateaus at node degree 6 and increases only slightly thereafter. The reason for this is that the number of update messages that GDSTR requires is a function of the network diameter D. It turns out that since the nodes have unit radio range and are all contained within a 10×10 unit square, D is approximately constant for densities higher than node degree 6.

Incremental Costs. To quantify the bandwidth required to update routing state when a new node joins and to repair the routing state after a node fails, we measure the costs of adding and removing

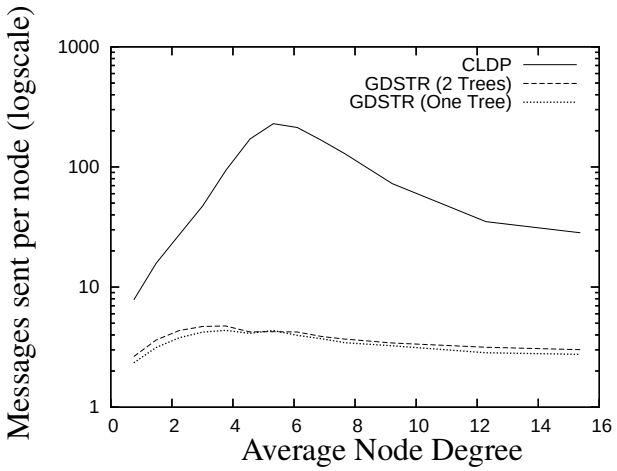


Figure 4-20: Packets sent or forwarded per node when a new node joins (averaged over only the nodes that are affected by a node join or failure).

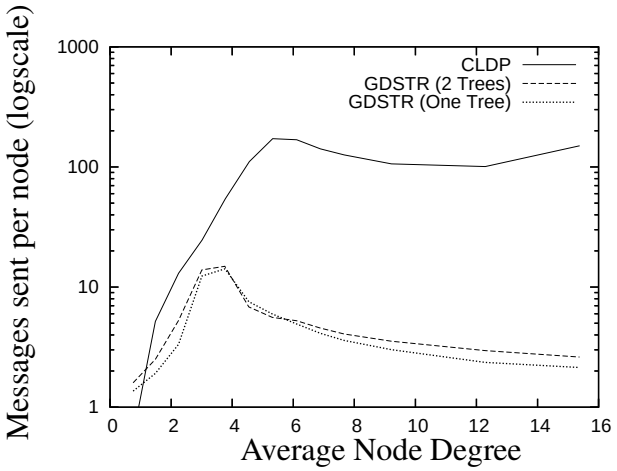


Figure 4-21: Packets sent or forwarded per node when an existing node fails (averaged over only the nodes that are affected by a node join or failure).

a single node from a stable network as follows: after a network has stabilized, we remove one node and count the number of messages sent per node. After the network has stabilized once again, we add the removed node back to the network and take the same measurement. We repeat this process on 20 randomly chosen nodes for each network and average the results to obtain the average cost per node change in each network.

In Figures 4-20 and 4-21, we plot the number of messages that are sent per node in order for the system to converge after one node join or departure, respectively. Note that these figures are averaged over only the nodes that are affected by a node join or failure. The peak for CLDP is about 200 messages per node at a node degree of 6. When a node joins the network for CLDP, new links are created between it and all its immediate neighbors, and these new links are probed independently by the various nodes; when a node fails, its adjoining neighbors will probe all the adjacent links that are marked *non-routable*, in case there is the need to revive a *non-routable* link to restore connectivity. We see that the costs for node joins and departures are comparable, except for high network densities. The likely reason for this is that at high densities, node failures are significantly more costly than node joins, because more links are re-probed for node failures and the number of such links is proportional to the node degree.

The join costs for GDSTR are uniformly low at approximately 3 messages per node; the repair costs after a node failure are highest in the region with node degree between 2 and 6 and fall gradually with increasing node density. The latter is because the likelihood of failure for an intermediate node is much higher at lower node densities (with a maximum of approximately 15 messages), whereas for high node densities, node failures are more likely to occur at the leaf nodes.

The bandwidth costs for updating a planar graph with CLDP incrementally are significantly lower than that for *en masse* stabilization at startup. In fact, they can also be interpreted as the cost to stabilize for CLDP when the network grows one node at a time.

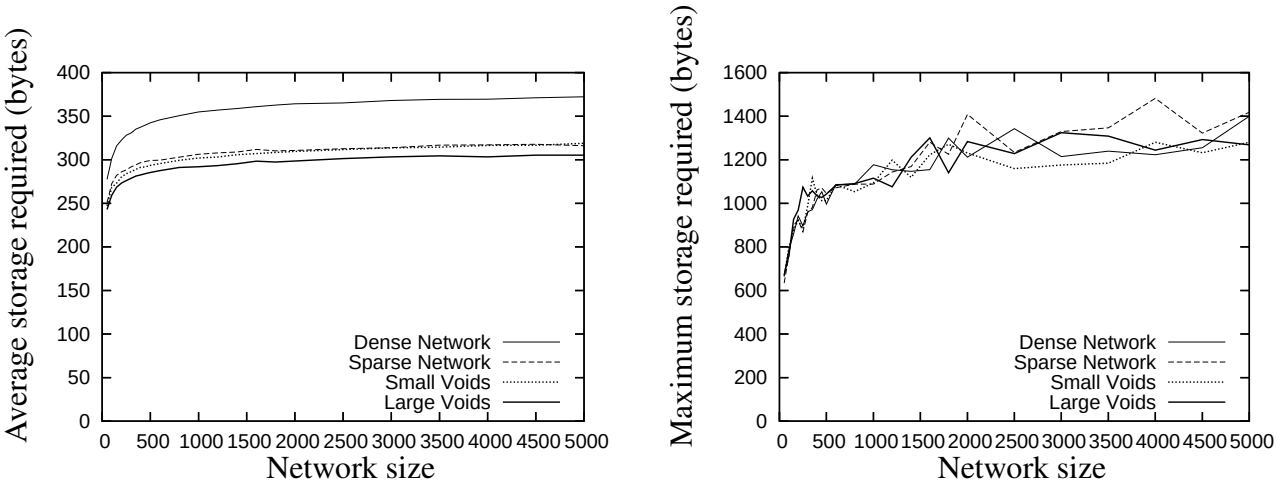


Figure 4-22: Amount of routing state stored at each node for networks with up to 5,000 nodes.

Discussion. We chose to evaluate startup and maintenance in terms of message count rather than convergence time, because the system parameters for both CLDP and GDSTR can be tuned to achieve faster convergence. For example, the probing rate for CLDP can be increased and for GDSTR, nodes can broadcast update messages as and when there are changes in its hull trees instead of waiting to piggyback the information on *keepalive* messages. The total amount of information to be transmitted to bring the routing information to a consistent state is however the same in all cases. In fact, we can work out the fundamental limit on convergence time by dividing the volume of messages to be transmitted by the maximal achievable bandwidth of the radios.

4.3.2 Scalability

In this section, we summarize what happens to cost as we scale networks up to 5,000 nodes for the networks described in Section 4.2.3. We refer to the UDG networks with average node degrees 6.5 and 12 as "Sparse" and "Dense," respectively, and to the non-UDG networks with high and low obstacle densities as "Large Voids" and "Small Voids," respectively.

Storage Costs. As shown in Figure 4-22, the average storage required per node is somewhat independent of network size and is about 300 bytes over the entire range of network topologies that we investigated; the maximal storage requirement increases steadily with network size, but it does not exceed 1,500 bytes even when the network size is scaled up to 5,000 nodes.

Bandwidth Costs. In Figures 4-23 and 4-24, we plot the total number of messages required by the entire network for stabilization for the UDG and non-UDG networks from a fresh state. Again, we see that CLDP requires about two orders of magnitude more packets for stabilization. Because the y axes of Figures 4-23 and 4-24 are in logscale, it may not be quite apparent, but it turns out that the total number of packets required for GDSTR seems to grow linearly with network size. Also, the total number of packets required for GDSTR seems to be somewhat independent of the topology.

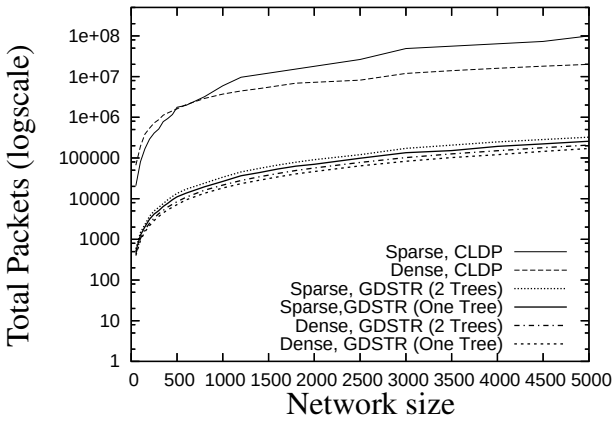


Figure 4-23: Total packets sent or forwarded during stabilization for UDG networks.

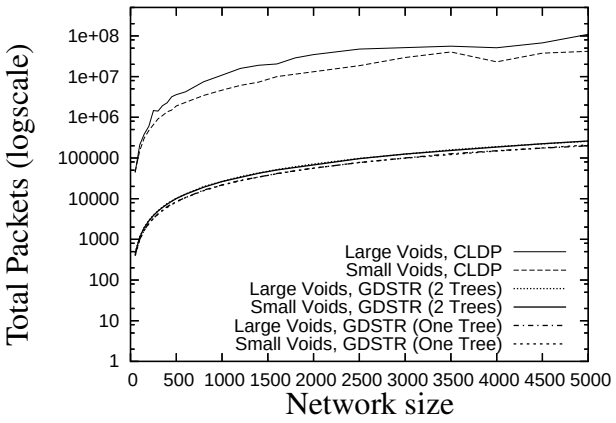


Figure 4-24: Total packets sent or forwarded during stabilization for non-UDG networks with cross-shaped obstacles.

For large networks, the initial startup costs where all nodes start from a fresh state are not important, since large networks will have to be turned on incrementally. In Figures 4-25 to 4-28, we plot the total number of messages (summed over all nodes) required to repair routing state when a node joins the network. For both CLDP and GDSTR, the average number of messages sent per node increase with network size. Node joins for GDSTR are relatively cheap, while repairs after network departures are significantly more costly, but still require almost an order of magnitude fewer messages than CLDP. Individual node joins and network repairs only affect a fraction of the nodes in the network (typically less than 20% and decreasing with increasing network size).

The reason node joins are cheap is that when the number of nodes is large, the probability that a new node will be a leaf node in both trees is high. The coordinates of leaf nodes often fall within existing convex hulls and propagate only a few hops up the hull tree and are somewhat localized. On the other hand, node departures will often cause changes to both the structure of the existing hull trees and to the convex hulls and are thus significantly more disruptive. Similarly, if a new extremal node is added to the network, a hull tree may have to be re-built. However, such an event is likely to be rare. It is likely that the repair algorithm can be optimized to reduce the number of messages exchanged by introducing some selective damping, i.e., we can allow local changes to propagate much faster than distant changes.

4.4 GDSTR+

In this section, we evaluate the performance of GDSTR+ by comparing it to GDSTR. Since GDSTR+ stores more state than GDSTR (with two hull trees), we compare GDSTR+ with one and two local hull trees to GDSTR with up to four hull trees. The width of the grids of the local trees in our implementation of GDSTR+ is five times the (unit) radio range.

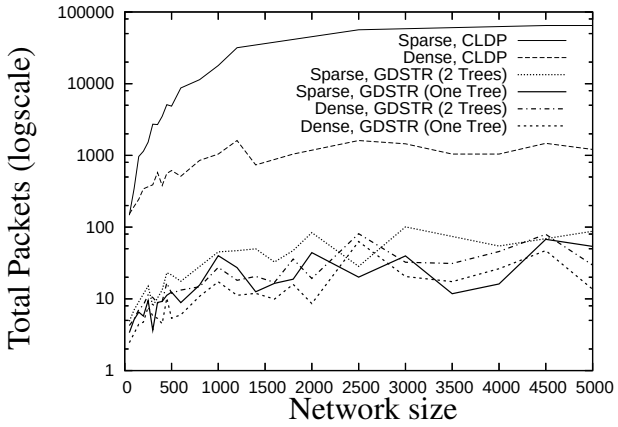


Figure 4-25: Total packets sent or forwarded when a new node joins for UDG networks.

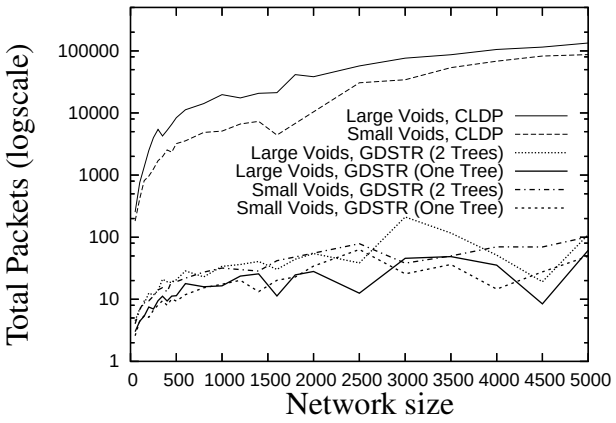


Figure 4-26: Total packets sent or forwarded when a new node joins for non-UDG networks with cross-shaped obstacles.

Typically, GDSTR uses only two global hull trees. To understand the tradeoffs, we compare the performance of GDSTR+ with one and two local trees (in addition to two global trees) to that for GDSTR with two to four global hull trees to determine whether we would see the same performance for GDSTR with three or four global trees.

4.4.1 Routing Performance

The stretch performance for networks with large and small voids is shown in Figures 4-29 and 4-30, respectively. The routing performance for GPVFR [52], the best face routing algorithm available, is also plotted for reference.

In Figure 4-29, we see that for networks with large voids, GDSTR performs better than GPVFR and GDSTR+ performs marginally better than GDSTR with the corresponding number of trees, i.e. GDSTR+ with one local tree is better than GDSTR with three global trees, and GDSTR+ with two local trees is better than GDSTR with four local trees.

In Figure 4-30, we see that the relative difference in performance between the hull-tree-based algorithms for networks with small voids is smaller than that for networks with large voids. Note also that the scale of the y-axis has changed between Figure 4-29 and Figure 4-30 to magnify these differences. It is noteworthy, however, that the relative performance for GPVFR is significantly better compared to that in Figure 4-29. In particular, we see that GPVFR achieves better routing performance than GDSTR with two trees. GDSTR+ is able to surpass the routing performance of both GDSTR and GPVFR. For 5,000-node networks, GDSTR+ with two local hull trees achieves a 17% improvement in stretch over GDSTR (with two global hull trees) and 8% lower stretch than GPVFR.

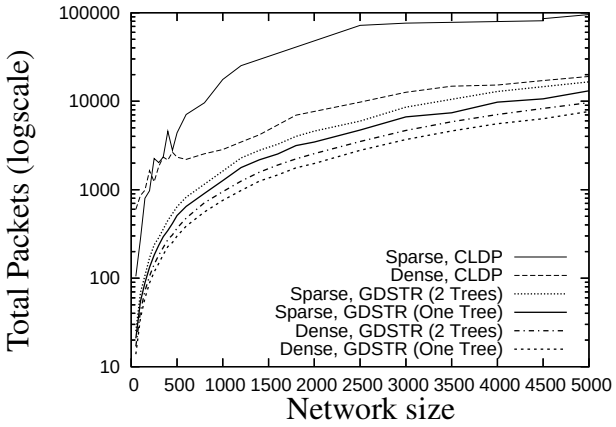


Figure 4-27: Total packets sent or forwarded when an existing node fails for UDG networks.

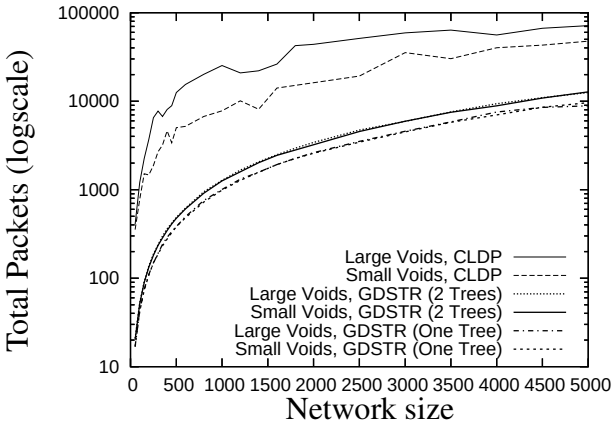


Figure 4-28: Total packets sent or forwarded when an existing node fails for non-UDG networks with cross-shaped obstacles.

We plot the stretch performance for sparse and dense networks in Figures 4-31 and 4-32, respectively. For the sparse networks in Figure 4-31, we find that GDSTR+ with one local tree performs slightly worse than GDSTR with three global hull trees, and GDSTR+ with two local trees performs slightly worse than GDSTR with four global hull trees. The corresponding hop stretch for the face routing algorithms is not plotted because it is greater than two. The reason GDSTR+ performs slightly worse than the corresponding version of GDSTR is that a packet can sometimes be forwarded in the wrong direction based on local information. In such instances, a packet would have to backtrack and hence, incur additional routing overhead. With tree traversal on global hull trees, packets are never forwarded in the wrong direction. It should be noted, however, that the difference in routing performance is less than 1%.

We see from the results in Figure 4-32 that, as expected, that GDSTR+ performs better than GDSTR. In such networks, the voids are relatively small, and hence, the information contained in the local hull trees are often sufficient to route packets around them in greedy-hull mode. Because the local trees are smaller and the roots are closer to the voids, the uptree region (see Section 3.4.1) for local trees tends to be smaller than that for global trees and the detour that packets may have to take to route up a tree is smaller, so void traversal is more efficient on average.

4.4.2 Costs

We had earlier shown that GDSTR has very low costs in terms of both storage requirements and maintenance bandwidth [51]. It turns out that the storage requirement for GDSTR+ with two local hull trees is approximately equal to the storage requirement for GDSTR with four global hull trees, and is less than 600 bytes on average across the entire range of networks studied.

While the size of the GDSTR+ maintenance messages is somewhat larger than that of the maintenance messages for GDSTR, since GDSTR+ needs to maintain two additional local trees, these

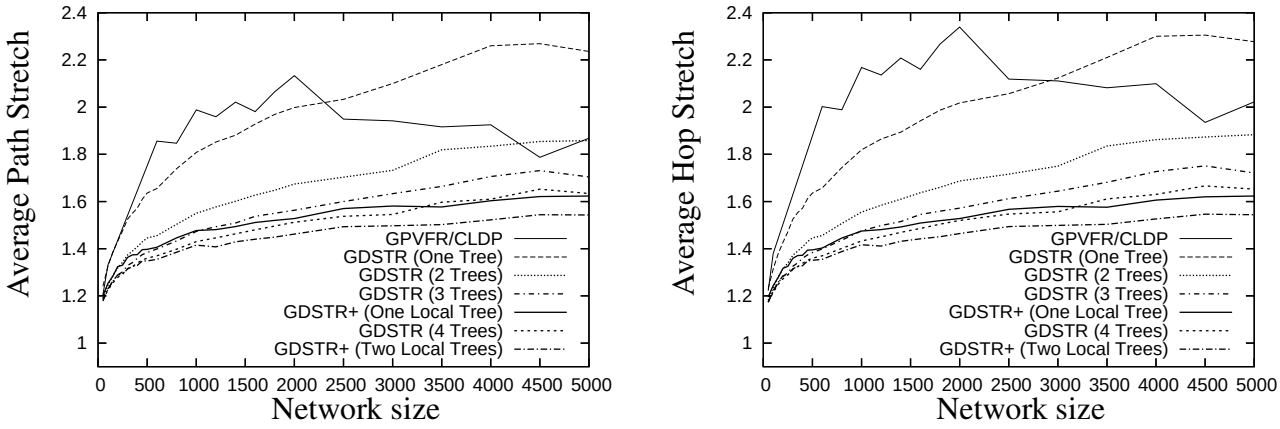


Figure 4-29: Plots comparing the routing performance of GDSTR to GDSTR+ for non-UDG networks with large voids (average node degree 6).

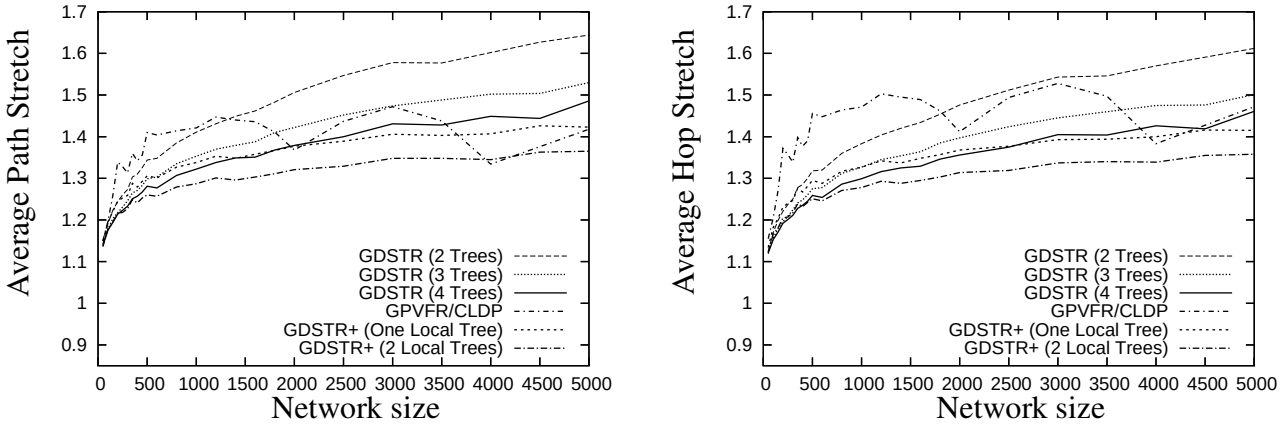


Figure 4-30: Plots comparing the routing performance of GDSTR to GDSTR+ for non-UDG networks with small voids (average node degree 7).

messages are small (< 500 bytes). The time taken for stabilization is not increased, however, since the local hull trees are shallow and it only takes a few hops to build them, while messages have to propagate across the entire network to build the global trees.

4.5 Geocast

In our evaluation of geocast efficiency, we do not compare our algorithm with previous algorithms since our objective is not to demonstrate that our hull-tree-based geocast algorithm is necessarily better, but that given an existing GDSTR/GDSTR+ deployment, geocast can be implemented relatively efficiently.

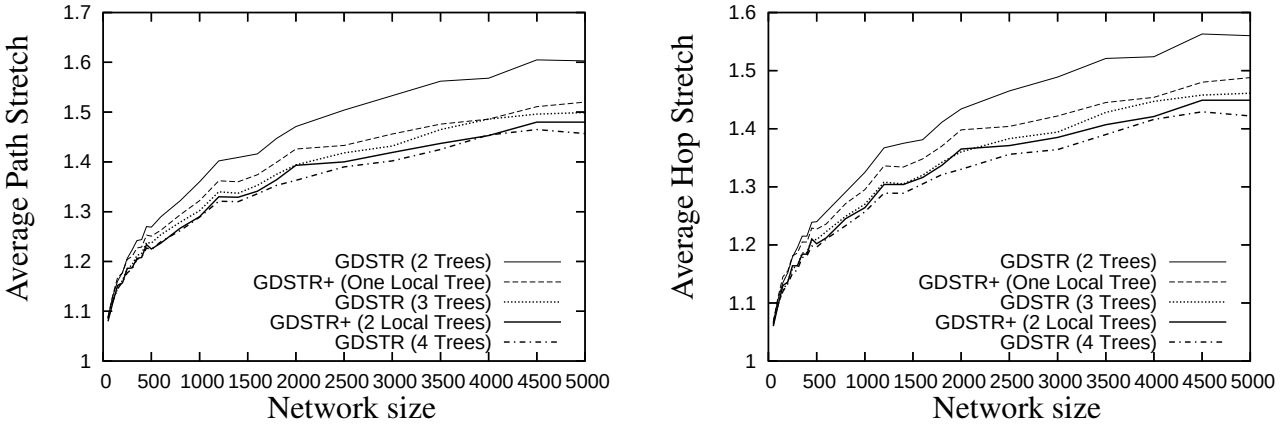


Figure 4-31: Plots comparing the routing performance of GDSTR to GDSTR+ for sparse UDG networks (average node degree 6.5).

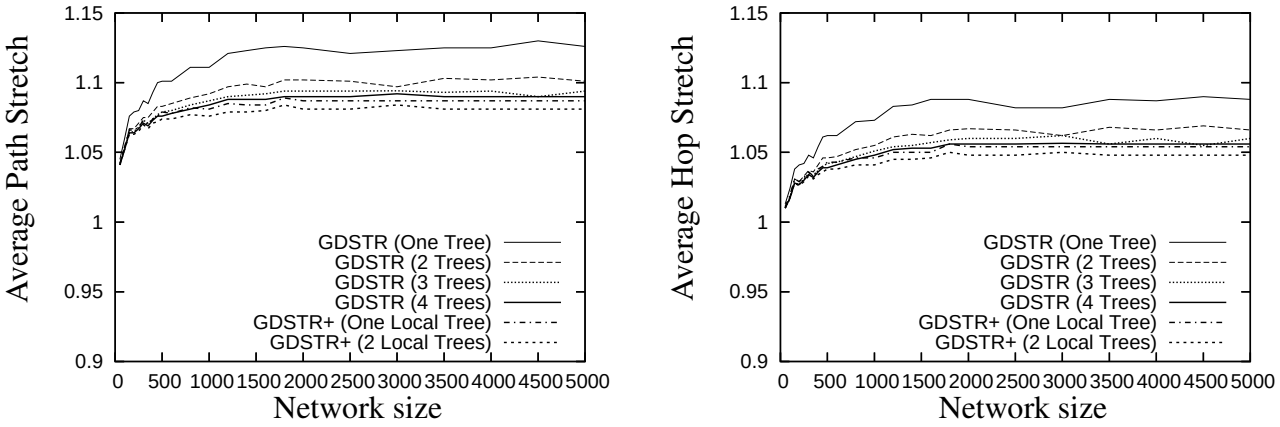


Figure 4-32: Plots comparing the routing performance of GDSTR to GDSTR+ for dense UDG networks (average node degree 12).

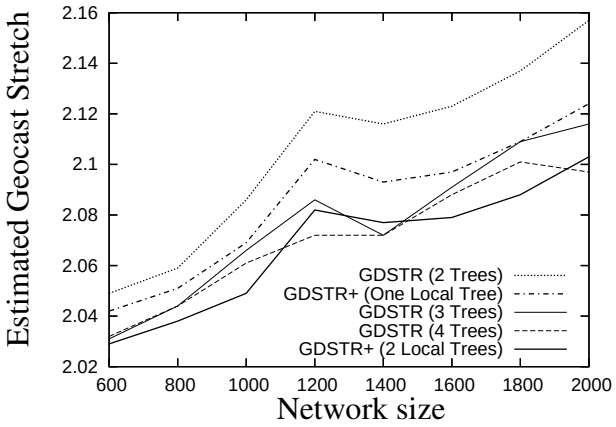


Figure 4-33: Estimated geocast stretch for sparse UDG networks (average node degree 6.5).

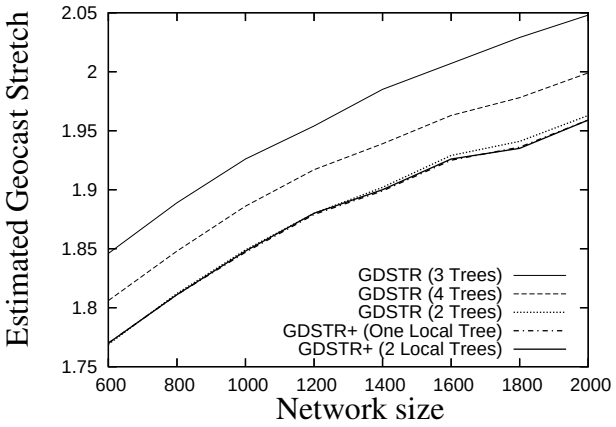


Figure 4-34: Estimated geocast stretch for dense UDG networks (average node degree 12).

We evaluate the performance of our geocast algorithm by measuring the number of geocast packets that is required to deliver a geocast message successfully to a unit square. To normalize across different network sizes, we define a ratio, *Geocast Stretch*, which is the ratio of the number of packets required by a geocast algorithm to the minimum number of packets required. The latter is the number of edges on the minimum-spanning tree containing the source and all the destination nodes.

We define h to be the sum of the minimum number of hops between the source and any node in the target region and the number of nodes in the target region minus one. It is easy to see that it is not possible to deliver the geocast message in fewer than h hops, since if we could route a packet to the node that is closest to the source and then take one hop to reach each of the remaining nodes in the target region, we have an optimal solution (though such a solution may not exist in general). Since it is costly to compute the minimum-spanning tree, we determine an upper bound on the maximum number of packets required by taking the ratio of the actual number of packets to h. We call this upper bound the *Estimated Geocast Stretch*.

We plot the results for sparse networks in Figure 4-33. In general, we can improve performance (i.e., achieve a lower geocast stretch) by having more trees and using a combination of two global and two local trees (GDSTR+) seems to work best. Using only one local tree (in addition to two global ones) seems to noticeably worse than using three global trees. This is likely because with only one forest of trees, it is relatively common that the target region does not fall completely within a GDSTR+ grid square, and hence, we have to resort to using a global tree for the broadcast.

We plot the results for sparse networks in Figure 4-34. The results with local trees are marginally better than those with only two global trees. Surprisingly, the results with three and four global trees are worse. We suspect that the latter is due to an artifact in the experimental setup. The effective difference in the results is small: it translates to a difference in two or three packets for each geocast instance.

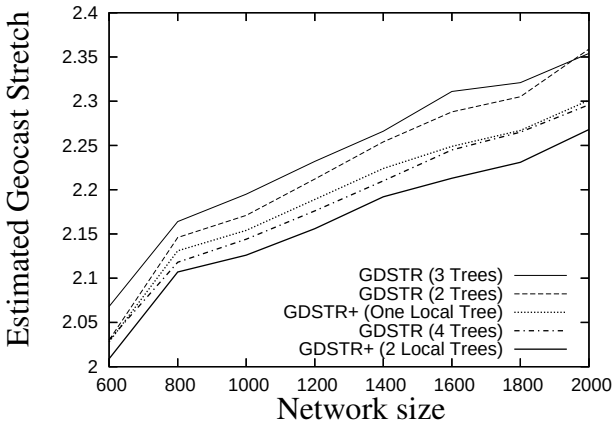


Figure 4-35: Estimated geocast stretch for non-UDG networks with high obstacle density (average node degree 6).

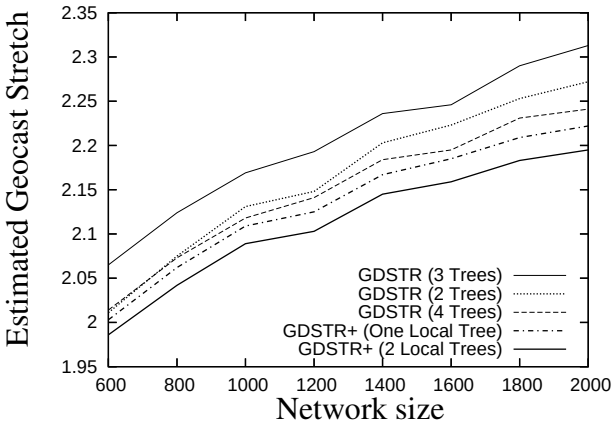


Figure 4-36: Estimated geocast stretch for non-UDG networks with low obstacle density (average node degree 7).

The results for networks with obstacles are shown in Figures 4-35 and 4-36. These results show that obstacles have a marginal effect on geocast performance. The fact that geocast with three global hull trees performs marginally worse than the rest is likely an artifact of the experimental set up, since the grid squares and target regions are squares that are aligned with the x and y axes. On the other hand, the rays that are used to choose the roots for the three global trees are not aligned in the same way. This is likely to have an effect on the orientation of the resulting hull trees.

Overall, geocast with local hull trees (GDSTR+) incurs 10% less overhead than geocast with only two global hull trees in sparse networks with large voids. All the variants seem to perform equally well in dense networks. These results also suggest that we can likely implement geocast using hull trees with no more than two times the minimum number of messages (since the Estimated Geocast Stretch is a loose upper bound).

4.6 Summary

Our simulations show that GDSTR achieves a peak improvement of about 20% in terms of path and hop stretch over the best available geographic face routing algorithm in situations where dead ends are common, and that GDSTR performance is consistently good over a wide range of network densities and sizes.

Simulation also shows that GDSTR generates significantly less maintenance traffic than CLDP. GDSTR sends two orders of magnitude fewer messages to build its trees initially than what CLDP sends to construct a planar subgraph, and GDSTR's communication when maintaining existing trees is one order of magnitude less than that of CLDP.

Finally, GDSTR+ is able to achieve up to a 17% improvement in stretch performance over GDSTR and an 8% lower stretch than GPVFR, the best existing face routing algorithm, for dense networks with small voids. We have also shown than we can implement geocast with 10% less overhead with GDSTR+ hull trees when compared to that for GDSTR (with two hull trees). Furthermore, our algorithm will likely require no more than twice the minimum number of messages for networks smaller than 5,000 nodes in size.

Chapter 5

Greedy Embedding Spring Coordinates (GSpring)

In this chapter, we describe Greedy Embedding Spring Coordinates (GSpring), an online virtual coordinate assignment algorithm that incrementally adjusts virtual coordinates to increase the convexity of voids in a virtual routing topology, thereby increasing the success rate of greedy forwarding and consequently improving geographic routing performance.

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, or t is a neighbor of s [65]. 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 [87], 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*.

5.1 Preliminaries

In this section, we describe the background of the geometric objects and properties that are employed by GSpring to incrementally improve the greedy forwarding success rate of virtual coordinates.

5.1.1 Region of Ownership

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 the network connectivity graph. For example, in Figure 5-1(a), the region

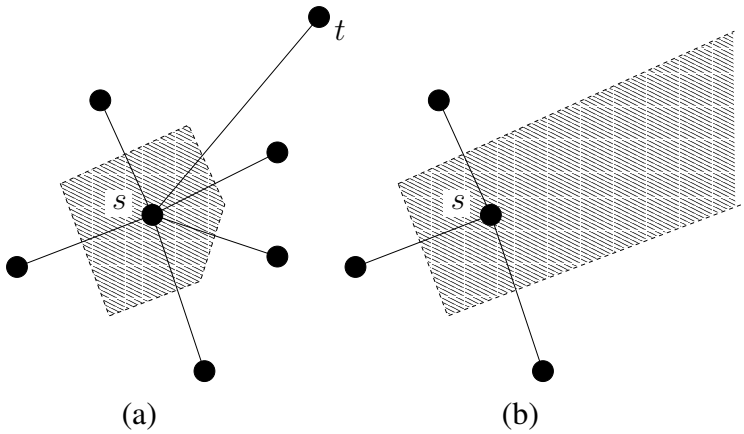


Figure 5-1: Regions of ownership for the node s is shaded.

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 Figure 5-1(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. In contrast, the regions of ownership for two nodes can sometimes intersect because the network can have arbitrary connectivity and nodes have only local (one-hop) knowledge. Figure 5-2(a) shows a topology for which a node s has another node t in its region of ownership.

GSpring uses the following result to incrementally adjust coordinates to increase the greedy forwarding success rate of an embedding:

Theorem 3 *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 we have 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. ■

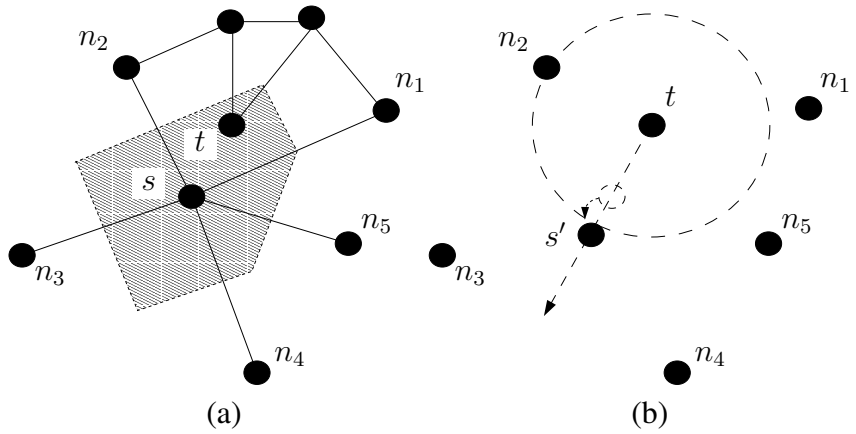


Figure 5-2: Required adjustment to move node s so that node t is no longer in its region of ownership. Original region of ownership for s is shaded in gray.

5.1.2 Adjustment of Coordinates to Increase Greedy Forwarding Success Rate

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 Figure 5-2(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). This is illustrated in Figure 5-2(b). 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. While this describes 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 will 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.

5.1.3 A Greedy Embedding Does Not Always Exist

It is known that a greedy embedding exists for all graphs that have a *Hamiltonian path*. A Hamiltonian path is a simple path through a graph that includes every vertex in the graph. The embedding is trivial – it is one with all the nodes laid out in a straight line in sequence along the Hamiltonian path. Unfortunately, the problem of determining if an arbitrary graph has a Hamiltonian path is known to be NP-complete, so it is not likely that this condition will be useful for determining if a greedy embedding exists for an arbitrary network in a distributed setting.

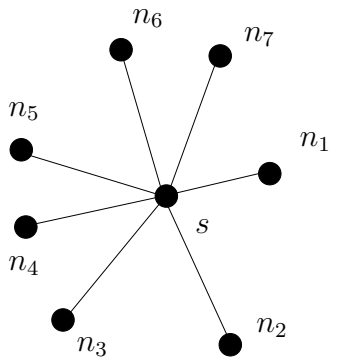


Figure 5-3: Network for which a greedy embedding does not exist.

It is easy to see that given a graph G, with vertex set $\mathcal{V}$ and edge set $\mathcal{E}$, if a greedy embedding exists for subgraph G' with vertex set $\mathcal{V}$ and edge set $\mathcal{E}' \subseteq \mathcal{E}$, the same greedy embedding will work for G.

However, the converse is false, i.e., if a greedy embedding exists for G, it is not always true that a greedy embedding will exist for G'. An example is the simple star-like graph shown in Figure 5-3. A greedy embedding in the Euclidean plane does not exist for this graph.

Proof: Assume that a greedy embedding exists. Consider the angles made by the neighbors of s at s. Since there are seven neighbors, it is most certainly the case that there exists at least one pair of nodes n_i and n_j such that the angle subtended at s is less than $\frac{2\pi}{7}$. If $|sn_i| = |sn_j|$, then $|n_i n_j| < |sn_j|$, which means that a packet at n_i with destination n_j is undeliverable. Without loss of generality, assume therefore $|sn_i| < |sn_j|$. Again $|n_i n_j| < |sn_j|$, and a packet at n_i with destination n_j is again undeliverable. ■

5.2 Overview of GSpring

In this section, we describe our online incremental virtual coordinate assignment algorithm, *Greedy Embedding Spring Coordinates (GSpring)*.

GSpring implements the idea described in Section 5.1.2, i.e., to make concave voids more convex by having each node gradually move away from the nodes in its conflict set. To do so, each node sends a geocast message to discover the nodes in its conflict set. Once the nodes in the conflict set are determined, a node adjusts its coordinates so as to "move away" from the nodes in its conflict set.

We model the nodes as particles and the required adjustment as a repulsion force acting between particles. While the repulsion arising from conflict sets is the critical factor that improves the greedy forwarding success rate, we use spring forces to keep the local relationship between nodes

consistent. Hence, GSpring acts as a distributed relaxation of a system of particles acted upon by a set of springs and some repulsion forces.

The final coordinates are those to which the system converges at equilibrium. Since GSpring simulates a spring system, nodes can in principle oscillate forever. Like others [11], we achieve stability by introducing *damping* and *hysteresis*.

5.2.1 Spring Rest Length

Each link between two neighboring nodes i and j is represented with a spring of rest length l_{ij}. 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 which are not shared by the other, $\mathcal{S}_i$ and $\mathcal{S}_j$, respectively. Then,

$$r_{ij} = \begin{cases} 1, & \text{if } |\mathcal{S}_{ij}| + |\mathcal{S}_i| = 0 \\ & \text{or } |\mathcal{S}_{ij}| + |\mathcal{S}_j| = 0 \\ \frac{|\mathcal{S}_{ij}|}{|\mathcal{S}_{ij}|+|\mathcal{S}_i|+|\mathcal{S}_j|}, & \text{otherwise} \end{cases} \tag{5.1}$$

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:

$$l_{ij} = l_{min} + (1 - r_{ij})(l_{max} - l_{min}) \tag{5.2}$$

where l_{min} and l_{max} are constants such that $l_{min} < l_{max}$. In our implementation, we set $l_{max} = 100$ (equal to the radio range) and $l_{min} = \frac{1}{10} l_{max}$.

5.2.2 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:

$$F_{ij} = \kappa \times (l_{ij} - \parallel x_i - x_j \parallel) \times u(x_i - x_j) \tag{5.3}$$

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:

$$F_i = \sum_{j \neq i} F_{ij} \tag{5.4}$$

Each node will periodically broadcast a *keepalive* message to inform its neighbors of its current coordinates. At the same time, a node will also update its coordinates based on the virtual coordinates

of its immediate neighbors using the following rule:

$$x_i = x_i + \frac{\min(|F_i|, \alpha_t)}{|F_i|} F_i \tag{5.5}$$

where α_t is a damping constant that decreases over time.

5.2.3 *Greedy Embedding* Update Rule

After a node has joined the network and the update rule described in Equation (5.5) no longer yields any significant changes to its virtual coordinates, it will geocast a *HELLO* message to all nodes within its region of ownership, which 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.

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:

$$R_{ik} = \delta \times u(x_i - x_k) \tag{5.6}$$

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:

$$F_i = \overbrace{\sum_{j \neq i} F_{ij}}^{\text{spring forces}} + \overbrace{\frac{\min(|\sum_{k \neq i} R_{ik}|, R_{max})}{|\sum_{k \neq i} R_{ik}|} \sum_{k \neq i} R_{ik}}^{\text{capped conflict set repulsion forces}} \tag{5.7}$$

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 (as described in Section 5.1.2). 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. We found that one way to break out of such minima is to give the system a "jolt" every once in a while. To achieve this effect, 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 (5.5), it will set its coordinates as the point s' where it has no nodes in its conflict set as described in Section 5.1.2, 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 $p = 0.1$.

5.3 Determination of Initial Coordinates

In essence, GSpring is a relaxation technique that incrementally improves the greedy forwarding success rate of a given set of virtual coordinates. Because GSpring only makes local changes, it requires a set of relatively "good" coordinates as input. True physical coordinates are usually "good," but they are not always available. Random coordinates will not work well because the small local adjustments that GSpring makes at each step are also not sufficient to allow the system to "unfold" the topology. A related point is that if we do not have access to true physical coordinates and hence have to guess some starting coordinates, it is a bad idea to guess coordinates for every node. While GSpring can "undo" the damage caused by bad guesses on occasion, a better strategy is to guess coordinates for a subset of the nodes, run GSpring for a while so that the subset of nodes converges on relatively good coordinates first and then have the remaining nodes take the cue from these nodes. For this reason, new nodes will also not attempt to "join" the network all at once, but will do so in a locally incremental way (See Section 5.4.1).

The challenge in GSpring is to seed a subset of the network with relatively "good" coordinates. Depending on the nature of the network deployment, one of the following strategies might be applicable:

- **Nodes in the subset have access to their physical locations**. The straightforward approach is this case is to seed all the nodes with their true physical coordinates.
- **Nodes do not have access to their physical locations**. We generate good "seed" coordinates using the *hop-count* algorithm (See Section 5.3.1).

Once a small fraction of the nodes in the system are seeded with starting coordinates, which can either be obtained from a positioning device, manually configured, or derived with the hop-count algorithm, a new node i that does not start with coordinates will derive its initial coordinates from its immediate neighbors as follows:

- 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 furthest apart and pick the midpoint between these nodes as the initial coordinates.

These two cases are illustrated in Figure 5-4. In summary, a network starts with a handful of nodes initialized with coordinates. Then, as nodes derive coordinates from their neighbors, coordinates

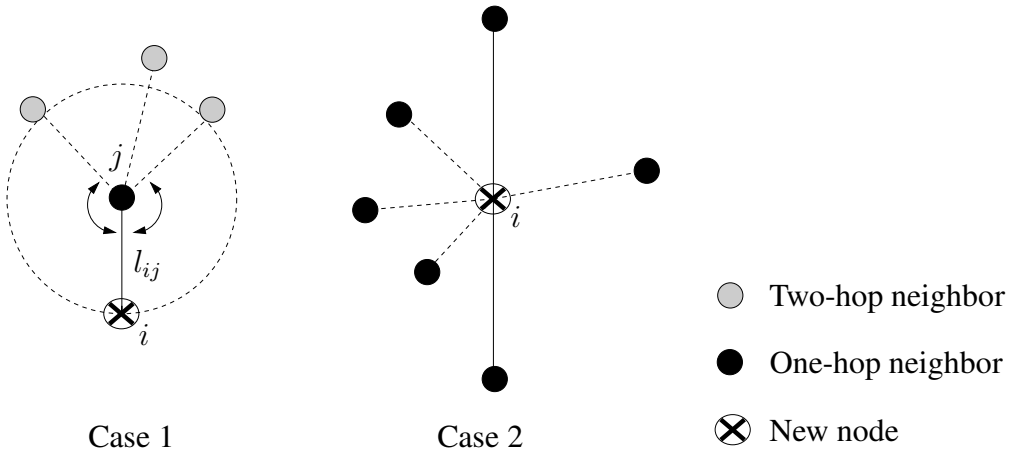


Figure 5-4: Choosing neighbors from which to derive initial coordinates.

gradually percolate through the system. Nodes apply the update rule described in Equation (5.5) as soon as coordinates become available.

For the nodes that are initialized with their actual physical coordinates, these coordinates are only used as the initial seed values. Their virtual coordinates will be updated according to the relaxation algorithm, and the final coordinates are likely to be different from the actual physical coordinates. As mentioned, there are two reasons why we do not pin nodes to their actual physical coordinates: (i) virtual coordinates that are somewhat different from the physical coordinates may yield more convex voids and hence higher greedy forwarding success rates; and (ii) allowing the virtual coordinates to be different from the actual physical coordinates provides greater flexibility in the choice of simulation parameters like the rest length of the springs. If we pin the initialized nodes to their actual physical coordinates and do not choose an appropriate spring rest length, it is likely that the spring relaxation process will cause the virtual topology to converge into a "contorted" configuration. By allowing the virtual coordinates of the nodes to change accordingly, we avoid such a situation and can afford some degree of freedom in choosing the spring rest length and other simulation parameters.

5.3.1 Deriving Seed Coordinates with Hop-Count Algorithm

In this section, we describe the algorithm that we use to derive initial seed coordinates where location information is not available or the available information is limited.

Boundary Detection. We begin by attempting to detect 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. To achieve a consensus, each node will record and broadcast the identity of the node that it thinks is the one with the lowest identifier. Also, by recording the hop count to this node, all

nodes will eventually come to an agreement on this reference node and also its hop count to that node. This process will take no longer than $O(D)$ time, where D is the diameter of the network.

Next, the network will try to come to a consensus on the perimeter node p_1 which is furthest from r in terms of hop count. As before, each node will broadcast the node that they think is the one that is furthest from r and also that node's hop count to r. Ties are broken consistently by comparing node identifiers. Again, it will take no more than $O(D)$ time for the system to come to a consensus. Once p_1 is determined, p_2 is determined in a similar way, as the node that is furthest from p_1 in terms of hop count. In a similar fashion, p_3, is the node that has the greatest sum of the square roots of the hop counts from p_1 and p_2, i.e., p_3 is the node which has the maximal value V_3 defined by:

$$V_3 = \sqrt{h_{j1}} + \sqrt{h_{j2}}$$

where h_{j1} and h_{j2} are the hop counts from node j to nodes p_1 and p_2. We choose to define V_3 as a sum of square roots because such an approach would prefer nodes that are approximately halfway between p_1 and p_2. Similar, p_k for $k > 3$ are likewise defined by the cost function V_k:

$$V_k = \sqrt{h_{j1}} + \sqrt{h_{j2}} + \cdots + \sqrt{h_{jk-1}}$$

The reason why we use the sum of square roots instead of the sum of hop counts is that a sum 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. It is preferable for the perimeters to be spread evenly on the boundary of the network. Using the sum of the square roots will tend to prefer the latter.

As more perimeter nodes p_k's are defined, each p_k is associated with a vector of its hop counts to each of the other perimeter nodes $p_j, j \neq k$. Overall, this perimeter detection scheme will stabilize in $O(D)$ time, and the constant is relatively small when 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 equivalent to a square matrix consisting of $\binom{p}{2} = \frac{p(p-1)}{2}$ hop counts, where p is the number of perimeter nodes, since p is small. It turns out that we do not need a large number of perimeter nodes, and we used eight perimeter nodes in our implementation, which yields excellent results.

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 like Rao et al. [73], it seemed that a natural approach is to arrange them in a virtual circle. Our algorithm for doing so is based on a very simple observation: given that we have a hop count matrix between the set of perimeter nodes, we can in general determine the adjacency relationship between the nodes by looking at the hop counts. We determine the adjacency relationship using the following algorithm:

> **Algorithm 7 (Determination of Cyclical Ordering)** *Given a set of perimeter points $\mathcal{P}$ and an empty array $\mathcal{A}$ of size $|\mathcal{P}|$.*

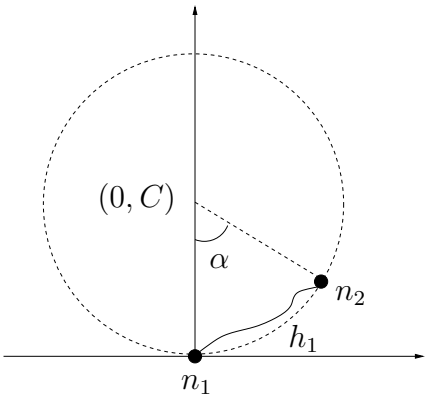


Figure 5-5: Determination of starting coordinates for perimeter nodes.

Set $i = 0$
Remove a node p_k *from* $\mathcal{P}$ *at random and* $cNode = p_k$
Set $\mathcal{A}[0] = cNode$
While $\mathcal{P}$ *is non-empty* $\{$
    Increment i
    Find p_j *in* $\mathcal{P}$ *that is of minimal hop count to* $cNode$
    (Break ties by choosing p_j *such that* j *and the corresponding index of* $cNode$ *is maximal modulo* $|\mathcal{P}|$*)*
    Set $\mathcal{A}[i] = cNode$
    Remove p_j *from* $\mathcal{P}$
$\}$

The array $\mathcal{A}$ *contains a cyclical ordering of the perimeter nodes.*

Coordinates for Perimeter Nodes. Once we have obtained the cyclical ordering of the nodes in $\mathcal{P}$, $\{n_1, n_2, \cdots, n_k\}$, where k is the number of perimeter nodes, we estimate 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:

$$C = \frac{H \times l_{max}}{2\pi} \tag{5.8}$$

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 Figure 5-5. 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.

Interpolation. The above procedure will derive coordinates for k perimeter nodes. One observation is that the hop matrix between all the perimeter nodes is known by all nodes since they are

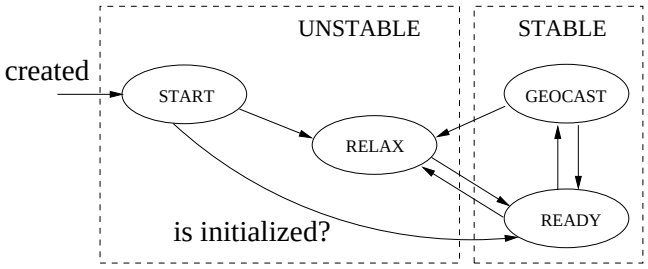


Figure 5-6: Node state transitions for GSpring.

propagated by broadcast. In addition, each node also knows its hop count from each of the perimeter nodes. A node x can 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 calculated from the hop matrix.

When this algorithm terminates, a small number of nodes at the boundary of the network and some nodes in the middle of the network will be initialized with some initial coordinates. The remaining nodes will derive their coordinates from these nodes as these node run the GSpring algorithm and stabilize.

5.4 Implementation

Sections 5.2.2 and 5.2.3 describe how nodes update their virtual coordinates. This section explains the node state transitions required to implement GSpring, and how damping and hysteresis are employed to ensure that the algorithm converges.

5.4.1 Node State Transitions

Nodes implement the GSpring algorithm with a simple state machine. The node state transitions for GSpring are shown in Figure 5-6, and a brief explanation of the various states is as follows:

- *Start:* Nodes are created in this state. A node in this state will attempt to determine its initial coordinates from the coordinates of immediate neighbors that are initialized. However, before a node does so, it will ensure that it does not have a neighbor with a smaller node identifier that is in the *Start* state or a neighbor in one of the other states that is not initialized. Some nodes are initialized with coordinates when the system starts; the other nodes

are considered to be initialized once they have switched to *Geocast* state and subsequently returned to the *Ready* state. Once a node picks its initial coordinates, it changes to the *Relax* state.

- *Relax:* A node in this state will change to the *Ready* state when the magnitude of the updates to its virtual coordinates during periodic updating/relaxation is smaller than a minimum threshold, α_{min}.

- *Ready:* If there are changes to the neighboring nodes that cause periodic updates to differ from the last update vector by a magnitude greater than a threshold α_{max}, a node will change back to the *Relax* state; a node in this state will periodically change to the *Geocast* state to probe for nodes in its region of ownership, even if no changes are detected in the immediate vicinity of the node.

- *Geocast:* When a node switches to this state, it sends a geocast message to detect if there are nodes in its region of ownership. Only the nodes in one of the two stable states (*Ready* or *Geocast*) will respond to the querying node; nodes in the unstable states will simply ignore the geocast query. A node will stay in the *Geocast* state for a pre-determined interval. If no new nodes are discovered, it will simply revert to the *Ready* state; if a new node is discovered, the node will switch to the *Relax* state.

5.4.2 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. Once the size of the change for a given time step falls below a threshold, α_{min}, a node will consider itself stabilized and no longer updates its coordinates.

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 in *Relax* state. Once the number of iterations exceeds a pre-determined threshold T, α_t is scaled by an exponentially decreasing constant as follows:

$$\alpha_t = \begin{cases} \alpha_{max}, & \text{if } t < T \\ \alpha_{max} e^{-\frac{t}{T}}, & \text{otherwise} \end{cases} \tag{5.9}$$

where α_{max} and T are constants and t is the count of the number of iterations after a node switches to the *Relax* state. If the magnitude of the displacement $\min(|F_i|, \alpha_t)$ falls below a minimum threshold α_{min}, a node in *Relax* state will switch to the *Ready* state.

α_{max} is the parameter that controls the *hysteresis* factor in the system. When a node switches to the *Ready* state at time t, it will record the force vector $F_{i,t}$; it will revert to the *Relax* state when and if the force vector $|F_{i,t+1} - F_{i,t}| > \alpha_{max}$ at some point $t+1$. In our implementation, $\alpha_{min} = 1$, $\alpha_{max} = 5$ and $T = 50$.

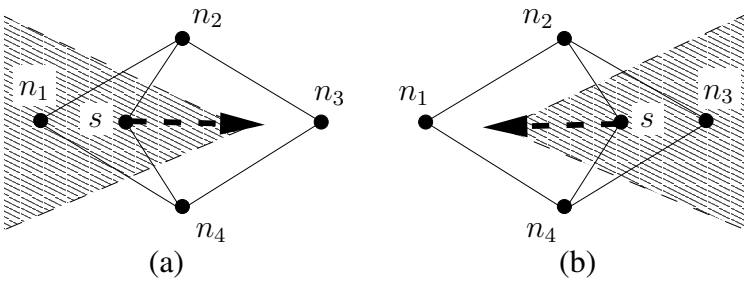


Figure 5-7: Configuration where a node, s, will oscillate and is unable to obtain virtual coordinates that will keep other nodes out of its region of ownership (shaded in gray).

There are configurations in which a node is unable to find good coordinates for which it is able to avoid having other nodes in its region of ownership. In fact, oscillation due to conflict set repulsion is possible. One example configuration is shown in Figure 5-7.

To avoid oscillations due to repulsion from a changing conflict set, 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 (5.7), only if it hears from new nodes in the new conflict set. In the example shown in Figure 5-7(a), s is first repelled by n_1, and subsequently by n_3 when it ends up as shown in Figure 5-7(b), and hence the system returns to the configuration in Figure 5-7(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 Figure 5-7(a).

5.5 Geocast can be replaced with Location Service

While GSpring as described requires the availability of a geocast mechanism, what is truly required is not a geocast mechanism, but rather a rendezvous mechanism that will allow a node to determine the other nodes that are within its region of ownership.

As mentioned in Chapter 1, geographic routing must be augmented by a location service. A location service is a rendezvous mechanism that can be used in place of geocast. Since nodes have to update their coordinates with the location service when they join the network and when they move, the location service contains information on the coordinates of nodes in the network.

Hence, in place of geocast, GSpring can also resort to querying the location service as long as nodes update their locations with the location service often enough. The relative costs of geocasting versus the querying of the location service will depend on the design and implementation of the respective services.

5.6 Summary

We have described GSpring, a simple virtual coordinate assignment algorithm that finds good virtual coordinates for geographic routing by incrementally adjusting coordinates to increase the convexity of voids in the virtual routing topology. GSpring is a combination of the following mechanisms:

1. A hop-count-based algorithm to detect a small number of perimeter nodes and seed nodes that lie on a suitably scaled circle;
2. The initialization of only a small number of nodes at the beginning and incremental addition of nodes only after the neighbors have stabilized to reduce system inertia;
3. Spring forces between adjacent nodes to keep the local relationship between nodes consistent;
4. Repulsion between nodes when one node is within the region of ownership of the other to incrementally improve the greedy forwarding success rate; and
5. The use of damping and hysteresis to ensure that the system converges.

Chapter 6

GSpring Evaluation

In this chapter, we evaluate the performance of GSpring, by comparing the routing performance of existing geographic face routing algorithms using coordinates obtained with the GSpring algorithm, actual physical coordinates, and coordinates obtained with the NoGeo algorithm [73]. In addition to path stretch and hop stretch, we also evaluate the scalability of GSpring with respect 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 geocast messages sent and received per node.

6.1 Simulation Setup

In this section, we describe our simulation setup. The goal of our simulations is to understand the effect of network density, network size and obstacles on the GSpring algorithm.

As before, we evaluate the performance of GSpring with simulations. The simulations are performed using the same high-level event-driven simulator [50] that was used to evaluate GDSTR, as described in Section 4.1.

Effect of Network Density. We use a different set of networks from those used for evaluating GDSTR to investigate the effect of network density on GSpring. The random scattering of nodes over a fixed area tends to generate separate networks instead of one connected network when the node density is low.

Hence, to generate a set of connected networks over a range of network densities, we randomly scatter 500 nodes of unit radio range over a number of $x \times x$ unit squares, where x ranges from 10 to $20\sqrt{5}$. Then, we determine the largest set of connected nodes among the scattered nodes, remove the remaining nodes and use this set as the network for our evaluation. While this procedure produces networks of randomly varying sizes, we are able to obtain a good spread of networks with densities spanning the entire region of interest.

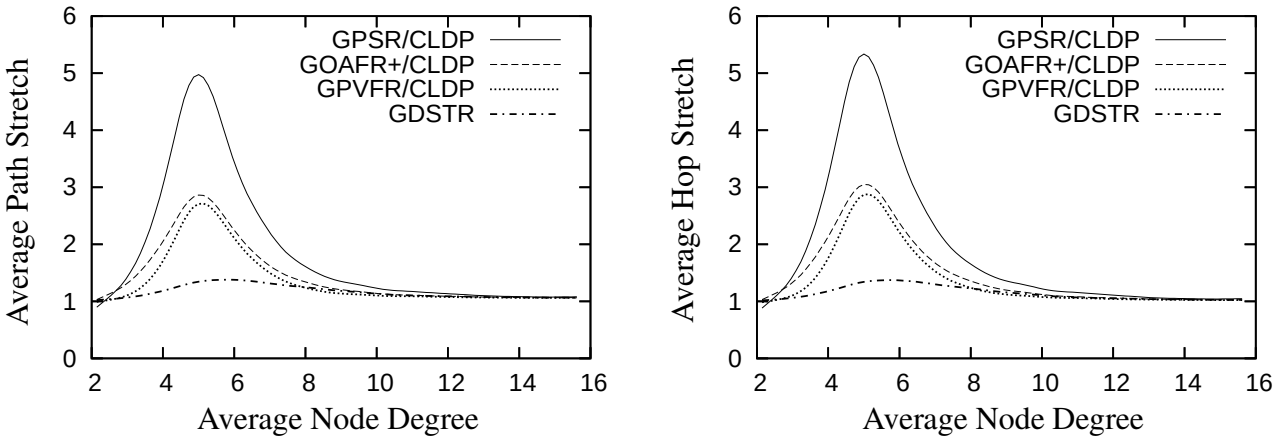


Figure 6-1: Plot of hop stretch with actual physical coordinates.

Effect of Network Size and Obstacles. To evaluate the scaling properties of GSpring and the effectiveness of GSpring for different kinds of networks (including networks with obstacles), we use the topologies described in Section 4.1. As before, we refer to the UDG networks with average node degrees 6.5 and 12 as "Sparse" and "Dense," respectively, and to the non-UDG networks with high and low obstacle densities as "Large Voids" and "Small Voids," respectively. We evaluate these networks for the sizes ranging from 50 to 2,000.

6.2 Routing Performance

In this section, we evaluate the routing performance for existing geographic routing algorithms with GSpring coordinates. We compare this with performance achieved using actual physical coordinates and for NoGeo coordinates. In Section 6.2.1, we evaluate the routing performance for small networks (with less than 500 nodes). In Section 6.2.2, we evaluate how routing performance scales with increasing network size from 50 to 2,000 nodes for sparse and dense unit disk graph (UDG) networks (see Section 2.2) with constant network density. We evaluate routing performance in networks with obstacles in Section 6.2.3.

6.2.1 Effect of Network Density

First, we compare the routing performance of the following geographic face routing algorithms over GSpring coordinates to the routing performance over actual physical coordinates: GPSR [39], GOAFR+ [48] and GPVFR [52] with CLDP planarization [42], the only algorithm that is known to work for practical networks, and with GDSTR [51], over random networks for a range of average node degrees up to about 16. The results are shown in Figures 6-1 and 6-2.

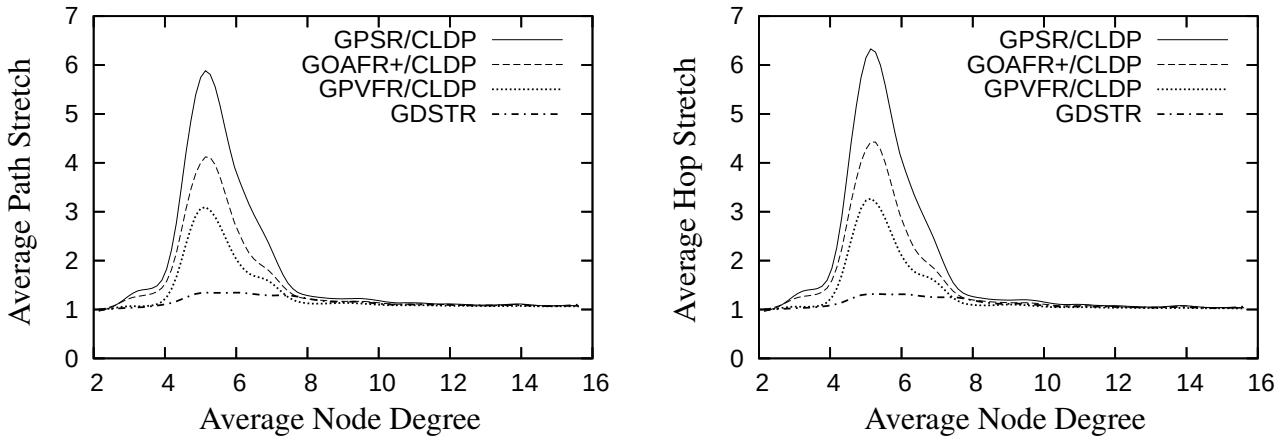


Figure 6-2: Plot of hop stretch with GSpring coordinates.

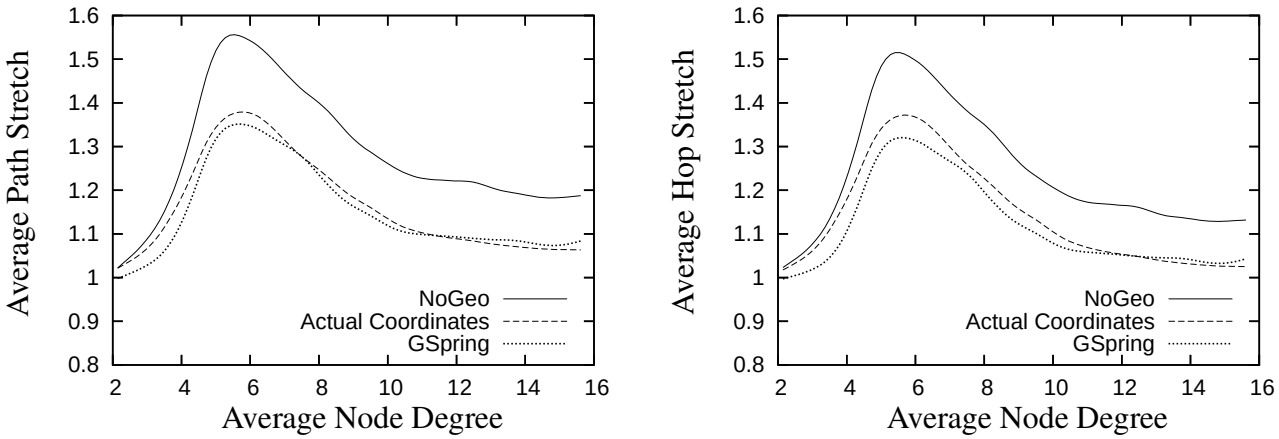


Figure 6-3: Plot comparing the stretch for GDSTR/GSpring over networks of different average node degrees (Note change of scale from Figure 6-2).

As before, our implementations of these routing techniques are based on the algorithms as described in [39], [47], [52] and [51], respectively. The configuration parameters for GOAFR+ are $\rho_0 = 1.4, \rho = \sqrt{2}$ and $\sigma = \frac{1}{100}$ as suggested in [47]; for GPVFR, we limit the length of the propagated path vectors to 3; our implementation of CLDP follows the description in [42] and we use two hull trees for all experiments with GDSTR.

Our results show that the virtual coordinates produced by GSpring allow existing geographic routing algorithms to achieve routing performance that is comparable to that obtained with real physical coordinates for the face routing algorithms. Since GDSTR is generally more efficient and significantly cheaper to deploy than geographic face routing algorithms [52], we focus on evaluating the performance of GSpring with GDSTR.

In Figure 6-3, we plot the routing performance of GDSTR over NoGeo coordinates, GSpring coordinates, and actual physical coordinates. We make two observations from these results: (i) GSpring

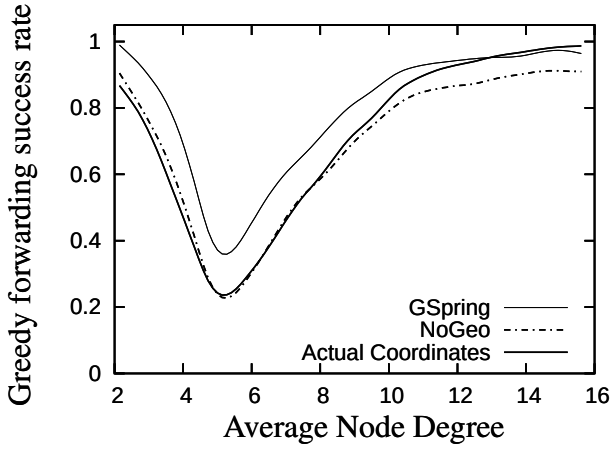


Figure 6-4: Greedy forwarding success rates for GSpring under various conditions.

coordinates allow GDSTR to achieve better routing stretch than NoGeo over the entire range of network densities; and (ii) GSpring produces coordinates that achieve slightly better performance than the actual physical coordinates. It turns out that observation (ii) is true only because the networks evaluated in this section are relatively small (i.e., less than 500 nodes in size) and because of the way that the networks have been generated.

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 Figure 6-4. 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.

6.2.2 Unit Disk Graph Networks

We plot the routing stretch for GDSTR with coordinates derived with NoGeo and GSpring, as well as with actual physical coordinates in Figures 6-5 and 6-6, respectively.

As shown in Figure 6-5, the actual physical coordinates yield the best routing performance for sparse UDG networks. GSpring achieves routing stretch that is approximately 10% higher than that for actual physical coordinates and about 20% lower than that for NoGeo coordinates.

As shown in Figure 6-6, the actual physical coordinates achieve close to optimal (unit) stretch for dense UDG networks. Note that the scale of the y axis was increased to improve clarity. GSpring is not only able to match the routing performance of the actual physical coordinates, it is in fact able to achieve slightly lower stretch when the networks are large. The difference however is very small. As before, GDSTR achieves 20% lower stretch with GSpring coordinates than with NoGeo coordinates.

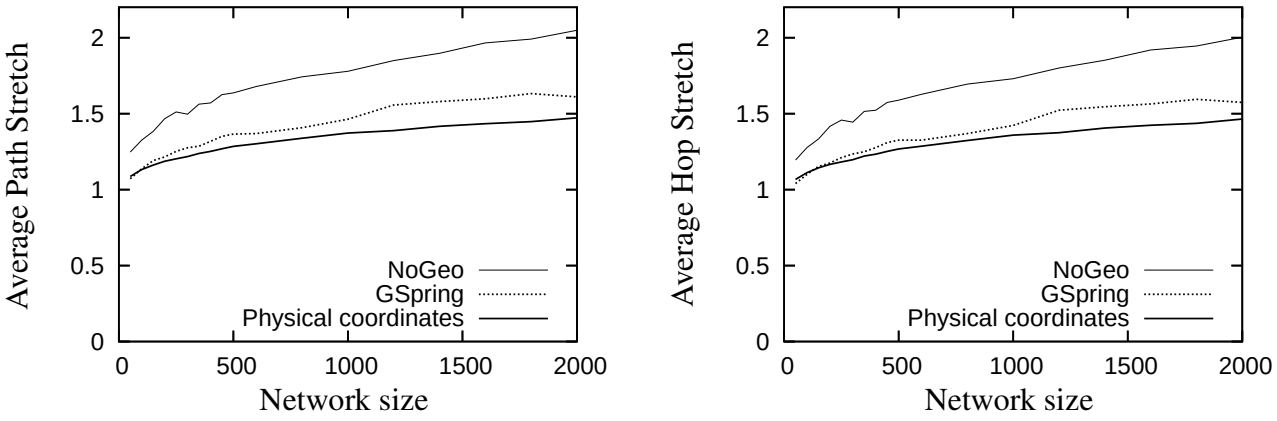


Figure 6-5: Plot of GDSTR stretch for sparse UDG networks (average node degree 6.5).

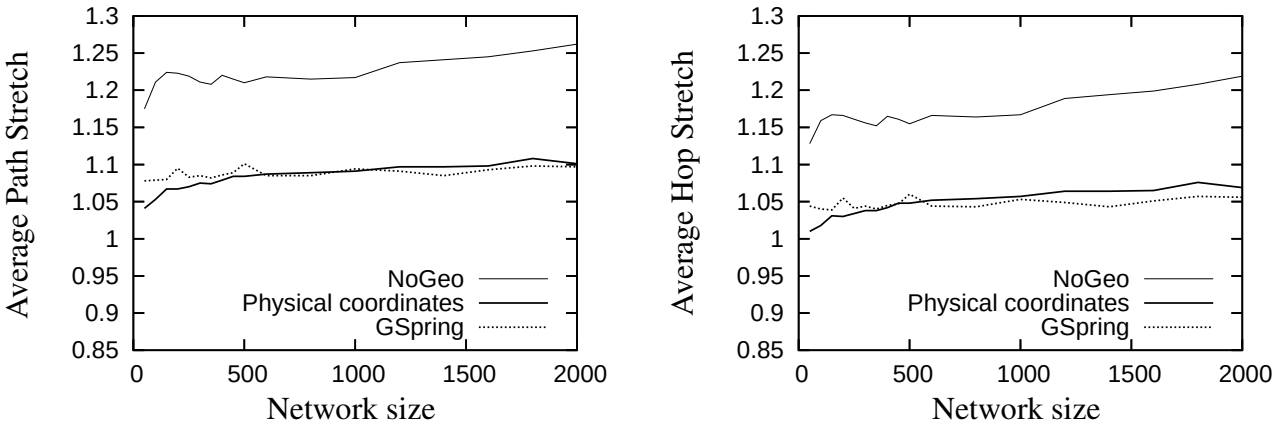


Figure 6-6: Plot of GDSTR stretch for dense UDG networks (average node degree 12).

6.2.3 Obstacles

Given that we adopt uniform radio ranges for the nodes in our evaluation in Section 6.2.2, it might not be surprising that GSpring converges to a virtual topology that has the same approximate shape as the actual physical topology. Since obstacles are common in real networks, it is important to understand the performance of GSpring in the presence of obstacles.

The routing performances for non-UDG networks with small and large voids are shown in Figures 6-7 and 6-8, respectively. We see from these results that the density of obstacles does not seem to have a very large effect on routing stretch.

Overall, NoGeo performs universally poorly for networks with obstacles. The routing performance for NoGeo worsens progressively with increasing network size. For 2,000-node networks, NoGeo incurs up to 40% higher stretch than actual physical coordinates. The Spring algorithm seems be perform universally better than NoGeo, though like NoGeo, the routing performance seems to worsen with increasing network size.

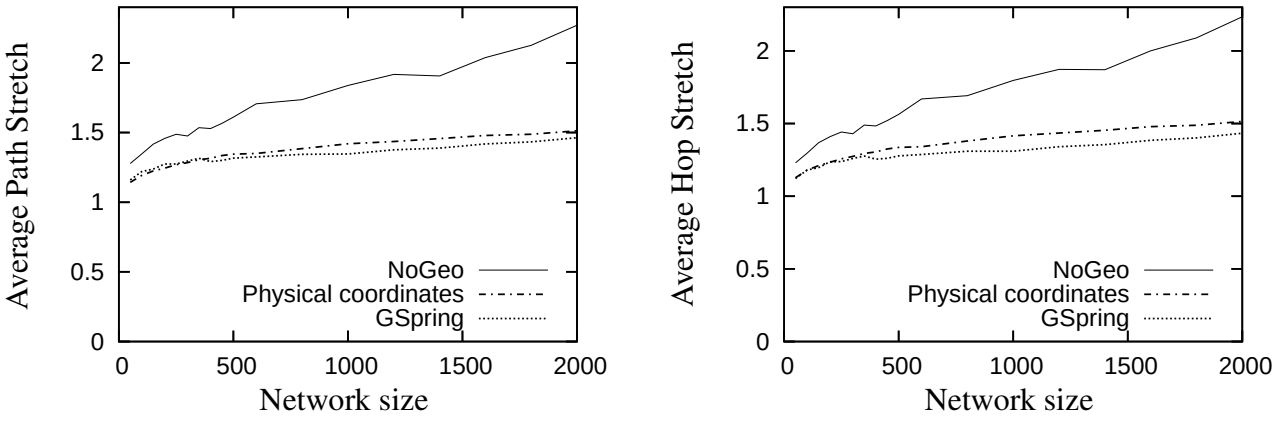


Figure 6-7: Plot of GDSTR stretch for networks with small voids (average node degree 8).

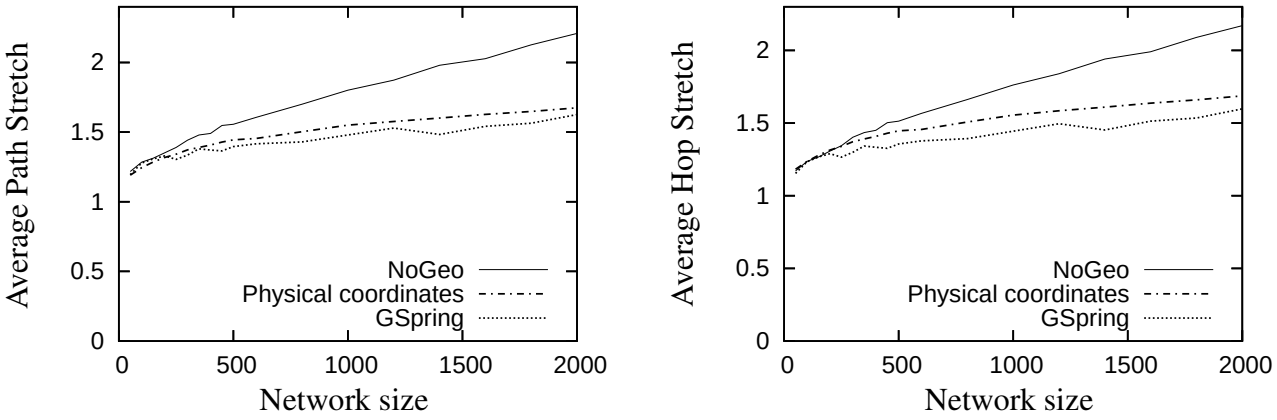


Figure 6-8: Plot of GDSTR stretch for networks with high obstacle density (average node degree 7).

GSpring on the other hand, demonstrates better performance than the actual physical coordinates and its performance is somewhat independent of the obstacle density. Not surprisingly, its performance when compared to the actual physical coordinates seems to depend on the size of the voids. For the networks with large voids (and higher obstacle density), it achieves up to 10% lower stretch than actual physical coordinates. The improvement is halved at 5% for the networks with smaller voids.

6.3 Seeding Some Nodes with Location Information

While GSpring is able to derive virtual coordinates when no location information is available, some networks may have a small number of nodes equipped with positioning devices. In this section, we investigate the routing performance for GDSTR with GSpring under scenarios where a small fraction ($< 5\%$) of the nodes are seeded with their true coordinates. We also consider the case

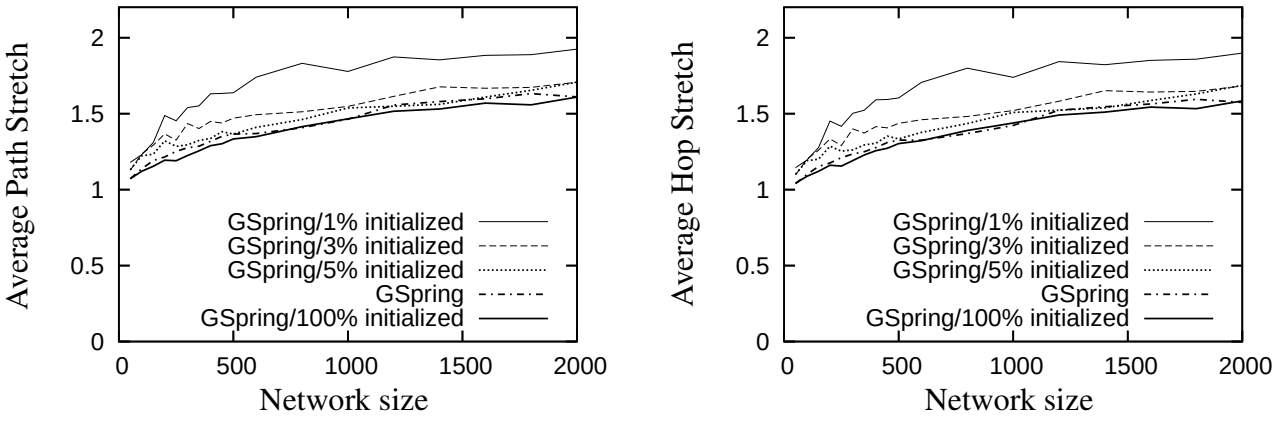


Figure 6-9: Plot of GDSTR stretch with location information seeding for sparse UDG networks (average node degree 6.5).

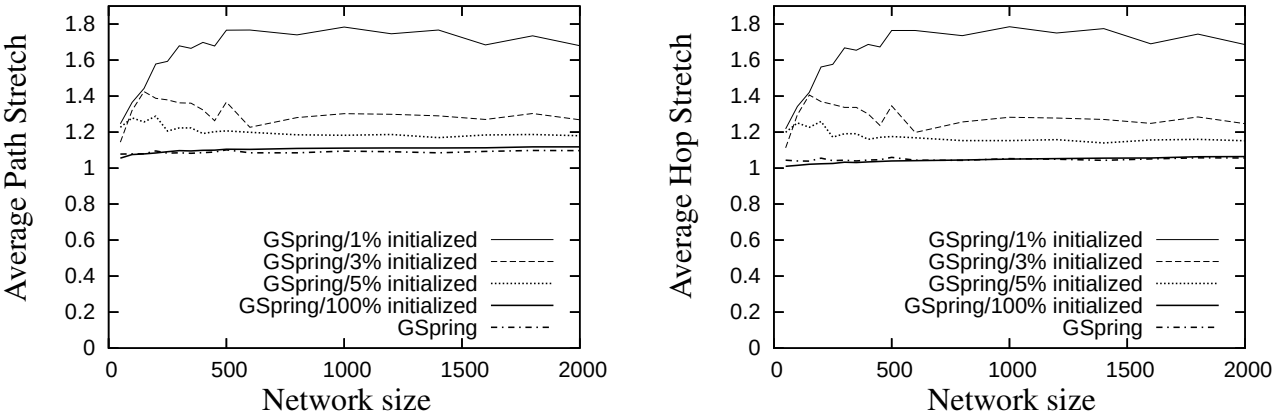


Figure 6-10: Plot of GDSTR stretch with location information seeding for dense UDG networks (average node degree 12).

where 100% of the nodes are initialized with their actual physical coordinates.

Overall, we found that seeding a small fraction ($< 5\%$) of the nodes with their true coordinates does not work as well as initializing the system with the hop-count algorithm as described in Section 5.3.

6.3.1 Unit Disk Graph Networks

In Figures 6-9 and 6-10, we plot the corresponding routing performance when nodes are seeded with their true physical coordinates instead of using the hop-count algorithm. Not surprisingly, we find in general that routing performance improves when more nodes are initialized. The case where 100% of the nodes are initialized provides an upper bound of how well we can do with more information.

Our results in Figure 6-9 shows that for sparse UDG networks, initializing 100% of the nodes with their true location information yields comparable routing performance to the hop-count algorithm. Earlier in Figure 6-5, we saw that GSpring coordinates did not perform as well as actual physical coordinates for such networks. This means that GSpring can make coordinates worse in sparse networks.

A likely explanation for this phenomenon is that the configuration for sparse random UDG networks is already relatively greedy. Because GSpring is a localized algorithm, it can in fact disrupt an existing configuration that is relatively greedy for large networks. Also, to ensure that the algorithm converges, we have to introduce damping and hysteresis. It is likely that as a result, the network often cannot "undo" bad adjustments.

The results for dense UDG networks in Figure 6-10 seem to suggest that the routing performance for dense UDG networks is dependent mostly on the fraction of nodes initialized and is somewhat independent of size. Like in the case of sparse networks, when some of the nodes are initialized with their true physical coordinates, initializing only a small fraction of the nodes does not seem to work well.

We found that this is because without sufficient initial points to "pin" the initial coordinates of the nodes down, it is quite easy for the network to converge to a set of coordinates that "folds over 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. GSpring is analogous to an attempt at "unfurling" a mesh of springs. When the mesh is small, it is relatively easy to unfurl it; but when the mesh is large, the mesh tends to get "entangled" in a bad configuration. If we can start by pinning some points of the mesh to "good" points in space, we can generally do better.

6.3.2 Obstacles

In Figures 6-11 and 6-12, we plot the corresponding routing performance for networks with obstacles when instead of using the hop-count algorithm, we seed some nodes with their true physical coordinates.

While seeding some nodes with their actual coordinates seemed to work fairly well for UDG networks, the same cannot be said for non-UDG networks with obstacles. Even in the limit when all the nodes are initialized with their actual physical coordinates, the performance of the final routing topology is significantly worse than that for actual physical coordinates and GSpring with the hop-count algorithm.

We suspect that the reason for this is that when damping and hysteresis are introduced, a network with all its nodes initialized will stabilize before the system reaches an "optimal" configuration. On the other hand, when the hop-count algorithm is applied, the majority of the nodes will wait for the system to stabilize before joining, and therefore the approach does not introduce quite as much rigidity and inertia. This suggests that it is helpful to initialize only a small number of nodes with good coordinates and incrementally add nodes only after the neighbors have stabilized.

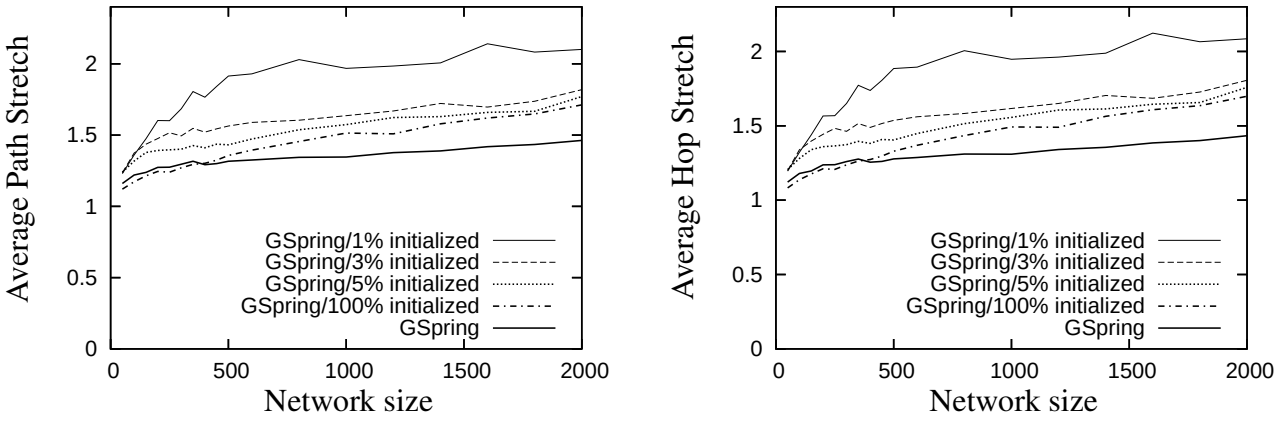


Figure 6-11: Plot of GDSTR stretch with location information seeding for networks with small voids (average node degree 8).

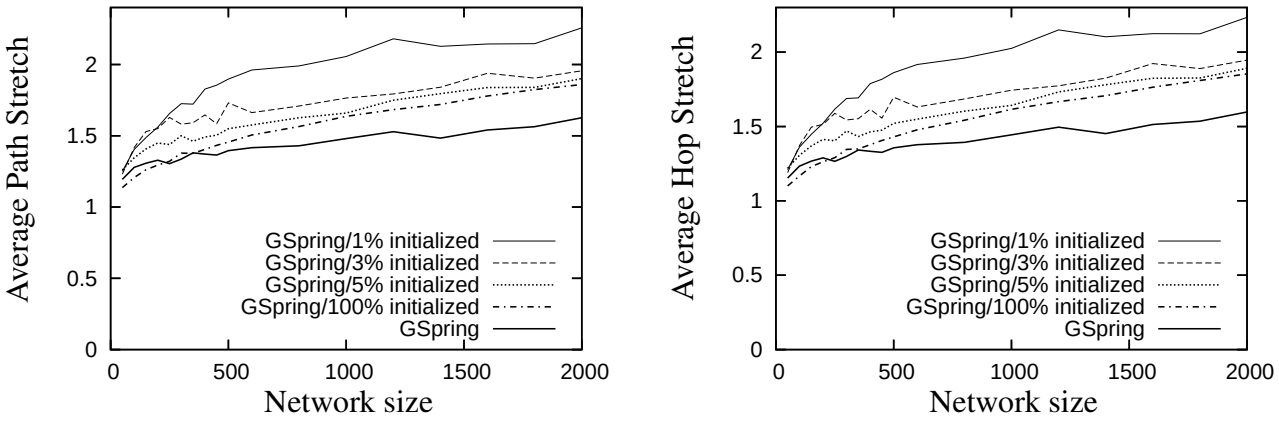


Figure 6-12: Plot of GDSTR stretch with location information seeding for networks with large voids (average node degree 7).

6.4 Understanding the Effect of Greedy Embedding Repulsion

We will refer to a variant of the GSpring algorithm without the greedy embedding repulsion component, i.e., it consists of using the hop-count algorithm to place 8 perimeter nodes on a virtual circle, and running the relaxation algorithm using only the Spring Relaxation Update Rule (see Section 5.2.2), as the "Spring" algorithm. We compare the hop stretch performance of Spring to GSpring and NoGeo in Figures 6-13 to 6-16. The corresponding path stretch performance follows the same trend.

From these results, it is clear that the greedy embedding repulsion is crucial for good routing performance, since the performance for Spring is significantly worse than GSpring in all cases. The Spring algorithm performs slightly better than NoGeo for sparse networks and for smaller dense networks. However, NoGeo seems to scale better than Spring for dense networks. The

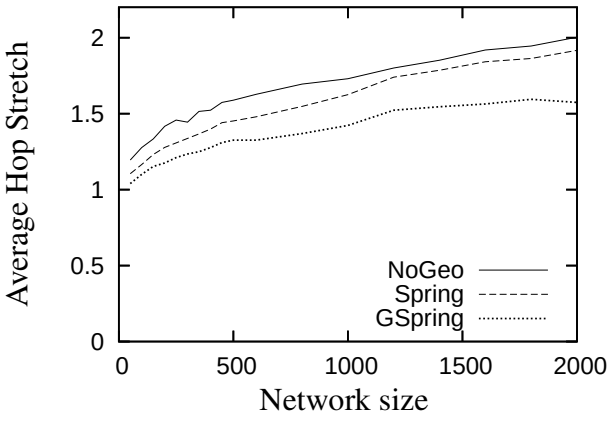


Figure 6-13: Plot of GDSTR stretch for sparse UDG networks (average node degree 6.5).

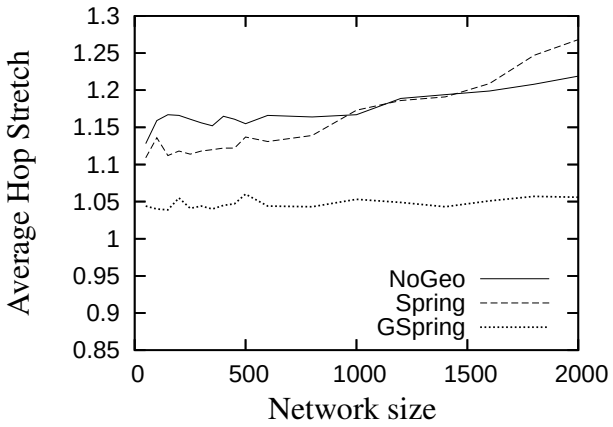


Figure 6-14: Plot of GDSTR stretch for dense UDG networks (average node degree 12).

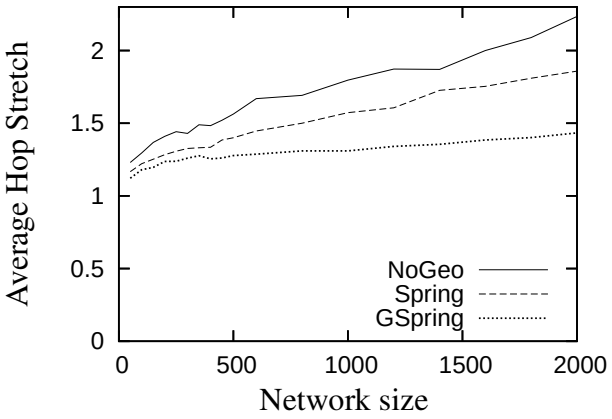


Figure 6-15: Plot of GDSTR stretch for networks with small voids (average node degree 8).

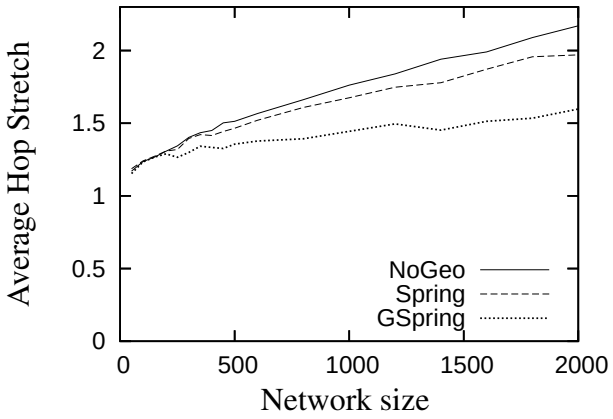


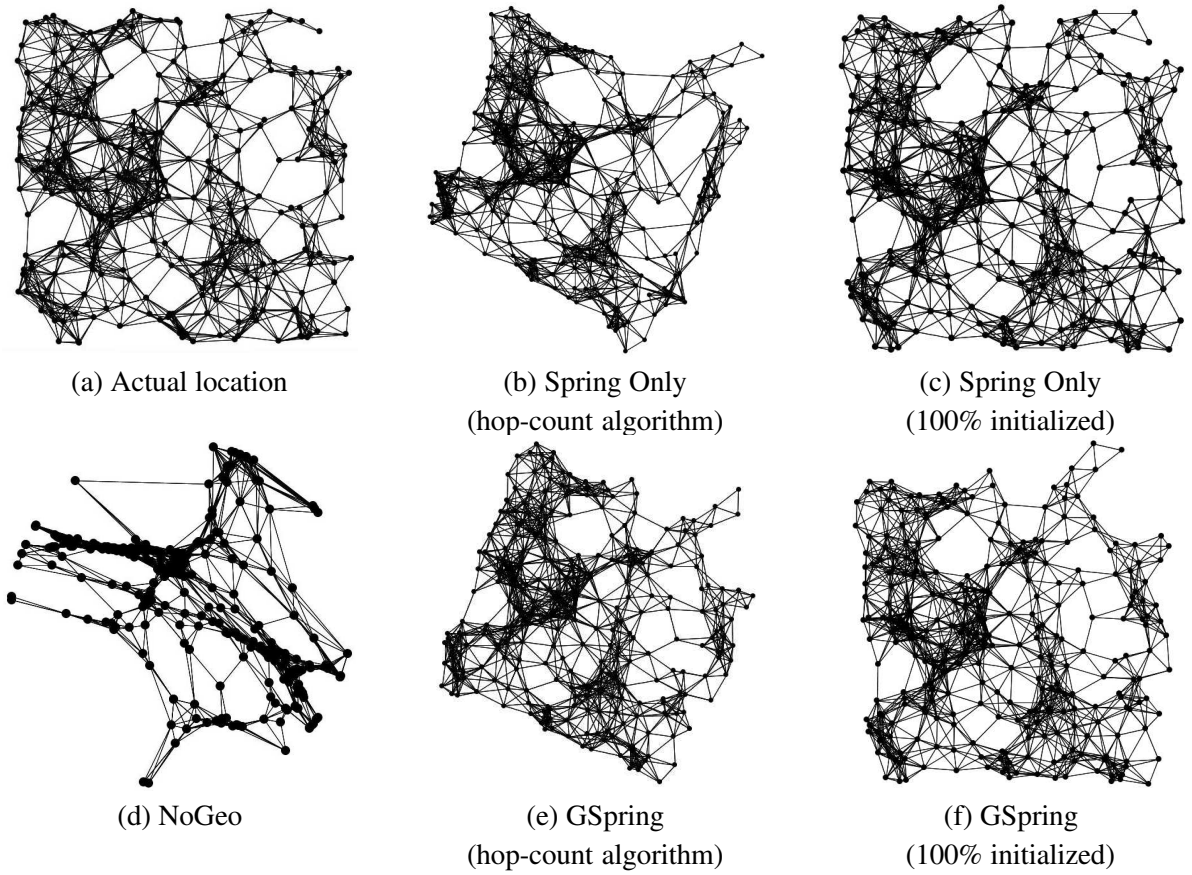
Figure 6-16: Plot of GDSTR stretch for networks with large voids (average node degree 7).

reason for this is that when the network is both large and dense, the virtual springs tend to "clump" together. In fact, the untangling of springs in a spring-like system is known to be challenging.

While our experiments described in the previous sections demonstrate that GSpring is able to converge to virtual coordinates that yield good routing performance, it is important to understand why. In the following sections, we present some examples of the routing topologies that are generated by GSpring to provide us with some insights and intuitions on how GSpring works.

6.4.1 Unit Disk Graph Networks

An example of a dense UDG network containing 300 nodes is shown in Figure 6-17(a). Figure 6-17(b) shows that if the basic spring relaxation algorithm is used by itself without conflict



(a) Actual location

(b) Spring Only (hop-count algorithm)

(c) Spring Only (100% initialized)

(d) NoGeo

(e) GSpring (hop-count algorithm)

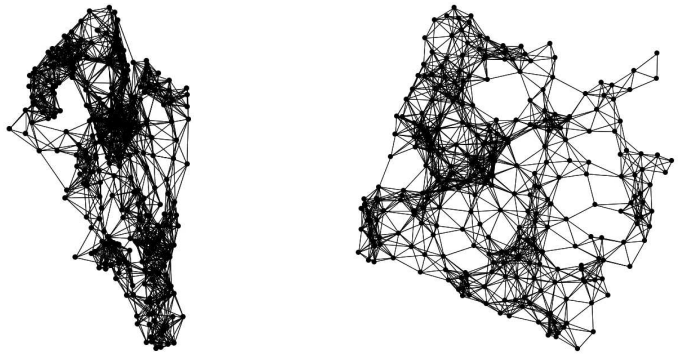
(f) GSpring (100% initialized)

Figure 6-17: Derived coordinates for sample dense 300-node unit disk graph (UDG) network with GSpring and NoGeo.

set repulsion, the resulting topology is fairly close to that of the real topology. When 100% of the nodes are initialized with their true physical coordinates, the basic spring relaxation algorithm tends to preserve the shape of the actual physical topology as shown in Figure 6-17(c).

The routing topology produced by GSpring is shown in Figure 6-17(e). We note that the topology produced by GSpring is similar to the actual topology. The effect of the conflict set repulsion can be seen by comparing Figure 6-17(f) to Figure 6-17(a), since Figure 6-17(f) shows the final routing topology by running the GSpring algorithm with all nodes initially seeded with their actual physical coordinates. In this particular network, the effect of the transformation induced by GSpring is not pronounced, though it should be noted that some nodes are shifted to positions that make the topology somewhat more "greedy". The virtual topology generated by NoGeo is shown in Figure 6-17(d) for reference.

We explained in Section 5.3.1 that we use the sum of square roots in the hop-count algorithm instead of the sum of hop counts to differentiate between two configurations with the same sum. It is not ideal to use the sum of hop counts. To illustrate this, consider Figure 6-18. Figure 6-18(a) shows the network configuration when we use the sum of the hop counts instead of their



(a) Spring Only (b) GSpring

Figure 6-18: Topologies for bad implementation of hop-count algorithm.

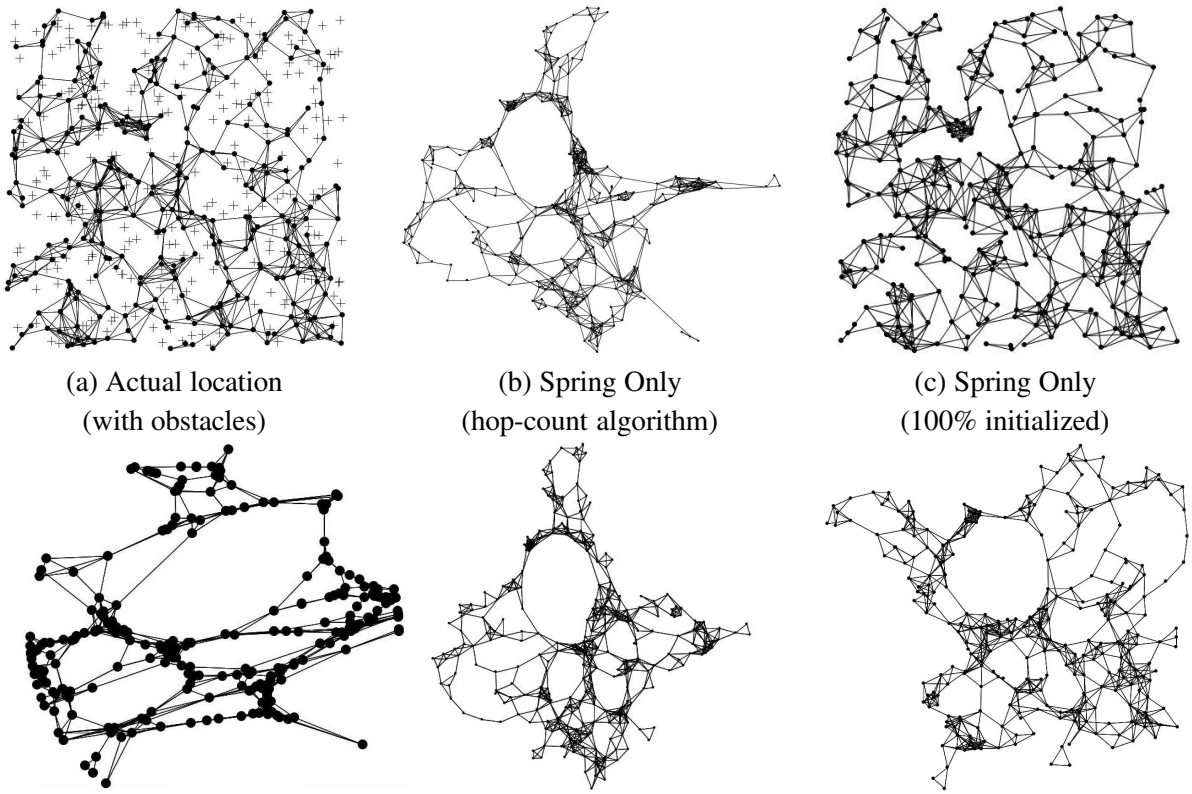
square roots with the Spring algorithm only. As shown, using the sum of the hop counts does not detect perimeter nodes well, and the resulting basic topology resembles a tangled mess of springs instead of the one in Figure 6-17(b). The figure in Figure 6-18(b) is the final configuration with the above hop-count algorithm implementation when we apply GSpring. We see that it is almost identical to Figure 6-17(e). This example demonstrates how GSpring is able to "unfurl" a tangled initial configuration and why the greedy embedding repulsion rule is absolutely essential for good routing performance.

6.4.2 Network with Obstacles

To provide some physical intuition for the effect of GSpring on networks with obstacles, we present an example a network with 300 nodes and a high density of cross-shaped obstacles in Figure 6-19(a).

As shown in Figure 6-19(b), applying only the basic spring relaxation algorithm results in a topology that is quite different from the actual topology. This is because the hop-count algorithm is relatively successful at stretching out the topology. If all the nodes in the network are initialized with the true coordinates, the basic spring algorithm will however tend to preserve the basic shape of the actual physical topology as shown in Figure 6-19(c).

Unlike the earlier example in Figure 6-17, the changes induced by GSpring on the routing topology are somewhat more pronounced on this network. We see this by comparing Figure 6-19(e) to Figure 6-19(f). The effect of the greedy embedding repulsion forces is also clear from comparing Figure 6-19(c) to Figure 6-19(f). As before, the virtual topology generated by NoGeo is shown in Figure 6-19(d) for reference.



(a) Actual location (with obstacles)
(b) Spring Only (hop-count algorithm)
(c) Spring Only (100% initialized)
(d) NoGeo
(e) GSpring (hop-count algorithm)
(f) GSpring (100% initialized)

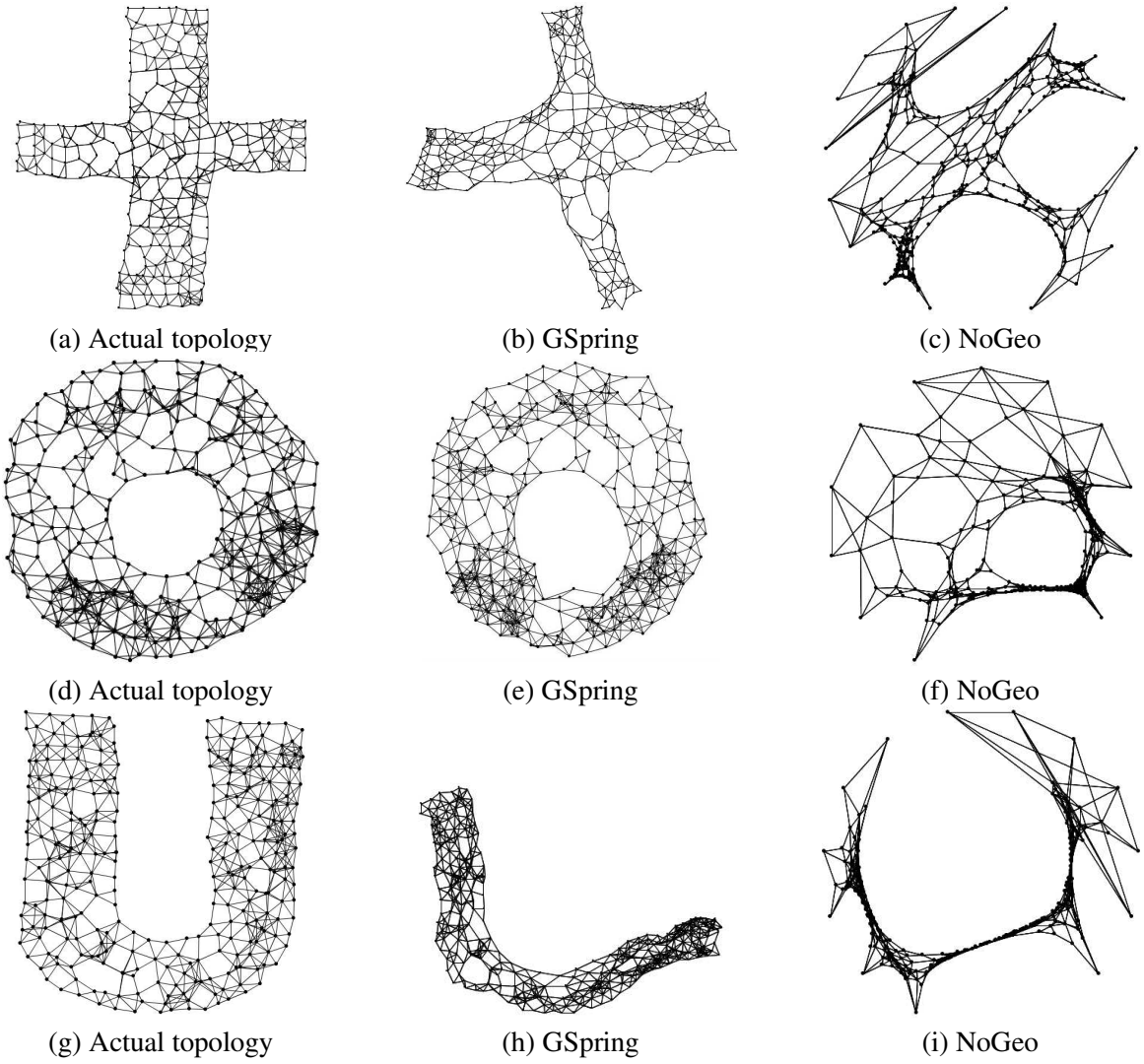
Figure 6-19: Derived coordinates for sample 300-node network with cross-shaped obstacles with GSpring and NoGeo.

6.4.3 Irregular Shapes

We present the resulting virtual topologies for three irregular shapes – cross, donut and U-shape, in Figure 6-20. For each shape, the first figure is the actual physical layout of the nodes; the second figure is the virtual topology generated by GSpring; and the third figure is the topology generated by NoGeo.

In the case of the cross and the donut topologies, the virtual routing topologies generated by GSpring are quite similar in shape to that of the actual topology. However, for the U-shaped topology, GSpring tends to "flatten" out the U. As discussed in Section 1.2, this is desirable because the original topology causes packets routed between nodes at the two ends of the U to end up in a dead end.

Two drawbacks of the NoGeo algorithm for small and relatively sparse networks are also apparent from these examples. First, small networks will result in NoGeo electing fewer perimeter nodes. Having only a small number of perimeter nodes will cause the nodes in the virtual topology to "clump together," leading to regions of unnecessarily high density in the virtual routing space. The



(a) Actual topology (b) GSpring (c) NoGeo

(d) Actual topology (e) GSpring (f) NoGeo

(g) Actual topology (h) GSpring (i) NoGeo

Figure 6-20: Routing topologies for sample 300-node networks with irregular shapes – cross, U-shape and donut.

other major drawback for NoGeo is clear from the example of the cross and U-shaped networks: the fact that NoGeo attempts to "map" a network onto a circle makes it likely to generate more dead ends for networks for which the greedy embedding is not circular.

6.5 Convergence and Costs

In this section, we evaluate the costs of the GSpring algorithm, in terms of the time taken to converge and the number of geocast messages that have to be sent. We also examine how damping and hysteresis affects both the convergence time and the number of geocast messages sent.

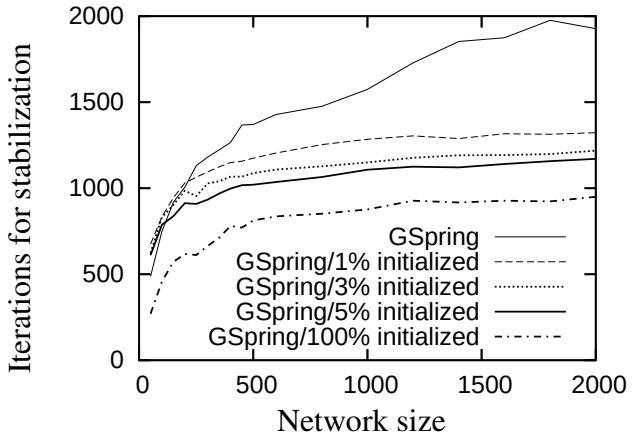


Figure 6-21: Iterations required for stabilization for sparse UDG networks (average node degree 6.5).

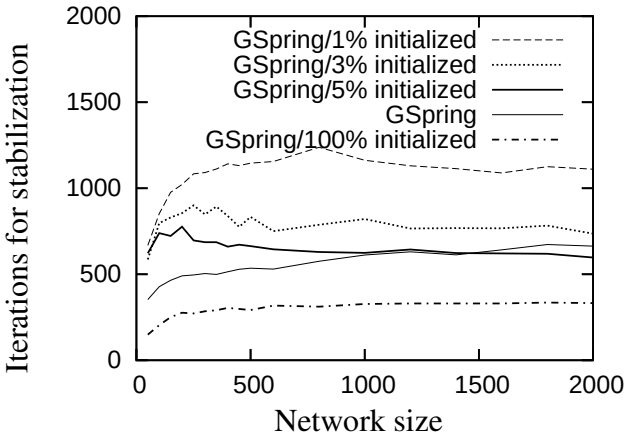


Figure 6-22: Iterations required for stabilization for dense UDG networks (average node degree 12).

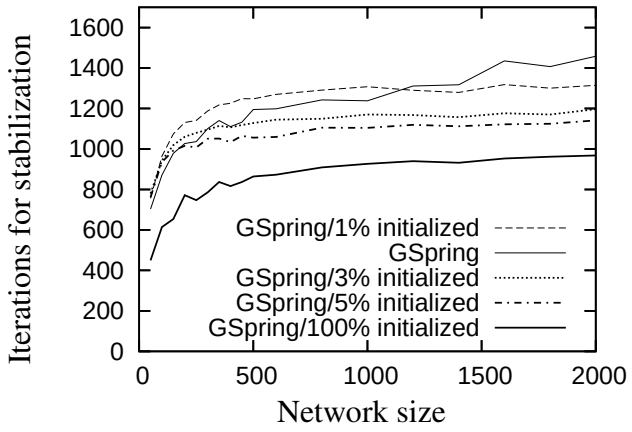


Figure 6-23: Iterations required for stabilization for networks with small voids (average node degree 8).

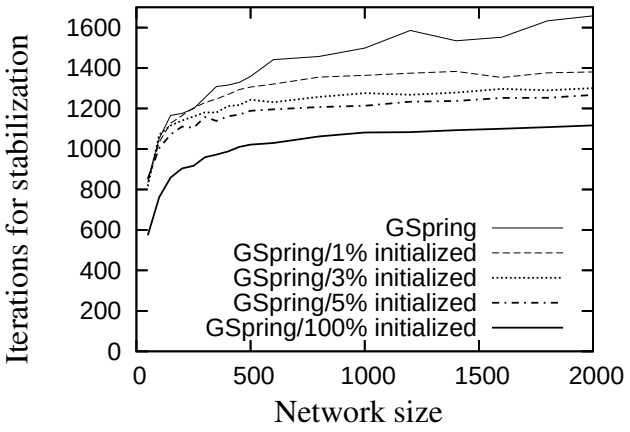


Figure 6-24: Iterations required for stabilization for networks with large voids (average node degree 7).

6.5.1 Convergence Time

With our current implementation of GSpring, each node typically requires between a thousand to two thousand iterations to stabilize (where one iteration involves an application of the update step described in Equation (5.5)). The average number of iterations required by a node to stabilize for the networks studied in Sections 6.3.1 and 6.2.3 is shown in Figures 6-21 to 6-24. These figures represent the scenario where, except for the initialized nodes, the rest of the nodes start off without any state.

As expected, if a larger proportion of the nodes is initialized with their actual physical coordinates, GSpring tends to converge more quickly. GSpring takes more iterations to converge in sparse networks than in dense networks. In sparse networks, the time to convergence seems to increase with network size, while in dense networks, the time to convergence seems to be somewhat indepen-

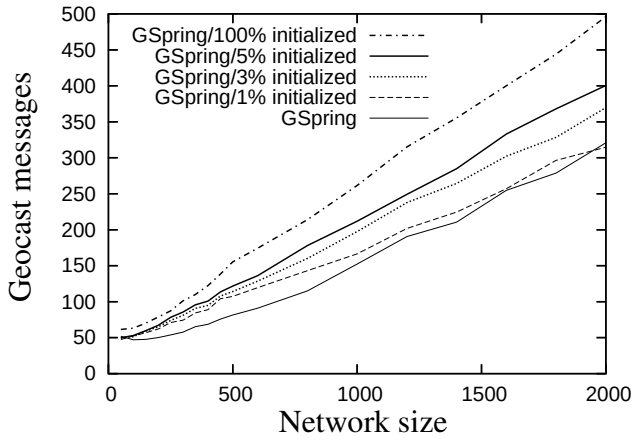


Figure 6-25: Geocast messages sent and received per node for sparse UDG networks (average node degree 6.5).

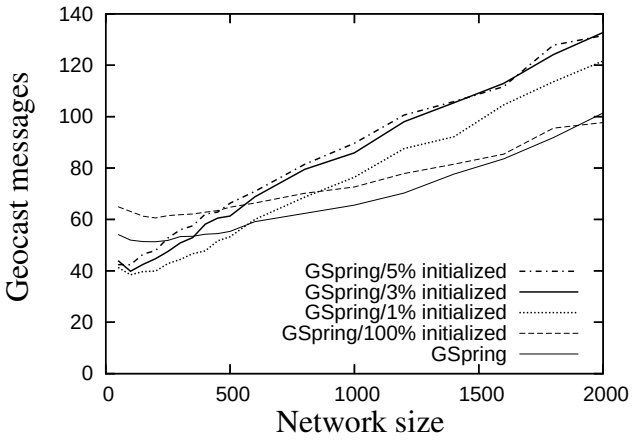


Figure 6-26: Geocast messages sent and received per node for dense UDG networks (average node degree 12).

dent of size. If the hop-count algorithm is not used to derive initial coordinates and some of the nodes are seeded with their actual physical coordinates, the time to convergence decreases with the proportion of nodes seeded as expected. Obstacles seem to have a marginal effect on convergence time, though a high density of obstacles seems to increase the convergence time for GSpring only marginally.

6.5.2 Geocast Messages

The average number of geocast messages sent and received by each node during stabilization for the networks evaluated is shown in Figures 6-25 to 6-28. These figures show that the number of geocast messages depends to a large extent on the density of the network. Dense networks without obstacles seem to require significantly fewer geocast messages than sparse networks and networks with obstacles. This is because the regions of ownership for dense networks are smaller and the likelihood that a node will be found in the region of ownership of another node is low.

The intuition for this phenomenon is that there are large voids in the latter networks and these large voids will tend to give rise to large conflict sets. While the number of messages seems somewhat large, note that our current implementation is naive: every node that is within the region of ownership will reply to a querying node. It is quite likely that the number of messages required can be reduced substantially by pruning some replies from the nodes in a conflict set. Also, as nodes observe geocast messages that are routed to them and through them, it is likely that they can use the information contained in the geocast queries and thus save on some geocast query messages as well.

Given these figures, it is natural to ask if GSpring is feasible for deployment in a practical setting. If the interval between each iteration is one minute, then 1,000 iterations is equivalent to about

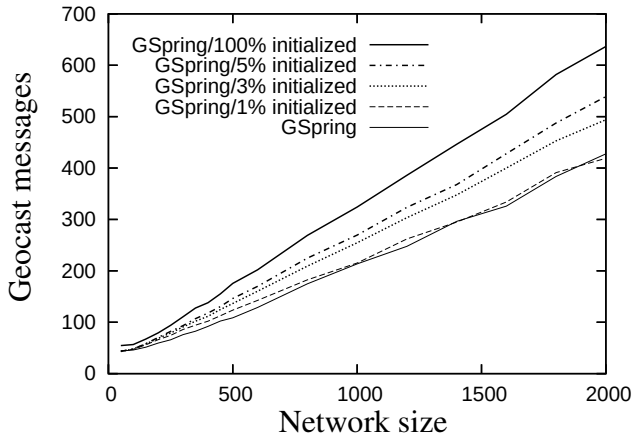


Figure 6-27: Geocast messages sent and received per node for networks with small voids (average node degree 8).

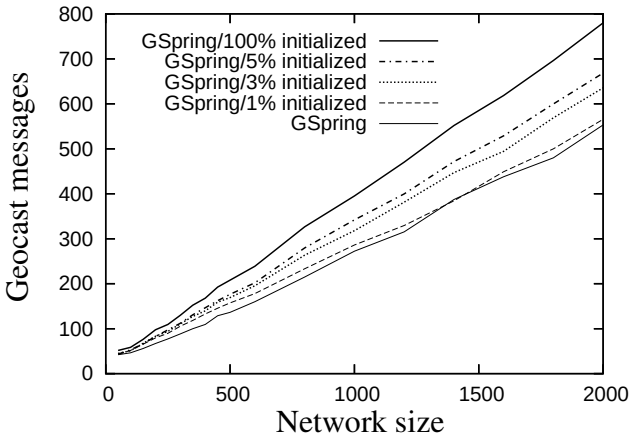


Figure 6-28: Geocast messages sent and received per node for networks with large voids (average node degree 7).

17 hours. If the deployment scenario is a sensor network that is expected to be deployed for weeks and months at a time, then perhaps 17 hours is not too long to wait. In any case, we show in Section 6.5.3 that the number of iterations can be reduced without major adverse effect by increasing damping and hysteresis.

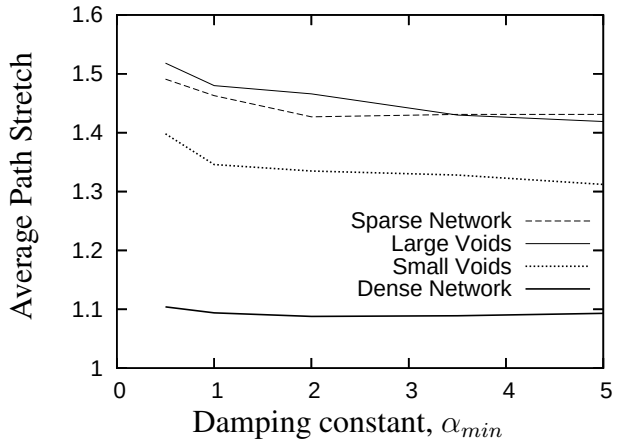
In the same way, with an iteration interval of one minute, to send or receive 200 geocast messages before stabilization is equivalent to processing one geocast message every 5 minutes and thus the geocast message overhead seems modest.

6.5.3 Damping and Hysteresis

We have been conservative in our implementation by setting $\alpha_{max} = 5$. It is likely that with a larger value of α_{max}, GSpring would be able to stabilize with fewer iterations. Similarly, GSpring can be forced to stabilize faster by imposing more damping, i.e., by reducing T in Equation (5.9), or hysteresis, i.e., by increasing α_{min}. Overall, the tradeoff is that increasing α_{max} might introduce some instability in the relaxation process and increasing the amount of damping and hysteresis might cause the system to stabilize in a less optimal configuration. Overall, our current implementation serves only as a proof of concept and has not been fully optimized.

Effect of Damping Constant, α_{min}. The effect of varying the damping constant, α_{min}, from 0.5 to 5 on routing stretch is shown in Figure 6-29. The default value of α_{min} is 1.0. As expected, routing performance gets worse as damping is increased. However, these figures show that the change is not significant.

The corresponding number of iterations required for convergence and the number of geocast messages sent and received per node is shown in Figures 6-30 and 6-31, respectively. These results



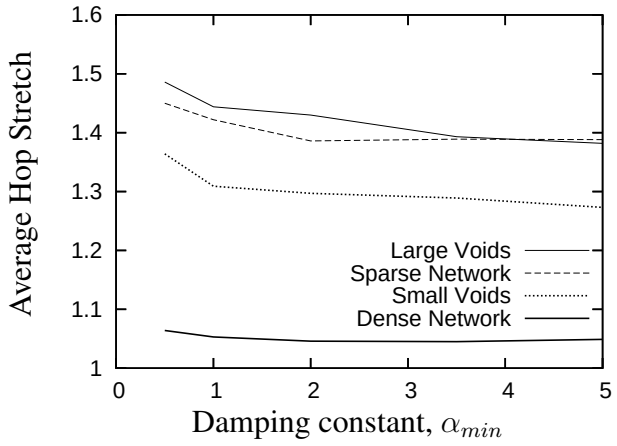


Figure 6-29: Plot of α_{min} against GDSTR stretch.

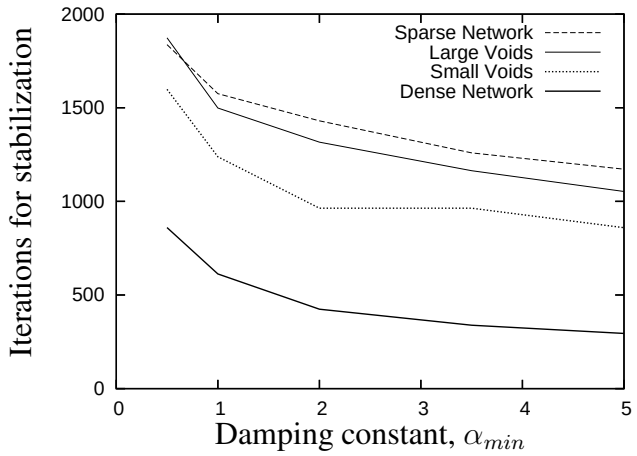


Figure 6-30: Plot of α_{min} against iterations required for stabilization.

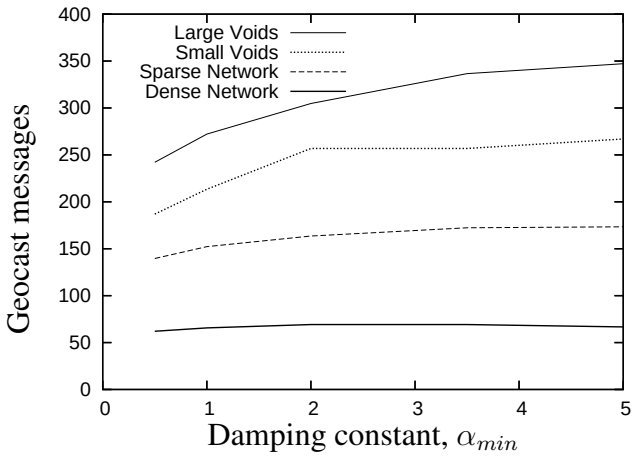
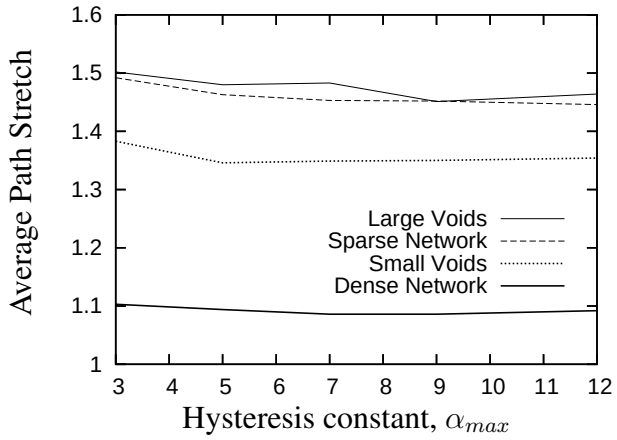


Figure 6-31: Plot of α_{min} against geocast messages sent and received.

demonstrate that we can reduce the number of iterations required for convergence by almost half by increasing the damping constant to 5. The number of geocast messages will be increased slightly in most cases. The reason for this is that with increased damping, nodes are likely to stop adjusting their coordinates sooner, often even before they have completely eliminated all the nodes in their region of ownership.

Effect of Hysteresis Constant, α_{max}. The effect of varying the hysteresis constant, α_{max}, from 3 to 12 on routing stretch is shown in Figure 6-29. The default value of α_{max} is 5.0. As expected, routing performance gets worse as hysteresis is increased. Again, these figures show that the change is not significant.

The corresponding number of iterations required for convergence and the number of geocast messages sent and received per node is shown in Figures 6-33 and 6-34, respectively. These results are similar to that for increasing the damping constant: we can reduce the number of iterations



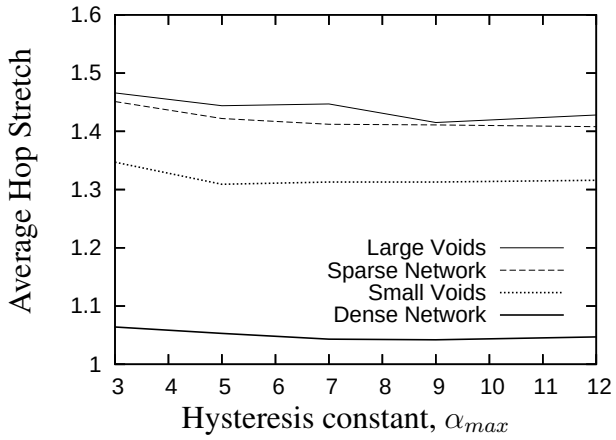


Figure 6-32: Plot of α_{max} against GDSTR stretch.

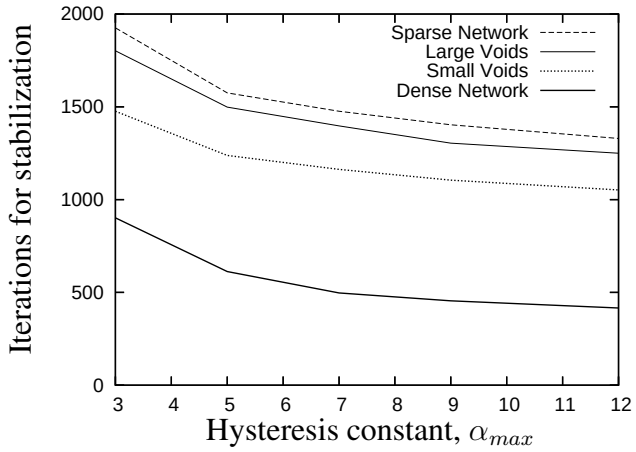


Figure 6-33: Plot of α_{max} against iterations required for stabilization.

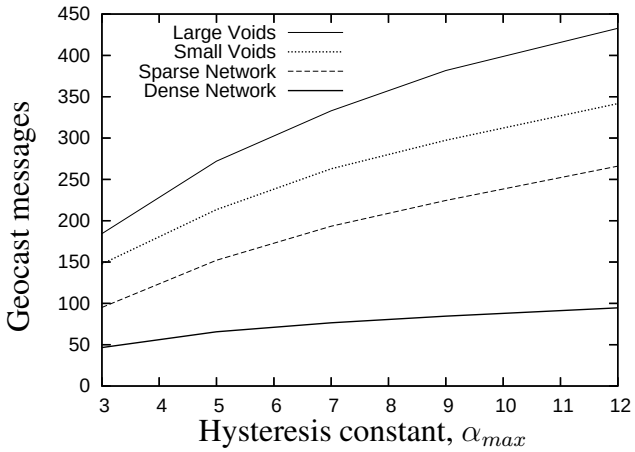
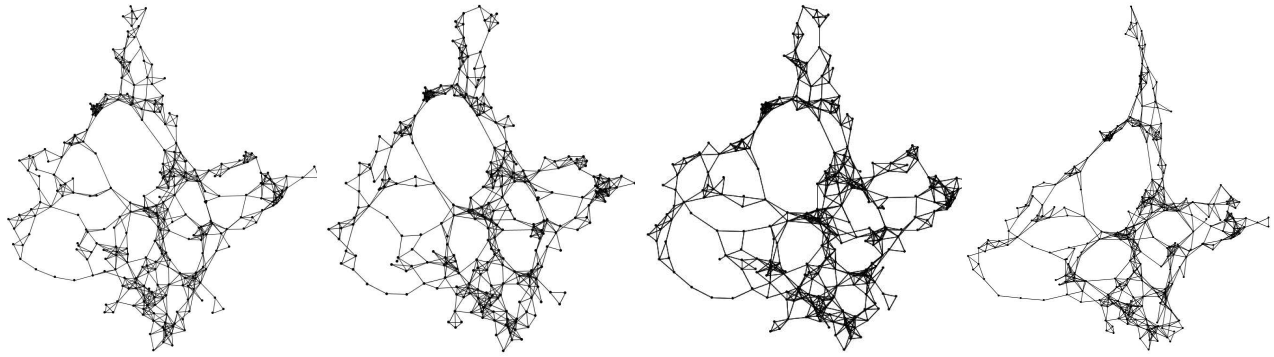


Figure 6-34: Plot of α_{max} against geocast messages sent and received.

required for convergence by almost half by increasing the hysteresis constant to 12. The number of geocast messages will also be increased as the hysteresis applied is increased. The increase rises up to 100% for networks with obstacles as shown in Figure 6-34. Nevertheless, because the total number of messages is not large, we do not expect this to be a major issue.

6.6 Getting Simulation Parameters Right

Since GSpring involves a large number of simulation parameters (i.e., spring rest length, spring constant, repulsion constant, etc.), one major concern is how the various parameters should be set to achieve good performance. In our work, we systematically tried a range of values for each parameter and we found that GSpring seems to be relatively robust to the parameter settings, i.e.,



(a) 20% repulsion (b) 50% repulsion (c) 200% repulsion (d) 500% repulsion

Figure 6-35: Effect of varying repulsion force (δ, R_{max}).

GSpring seems to work reasonably well over a wide range of parameter settings and we do not need to try too hard to get the parameters "right."

The following example serves to illustrate this point: in Figure 6-35, we present the corresponding topologies for the network in Figure 6-19(a) when none of the nodes are initialized, and the conflict node repulsion constant δ and the maximum total repulsion force R_{max} are varied. Comparing these figures with Figure 6-19(e), we see that increasing the repulsion constants will tend to make the final topology somewhat more "spread out", while the general shape of the topology is preserved.

6.7 Summary

We have shown the GSpring is able to converge to virtual coordinates that yield routing performance that is comparable to actual physical coordinates and significantly better than that for NoGeo coordinates for geographic routing algorithms.

In particular, we show that GDSTR [51] is able to achieve 20% lower hop stretch by routing with GSpring coordinates instead of NoGeo coordinates for sparse networks. GSpring achieves even better performance (up to 40% lower hop stretch) for networks with obstacles. For sparse networks, GDSTR routing with GSpring coordinates incurs only a small overhead of about 10% in stretch compared to routing with actual physical coordinates. For dense networks, geographic routing algorithms are often able to achieve unit (optimal) stretch; GSpring coordinates are able to match the routing performance of actual physical coordinates in such cases.

GSpring is the first known algorithm that can derive coordinates that achieve better geographic routing performance than actual physical coordinates. For networks with obstacles, GDSTR routing with GSpring coordinates achieves up to a 10% improvement in stretch compared to routing with actual physical coordinates.

While GSpring typically requires about a thousand iterations for a network to converge to a set of stable coordinates, this is not an issue in a practical setting since GSpring converges relatively quickly to a set of coordinates that are good enough to route with. The higher-level geographic routing algorithm will ensure packet delivery during the transitional period. After a network has reached stable coordinates, a new node joining the system can often stabilize within a few hundred iterations and will typically only cause small and local changes to the virtual routing topology. Also, as described, GSpring is not fully optimized and we can achieve convergence by increasing the damping or hysteresis that is applied to the system.

Chapter 7

Conclusion

In this chapter, we summarize our work and discuss some open questions and possible directions for future research.

7.1 Summary

In this dissertation, we describe and evaluate two new algorithms for geographic routing: *Greedy Distributed Spanning Tree Routing (GDSTR)* and *Greedy Embedding Spring Coordinates (GSpring)*.

7.1.1 Greedy Distributed Spanning Tree Routing (GDSTR)

GDSTR is a geographic routing algorithm that uses a new data structure called a hull tree as the backup routing topology instead of a planar graph like the geographic face routing algorithms. Our simulations show that GDSTR achieves a peak improvement of about 20% in terms of path and hop stretch over the best available geographic face routing algorithm in situations where dead ends are common. GDSTR performance is consistently good over a wide range of network densities and sizes.

Simulation also shows that GDSTR generates significantly less maintenance traffic than CLDP and hence makes geographic routing more practical. GDSTR sends two orders of magnitude fewer messages to build its trees initially than what CLDP sends to construct a planar subgraph, and GDSTR's communication when maintaining existing trees is one order of magnitude less than CLDP.

We found that for sparse networks, the routing performance of GDSTR is consistently better than that for existing face routing algorithms, while for denser and larger networks, existing face routing

algorithms can sometimes achieve slightly lower stretch. We address this issue with GDSTR+, a variant of GDSTR that maintains two additional local hull trees. The key insight is that a few hops of information is often sufficient to guarantee that a node picks the correct forwarding direction around a void.

Our simulations show that GDSTR+ is able to achieve up to a 17% improvement in stretch performance over GDSTR and an 8% lower stretch than GPVFR, the best existing face routing algorithm, for networks where the routing performance of GPVFR surpasses that of GDSTR, and slightly better routing performance than GDSTR with two global trees for sparse networks. This makes GDSTR+ suitable for deployment over highly heterogeneous networks.

We have also shown GDSTR can be extended to implement approximate routing and how geocast can be implemented with hull trees. Our geocast algorithm incurs 10% less overhead with GDSTR+ hull trees when compared to that for GDSTR (with two hull trees) and will likely require no more than twice the minimum number of messages.

GDSTR is immediately applicable to a large class of stationary wireless networks, e.g. roofnets [1, 81] and sensor networks [36, 75]. While we have not explicitly evaluated the performance of GDSTR for mobile networks, our simulations show that GDSTR requires only a small number of packets to set up and repair its routing state. This suggests that it is quite plausible that GDSTR will work well in a mobile setting with some tuning and optimization.

7.1.2 Greedy Embedding Spring Coordinates (GSpring)

GSpring is a new virtual coordinate assignment algorithm that derives good coordinates for geographic routing of location-aware wireless nodes. Starting from a set of initial coordinates derived with a set of elected perimeter nodes, GSpring use a modified spring relaxation algorithm to adjust virtual coordinates incrementally 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 and improves the routing performance of existing geographic routing algorithms.

GDSTR is able to achieve 20% lower hop stretch by routing with GSpring coordinates instead of NoGeo coordinates for sparse networks. GSpring achieves even better performance (up to 40% lower hop stretch) for networks with obstacles. For sparse networks, GDSTR routing with GSpring coordinates incurs only a small overhead of about 10% in stretch compared to routing with actual physical coordinates. For dense networks, geographic routing algorithms are often able to achieve unit (optimal) stretch; GSpring coordinates are able to match the routing performance of actual physical coordinates in such cases.

GSpring is the first known algorithm that can derive coordinates that support better geographic routing performance than actual physical coordinates. For networks with obstacles, GDSTR routing with GSpring coordinates can achieve up to a 10% improvement in stretch compared to routing with actual physical coordinates.

While GSpring typically requires about a thousand iterations for a network to converge to a set of stable coordinates, this is not an issue in a practical setting since GSpring converges relatively quickly to a set of coordinates that are good enough to route with. The higher-level geographic routing algorithm will ensure packet delivery during the transitional period. After a network has reached stable coordinates, a new node joining the system can often stabilize within a few hundred iterations and will typically only cause small and local changes to the virtual routing topology. Also, as described, GSpring is not fully optimized and we can achieve convergence by increasing the damping or hysteresis constants thereby allowing the network to converge at least twice as fast.

7.1.3 Insights

The following is a summary of the insights gained on geographic routing over the course of this research:

1. **Planarization is unnecessary for geographic routing**. Most of the previous work on planarization has focused on deriving planarization algorithms that yielded better geographic routing performance for Unit Disk Graph (UDG) networks [2, 21, 54, 85, 86] and CLDP solves the practical distributed planarization in the general case [42].

   A major contribution of our research is a paradigm shift to improve geographic routing performance, not by incrementally improving the planarization or face routing algorithm, but by thinking about geographic routing in terms of a greedy forwarding mode and a guaranteed-delivery forwarding mode. To some extent our approach is a natural response to the realization that perhaps it is both too expensive and unnecessary to attempt to planarize the network graph.

2. **Convex hulls can aggregate location information efficiently**. The underlying intuition for geographic routing is that the physical location of a node relative to the destination of packet provides a good hint of the correct general forwarding direction. Brad Karp refers to this as the *self-describing* nature of position information [37]. These hints however are insufficient to guarantee packet delivery, which we will also define as *correctness*.

   For geographic face routing algorithms, *correctness* is achieved by exploiting the geometric properties of a planar graph. Our key observation is that a similar guarantee can be obtained by using a hull tree.

   This result is not surprising, considering that a common technique for achieving scalability in traditional networking is the aggregation of information about the address space. It turns out that we can apply the same principle in GDSTR to help it route along a spanning tree, by aggregating the locations covered by subtrees using convex hulls. GDSTR uses the convex hulls to decide which direction in the tree is most likely to make progress towards a given geographic destination.

3. **Getting the forwarding direction right is key when routing around voids**. Existing geographic face routing algorithms [7, 39, 47, 52] differ mainly in the way they handle situations when a packet is routed to a dead end during greedy forwarding. These dead ends usually arise because of voids in the routing topology.

   We make three key observations about such voids:

   (a) The backup routing topology employed when greedy routing fails should attempt to "approximate" these voids, since a topology that conforms to the shape of the voids will allow a packet to be routed around them most efficiently. A planar graph is a natural topology that approximates such voids very well; our insight in developing GDSTR is that it is often no less efficient to use two hull trees instead of a planar graph as the backup routing topology when greedy forwarding fails, and that it is significantly easier to build and maintain hull trees than a planar graph.

   (b) In sparse networks with large voids, it is critical to choose the correct forwarding direction when attempting to route around a void in order to achieve good routing performance, even though either forwarding direction will allow us to guarantee packet delivery. In general, it is impossible to guarantee that the optimal forwarding direction is chosen when the voids are large without global information.

   (c) That said, in a dense network where voids are generally small, we can often pick the optimal forwarding direction by maintaining several hops of information about the routing topology.

4. **Local Information Can Improve Performance by Providing Hints on the Optimal Forwarding Direction**. While it may seem obvious that maintaining more information about the network beyond the one-hop neighborhood would improve routing performance, it was not entirely clear how such information helps. We found that maintaining a little extra information along each planar face is sufficient to allow a node to guess the right forwarding direction more often when the voids are large and be always correct when the voids are small enough for nodes to have complete information about the adjacent voids.

5. **Routing performance can be improved with a judicious choice of routing coordinates**. Geographic routing systems should exploit flexibility in choosing coordinates when location information is not available, to choose virtual coordinates that result in virtual routing topologies with convex voids and fewer crossed links. The former reduces the probability that packets will get trapped in dead ends, while the latter allows the routing address space to be aggregated more efficiently with convex hulls.

   The hypothesis that virtual coordinates which allow greedy forwarding to succeed more often will improve the routing performance for existing geographic routing algorithms is validated over a wide range of random network graphs. To the best of our knowledge, GSpring is the first algorithm to derive coordinates that can achieve better geographic routing performance than actual physical coordinates. In some sense, we can view GSpring as an algorithm that "embeds" part of the required routing tables into the coordinate system.

Finally, we found that it was extremely important to have a good visualization tool when developing and debugging geographic routing algorithms. Without such a tool, it would be very different to understand why things fail if there is a bug in the implementation. It is for this reason that a lot of effort in this work was devoted to the development of a good visualization tool for our network simulator [50].

7.2 Open Issues and Future work

Existing work on geographic routing has focused almost exclusively on two dimensional Euclidean coordinates because face routing only works with a planar graph and is not applicable to higher dimensions. Both GDSTR and GSpring can be generalized to higher dimensions in a straightforward way – both the convex hull and the region of ownership are well-defined entities in higher dimensions. We believe that it is likely that we can achieve better stretch for some networks by routing in higher dimensions. We conjecture that it is easier for GSpring to converge to a greedy embedding in higher dimensional space. Perhaps given d dimensions where $2 < d \leq O(\log n)$, it is possible to show that GSpring will always converge to a greedy embedding.

The aim of our research is to explore new ideas, and due to time constraints, only a limited amount of effort was spent optimizing and exploring the effect of various parameter settings for GSpring. While anecdotal evidence shows that GSpring is relatively robust to many parameter settings, the effect of the various parameters on the convergence time has not been systematically explored. Before GSpring can be deployed in real system, it is essential to understand how parameters should be set to minimize the convergence time of the algorithm without sacrificing the routing performance of the resulting coordinates.

A major motivation for geographic routing is an assumption that geographic routing is cheap because only local (one-hop) information is sufficient to guarantee delivery [37]. The difficulties of practical planarization were not well-understood then. Recent advances in our understanding of distributed planarization algorithms have shown that the cost of planarization can be prohibitive in practice [42, 51]. While GDSTR partially mitigates this cost by obviating the need for planarization, it is perhaps timely to compare the associated costs of GDSTR to traditional ad hoc routing algorithms [35, 66, 68, 69], which maintain $O(n)$ routing state. From our work, we know that the cost of maintaining the routing state for GDSTR is $O(n)$, like traditional ad hoc routing algorithms. We believe that GDSTR is likely to have a lower constant than traditional ad hoc routing algorithms since it has access to location information, but the constants remain to be evaluated in a practical setting.

The effect of mobility on GDSTR and GSpring is another area that remains to be explored. In particular, it may be useful to see if variants of GDSTR and GSpring can be developed for heterogeneous networks, consisting of a mixture of both mobile and stationary nodes, and where individual nodes have differing amounts of available energy.

Appendix A

GDSTR Algorithm Design

In this Appendix, we describe a variant of GDSTR, called GDSTR II, that uses a different convex hull aggregation mechanism that presents a node with a view of the locations accessible via each neighboring node. We also present the results of some simulations for GDSTR and GDSTR II to support our design choices.

In particular, we explore the following dimensions in the GDSTR design space to determine how GDSTR performance and maintenance costs are affected by the following design choices:

1. Hull tree maintenance/tree traversal algorithm (GDSTR versus GDSTR II)
2. Spanning tree algorithm
3. Node traversal ordering
4. Hull tree choosing heuristic

A.1 GDSTR II: A Node-Centric Approach to Hull Tree Maintenance

In Section 3.8.1, we described a different convex hull aggregation mechanism that does not support the notion of a "root". The global hull trees can also use this algorithm to maintain its trees. We call the variant of GDSTR that uses such hull trees for routing GDSTR II.

Each node orders its neighbors in a cyclical fashion and during tree traversal, there is no notion of forwarding up or down a tree. A packet is simply forwarded to the next neighbor in the cyclical ordering. The following is an modified routing algorithm:

Algorithm 8 (GDSTR II) *When a node v receives packet p for destination node t from a neighboring node u, do:*

1. **Preliminary Checks:**
   (a) **Packet Delivery:** *If $v = t$, the packet has been delivered.*
   (b) **Check for switch to Greedy mode:** *If $p.mode \neq$ Greedy and there is at least one immediate neighbor w such that $|wt| < |(p.n_{min})t|$, then set $p.mode :=$ Greedy, $p.n_{min} := w$ and clear $p.n_{anchor}$ and $p.tree$ if they are set. Execute step 2 or 3 according to $p.mode$.*
2. **Greedy Mode:** *Find the node w in the set of immediate neighbors that is closest to the destination t.*
   (a) **Greedy Forwarding:** *If $|wt| < |vt|$, set $p.n_{min} := w$ and forward the packet to w.*
   (b) **Switch to Tree Traversal Mode:** *Choose one of the hull trees for forwarding and set $p.tree$ to the chosen tree's identifier, $p.mode :=$ Tree and $p.n_{anchor} := v$. Then, find the set of child nodes with convex hulls that contain the destination t.*
      - *If set is non-empty, arrange the nodes (relative to $p.tree$) with convex hulls that contain the destination point in an ascending sequence according to the global ordering of node identifiers and forward the packet to the first such node.*
      - *Else, conclude that packet is* **not deliverable**.
3. **Tree Mode:** *If the root for $p.tree$ has changed, follow step 2(b), else follow step 3(a).*
   (a) **Check Termination Condition:** *If $v = p.anchor$ and u is the last child, conclude that packet is* **not deliverable**.
   (b) **Tree Traversal:** *Given an ascending sequence of nodes with convex hulls that contain the destination point, forward to the next neighboring node in this sequence.*

The correctness of GDSTR II follows from the correctness of GDSTR. GDSTR II is effectively equivalent to GDSTR. The key difference lies only in how the convex hull information is recorded and propagated. Corresponding algorithms for approximate routing and geocast can also be derived for GDSTR II.

While the formulation of GDSTR II and its associated geocast algorithm is simpler than GDSTR, there are two tradeoffs: (i) GDSTR II incurs higher maintenance bandwidth even though it does not require conflict hulls since each node has to broadcast more convex hull information. Instead of broadcasting only its own hull, it has to broadcast the convex hulls of all its neighbors from its perspective; and (ii) it turns out that the use of conflict hulls in GDSTR is a more efficient way to prune the routing subtree and hence GDSTR tends to achieve marginally better routing

performance in practice. The reason for the latter is that GDSTR II's aggregation algorithm is rather inefficient for extremal-rooted trees; it is much better for radially-balanced trees.

Another key distinction between GDSTR and GDSTR II is the following: in GDSTR, only the root nodes have a global view of the network; in GDSTR II, every node has a perspective of the global view. While this view does not provide much benefit with respect to routing since the convex hull associated with the parent node in a hull tree (in GDSTR II) is likely to be large and intersect with other convex hulls, it does allow a node to drop packets that are destined for points that are outside of the perimeter of the network. Unfortunately, it is unlikely that such a feature would turn out to be useful in practice since such circumstances are likely to be rare and for data centric applications, the goal is to route a packet to the node that is closest to the destination point even if the destination is outside the perimeter of the network. Finally, there is a cost associated with providing each node with a global view of the network: network changes are likely to require more nodes to be updated and thereby increase maintenance cost.

We also explored the effect of various design choices and heuristics on GDSTR II, and found that unlike GDSTR, the *minimal-path* spanning tree works best and also that picking the neighboring node with a convex hull containing the destination that is nearest to the destination works best. The storage savings the GDSTR II has over GDSTR also turn out to be quite insignificant in practice.

A.2 Design Choices

In this section, we describe the various options for the major design choices, including the spanning tree algorithm, the node traversal ordering and the hull tree choosing heuristic.

A.2.1 Spanning Tree Algorithms

Because the constraints for correctness are much stricter for planarizations, it will in general require more effort to maintain a planar subgraph than a spanning tree. In fact, a distributed spanning tree has only two criteria for correctness:

1. Each node, except for the root node, has exactly one parent node.
2. Each node must be connected. We guarantee this by ensuring that every node has a common view of the root of the tree.

Both these conditions can be checked locally by a node by communicating only with immediate neighbors. On the other hand, the only known technique for detecting and eliminating non-planar edges in a connected graph requires non-local face traversals [42]. The GDSTR routing algorithm

will work correctly as long as we have a rooted spanning tree. In this section, we describe some algorithms that will produce rooted spanning trees.

Given a hull tree with a specific root, GDSTR employs the following *Minimal-Depth Spanning Tree* algorithm. A node, n, chooses its parent node as follows:

> **Algorithm 9 (Minimal-Depth Spanning Tree)** *Determine the set of neighboring nodes that have minimal depth, i.e., are at the smallest number of hops from the root.*
>
> - *If there is only one node in the set, choose that node as the parent.*
> - *If there is more than one node in the set, choose the node that is closest in geometric distance to n as the parent.*

Closely related is the following *Minimal-Path Spanning Tree*:

> **Algorithm 10 (Minimal-Path Spanning Tree)** *Determine the set of neighboring nodes that have minimal path length to the root.*
>
> - *If there is only one node in the set, choose that node as the parent.*
> - *If there is more than one node in the set, choose the node that is closest in geometric distance to n as the parent.*

These algorithms will produce minimal spanning trees (in terms of either path length or hops) rooted at extremal nodes. The expected advantage of these trees is that a packet is will be able to traverse the entire tree in a small number of hops or path length. The actual routing performance is related to D, the diameter of the network. The disadvantage of such trees is that when the network density is high, some intermediate nodes may end up with a large number of children. Since each child has an associated hull, the amount of state stored per node will therefore be proportional to network density, and not constant.

To keep the amount of state stored at each node constant and independent of network density, we can reduce the number of children at each node while still avoiding intersections of the convex hulls as far as possible with the following variants of the above algorithms as follows:

> **Algorithm 11 (Sparse Minimal-Depth Spanning Tree)** *Given a global ordering for node identifiers, find the set of neighboring nodes with a higher priority that is at the same number of hops from the root.*
>
> - *If this set is non-empty, then:*
>   - *If there is only one node in the set, choose that node as the parent.*
>   - *If there is more than one node in the set, choose the node that is closest in geometric distance as the parent.*

- *Else, if there are no neighboring nodes at the same number of hops from the root, pick the node that is closest in geometric distance from the set of nodes that are one hop nearer to the root as the parent.*

Algorithm 12 (Sparse Minimal-Path Spanning Tree) *Given a global ordering for node identifiers, find the set of neighboring nodes with a higher priority that is at the same number of hops from the root.*

- *If this set is non-empty, then:*
  - *If there is only one node in the set, choose that node as the parent.*
  - *If there is more than one node in the set, choose the node that is closest in geometric distance as the parent.*
- *Else, if there are no neighboring nodes at the same number of hops from the root, pick the node that has the shortest path to the root as the parent.*

The distributed spanning tree algorithms described above are very strict, i.e. given a network configuration, each spanning tree is uniquely specified. One drawback of having a strict specification is that whenever a node joins or leaves the network, all the trees will often need to re-configured. Since the criteria for correctness for a distributed spanning tree are relatively relaxed, the following "lazy" variants can be used to reduce the need for topological updates:

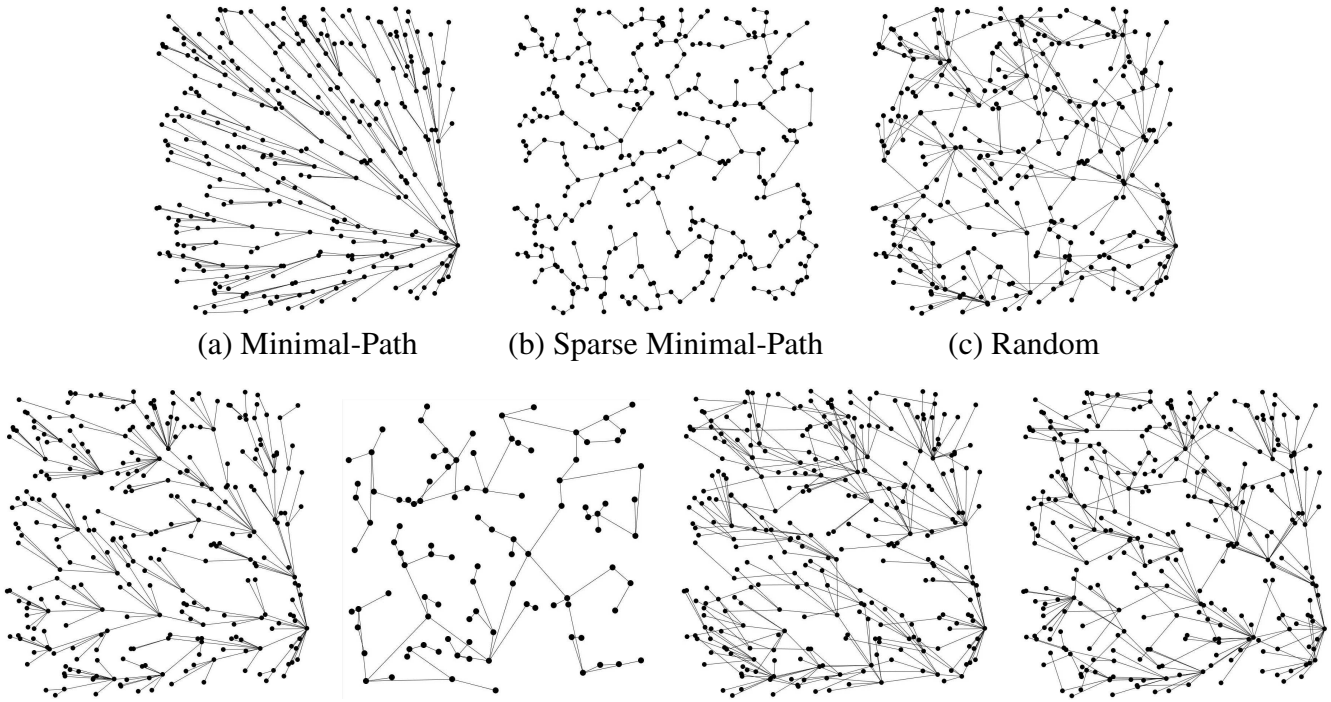
Algorithm 13 (Lazy Minimal-Depth Spanning Tree) *Assume we have a global ordering for node identifiers.*

- *If current parent is closer to the root in hop count, or at the same number of hops from the root and has a higher priority node identifier, do nothing.*
- *If there is only one node in the set, choose that node as the parent.*
- *If there is more than one node in the set, choose the node that is closest in geometric distance to n as the parent.*

Algorithm 14 (Lazy Sparse Minimal-Depth Spanning Tree) *Assume we have a global ordering for node identifiers.*

- *If current parent is closer to the root in hop count, or is at the same number of hops from the root and has a higher priority node identifier, do nothing.*
- *Find the set of neighboring nodes that has a lower hop count to the root or that are of equal hop count and a higher priority. Select the node in this set that is closest in geometric distance as the parent.*

The key drawback of these "lazy" variants is that the resulting trees will likely have more intersecting convex hulls because the trees formed have less structure. Sample spanning trees formed with the various tree building algorithms are shown in Figure A-1. We see from these figures that the trees produced by these two "lazy" algorithms are somewhat similar in structure.



(a) Minimal-Path (b) Sparse Minimal-Path (c) Random

(d) Minimal-Depth (e) Sparse Minimal-Depth (f) Lazy Minimal-Depth (g) Lazy Sparse Minimal-Depth

Figure A-1: Sample trees generated by the various distributed spanning tree algorithms.

A.2.2 Tree Traversal Heuristics

Tree Traversal Ordering. The tree traversal algorithm is correct if we impose a strict global ordering on all the nodes in the system: when a node has to pick the next node to forward a packet during tree traversal, it simply picks the next neighbor according to this global ordering.

However, we observe that a global ordering is not necessary for correctness. Any consistent ordering for each source-destination pair will work. Hence, we investigated the effect of the following neighbor orderings on GDSTR routing performance:

1. **Minimal-Distance:** Order the neighboring nodes in ascending order in terms of their distance from the destination point.
2. **Smallest-Angle:** Order the neighboring nodes in ascending order in terms of the angle that they make relative to the imaginary line between a node and the destination. The idea is that we want to try the nodes that are in approximately the right direction first.
3. **Smallest-Area:** Order the neighboring nodes in ascending order according to the areas of their convex hulls, i.e., try the nodes that have smaller hulls first.

4. **Minimal-Height:** Order the neighboring nodes in ascending order according to their height, i.e., try the nodes with hulls that have a smaller number of hops to the leaves first[1].

5. **Random:** Order the neighboring nodes in some arbitrary (but fixed) order.

The neighboring nodes referred to above are the neighboring nodes that are neighbors in the associated hull tree, with convex hulls that contain the destination node. Nodes that are immediate neighbors but which are not neighbors in the hull tree or which have hulls that do not contain the destination node are not traversed.

Tree Choosing Heuristics When a packet switches from greedy forwarding to tree traversal and there are multiple trees, a node will have to decide which hull tree to use for tree traversal mode. In principle, any tree will work, but as explained in Section 3.3, it is critical to choose the correct tree to achieve good routing performance. For GDSTR, there are two possible scenarios: (i) at least one hull tree has a convex hull that contains the destination and (ii) none of the hull trees have a convex hull that contains the destination. For GDSTR II, only the former matters because in the latter case, GDSTR II can conclude that a packet is undeliverable.

We define the *height* of a node as the maximal number of hops from the node to the descendant node that is farthest down the tree; similarly, we define the *depth* of a node as its hop count from the root of the tree.

The following are the tree choosing heuristics when there are more than one hull trees that contain the destination that we evaluated:

1. **Minimal Distance:** Choose the tree for which the child node with a hull containing the destination is nearest to the destination point in geometric distance.

2. **Minimal Angle:** Choose the tree for which the child node with a hull containing the destination makes the smallest angle relative to the imaginary line between a node and the destination.

3. **Minimal Area:** Choose the tree for which the child node with a hull containing the destination has a hull with the smallest area.

4. **Minimal Height:** Choose the tree for which the child node with a hull containing the destination has the lowest height.

5. **Nearest Destination:** Choose the tree whose root that is nearest to the destination in terms of geometric distance.

6. **Nearest Root:** Choose the tree which has a root that is nearest to the forwarding node in terms of geometric distance.

[1]To implement this heuristic, as convex hull information is aggregated back up a tree, we compute the *height* of each node in the tree. We define the *height* of a node as the maximal number of hops from the node to the descendant node that is farthest down the tree; similarly, we define the *depth* of a node as its hop count from the root of the tree.

The following are the tree choosing heuristics when none of the hull trees contain the destination point that we evaluated (applicable to GDSTR only):

1. **Minimal Angle:** Choose the tree for which the parent node makes the smallest angle relative to the imaginary line between a node and the destination.
2. **Minimal Depth:** Choose the tree for which the forwarding node is at the minimal depth, i.e., fewest hops from the root.
3. **Minimal Distance:** Choose the tree that has a parent node that is nearest to the destination point in geometric distance.
4. **Nearest Destination:** Choose the tree whose root that is nearest to the destination in terms of geometric distance.
5. **Nearest Root:** Choose the tree which has a root that is nearest to the forwarding node in terms of geometric distance.

For comparison, we also evaluated the simple strategy of choosing from the set of available trees at random (which we call **Random**).

When none of the hull trees contain the destination point, GDSTR will forward packet to the parent node in some tree. Our objective is to find a heuristic that will produce the best choice for the parent node. Some of these heuristics take into account the position of the parent relative to the destination, while others take into account the structure of the tree. For example, by choosing the tree with the minimal depth, we surmise it will take fewer hops for a packet to traverse the tree if the destination is to be found in a distant branch of the hull tree.

A.3 Evaluation Methodology

The dimensionality of the GDSTR design space is too high for us to probe the entire space. In addition to the hull maintenance/tree traversal algorithm, the spanning tree algorithm, the node traversal ordering and hull tree choosing heuristics, there are other system parameters like the number of hull trees, and the value of r, the maximum size for the convex hulls, that may interact to affect relative routing performance. The actual topology of the underlying network is also likely to affect the relative performance of the various design choices or heuristics.

Our goal is to explore a new approach to geographic routing using hull trees and not attempt to fine-grained optimization of routing performance, so we do not attempt to completely characterize the entire design space. Instead we adopt the following simple approach: we vary each design parameter in turn and systematically derive a set of choices that we believe is "optimal" and run further simulations varying one choice at a time while holding the other choices constant. The

process then allows us to conclude that our design choices are optimal subject to the variation of one parameter.

As the underlying evaluation topology, we use the sets of networks with 25 to 500 nodes randomly scattered over a 10×10 unit square, described in Section 4.1. For each density, we generate 200 networks, and then route 20,000 packets using each algorithm between randomly chosen pairs of source and destination nodes. The performance measurements presented are the average over the 200 times 20,000 data points.

In our evaluation, we focus on evaluating routing performance with two hull trees and we do not limit the value of r, the maximum size for the convex hulls. We measure routing performance with respect to two metrics: (1) *path stretch*, and (ii) *hop stretch*. Path stretch is the ratio of the total path length to the shortest path (in Euclidean distance) between two nodes; 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 measure maintenance costs with respect to the following metrics:

- Storage required at each node
- Size of broadcast message

For each of these metrics, we not only consider the average values over many trials, but we also note the maximal value over all trials. The latter is important since we are working with trees, and one of the issues with trees is that the distribution of such characteristics is likely to be highly non-uniform. A small number of nodes in tree-like systems will tend to have significantly higher than average loads and hence we want to ensure that our final algorithm does not consume an inordinate amount of resources for such loads. It is unacceptable if the consumption for most loads is low, but a few of the nodes are so overloaded that they break down.

A.4 Results for GDSTR

We found that the optimal combination of parameter choices for GDSTR is as follows:

- **Spanning tree algorithm:** Minimal-Depth
- **Traversal ordering:** Negligible effect, any fixed ordering will work
- **Tree Choosing (Hull Trees do not contain destination):** Nearest Destination
- **Tree Choosing (Hull Trees contain destination):** Negligible effect, any random choice among the trees with a convex hull that contains the destination works.

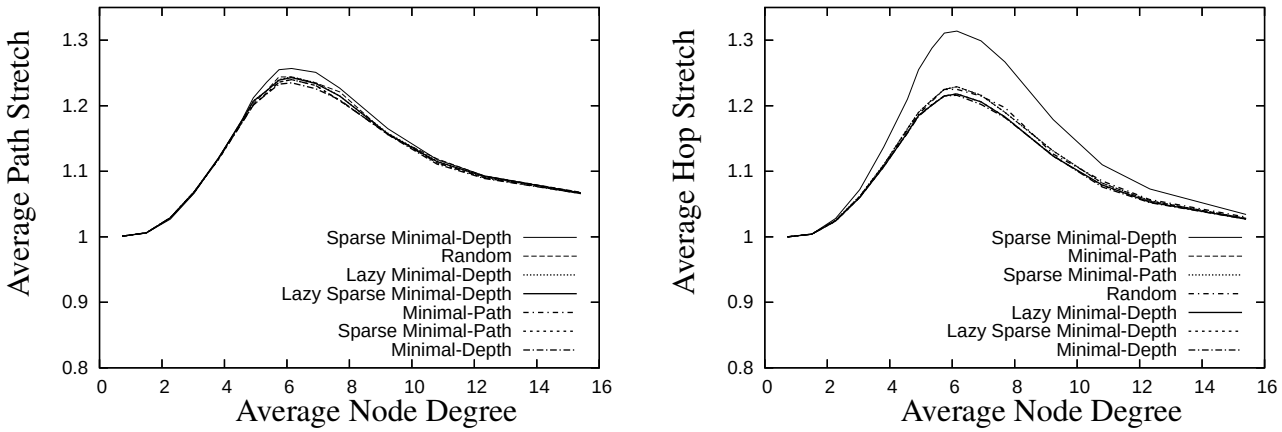


Figure A-2: Effect of the spanning tree algorithm on GDSTR routing performance.

Spanning tree algorithm. The effect of the spanning tree algorithm on GDSTR routing performance is shown in Figure A-2. These results show that while the sparse minimal-depth spanning tree yields noticeably worse routing performance compared to the other spanning trees, the other spanning tree algorithms seem to yield comparable routing performance. The minimal-depth tree seems to yield the best stretch overall. Also, not considering the sparse minimal-depth spanning tree, the peak difference in the routing performance of the different spanning tree algorithms is less than 3%. It is interesting to note that even a random tree will yield relatively good routing performance.

While the lazy variants of the spanning tree algorithms also yield good routing performance, we found that they do so only in the case of point-to-point routing to a existing destination node in the network. To evaluate the performance of GDSTR under a scenario where non-deliverable packets are common, we also route 20,000 packets with undeliverable destinations for each network. We take the average number of hops taken by these packets before GDSTR terminates and concludes that they are undeliverable, and divide this number by the average of the minimal hop paths between two randomly chosen nodes in the network. We refer to the resulting ratio as *undeliverable stretch*. The corresponding results are shown in Figure A-3.

We see from these results that lazy variants require a packet to visit more nodes before GDSTR can conclude that it is undeliverable for dense networks. This arises from the fact that the lazy variants will generate hull trees with more intersecting convex hulls. When a destination falls within the intersection of two or more convex hulls, the traversed subtree is increased in size and it takes more hops to completely traverse the tree.

In Figure A-4, we plot the storage requirements for the various spanning tree algorithms. The sparse minimal-depth tree imposes a somewhat higher requirement on the nodes, and a slightly lower maximal requirement. This is expected because the sparse minimal-depth tree limits the degree of each node in the tree, but increases the depth of the tree and the number of non-leaf nodes. As for the remaining spanning tree algorithms, they seem to have similar storage requirements.

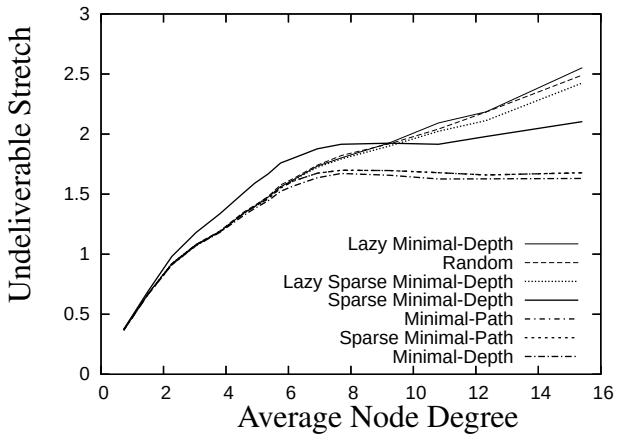


Figure A-3: Effect of the spanning tree algorithm on GDSTR undeliverable stretch.

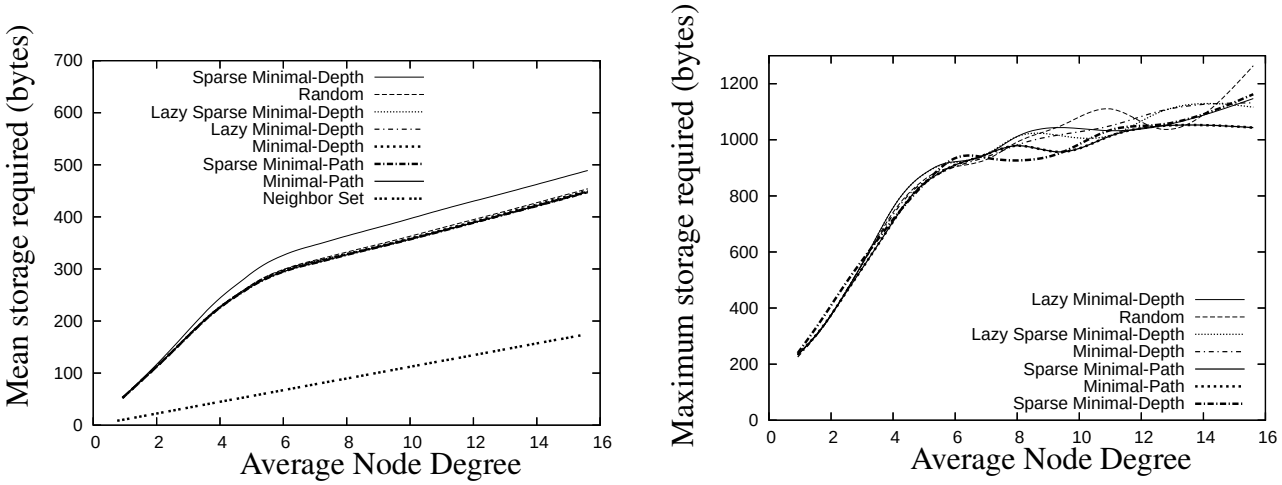


Figure A-4: Effect of the spanning tree algorithm on GDSTR storage requirement.

In Figure A-5, we plot the maximum size of the broadcast messages. From these results, we see that the average size of the messages for the deterministic trees (minimal-depth, minimal-path and sparse minimal-path) are somewhat constant at higher network densities, while that for the remaining trees increases with network density. This is due to the increase in the number of conflict hulls at higher network densities.

Traversal Ordering. In our simulations, we found that traversal ordering does not have any noticeable effect on routing performance. The results are not presented here since the plots for different traversal orderings are visually indistinguishable.

Tree Choosing Heuristic. When GDSTR has to be switch to tree traversal mode, a node can either find that some of the hull trees contain the destination or that none of its hull trees contains the destination. In the latter, the logical thing is to forward a packet down the hull tree to a child node that has a convex hull containing the destination. It turns out that if there are several hull trees that satisfy this condition, the choice of hull tree does not have any noticeable effect on routing

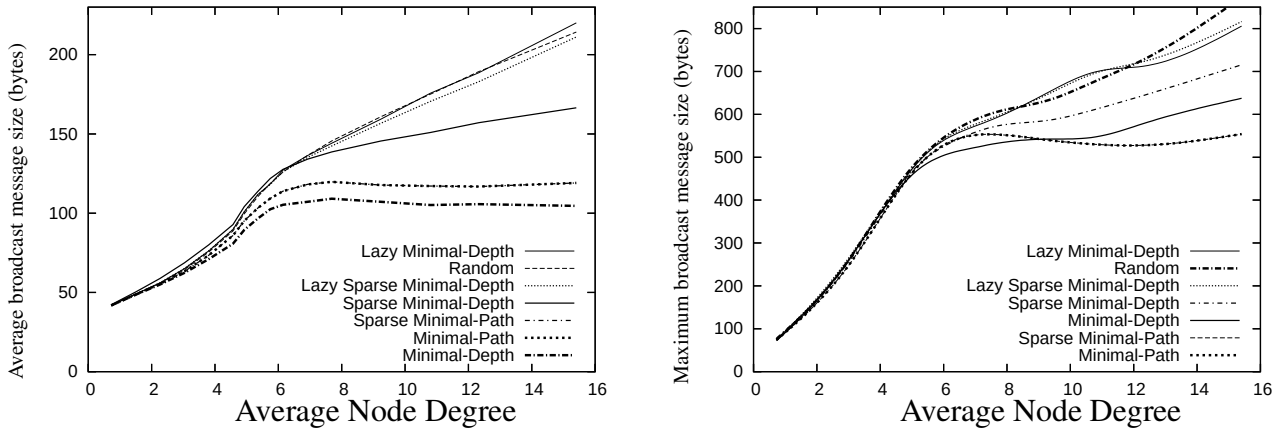


Figure A-5: Effect of the spanning tree algorithm on size of GDSTR broadcast messages.

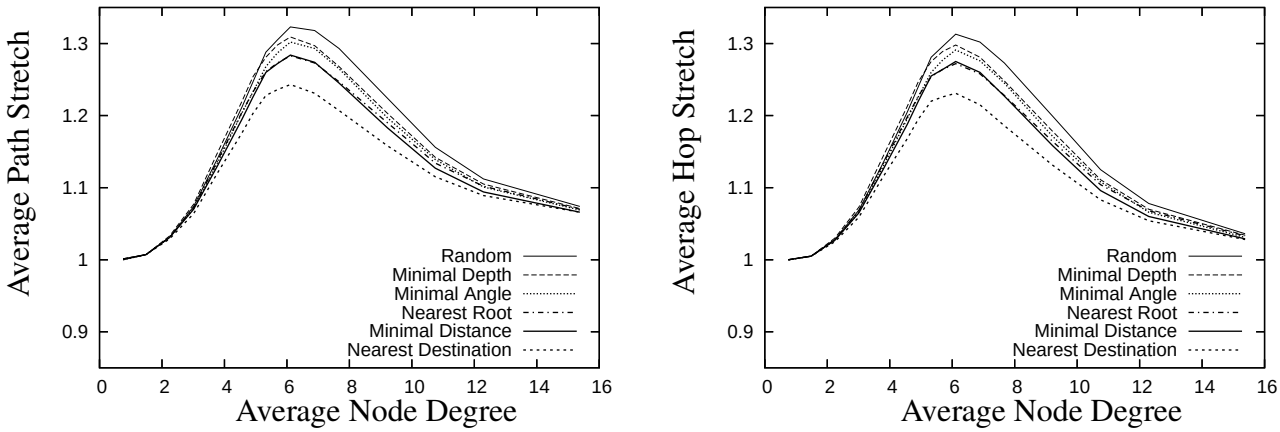


Figure A-6: Effect of the tree choosing heuristic (when none of the hull trees contain the destination) on GDSTR routing performance.

performance and the results are not presented here since the plots for different traversal orderings are visually indistinguishable. In the latter, the choice of hull tree has a significant effect as shown in Figure A-6. We see that optimal routing performance is achieved by choosing the hull tree with a root node that is closest to the destination. It turns out that with this heuristic a packet will be forwarded in the correct general direction most often and hence achieves the best performance.

A.5 Results for GDSTR II

We found that the optimal combination of parameter choices for GDSTR is as follows:

- **Spanning tree algorithm:** Minimal-Path

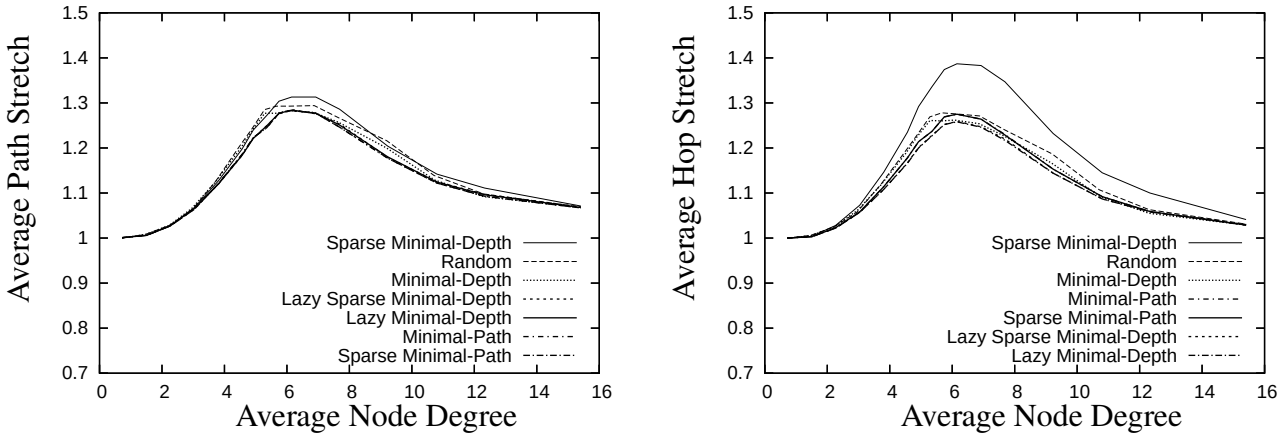


Figure A-7: Effect of the spanning tree algorithm on GDSTR II routing performance.

- **Traversal ordering:** Minimal Height, has a marginal effect.
- **Tree Choosing (Hull Trees contain destination):** Minimal Distance

Spanning tree algorithm. The effect of the spanning tree algorithm on GDSTR routing performance is shown in Figure A-7. Again, the sparse minimal-depth tree yields significantly worse path and hop stretch performance, while the difference between the other spanning tree algorithms is relatively small.

The undeliverable stretch for the various spanning tree algorithms is shown in Figure A-8. Like for GDSTR, the sparse minimal-depth tree yields markedly worse performance than the other algorithms and the minimal-depth tree seems to yield the best performance. Comparing Figures A-7 and A-3, we see that the undeliverable stretch for GDSTR II is higher than that of GDSTR for dense networks. This is not unexpected since GDSTR II is similar to GDSTR without conflict hulls. In these experiments, the majority of the packet destinations are within the boundaries of the network. In cases where the majority of packet destinations lies outside the boundaries of the network, it is likely that GDSTR II will yield significantly better performance, since each node in the network has a view of the boundaries and can conclude immediately that such packets are undeliverable. The conflict hulls for GDSTR allow nodes to conclude that a packet is undeliverable only if the destination falls within a convex hull, packets with destinations that fall outside the boundaries of the network will still only terminate at the root. This drawback can however be easily overcome by having the root node broadcast the hull of the entire network down a hull tree.

The storage requirements per node and the size of broadcast messages for GDSTR II are shown in Figures A-9 and A-10, respectively. These figures show that the spanning tree algorithm has a marginal effect on the storage requirements and broadcast message size. The mean storage requirement for GDSTR II is higher than GDSTR, whole the maximum storage requirement is lower. This is not surprising since with GDSTR II, all nodes (including the leaf nodes) will have convex hulls that cover all the nodes in the network. In contrast, with GDSTR, leaf nodes will store almost no state and have only a partial view of the network.

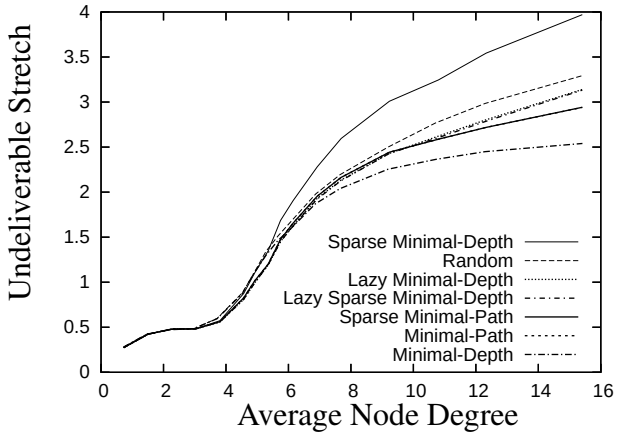


Figure A-8: Effect of the spanning tree algorithm on GDSTR II undeliverable stretch.

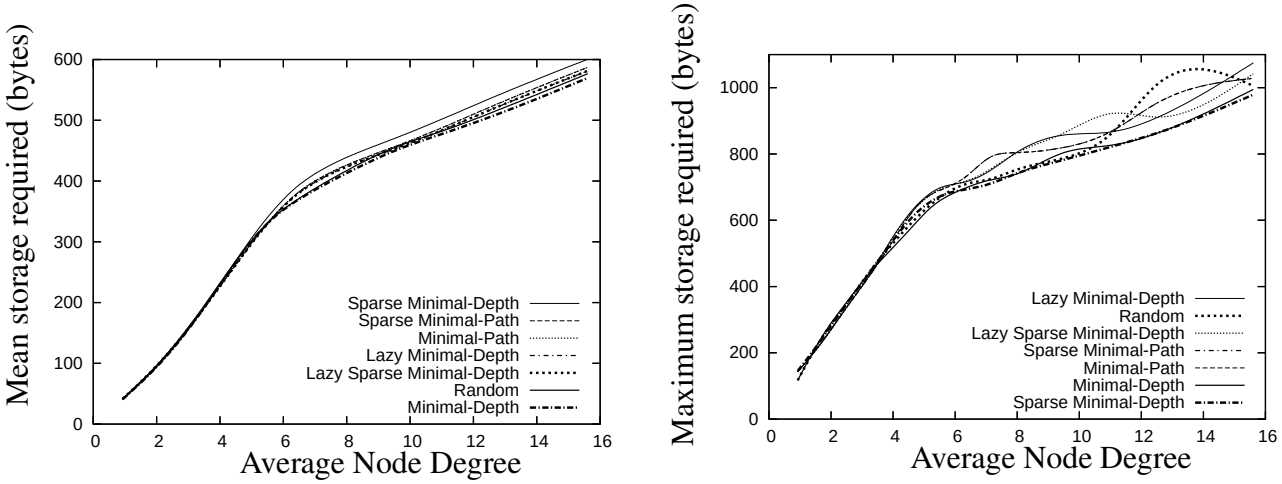


Figure A-9: Effect of the spanning tree algorithm on GDSTR II storage requirement.

Traversal Ordering. The effect of tree traversal ordering is shown in Figure A-11. The minimal height heuristic seems to be the best choice, but the overall effect is small and seen only for the networks with average node degrees between 5 and 9.

Tree Choosing Heuristic. The effect of tree traversal ordering is shown in Figure A-12. The minimal height heuristic seems to be the best choice, but the overall effect is small and seen only for the networks with average node degrees between 5 and 9.

A.6 Summary

In summary, the routing performance of GDSTR is slightly better than that for GDSTR II. The difference in the simulation study above may not be large, but we found that the difference in

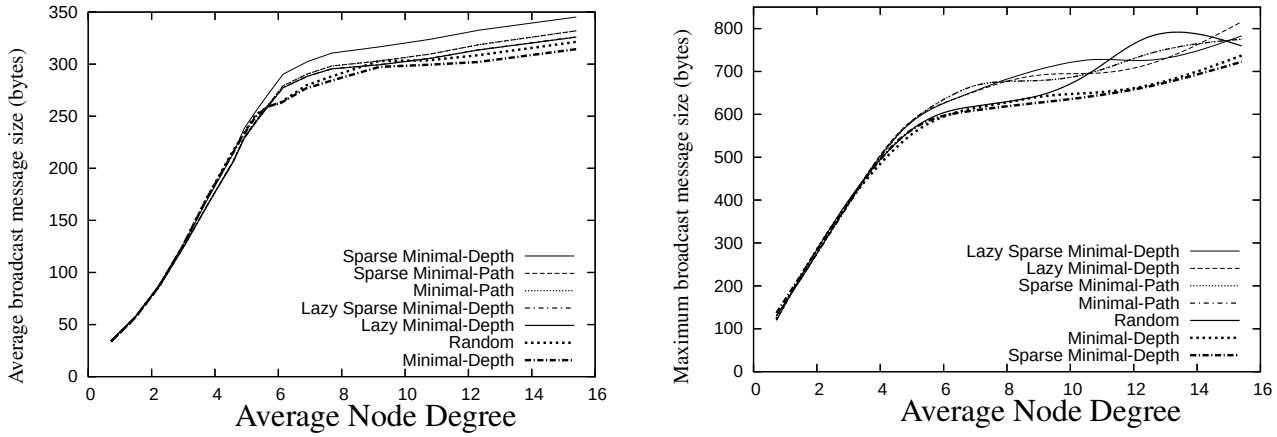


Figure A-10: Effect of the spanning tree algorithm on size of GDSTR II broadcast messages.

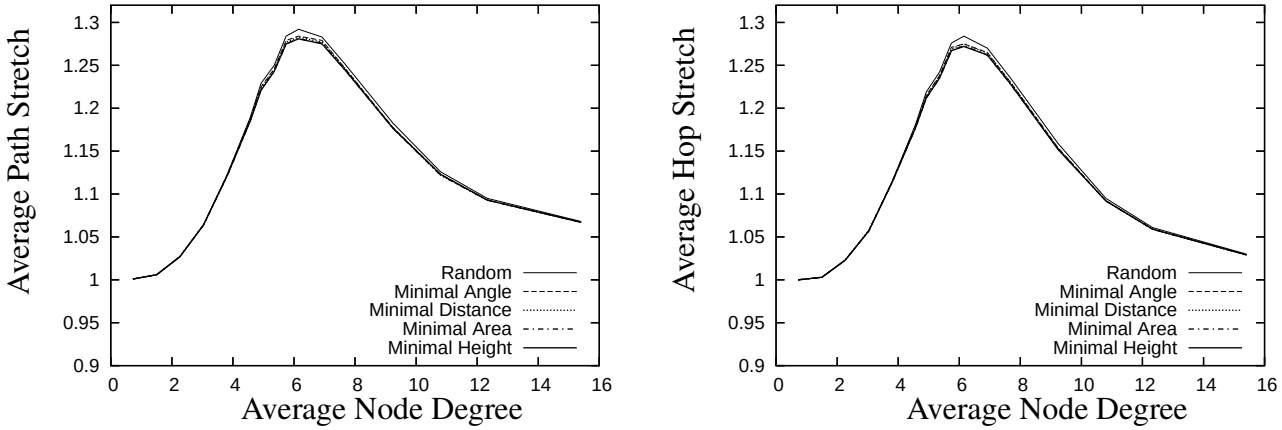


Figure A-11: Effect of the tree traversal ordering on GDSTR II routing performance.

routing performance is magnified as the network size increases. Also for data-centric applications, it is likely that packets will often be routed to destinations that do not correspond to specific nodes. In such cases, the undeliverable stretch is the more important criterion and hence GDSTR is again the preferred algorithm. The storage requirements for GDSTR are slightly lower and the GDSTR message sizes are slightly smaller than those for GDSTR II.

Hence, while GDSTR is somewhat more complicated than GDSTR II and involves the maintenance of conflict hulls, it is preferable over GDSTR II. Our simulations show that the optimal spanning tree algorithm is the *minimal-depth tree* and the only heuristic that has a significant effect on routing performance is the choice of tree hull when a node finds that none of its child nodes have convex hulls that contain the destination. In such cases, the hull tree with a root nearest to the destination should be picked.

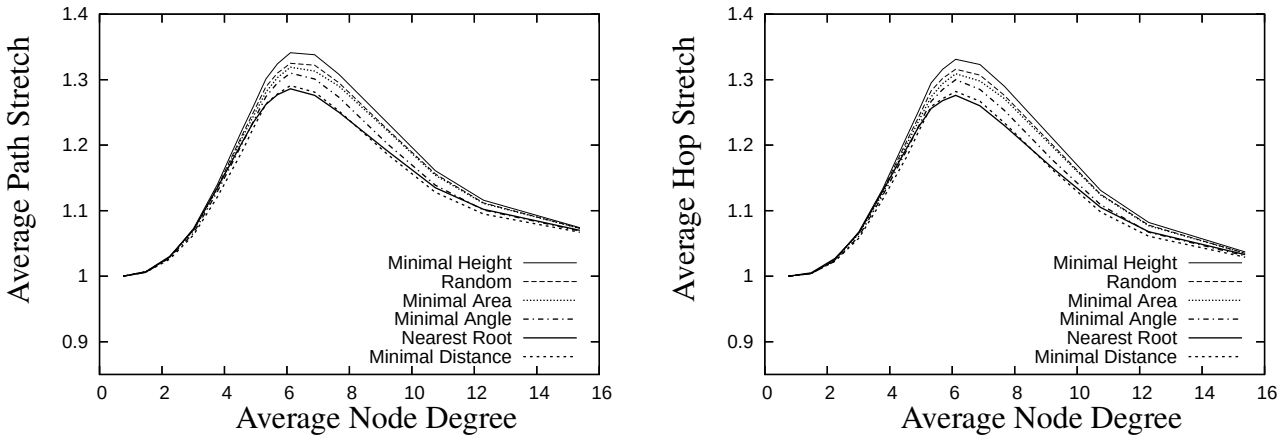


Figure A-12: Effect of the tree choosing heuristic on GDSTR II routing performance.

Bibliography

[1] Daniel Aguayo, John Bicket, Sanjit Biswas, Glenn Judd, and Robert Morris. Link-level measurements from an 802.11b mesh network. In *Proceedings of ACM SIGCOMM Conference 2004*, August 2004.

[2] Khaled Alzoubi, Xiang-Yang Li, Yu Wang, Peng-Jun Wan, and Ophir Frieder. Geometric spanners for wireless ad hoc networks. *IEEE Transactions on Parallel and Distributed Systems*, 14(5), May 2003.

[3] Stefano Basagni, Imrich Chlamtac, Violet R. Syrotiuk, and Barry A. Woodward. A distance routing effect algorithm for mobility (dream). In *Proceedings of the 4th annual ACM/IEEE international conference on Mobile computing and networking (MobiCom '98)*, pages 76–84, New York, NY, USA, 1998. ACM Press.

[4] Prosenjit Bose and Pat Morin. Online routing in triangulations. In *ISAAC: 10th International Symposium on Algorithms and Computation (formerly SIGAL International Symposium on Algorithms))*, 1999.

[5] Prosenjit Bose and Pat Morin. Competitive online routing in geometric graphs. In *Proceedings of the 8th International Colloquium on Structural Information and Communication Complexity (SIROCCO 2001)*, 2001.

[6] Prosenjit Bose, Pat Morin, Andrej Brodnik, Svante Carlsson, Erik D. Demaine, Rudolf Fleischer, J. Ian Munro, and Alejandro Lopez-Ortiz. Online routing in convex subdivisions. In *International Symposium on Algorithms and Computation*, pages 47–59, 2000.

[7] Prosenjit Bose, Pat Morin, Ivan Stojmenovic, and Jorge Urrutia. Routing with guaranteed delivery in ad hoc wireless networks. *Wireless Networks*, 7(6):609–616, 2001.

[8] Nirupama Bulusu, John Heidemann, and Deborah Estrin. GPS-less low-cost outdoor localization for very small devices. *IEEE Personal Communications*, 7(5):28–34, October 2000.

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

[10] Edgar Chavez, Stefan Dobrev, Evangelos Kranakis, Jaroslav Opatrny, Ladislav Stacho, and Jorge Urrutia. Traversal of a quasi-planar subdivision without using mark bits. *Journal of Interconnection Networks*, 5(4):395–408, 2004.

[11] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: A decentralized network coordinate system. In *Proceedings of the ACM SIGCOMM '04 Conference*, Portland, Oregon, August 2004.

[12] Douglas S. J. De Couto and Robert Morris. Location proxies and intermediate node forwarding for practical geographic forwarding. Technical report, MIT, 2001.

[13] Lance Doherty, Kristofer S. J. Pister, and Laurent El Ghaoui. Convex position estimation in wireless sensor networks. In *Proceedings of IEEE Infocom '01*, pages 1655–1663, 2001.

[14] Cheng Tien Ee, Sylvia Ratnasamy, and Scott Shenker. Practical data-centric storage. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI 2006)*, May 2006.

[15] Qing Fang, Jie Gao, and Leonidas J. Guibas. Locating and bypassing routing holes in sensor networks. In *Proceedings of IEEE Infocom '04*, March 2004.

[16] Qing Fang, Jie Gao, Leonidas J. Guibas, Vin de Silva, and Li Zhang. GLIDER: Gradient landmark-based distributed routing for sensor networks. In *Proceedings of IEEE Infocom '05*, March 2005.

[17] Gregory G. Finn. Routing and addressing problems in large metropolitan-scale internetworks. Technical Report ISI/RR-87-180, ISI, March 1987.

[18] Sally Floyd and Van Jacoboson. The synchronization of periodic transmission messages. *IEEE/ACM Transactions on Networking*, 2(2):122–136, April 1994.

[19] Rodrigo Fonseca, Sylvia Ratnasamy, Jerry Zhao, Cheng Tien Ee, David Culler, Scott Shenker, and Ion Stoica. Beacon vector routing: Scalable point-to-point routing in wireless sensornets. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI 2005)*, May 2005.

[20] K. Ruben Gabriel and Robert R. Sokal. A new statistical approach to geographic variation analysis. *Systematic Zoology*, 18:259–278, 1969.

[21] Jie Gao, Leonidas J. Guibas, John Hershberger, Li Zhang, and An Zhu. Geometric spanner for routing in mobile networks. In *Proceedings of the 2001 ACM Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc 2001)*, pages 45–55, July 2001.

[22] Silvia Giordano and Ivan Stojmenovic. Position based routing algorithms for ad hoc networks: A taxonomy. *Ad Hoc Wireless Networking*, pages 103–136, 2004.

[23] Ronald L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Info. Proc. Letters*, 1:132–133, 1972.

[24] Ramakrishna Gummadi, Nupur Kothari, Young-Jin Kim, Ramesh Govindan, Brad Karp, and Scott Shenker. Reduced-state routing in the internet. In *Proceedings of the Third ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets 2004)*, November 2004.

[25] Zygmunt J. Haas. A routing protocol for the reconfigurable wireless networks. In *IEEE International Conference on Universal Personal Communications (ICUPC'97)*, volume 2, pages 562–566, 1997.

[26] Zygmunt J. Haas and Marc R. Pearlman. The performance of query control schemes for the zone routing protocol. In *Proceedings of ACM SIGCOMM Conference 1998*, pages 167–177, August 1998.

[27] Andy Harter, Andy Hopper, Pete Steggles, Andy Ward, and Paul Webster. The anatomy of a context-aware application. In *Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking (MobiCom '99)*, pages 59–68, New York, NY, USA, 1999. ACM Press.

[28] Tingh-Chao Hou and Victor O.K. Li. Transmission range control in multihop packet radio networks. *IEEE Transactions on Communications*, 34(1):38–44, 1986.

[29] Qingfeng Huang, Chenyang Lu, and Gruia-Catalin Roman. Spatiotemporal multicast in sensor networks. In *Proceedings of the 1st international conference on Embedded networked sensor systems (SenSys'03)*, pages 205–217, New York, NY, USA, 2003. ACM Press.

[30] Qingfeng Huang, Chenyang Lu, and Gruia-Catalin Roman. Reliable mobicast via face-aware routing. In *Proceedings of IEEE Infocom '04*, March 2004.

[31] Tomasz Imielinski and Julio C. Navas. GPS-based addressing and routing, November 1996. RFC 2009.

[32] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *Proceedings of the 6th annual international conference on Mobile computing and networking (MobiCom '00)*, pages 56–67, New York, NY, USA, 2000. ACM Press.

[33] Chalermek Intanagonwiwat, Ramesh Govindan, Deborah Estrin, John Heidemann, and Fabio Silva. Directed diffusion for wireless sensor networking. *IEEE/ACM Transactions on Networking*, 11(1):2–16, 2003.

[34] Philippe Jacquet, Paul Muhlethaler, and Amir Qayyum. Optimized link state routing protocol, October 2003. RFC 3626.

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

[36] Joe M. Kahn, Randy H. Katz, and Kris S. J. Pister. Next century challenges: Mobile networking for "smart dust". In *Proceedings of the 5th ACM International on Mobile Computing and Networking (MobiCom '99)*, pages 271–278, August 1999.

[37] Brad Karp. *Geographic Routing for Wireless Networks*. PhD thesis, 2000.

[38] Brad Karp. Challenges in geographic routing: Sparse networks, obstacles, and traffic provisioning, May 2001.

[39] Brad Karp and H. T. Kung. GPSR: greedy perimeter stateless routing for wireless networks. In *Proceedings of the 6th ACM International on Mobile Computing and Networking (MobiCom '00)*, pages 243–254, Boston, MA, August 2000.

[40] Yongjin Kim, Jae-Joon Lee, and Ahmed Helmy. Modeling and analyzing the impact of location inconsistencies on geographic routing in wireless networks. *Mobile Computing and Communications Review*, 8(1):48–60, 2004.

[41] Young-Jin Kim, Ramesh Govindan, Brad Karp, and Scott Shenker. Practical and robust geographic routing in wireless networks. Technical Report 04-832, University of Southern California, 2004.

[42] Young-Jin Kim, Ramesh Govindan, Brad Karp, and Scott Shenker. Geographic routing made practical. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI 2005)*, May 2005.

[43] Young-Bae Ko and Nitin H Vaidya. Geocasting in mobile ad hoc networks: Location-based multicast algorithms. Technical Report TR-98-018, Texas A&M, September 1998.

[44] Young-Bae Ko and Nitin H. Vaidya. Location-aided routing (LAR) in mobile ad hoc networks. In *Proceedings of the 4th ACM International Conference on Mobile Computing and Networking (MobiCom '98)*, pages 66–75, October 1998.

[45] Young-Bae Ko and Nitin H. Vaidya. GeoTORA: A protocol for geocasting in mobile ad hoc networks. Technical report, Texas A&M, March 2000.

[46] Evangelos Kranakis, Harvinder Singh, and Jorge Urrutia. Compass routing on geometric networks. In *Proceedings of the 11th Canadian Conference on Computational Geometry*, pages 51–54, Vancouver, August 1999.

[47] Fabian Kuhn, Roger Wattenhofer, Yan Zhang, and Aaron Zollinger. Geometric ad-hoc routing: Of theory and practice. In *Proceedings of PODC 2003*, July 2003.

[48] Fabian Kuhn, Roger Wattenhofer, and Aaron Zollinger. Worst-Case Optimal and Average-Case Efficient Geometric Ad-Hoc Routing. In *Proceedings of 4th ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc 2003)*, June 2003.

[49] Fabian Kuhn and Aaron Zollinger. Ad-hoc networks beyond unit disk graphs. In *Proceedings of the 2003 joint workshop on Foundations of mobile computing (DIALM-POMC '03)*, pages 69–78, 2003.

[50] Ben Leong. Geographic routing network simulator, 2004. http://web.mit.edu/benleong/www/netsim.

[51] Ben Leong, Barbara Liskov, and Robert Morris. Geographic routing without planarization. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI 2006)*, May 2006.

[52] Ben Leong, Sayan Mitra, and Barbara Liskov. Path vector face routing: Geographic routing with local face information. In *Proceedings of the 13th IEEE International Conference on Network Protocols (ICNP 2005)*, pages 147–158, November 2005.

[53] Jinyang Li, John Jannotti, Douglas S. J. De Couto, David R. Karger, and Robert Morris. A scalable location service for geographic ad hoc routing. In *Proceedings of the 6th ACM International Conference on Mobile Computing and Networking (MobiCom '00)*, pages 120–130, 2000.

[54] Xiang-Yang Li, Gruia Calinescu, and Peng-Jun Wan. Distributed construction of planar spanner and routing for ad hoc wireless networks. In *Proceedings of IEEE Infocom '02*, June 2002.

[55] Xin Li, Young Jin Kim, Ramesh Govindan, and Wei Hong. Multi-dimensional range queries in sensor networks. In *Proceedings of the 1st international conference on Embedded networked sensor systems (SenSys'03)*, pages 63–75, 2003.

[56] W.H. Liao, Y.C. Tseng, K.L. Lo, and J.P. Sheu. Geogrid: A geocasting protocol for mobile ad hoc networks based on grid. *Journal of Internet Technology*, 1(2):23–32, 2000.

[57] Martin Mauve, Jörg Widmer, and Hannes Hartenstein. A survey on position-based routing in mobile ad hoc networks. *IEEE Network Magazines*, 15(6), November 2001.

[58] David Moore, John Leonard, Daniela Rus, and Seth Teller. Robust distributed network localization with noisy range measurements. In *Proceedings of the 2nd international conference on Embedded networked sensor systems (SenSys'04)*, pages 50–61, November 2004.

[59] Radhika Nagpal, Howard Shrobe, and Jonathan Bachrach. Organizing a global coordinate system from local information on an amorphous computer. In *Proceedings of the 2nd International Workshop on Information Processing in Sensor Networks (IPSN'03)*, April 2003.

[60] Julio C. Navas and Tomasz Imielinski. Geocast – geographic addressing and routing. In *Proceedings of the 3rd ACM International Conference on Mobile Computing and Networking (MobiCom '97)*, pages 66–76, New York, NY, USA, 1997. ACM Press.

[61] James Newsome and Dawn Song. GEM: Graph EMbedding for routing and data-centric storage in sensor networks without geographic information. In *Proceedings of the 1st international conference on Embedded networked sensor systems (SenSys'03)*, November 2003.

[62] T. S. Eugene Ng and Hui Zhang. Towards global network positioning. In *Proceedings of IEEE Infocom '02*, June 2002.

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

[64] Richard G. Ogier, Fred L. Templin, Bhargav Bellur, and Mark G. Lewis. Topology broadcast based on reverse-path forwarding. Internet Draft, November 2002.

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

[66] Vincent D. Park and M. Scott Corson. A highly adaptive distributed routing algorithm for mobile wireless networks. In *Proceedings of IEEE Infocom '97*, pages 1405–1413, 1997.

[67] Vincent D. Park and M. Scott Corson. Temporally-ordered routing algorithm (TORA). Internet Draft, October 1999.

[68] Charles Perkins. Ad-hoc on-demand distance vector routing. In *Proceedings of IEEE Military Communications Conference (MILCOM '97)*, November 1997.

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

[70] Nissanka B. Priyantha, Hari Balakrishnan, Erik Demaine, and Seth Teller. Anchor-free distributed localization in sensor networks. Technical Report LCS 892, MIT, March 2000.

[71] Nissanka B. Priyantha, Allen Miu, Hari Balakrishnan, and Seth Teller. The cricket compass for context-aware mobile applications. In *Proceedings of the 7th ACM International Conference on Mobile Computing and Networking (MobiCom '01)*, July 2001.

[72] S Radhakrishnan, Gopal Racherla, Chandra N. Sekharan, N. S.V Rao, and S.G. Batsell. DST – a routing protocol for ad hoc networks using distributed spanning trees. In *IEEE Wireless Communications and Networking Conference*, 1999.

[73] Ananth Rao, Christos H. Papadimitriou, Scott Shenker, and Ion Stoica. Geographic routing without location information. In *Proceedings of the 9th ACM International Conference on Mobile Computing and Networking (MobiCom '03)*, pages 96–108, San Diego, CA, September 2003.

[74] S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S. Shenker. GHT: A geographic hash table for data-centric storage in sensornets. In *Proceedings of the First ACM International Workshop on Wireless Sensor Networks and Applications (WSNA)*, September 2002.

[75] Sylvia Ratnasamy, Brad Karp, Scott Shenker, Deborah Estrin, Ramesh Govindan, Li Yin, and Fang Yu. Data-centric storage in sensornets with ght, a geographic hash table. *Mobile Networks and Applications (MONET), Journal of Special Issues on Mobility of Systems, Users, Data, and Computing*, 2003.

[76] Chris Savarese, Jan M. Rabaey, and Jan Beutel. Locationing in distributed ad-hoc wireless sensor networks. In *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2001)*, pages 2037–2040, May 2001.

[77] Chris Savarese, Jan Rabay, and Koen Langendoen. Robust positioning algorithms for distributed ad-hoc wireless sensor networks. In *Proceedings of the USENIX Technical Annual Conference*, June 2002.

[78] Andreas Savvides, Chih-Chieh Han, and Mani B. Strivastava. Dynamic fine-grained localization in ad-hoc networks of sensors. In *Proceedings of the 7th ACM International Conference on Mobile Computing and Networking (MobiCom '01)*, pages 166–179, 2001.

[79] Karim Seada, Ahmed Helmy, and Ramesh Govindan. On the effect of localization errors on geographic face routing in sensor networks. In *Proceedings of the 3rd International Symposium on Information Processing in Sensor Networks (IPSN'04)*, pages 71–80, 2004.

[80] Yuval Shavitt and Tomer Tankel. Big-bang simulation for embedding network distances in Euclidean space. In *Proceedings of the IEEE Infocomm*, April 2003.

[81] Timothy J. Shepard. A channel access scheme for large dense packet radio networks. In *Proceedings of the ACM SIGCOMM '96 Conference*. ACM SIGCOMM, August 1996.

[82] Hideaki Takagi and Leonard Kleinrock. Optimal transmission ranges for randomly distributed packet radio terminals. *IEEE Transactions on Communications*, 32(3):246–257, 1984.

[83] Crossbow Technlogies. Mica2 series wireless measurement system. http://www.xbow.com.

[84] Godfried T. Toussaint. The relative neighbourhood graph of a finite planar set. *Pattern Recognition*, 12:261–268, 1980.

[85] Roger Wattenhofer and Aaron Zollinger. XTC: A practical topology control algorithm for ad-hoc networks. Technical Report 407, ETH Zurich, 2003.

[86] Roger Wattenhofer and Aaron Zollinger. XTC: A practical topology control algorithm for ad-hoc networks. In *Proceedings of the of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, 2004.

[87] Guoliang Xing, Chenyang Lu, Robert Pless, and Qingfeng Huang. On greedy geographic routing algorithms in sensing-covered networks. In *Proceedings of the 5th ACM International Symposium on Mobile Ad-Hoc Networking and Computing (MobiHoc 2004)*, pages 31–42, 2004.

[88] Jerry Zhao and Ramesh Govindan. Understanding packet delivery performance in dense wireless sensor networks. In *Proceedings of the 1st international conference on Embedded networked sensor systems (SenSys'03)*, pages 1–13, 2003.

[89] Yao Zhao, Bo Li, Qian Zhang, Yan Chen, and Wenwu Zhu. Hop ID based routing in mobile ad hoc networks. In *Proceedings of the 13th IEEE International Conference on Network Protocols (ICNP 2005)*, pages 179–190, November 2005.