ROUTING PROTOCOLS FOR AD HOC NETWORKS WITH UNCERTAINTY IN THE POSITION OF THE DESTINATION

ANUP PATNAIK

A THESIS
IN
THE DEPARTMENT
OF
COMPUTER SCIENCE AND SOFTWARE ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

FEBRUARY 2006

Your file Votre référence
ISBN: 0-494-14333-9
Our file Notre référence
ISBN: 0-494-14333-9

Abstract

Routing Protocols for Ad Hoc Networks with Uncertainty in the Position of the Destination

Anup Patnaik

An ad hoc network comprises a set of computing devices, often mobile, that communicate using wireless transmissions with the devices within their transmission range. These devices or nodes are able to detect the presence of other nodes in their transmission range as well route packets on behalf of other nodes. Hence, the network does not need to depend on any predefined infrastructure. This lack of infrastructure makes the problem of routing and packet forwarding a challenging task in such networks. To carry out this task in an efficient and scalable manner, several authors have proposed position-based routing algorithms. Position-based routing algorithms utilize the position or location of the destination node to inform routing decisions. However, obtaining the accurate position of the destination may not be feasible in some settings.

In this thesis, we consider the problem of routing in an ad hoc network where the source node knows the approximate position of the destination node, but is uncertain about its exact current location. We investigate two approaches to this problem: one, based on a traversal of the faces of a planar sub-graph of the graph representing the network, and the second, based on flooding a limited area of the graph that represents the region the destination is likely to be found. We propose several variants of both approaches, and do extensive simulations to analyze the performance of the algorithms. Our results indicate that a simple modification of the basic flooding approach yields the best trade-off for optimizing delivery rate, stretch factor, as well as transmission cost. If however, delivery is required to be guaranteed, then a variant of the face tree traversal approach that we propose has the best performance.

Acknowledgments

This thesis has been much more than an academic experience. Many wonderful people have contributed towards this experience in very many ways. I would begin by expressing my gratitude towards my mother, Deepti, and father, Rabindra. It is their care and unconditional love that has made me the person that I am. I dedicate this work to them. Also, I would like to mention my little sister, Anuradha, whose non stop chattering was always a welcome break.

I have been very fortunate to have been working under Dr. Lata Narayanan. She has guided, motivated and supported me in all respects all throughout my stay at Concordia. Her energy is contagious. And, her dedication to her work and her students has and will always inspire me. A big thanks to her for everything.

Dr. V. S. Alagar taught me many important lessons. I have enjoyed the many technical and non-technical discussions with him, which have helped me develop a systematic way of thinking. He has been a true mentor to me. My sincere thanks to him.

I thank my friends: Ankur, for filling up the role of a mother, GB, for his idiosyncrasies, Israat, for making me think on her problems, Kruthi, for all that nagging, Pari, for those beautiful songs that are hard to let go, Ram, for being eternally confused, Sabeel, for being eternally bored, Sandesh, for those morning strawberry milk shakes and, Tejas, for being T. It is for their presence that my stay in Montreal was memorable.

I would like to thank my co-authors Evangelos Kranakis, Danny Krizanc, and Sunil Shende with whom we have submitted a paper titled "Routing with uncertainty in the position of the destination".

I would also like to thank Halina for taking care of all my administrative needs and Pauline for giving me interesting tutoring assignments.

Finally, I would like to thank the Art of Living Foundation and all my friends there.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Internet was born thirty six years ago with the aim of building a nationwide network of computers [Tug69]. A key element in the vision for the Internet, as articulated by its founders, was that anyone will be able to plug in from any location with any device at any time [Kle03]. Truly, in recent times, this anytime anywhere information access has become the need of our society and industry.

This need for computing and communications on the move has seen the emergence of Wireless Local Area Networks (WLANs), computer networks where radio waves are used as the physical medium for transmitting data. Owing to the standardization of IEEE 802.11 protocols for the physical and MAC layers, deployment of WLANs has seen tremendous growth. Such networks consist of special devices called *access points*, which are responsible for serving wireless hosts over a range of 100-200 m. The access points act as an interface between the wired and the wireless networks. The wireless hosts connect to the access points to communicate with other hosts on the wired/wireless network using wireless network interface cards (NICs). The mobility of a host is not restricted to the range of a single access point. Hosts are free to move from the coverage of one access point to the other by means of a process called a handoff. IEEE 802.11 based WLANs as described above are known as *infrastructure networks*. An illustration of an infrastructure WLAN is given in Figure 1.1. Apart from this, wireless hosts can also directly communicate with other wireless hosts within their transmission range without the need of any access points. Such a mode of operation of the hosts is called *ad hoc mode*. This thesis concerns itself with such *infrastructure-less networks* or *ad hoc networks*.

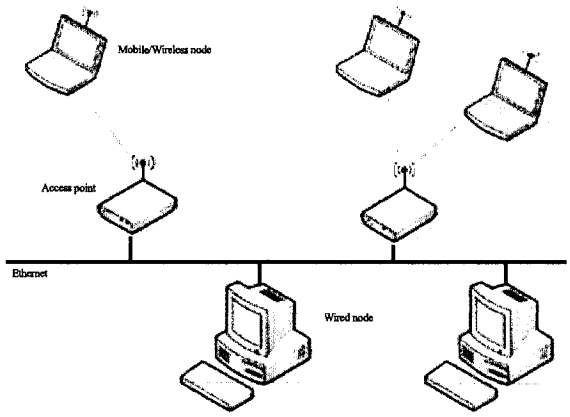


Figure 1.1: An infrastructure WLAN.

1.1 Ad Hoc Networks

An ad hoc network comprises a set of computing devices, often mobile, that communicate using wireless transmissions, directly with the devices within their transmission range. Since the transmission range is limited, typically 50-200 feet, not all devices are in each other's range. To communicate with devices outside their transmission range, a device relies on intermediate devices to *relay* or *forward* packets. These devices or nodes are able to detect the presence of other nodes in their transmission range as well route packets on behalf of other nodes. Therefore, every node in the network acts as both an end machine as well as a router. The network has a mesh topology (see Figure 1.2) as opposed to the star topology of an infrastructure network (refer to Figure 1.1). Unlike the infrastructure network, in an ad hoc network the data is transmitted over multiple wireless links, hence such networks are also called *multi-hop wireless networks.*

Such networks are very different from the traditional wireless/wired networks. Since the nodes are free to move about randomly, the network's topology may change rapidly and unpredictably. An example of this can be seen in Figure 1.3. Here, the topology of the network drastically changes as node D moves out of node A, B and C's transmission range into node E's transmission range. As more and more devices participate in the network, the network becomes more robust with multiple paths between any two set of nodes. The network is self-organizing, meaning the network is formed and destroyed on-the-fly without the need of any administration and whenever needed, hence the name ad hoc.

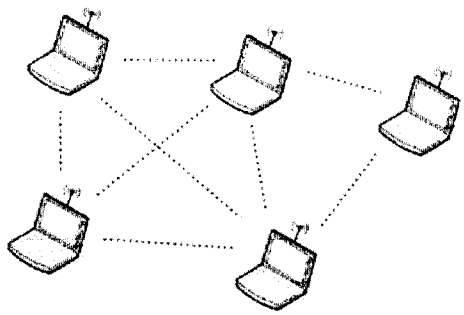


Figure 1.2: Mesh topology of an ad hoc network.

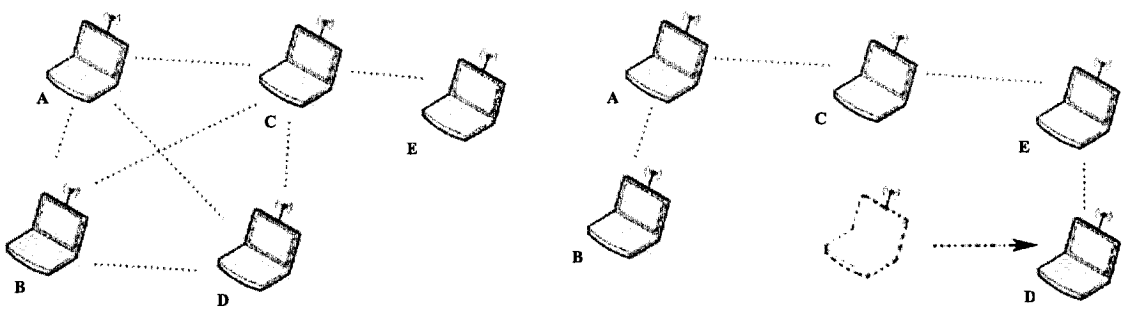


Figure 1.3: An ad hoc network with changing topology.

1.1.1 A Brief History

Multi-hop relaying, the principle behind ad hoc networking dates back to 500 B.C. when Darius I (522-486 B.C.), the king of Persia, made use of a line of shouting men positioned on tall structures to send messages from his capital to the remote provinces of his empire [MM04]. In 1972, this principle was applied to broadcast radio networks in the Packet Radio Network (PRNET) project sponsored by the Defense Advanced Research Projects Agency (DARPA) [DAR]. The PRNET consisted of mobile nodes having radio interfaces that used a combination of ALOHA [Tan02] and carrier sense multiple access (CSMA) [Tan02] to access the shared radio channel. The network consisted of mobile radio repeaters responsible for relaying packets from one repeater to another, until the packets eventually reach the destination node. The classical Bellman-Ford algorithm was used as the routing algorithm. Early prototypes of the system were very successful proving the feasibility and efficiency of infrastructure-less networks. In the 1980s, the success of PRNET led DARPA to extend the work on such networks through the *survivable radio networks* (SURAN) project [FL01]. This project mainly focused on improving the radio devices by making them small, low-cost and power efficient. In this project they were also able to improve on the network's scalability and survivability (i.e., resilience to node and link failure). In the mid 1990s, DARPA initiated the Global Mobile (GloMo) Information Systems project to take advantage of the rapidly developing Internet infrastructure and technologies [DAR]. The main goal was to provide mobile users anytime anywhere connectivity. Many novel ideas of networking within an ad hoc network as well as between an ad hoc network and the Internet were tried out in this project. Around the same time, with the availability of laptop computers, the concept of ad hoc networking was taken to the commercial world. Realizing the need for open standards in the protocols involved in ad hoc networks, a working group within the Internet Engineering Task Force (IETF), called *the mobile ad hoc networks (MANET) working group* was formed [IET]. The goal of the MANET group is to "standardize IP routing protocol functionality suitable for wireless routing applications within both static and dynamic topologies" [IET]. Soon after, the IEEE 802.11 subcommittee standardized a medium access protocol based on collision avoidance and resilient to the hidden terminal problem. The standard also specified the physical layer. This led to wide scale manufacturing of cheap radio/wireless cards that could be plugged into laptops and other mobile devices. As a result now ad hoc networks could be built out of cheap off-the-shelf equipment. Currently, companies such as Packethop, Tropos and Motorola

are providing ad hoc networking solutions for areas ranging from public safety to wireless broadband.

1.1.2 Applications of Ad Hoc Networks

Being infrastructure-less, these networks can be deployed rapidly anytime anywhere in a cost-effective manner. They can also provide reliable connectivity in spite of mobility of nodes. Owing to these, ad hoc networks find applications in several areas. Some of these include: military communications, disaster management, emergency operations, healthcare, vehicular networks, collaborative computing, wireless mesh networks and wireless sensor networks. We discuss some of these below.

Military communications was one of the foremost areas of application for ad hoc networks. In battleground scenarios of hostile enemy territories and difficult terrains such as mountains and desserts, availability of a communication infrastructure cannot be assumed. In such environments, ad hoc networks help in instantly setting up communications between groups of soldiers and mobile military vehicles such as vans, tanks and planes.

Ad hoc networks also meet temporary communication needs of **collaborative computing** by enabling communication without the need of any prior setup. Such scenarios arise in conferences where a group of people coming together for a purpose may need to coordinate, communicate and share resources.

Ad hoc networks are very useful in **emergency, search and rescue, and disaster management** scenarios. In cases of earthquake and hurricanes, conventional infrastructure could be broken. Here, ad hoc networks immediately provide communication for coordinating rescue and restoration activities. Also, in emergency situations, ad hoc networks provide more localized and dedicated communication infrastructure as compared to the conventional ones [MM04].

Wireless Mesh Networks are ad hoc networks with dedicated nodes forming a mesh topology to provide communication infrastructure for fixed/mobile users. Unlike other ad hoc networks as discussed above, in such networks the end user terminals do not carry out any routing. Maintaining of the topology and routing are carried out exclusively by the dedicated nodes. In the context of 802.11 based networks, each of the access points, acts as a node in the ad hoc network. The user terminals connect to the access points which in turn connect to other access points over the wireless medium. A mesh of hundreds of such access points mounted on top of buildings and street lights can be used to provide

broadband wireless Internet access throughout a city. A survey of wireless mesh networks is given in [AWW05].

Sensor networks are another application of ad hoc networking. Sensors are tiny devices that are capable of sensing environmental parameters and communicating them over the wireless links. Sensor networks are ad hoc networks comprising a large collection of sensor nodes. The use of ad hoc networking enables such networks to be deployed in a cost effective manner and gives the flexibility of effortless addition, removal and movement of sensors. Some of the application scenarios for such networks are: detection of nuclear radiation, border intrusion and temperature sensing [ASSC02].

Recently, use of ad hoc networking is also being explored in the arena of cellular networks [WQDT01]. The main idea here is to enable each phone with multi-hop relaying so that the base station as well as other cell phones can be reached through other intermediate phones. Such a **hybrid (cellular/ad hoc) wireless network** architecture results in an increase in the number of users supported per cell.

1.1.3 Challenges in Designing Ad Hoc Networks

Due to their unique characteristics, ad hoc networks face many challenges. We discuss these below.

- **Wireless medium:** Because of the use of the wireless medium, transmissions in an ad hoc network are prone to noise and interference from outside signals. Hidden-terminal and exposed-terminal problems could also occur. Also, wireless links have severe bandwidth constraints as compared to the wired ones. Therefore, protocols need to be designed in a manner so as to shield the upper layers from such details.

- **Infrastructure-less:** In the absence of any centralized servers, all the network management, routing and security functionalities have to be distributed across different nodes. This implies that all the protocols must be designed in a perfectly distributed manner.

- **Multi-hop routing:** Since there are no dedicated routers in the network, every node in the network has to gather the topology information, build its routing tables and forward packets. This makes designing routing protocols very hard.

- **Dynamically changing topologies:** Because of arbitrary movement of nodes, the topology of the network can change frequently and unpredictably. To cope with this the routing protocols have to be designed in a manner so that they always maintain accurate topology information.

- **Asymmetric node and link capabilities:** Nodes in the network with radio interfaces may have different transmission and receiving capabilities. This can result in asymmetric links. Also, different nodes may have different processing and power capabilities. Designing protocols for such heterogeneous networks can be very complex.

- **Energy constrained operation:** Nodes, being mobile, are constrained by limited battery power. This is an important issue in ad hoc networks since each node expends energy in forwarding packets for other nodes. Hence, protocols have to be designed for optimal energy usage.

1.2 Routing in Ad Hoc Networks

In computer networks, *routing* is defined as carrying out the following two functions:

- Discovering paths (also called routes) between network terminals/nodes along which data packets can be sent.

- Forwarding, the passing of addressed packets from their source node to their destination node.

The nodes in the network that carry out this functionality are called *routers* and the distributed algorithm running at the network layer that performs the routing functionalities is called the *routing protocol.*

The forwarding function can be achieved in three main ways. One, by building and maintaining *routing tables* at each router, which maintain a record of next-hop node along the path for every destination node. Two, by storing the full path from source to destination in the header of the packet itself. Such a technique is called *source routing*. Three, by using the node location information. This does not require any routing state to be maintained at the nodes. We discuss algorithms based on this approach in detail in Chapter 2.

In protocols based on routing tables, the discovery of paths involves building and maintaining the routing tables. There are two approaches to achieve this: *Distance Vector* (DV), in which each node sends its neighbors its entire routing table, and *Link State* (LS), in which each node sends to all the other nodes the state of its link with its current neighbors via flooding. To guarantee that routing tables are up to date and reflect the current network topology, nodes continuously exchange route updates.

In the context of ad hoc networks, the real challenge in designing routing protocols is in maintaining accurate routing tables in the face of changing topology, and doing so in an efficient manner so as to meet the bandwidth and energy constraints. For this, many different strategies are deployed. We classify and discuss them below.

1.2.1 Proactive protocols

The table-driven DV and LS based protocols that calculate all possible paths irrespective of their use are called proactive protocols. The advantage here is that a path to the destination is always available without any delay. Two of the prominent proactive routing protocols are Destination Sequenced Distance Vector (DSDV) [PB94] and Optimized Link State Routing (OLSR) [CeA+03] protocols. We discuss these below.

Destination Sequenced Distance Vector

As it is a routing table based protocol, each node maintains a table that contains the shortest distance and the first node on the shortest path to every other node in the network. Additionally each entry also contains a sequence number. The tables are exchanged between neighbors periodically, or when a topology change is detected. These updates occur either by exchanging complete routing tables or parts of it incrementally. For each update for a destination, an entry in the routing table is updated only if the update has a more recent sequence number or if it has the same sequence number but a better path length. This mechanism prevents routing loops and the count-to-infinity problem.

Optimized Link State Routing

OLSR is a link state routing protocol with optimized flooding of the network with routing updates (called link state packets). Each node selects, independently from other nodes, a minimal set of nodes, called *Multipoint relays*, from its one-hop neighbors. Only these nodes are responsible for link state updates and for packet forwarding. Link state updates are generated periodically.

1.2.2 Reactive protocols

Another strategy in designing a protocol is to calculate a path to a destination only when there is a need. In the absence of any data transmission, such protocols do not generate any routing traffic. This class of protocols are called reactive or on-demand protocols. Dynamic Source Routing (DSR) [JM96] and Ad hoc On Demand Distance Vector routing (AODV) [PR99] are two main reactive routing protocols. We discuss them below.

Dynamic Source Routing

DSR involves two main mechanisms: Route Discovery and Route Maintenance. Route Discovery is triggered by the source node when it wants to find a path to the destination node. Route Request (RREQ) packets containing the sender address, destination address, a unique number to identify the request and route record are flooded in the network. An intermediate node receiving the RREQ relays it only if it has not relayed it before (this check is made by looking at the unique number). If the intermediate node has a route to the destination in its cache then it replies to the RREQ with the complete route, else it appends its address to the route record and broadcasts the packet to its neighbors. On receiving a RREQ packet, the destination replies to it with a Route Reply (RREP) packet containing the complete route record from the RREQ. This RREP is sent along the path obtained by reversing the route record in the RREQ. In some cases to restrict the flooding of the whole network, the Time To Live (TTL) parameter in the RREQ is set to a value less than the network diameter. Route Maintenance, as opposed to Route Discovery, involves invalidating stale entries in the cache. When an intermediate node detects a link break to its next-hop node towards the destination, it removes this link from its route cache and sends a Route Error message to the source. On receiving the Route Error (RERR) message the source removes the entry in the cache for the destination, and triggers a new route discovery.

Ad hoc On Demand Distance Vector Routing

Like DSR, AODV also makes use of RREQ and RREP packets to discover routes, but the main difference here is that the newly found route is stored at the nodes and not in the packet. Every node receiving the RREQ sets up the reverse path from itself to the source by adding the neighbor from which it received the RREQ as the next-hop towards the source node. Hence, by the time the RREQ packet reaches the destination node, the reverse path from the destination to the source node is set on all intermediate nodes. On receiving the RREQ, the destination node sends the RREP along this path. Similar to RREQ, as

the RREP is propagated towards the source node, the reverse path from the source to the destination is set on all intermediate nodes. To maintain freshness of routes, route entries are deleted if they are not used within a given route expiration time interval. Similar to DSDV, AODV also maintains sequence numbers corresponding to each destination. A path information towards a destination is updated only if the destination sequence number in the packet received is greater than the last destination sequence number stored at the node. Route maintenance involves periodic transmission of HELLO messages. Upon detecting a link break, an unsolicited RREP message is sent to all precursor nodes involving the broken link. A precursor node is a node that is one before the current node on the path to the destination through the next-hop. These nodes in turn relay packets to their precursor nodes so that all sources involving the broken link on their path to the destination are notified. On receiving the unsolicited route reply, the source nodes start a new route discovery.

1.2.3 Position-based protocols

The protocols in the previous two categories heavily rely on flooding and frequent updates of routing tables. These are very resource-intensive operations, especially in the context of ad hoc networks. Therefore, to reduce the amount of control traffic, several authors have proposed the use of node location information. We refer to the problem of routing from a source node s to a destination node d at position p by ROUTE(s,d,p). Protocols that solve this problem are often called *position-based* routing protocols. In such protocols, every node is assumed to know the locations of itself and its neighbors. In addition, the source node is assumed to know the location of the destination. These protocols are stateless, meaning there are no routing tables or any other routing states maintained at the nodes. An intermediate node picks the next-hop node on the path towards the destination based on the location information of the destination and its neighbors. One such protocol is the compass routing proposed by Kranakis, Singh and Urrutia [KSU99]. In compass routing, an intermediate node I forwards the packet to one of its neighbor N such that the direction IN is closest to the direction ID, where D is the destination node. Compass routing does not guarantee the delivery of packets. Our study in this thesis is centered on position-based protocols. We discuss them in detail in Chapter 2.

1.3 Routing with Uncertainty in Destination's Position

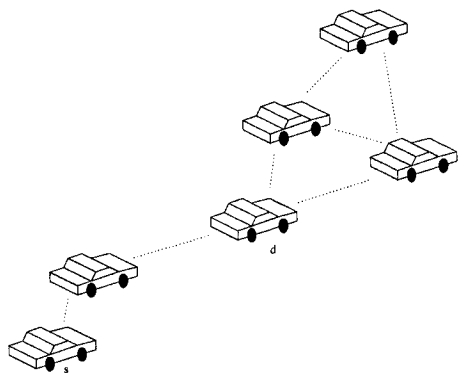
All the position-based routing protocols above assume the use of exact location information about the destination node. This information can be obtained using a location service [CBW02], or from messages previously received from the destination. However, in both these cases, there can be inaccuracies in the position information. In the presence of such inaccuracies, the routing algorithm may have a reduced rate of delivery. Also, in some settings maintaining a location service may be too resource consuming or may not be feasible at all.

In this thesis, we assume that the source node s knows the destination node d's position (x_0, y_0) at time t_0, but is interested in sending a packet to d at time t_1 where $t_1 > t_0$. If the maximum velocity of node d is v units per second, then the position of d at time t_1 is a point inside the circle with center (x_0, y_0) and radius $v(t_1 - t_0)$ units. This leads to the problem of routing with uncertainty in the position of the destination. Instead of routing to a specific destination at a known position (x, y), we are interested in routing to a specific destination whose position is somewhere inside a circle with known center and radius. We refer to the problem of routing from a source node s to a destination node d contained in the circle of radius r centered at position p as ROUTE-U(s, d, p, r) and r as the *uncertainty radius*.

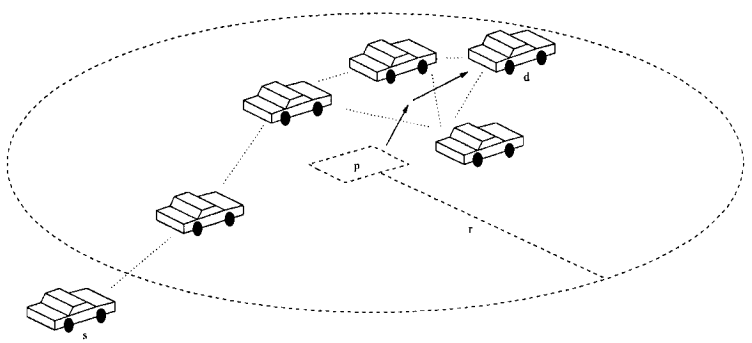
The problem of routing with uncertainty in destination's position may occur especially in vehicular ad hoc networks where the goal is to enable communication between vehicles without the need for a network infrastructure. In such networks, because of rapid movement of the vehicles, a source vehicle s wanting to send data to the destination vehicle d may not always have the accurate position of d. This is illustrated in Figure 1.4. Here, s has the exact location of d at time t_0, but when it wants to send the data to d at time t_1, vehicle d has moved to a new location (Figure 1.4 (b)). This movement is shown by the solid arrows. Note that other vehicles may also move, however, this need not affect the routing algorithm. Note that although s does not know the exact location of d at t_1, it can be safely assumed to be within the dotted circle centered at $d's$ old location and with r as its radius.

1.3.1 Related Work

To the best of our knowledge, this is the first work to deal with the problem of routing with uncertainty in the destination's position. However, the problem is similar to the problem of *geocasting*. In geocasting, the objective is to route packets from a single source node to *all*



(a) A vehicular network at time t_0



(b) The vehicular network at time t_1

Figure 1.4: A vehicular ad hoc network.

nodes in a given geographical region. From this, it follows that the problem of ROUTE-U is subsumed by the problem of geocasting. Therefore, a solution to the geocasting problem would necessarily solve the ROUTE-U problem. However, such a solution may not be efficient as compared to one that solves only the ROUTE-U problem. We discuss some of the geocasting algorithms proposed in the literature below.

In [KV99], Ko and Vaidya propose the Location Based Multicast (LBM) algorithm based on flooding. They restrict the flooding by defining forwarding zones, which include at least the destination region and a path between the sender and the destination region. An intermediate node forwards a geocast packet only if it belongs to the forwarding zone. They give two schemes to determine the forwarding zone. The first scheme defines the forwarding zone as the smallest rectangular shape that includes the sender and the destination region. The second scheme defines the forwarding zone by the coordinates of the sender, the destination region and the distance of a node to the center of the destination region. LBM does not guarantee delivery of packets. The authors of [SH04] give an algorithm that consists of reaching a node q contained in the circle with radius r centered at p, and then flooding all nodes contained in the circle. This approach also does not guarantee delivery. We call algorithms based on this approach *flooding-based* algorithms and propose several variants of this approach in Chapter 3. Recently, Stojmenovic [Sto04] gives two algorithms for geocast that guarantee delivery. The first algorithm is essentially flooding inside the circle, augmented by face traversals initiated by *inside border* nodes; this approach was also outlined in [SH04]. In the second algorithm, the packet is sent simultaneously to a grid of points just outside the circle, which then initiate a flooding inside the circle. In [dBvOO96], an algorithm that performs a traversal of a planar graph in $O(n^2)$ steps is given. The algorithm is based on constructing and traversing a tree of the faces of the planar graph. This was improved in [BM02], to an algorithm that can be used for geocasting with guaranteed delivery in a MANET in $O(n \log n)$ steps [BMSU99]. We discuss these *face tree traversal-based* algorithms in detail in Chapter 2.

1.4 Contribution of Thesis

In this work, we investigate two classes of protocols to solve the problem of routing with uncertainty in the position of the destination. The first class consists of variants of a

flooding-based approach, while the second class consists of variants of the face tree traversal algorithm proposed by Bose and Morin [BM02, Mor01]. We define and analyze all the protocols mentioned above. We also implement and simulate each of the protocols to compare performance. We also study the effect of the number of nodes and the size of the uncertainty radius on *delivery rate*, *stretch factor*, and the *transmission cost*. The delivery rate is the percentage of packets that get transmitted successfully to the destination. The stretch factor is the number of hops taken by a packet compared to the minimum hop path available in the network, averaged over all successfully delivered packets. The transmission cost is the ratio of total number of times that copies of the packet get transmitted in the course of successful delivery of the packet to the number of transmissions in the minimum hop path, averaged over all successfully delivered packets. It is a measure of the energy costs of the algorithm.

The surprising findings of our experiments are listed below:

- Flooding-based algorithms show an interesting behavior whereby the delivery rate first decreases and then increases as the uncertainty radius increases.
- A simple augmentation of flooding, that we call EXTENDED SN FLOODING , achieves very high delivery rate, at the same time as achieving very low stretch factor, and a drastically reduced transmission cost.
- While the accepted wisdom is that flooding is very resource-inefficient, and would have a high transmission cost, our results show that some variations of the face tree traversal approach, including the version given in [BMSU99], have a very high transmission cost as well. Indeed, there is considerable overlap between the transmission cost profiles of the two approaches. In particular, the cheapest algorithms among the ones studied are EXTENDED SN FLOODING and SN FLOODING, while the two most expensive algorithms are face tree traversal-based algorithms.
- The difference between geocasting and ROUTE-U is highlighted by the fact that a technique that provably improves the performance for geocasting appears to *degrade* the performance for ROUTE-U.
- The original face tree traversal-based approach is memoryless as compared to the flooding-based approaches, which require a constant amount of routing state at nodes. However, an obvious modification of FACE TREE that uses extra memory does not yield much benefit. In particular, its performance is still worse than the best flooding approach in our experiments.

- Finally, if marked bits are not practical, or if guaranteed delivery is required, then DFS FACE TREE would seem to be best approach, but otherwise, EXTENDED SN FLOODING would be best.

1.5 Organization of Thesis

In Chapter 2, first we give a discussion on graphs, unit disk graphs, planarization of graphs and the network model used in this thesis. Next, we describe three position-based routing protocols and give an outline of our solution to the problem of routing with uncertainty in destination's position. Finally, we present a class of algorithms based on face tree traversal.

In Chapter 3, we present a class of algorithms based on simple flooding. We illustrate cases where such algorithms fail to guarantee delivery and also suggest techniques to improve delivery rate.

In Chapter 4, we give details of the simulation environment used in our experiments. We compare the performance of all the algorithms categorizing them into face tree traversal-based and flooding-based. Finally, we give a discussion comparing the two approaches.

In Chapter 5, we present a summary of our findings and suggest future research directions.

Chapter 2

Face Tree Traversal Based Protocols

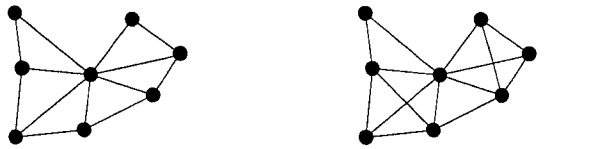
In this chapter we detail a class of algorithms that solve the problem of routing with uncertainty in the destination's position. We begin with a discussion on graphs, unit disk graphs and their planarization in Sections 2.1, 2.2 and 2.3 respectively. We give the network model used in this thesis in Section 2.4. In Section 2.5, we give background on position-based routing protocols and discuss the GFG ROUTING algorithm. Finally, in Section 2.7, we present the face tree traversal-based algorithms.

2.1 Graphs

A graph can be defined as follows [Car79]:

> A graph $G = (V, E)$ consists of
>
> 1. a finite set $V = \{v_1, v_2, \ldots, v_n\}$ whose elements are called *vertices*;
> 2. a subset E of the Cartesian product $V \times V$, the elements of which are called *edges*.

A *geometric graph* is a graph with points on the plane as vertices and straight line segments joining them as edges. A geometric graph is said to be *planar* if no two of its edges intersect other than at a common endpoint. A geometric graph with intersecting edges is called *nonplanar*. Examples of planar and nonplanar geometric graphs are given in Figure 2.1.



(a) Planar Graph **(b) Nonplanar Graph**

Figure 2.1: Planar and nonplanar geometric graphs.

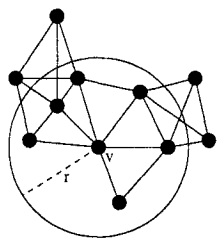


Figure 2.2: A unit disk graph, where r denotes the transmission radius of node v. The circular disk denotes the transmission area of node v.

2.2 Unit Disk Graph

A *unit disk graph* (UDG) is a specific type of geometric graph used to model broadcast networks. In a UDG, an edge exists between two vertices u and v if and only if the euclidean distance between u and v is at most 1. In the context of ad hoc networks, the vertices in the UDG represent network nodes. An edge exists between two nodes if the euclidean distance between the two nodes is less than or equal to a node's transmission range r. These neighboring nodes can directly communicate with each other over the wireless media. Here, it is assumed that all nodes have transceivers of equal power and hence the same transmission range r. Figure 2.2 shows an example of a UDG.

2.3 Planarization of a Graph

Planarization is the process of transforming a UDG into a planar graph by removing certain edges. Some of the position-based routing protocols, such as FACE ROUTING, require the underlying graph of network nodes to be planar. This is accomplished by the use of a

GABRIEL GRAPH **Algorithm**

1. **for** each $u \in N(v)$ **do**
2. **if** $disk(u,v) \cap (N(v) \backslash \{u,v\}) \neq \emptyset$ **then**
3. delete(u,v)
4. **end if**
5. **end for**

Figure 2.3: The GABRIEL GRAPH algorithm [BMSU99]. Here, $disk(u,v)$ is the disk with diameter (u,v) and $N(v)$ is the set of neighbors of node v in the UDG.

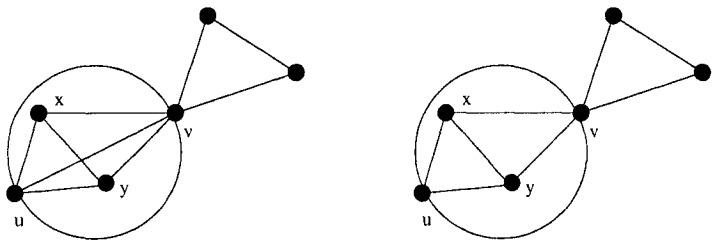
planarization algorithm. The nature of ad hoc networks requires that the planarization algorithms be completely distributed. Here, we discuss the *Gabriel Graph* (GG) planarization algorithm. In all of our algorithms we use the GG planarization.

Gabriel Graph Algorithm

The GABRIEL GRAPH algorithm is completely distributed and computes the planar subgraph of a UDG using only immediate neighbor information. Each node executes the algorithm to compute its neighbors in the planar subgraph. This neighbor set is used by the routing protocols that require planarization of the UDG. Given a unit disk graph $G = (V,E)$, the Gabriel graph of G is the subgraph $G' = (V,E')$ where $(u,v) \in E'$ if and only if $(u,v) \in E$ and $disk(u,v)$ contains no nodes in V other than u and v [GS69]. Here, $disk(u,v)$ is the disk with diameter (u,v) containing both u and v. Figure 2.3 gives the pseudocode for the GABRIEL GRAPH algorithm. Given that the UDG is connected, it has been proved that the corresponding Gabriel subgraph is connected and planar (*Lemma 1* from [BMSU99]). An example of Gabriel graph construction is given in Figure 2.4. Here, edge (u,v) is eliminated in the Gabriel graph since nodes x and y lie within the $disk(u,v)$, while all other edges are retained. The complexity of the GABRIEL GRAPH algorithm is $O(d \log d)$ where d is the maximum degree of a vertex in the UDG.

2.4 Network Model

In this thesis we model an ad hoc network as a UDG. Mobile devices/nodes, represented by vertices of the UDG, are randomly and uniformly distributed on a euclidean plane. Each node has a omnidirectional antenna to transmit and receive messages. All nodes transmit



(a) G **(b)** GABRIEL GRAPH (G)

Figure 2.4: Gabriel graph computation.

with the same power and hence have the same transmission range r, which is the unit distance for the corresponding UDG. An edge represents the communication link between a pair of nodes that are within the transmission radius of each other. All communication links/edges are *bidirectional*, i.e., if a node u can receive messages from node v, then v can also receive messages from u. Also, we assume the absence of any obstacles since presence of obstacles may not result in a UDG.

2.5 Position-based Routing

Position-based routing utilizes node location information for improving the efficiency of routing by reducing the overhead of control traffic. This location information can be obtained from the Global Positioning System (GPS) that is assumed to be present in each node. Every node knows the location of itself and its neighbors. The neighbors' locations can be obtained and maintained using a neighbor discovery beaconing protocol. The source node is assumed to know the destination location and includes this information in the packet for every intermediate node to see. Each intermediate node chooses the next hop for the packet in a localized manner based solely on the location of itself, its neighboring nodes, and the destination. As a result of this localization, position-based routing protocols are highly scalable.

In the rest of the section, we discuss two of the position-based routing protocols used in our work, GREEDY ROUTING and FACE ROUTING. Geographic distance routing (GEDIR)

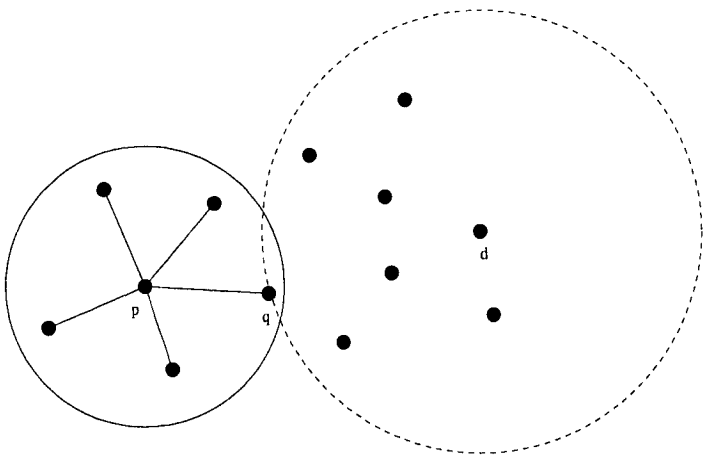


Figure 2.5: An example showing GREEDY ROUTING. The dotted circle centered at d is to show that q is closest to d amongst all neighbors of p

[SL01], compass routing [KSU99], terminode routing [BBC+01] and location-aided routing (LAR) [KV98] are some other position-based routing protocols. A survey of position-based routing protocols can be found in [GS04].

2.5.1 Greedy Routing

In [Fin87], Finn proposed a greedy heuristic for routing using location information. In GREEDY ROUTING a node p selects a neighboring node q (Figure 2.5) such that its euclidean distance to the destination d is least among all its neighbors. This node is the packet's next hop towards the destination. Note that GREEDY ROUTING works on the UDG and does not require any planarization. GREEDY ROUTING suffers from the problem of local maximum wherein a node is closer to the destination than all of its neighbors. Hence, in such a case the only route to the destination is to go through a node relatively farther from the destination. Figure 2.6 shows an example of such a topology. Here, p is closer to destination d than its neighbors x and y. Therefore, GREEDY ROUTING does not guarantee delivery. However, it gives a path with near optimal hop count.

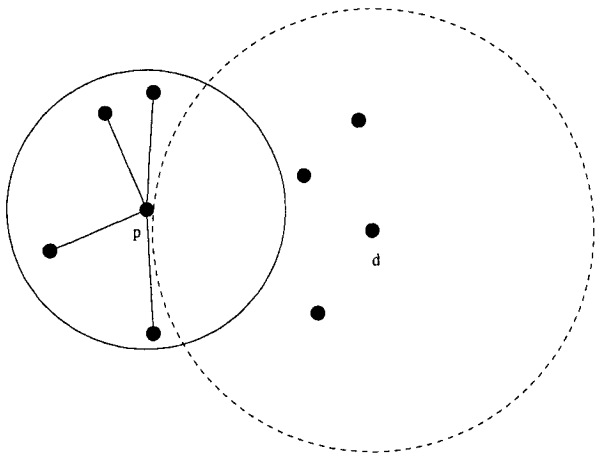


Figure 2.6: An example showing a topology for which GREEDY ROUTING fails. The dotted circle centered at d is to show that p is closer to d than all of its neighbors.

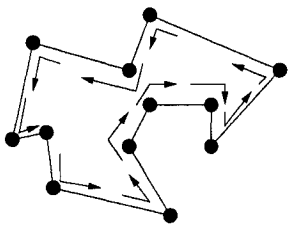


Figure 2.7: Traversal of a face using right hand rule.

2.5.2 Face Routing

A planar geometric graph G divides the plane into connected regions bounded by the edges of G called *faces*. FACE ROUTING relies on the traversal of these faces to route a packet from the source to the destination. For this it makes use of the *right hand rule* [BM76] which states that all the walls of a maze can be visited by keeping the right hand on the wall and walking in a forward direction. Since a maze essentially is one large face, this rule implies that all the edges of a face can be traversed by traversing the edge to the right of the current edge (which is also the first edge in clockwise order) moving in forward direction. An example of this is shown in Figure 2.7.

FACE ROUTING starts by sending the packet along the first edge in clockwise order with respect to the imaginary line segment $\overline{td}$ where t, to start with, is the location of the

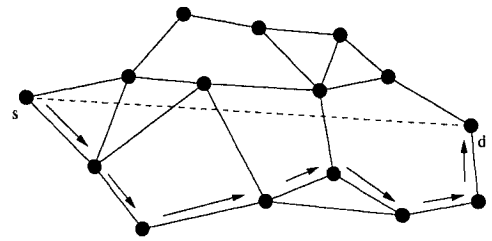


Figure 2.8: FACE ROUTING.

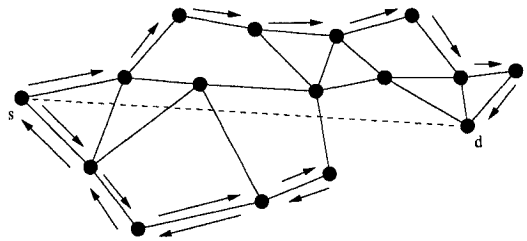


Figure 2.9: FACE ROUTING.

source node s and d is the location of the destination node. Every intermediate node sends the packet along the first edge in clockwise order with respect to the previous edge. If this edge intersects $\overline{td}$ and the distance between the intersection point and the destination location is less than the distance between t and d, then t is updated to this intersection point and the packet is sent along the first edge in clockwise order with respect to the intersecting edge. That is, the packet is forwarded to the next face. This updating and tracing of edges intersecting the imaginary line segment between the source and destination node locations ensures progress of the packet towards the destination. This process is followed until reaching the destination node. Figure 2.8 and Figure 2.9 show examples of the paths taken by FACE ROUTING. Note that in Figure 2.9, the exterior face of the planar graph G is traversed in *clockwise* direction while the interior faces are traversed in *counterclockwise* direction.

The FACE ROUTING algorithm used in this thesis is based on the FACE-2 algorithm given in [BMSU99]. The original FACE ROUTING algorithm which Bose *et al.* refer to as FACE-1 was given by Kranakis *et al.* in [KSU99]. In FACE-1, the entire face is traversed to determine the intersection point of line segment $\overline{td}$ and an edge, while in FACE-2, traversal

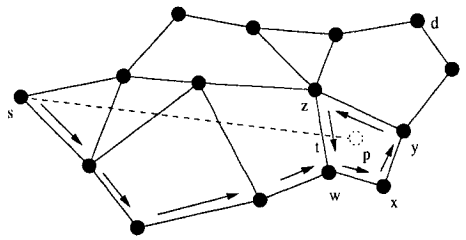


Figure 2.10: Failure of FACE ROUTING in presence of uncertainty in destination position.

of a face terminates on reaching an edge that intersects $\overline{td}$. The FACE ROUTING algorithm is known to guarantee delivery for all connected planar geometric graphs (*Theorem 5* of [BMSU99]).

However, FACE ROUTING cannot guarantee delivery with only an estimation of the destination's position. In particular, using FACE ROUTING to solve ROUTE-U(s,d,p,r) (refer to Section 1.3) results in the packet looping around the face enclosing p. See Figure 2.10 for an illustration of this phenomenon. Here, note that the intersection point t is discovered the second time while traversing the edge $[z,w]$, hence the packet does not change face and loops in face $[w,x,y,z]$.

2.5.3 GFG Routing

FACE ROUTING can sometimes lead to very long paths in the graph. Bose *et al.* [BMSU99] propose a combination of greedy and face routing called GFG ROUTING (Greedy-Face-Greedy) algorithm (also proposed by [KK00] as the GPSR routing protocol) that guarantees delivery of packets at the same time as reducing the length of paths. The GFG ROUTING algorithm follows GREEDY ROUTING until the current node has no neighbor closer to the destination than itself (such a node is called a *concave node*). At this node, the algorithm switches to FACE ROUTING. With the concave node as the source, face routing is followed until a node closer to the destination than the last concave node is encountered. At this node the algorithm switches back to GREEDY ROUTING. This switching between greedy and face routing continues till the destination is reached. An example of GFG ROUTING is shown in Figure 2.11. Here, the solid arrows denote the path taken by GREEDY ROUTING while the dotted arrows denote the path taken by FACE ROUTING, c is the concave node and the dotted circle centered at d is to show the distance of other nodes

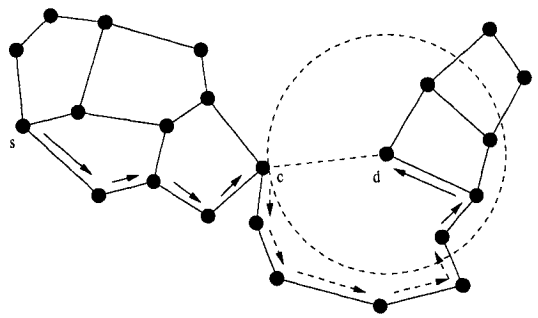


Figure 2.11: GFG ROUTING.

in comparison to the concave node c. The packet format and the pseudocode for the GFG ROUTING protocol are given in Appendix A.1.

2.6 Algorithms for Routing with Uncertainty

An obvious approach to solve ROUTE-U(s,d,p,r) is to use an algorithm to solve the ROUTE(s,d,p) problem, with an augmentation to stop if the destination node is found anywhere enroute. Since the destination may no longer be at position p, and may not be encountered enroute to the position p, the message may not be delivered. A second approach would be to ignore the imprecise location information and simply flood the entire network, but this is clearly resource-inefficient.

Therefore, we use the following approach to solve the ROUTE-U(s,d,p,r) problem. We refer to the circle of radius r centered at p as the *uncertainty zone* and r as the *uncertainty radius*. All our algorithms have two phases. In the first phase, we attempt to reach any node in the uncertainty zone. For this we use the GFG ROUTING algorithm as described above to solve the problem ROUTE(s,d,p). We follow the path given by this algorithm until we reach a node inside the uncertainty zone, we call this node the *entry node*. In the second phase, we attempt to find a path from the entry node to d, using either a face tree traversal-based approach or a flooding-based approach. The use of the face tree traversal approach to find the destination node within the uncertainty zone is based on the observation that all the nodes within a circular/rectangular region can be visited by traversing all the faces intersecting and contained in the region [BMSU99].

For convenience, in this thesis we always consider the uncertainty zone to be a circular

area. However, the algorithms presented in this thesis will work for uncertainty zone of any shape as long as it is a connected region in the plane.

In the following section we present several variants of the face tree traversal algorithm given in [BM02, Mor01]. In Chapter 3 we discuss the flooding-based algorithms.

2.7 Face Tree Traversal Algorithms

In [dBvOO96], de Berg *et al.* propose an algorithm to traverse a tree of all the faces in a planar geometric graph without using any additional memory. Bose and Morin applied this algorithm to the geocasting problem in the context of ad hoc networks to construct and traverse a tree of all the faces intersecting and contained in the geocasting region [BM02, Mor01]. The traversal of this tree, called the *face tree*, is followed until returning back to the start edge. This algorithm can clearly be used to solve the ROUTE-U(s,d,p,r) problem. In this section we describe the concept of face tree followed by various traversal algorithms. Based on their memory requirements, we classify the algorithms into memoryless, constant memory and non-constant memory. The memoryless algorithms do not maintain any routing state at the nodes. The constant memory algorithms require a constant amount of memory at every node for maintaining routing state. The non-constant memory algorithms require non-constant memory for the routing state at every node. In all algorithms, we assume that we are allowed to include $O(log\ n)$ bits of information in the header of the packet for routing purposes. Since the destination's identity always needs to be in the packet header, it always needs to contain $\Omega(log\ n)$ bits of information [1]. In all our protocols except one, the number of fields in the packet is constant; that is, it does not grow with the size of the network or the length of the path. Thus, all of our algorithms except one use $\Theta(log\ n)$ bits of information in the header for routing purposes.

2.7.1 Face Tree Construction

The input to the face tree construction is an arbitrary fixed point in the plane s'. With respect to this point, a total order on the edges of the planar graph G is defined. For every

[1] Here we assume that we use a simple addressing scheme in which every node is identified by a unique monotonically increasing number starting from zero. This helps us in comparing the algorithms in terms of their memory requirements. However, if a standard protocol such as IP is used for addressing, such an assumption would not be valid, and in that case an address field would have a fixed length.

edge $e = (u, v)$:

$$key_{s'}(e) = (distance(e, s'), \overrightarrow{s'c(e)}, \angle s'uv, \overrightarrow{uv})$$

where,

- $distance(e, s')$ is the radius of the smallest circle C centered at s' that intersects the edge e, let $c(e)$ be the point where C intersects e.
- $\overrightarrow{s'c(e)}$ is the angle of the line $\overline{s'c(e)}$ measured counterclockwise from the positive x-direction.
- $\angle s'uv$ is the smallest angle between s',u and v.
- $\overrightarrow{uv}$ is the angle of the line $\overline{uv}$ measured counterclockwise from the positive x-direction.

The total order on the edges is defined by lexicographic comparison of the key values of the edges. We denote this lexicographic comparison of the keys by $\prec$. Bose and Morin prove that for any two edges $e_1 \neq e_2$ of a planar geometric graph $key(e_1) \neq key(e_2)$ (*Lemma 10* from [Mor01]).

For every face f, an *entry edge* , $entry(f, s')$ is defined as the edge minimizing the value of the key amongst all other edges of the face. It follows from the total order defined above that an entry edge for a face is unique and well defined. Figure 2.12 shows the entry edge computation for face $[u, v, w]$. Here, as it can be seen that $distance([wv], s')$ is greater than $distance([uw], s')$ and $distance([uv], s')$. Therefore, $[wv]$ is not an entry edge. While comparing edges $[uw]$ and $[uv]$, it can be seen that $distance([uw], s') = distance([uv], s')$ and, $\overrightarrow{s'c(uw)} = \overrightarrow{s'c(uv)}$. However, the third component of the key, $\angle s'uw$ is smaller than $\angle s'uv$. Hence, edge $[uw]$ is the entry edge for the face $[uvw]$. The entry edges define a partial order between the faces of G in the following manner: for any face f, its parent f' is defined as the other face that the entry edge for f belongs to. Bose and Morin prove that for any face f of G that does not contain s', $entry(f, s')$ is on the boundary of two faces of G (*Lemma 11* from [Mor01]). This implies that all faces except the root face necessarily have a parent face, and that the relationship between faces defines a spanning tree of all faces in the graph. This tree is called the face tree, and the face containing s' is the root face. An example showing the face tree of a planar graph can be seen in Figure 2.13.

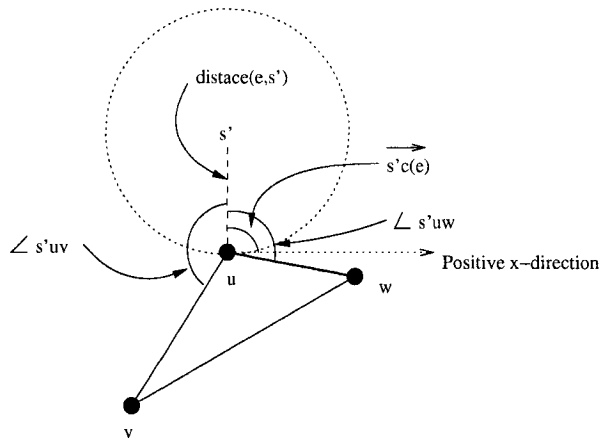


Figure 2.12: Entry edge computation for the face $[uvw]$. uw is the entry edge.

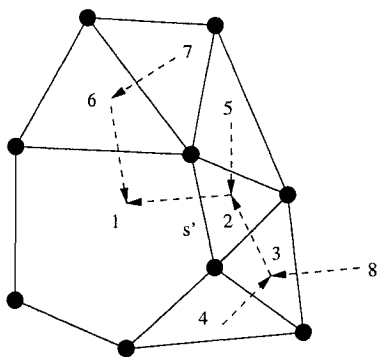


Figure 2.13: An example showing the face tree for a planar graph. The arrows moving from child to parent face, passing through the entry edges, represent the edges of the face tree.

2.7.2 Face Tree Traversal

In this section we describe various algorithms to traverse the face tree. These algorithms constitute the second phase of the solution to the ROUTE-U(s,d,p,r) problem with entry node q acting as their start node. The algorithms start at the *start edge* and traverse the faces intersecting or contained in the uncertainty zone until finding the destination or returning back to the start edge. The start edge is the first clockwise edge from $\overline{qp}$ incident on q. The arbitrary fixed point, s', with respect to which the face tree is defined, is computed by taking an arbitrary point to the left of the start edge. The algorithms differ in two ways: one, the order of traversal of the faces and, second, the way they compute the entry edge.

Doubling Face Tree Algorithm

This is essentially the algorithm given in [BM02, Mor01], modified to stop as soon as the destination is encountered. In this algorithm the faces of the face tree are traversed in depth-first order. Each face in turn is traversed using the right hand rule. Starting from the entry node on the face containing s', say $face\_of(s')$, the packet changes over to the opposite face (child of $face\_of(s')$) if the next edge to be traversed according to the right hand rule is an entry edge for the opposite face. The packet returns back to the parent face of the current face f if the next edge to be traversed is an entry edge for the current face. Note that a precondition for an edge to be an entry edge is that it should intersect the uncertainty zone. Figure 2.14 gives the pseudocode for this algorithm, which we call the FACE TREE algorithm. Here, $face\_of(s')$ is the face containing s', $next(e,f)$ is the next edge following the right hand rule on face f, $entry(f,s')$ is the entry edge for face f with respect to point s' and $opposite(e,f)$ is the other face that the edge e of face f belongs to.

As evident, the computation of an entry edge is an important part of the algorithm. For this, Bose and Morin give an innovative doubling approach to determine if a given edge e is the entry edge. Essentially, starting with $d = 1$, the packet traverses d edges to the left of e, then $2d$ edges to its right, and so on, until an edge e' is encountered such that $key_{s'}(e') \prec key_{s'}(e)$ (which confirms that e is not the entry edge), or until we are sure that all edges of the face have been seen (which confirms that e is the entry edge). The pseudocode for this entry edge computation algorithm is given in Figure 2.15. We call the FACE TREE algorithm combined with the entry edge computation as described above as the DOUBLING FACE TREE algorithm.

To reduce the path length, we make a slight change to the algorithm. We store the entry

FACE TREE **Algorithm**

1. $f \leftarrow face\_of(s')$
2. $e_{start} \leftarrow e \leftarrow$ first clockwise edge from $\overline{qp}$ incident on q //where q is the entry node
3. **repeat**
4. **if** e intersects the uncertainty zone **then**
5. **if** $e = entry(f, s')$ **then** // return to parent of f
6. $f \leftarrow opposite(e, f)$
7. **else if** $e = entry(opposite(e, f), s')$ **then** //visit child of f
8. $f \leftarrow opposite(e, f)$
9. **end if**
10. **end if**
11. $e \leftarrow next(e, f)$
12. **until** $destination =$ one of the end-points of e **or** $e = e_{start}$

Figure 2.14: The FACE TREE algorithm (from [Mor01]).

edge for the current face in the packet once it is found. Hence, the entry edge need not be computed again as long as we traverse the same face. For the example in Figure 2.13, the faces are traversed in the order 1, 2, 3, 4, 3, 8, 3, 2, 5, 2, 1, 6, 7, 6, 1. This algorithm is memoryless. Bose and Morin in [BM02] prove that it has a worst case complexity of $O(n \log n)$, where n is the number of nodes in the network. The packet format and the pseudocode for the DOUBLING FACE TREE protocol as implemented in this thesis are given in Appendix A.2.

Finally, the characteristics of DOUBLING FACE TREE can be summarized as follows:

Stretch factor : $O(n \log n)$
Memory at node : *Memoryless*
Memory in packet : $O(\log n)$
Duplicate packets : *No*

DFS Face Tree Algorithm

In the DFS FACE TREE algorithm, the faces of the planar graph are traversed in the same order as in the DOUBLING FACE TREE algorithm, hence, it uses the FACE TREE algorithm as given in Figure 2.14. However, the entry edge computation is different. In the DFS

DOUBLING ENTRY EDGE **Algorithm**

```
1.  k ← 1
2.  j ← i
3.  repeat
4.      while j ≠ i+k do
5.          j ← j+1
6.          if key_{s'}(e_i) ⪰ key_{s'}(e_j) then
7.              output false
8.          end if
9.      end while
10.     k ← 2k
11.     while j ≠ i−k do
12.         j ← j−1
13.         if key_{s'}(e_i) ⪰ key_{s'}(e_j) then
14.             output false
15.         end if
16.     end while
17.     k ← 2k
18. until k ≥ ⌈2n/3⌉
19. output true
```

Figure 2.15: The entry edge computation algorithm used by DOUBLING FACE TREE (from [Mor01]).

FACE TREE algorithm, to determine if a given edge is the entry edge, we simply do a right hand based traversal of the entire face keeping track of the edge with minimum key value. This entry edge computation algorithm is given in Figure 2.16. This is the same as proposed by de Berg *et al.* in [dBvOO96]. As in DOUBLING FACE TREE, we store the entry edge for the current face in the packet to avoid doing the same computation a second time. This algorithm is also memoryless but has a complexity of $O(n^3)$. The packet format and the pseudocode for the DFS FACE TREE protocol are given in Appendix A.3.

Interestingly, as our experimental results show, there are many situations where DFS FACE TREE has smaller path length than DOUBLING FACE TREE . The reason is that DFS FACE TREE can sometimes end up encountering the actual destination while simply doing its entry edge computation, which involves traversing the entire face. Note that when used for geocasting, DFS FACE TREE can never have a better performance than DOUBLING FACE TREE .

An example of this phenomenon is shown in Figure 2.17. The start edge is (q,r). DFS FACE TREE finds the destination d while checking if (q,v) is an entry edge for face 4, while visiting face 2. On the other hand, DOUBLING FACE TREE quickly determines that (q,v) is not the entry edge for face 4, therefore visits face 2 then returns to face 1, then visits face 3, then returns to face 1, and now while checking if edge (u,q) is an entry edge for the opposite face, it finds the destination d. For this example, the length of the path found by DOUBLING FACE TREE is 56 hops while that found by DFS FACE TREE is 14 hops.

Finally, the characteristics of DFS FACE TREE can be summarized as follows:

Stretch factor : $O(n^2)$
Memory at node : *Memoryless*
Memory in packet : $O(\log n)$
Duplicate packets : *No*

Mark Entry Edge Face Tree Algorithm

In this version of face tree traversal, faces are traversed in the same order as in DOUBLING FACE TREE (which is given in Figure 2.14), but the entry edge computation is different. In the MARK ENTRY EDGE FACE TREE algorithm, every time we enter a new face, we find the entry edge in the face and *mark* it. This means the next time we enter the face, we do

ENTRY EDGE **Algorithm**

1. $e_{start} \leftarrow e \leftarrow$ start edge for the face
2. $e_{entry} \leftarrow e_{start}$
3. **repeat**
4. **if** $key_{s'}(e) \prec key_{s'}(e_{entry})$ **then**
5. $e_{entry} \leftarrow e$
6. **end if**
7. $e \leftarrow next(e, f)$
8. **until** $e = e_{start}$
9. **output** e_{entry}

Figure 2.16: The entry edge computation algorithm used by DFS FACE TREE.

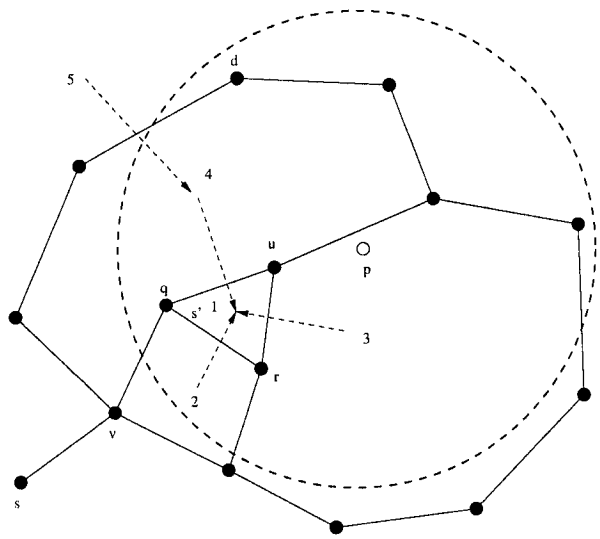


Figure 2.17: An example where the path taken by DOUBLING FACE TREE is much longer than that taken by DFS FACE TREE .

not need to do the entry edge computation, which potentially decreases the number of hops required. Hence, as shown in Figure 2.13, when the packet returns to face 3 from face 4, the entry edge for the face has already been computed and saved. This is unlike the DOUBLING FACE TREE and the DFS FACE TREE algorithms where the entry edge will need to be computed again. The entry edge for face 3 is computed when face 3 is entered for the first time through face 2. In the MARK ENTRY EDGE FACE TREE algorithm, first-time entry edge computation is done in the same way as in the DFS FACE TREE algorithm, that is, by traversing the entire face. The marking of the edges requires storing of constant memory routing state at the nodes. The pseudocode for the entry edge computation algorithm is given in Figure 2.18. The complexity of this algorithm is $O(n)$. The packet format and the pseudocode for the MARK ENTRY EDGE FACE TREE protocol are given in Appendix A.4.

Finally, the characteristics of MARK ENTRY EDGE FACE TREE can be summarized as follows:

Stretch factor : $O(n)$
Memory at node : $O(d)$
Memory in packet : $O(\log n)$
Duplicate packets : *No*

BFS Face Tree Algorithm

In this algorithm, the face tree is traversed in breadth first order. Therefore, for the example in Figure 2.13, the faces are traversed in the order 1, 2, 1, 6, 1, 2, 3, 2, 5, 2, 1, 6, 7, 6, 1, 2, 3, 4, 3, 8, 3, 2, 1. The algorithm maintains a queue Q of entry edges for the faces in the packet that determines the order of traversal of the faces. The first traversal of a face involves identifying the entry edges for the opposite faces and putting them in Q. Note that unlike DOUBLING FACE TREE and DFS FACE TREE, BFS FACE TREE does not visit the opposite face immediately on encountering the entry edge for it. The opposite face is visited, if the current edge is the first element in Q. This is while traversing the face a second time. Every time an opposite face f is visited, its corresponding entry edge, which is the first element in Q, is dequeued and enqueued. At this point the entry edge for f is the last element of Q. Again, a first traversal of face f enqueues the entry edges for the opposite faces. In Q, these edges are right after the entry edge for f. However, if f does

MARK ENTRY EDGE **Algorithm**

1. $e_{start} \leftarrow e \leftarrow$ start edge for the face
2. $e_{entry} \leftarrow e_{start}$
3. $entry[e] \leftarrow null \; \forall \; e \in$ GABRIEL-GRAPH(G)
4. **repeat**
5. **if** $key_{s'}(e) \prec key_{s'}(e_{entry})$ **then**
6. $e_{entry} \leftarrow e$
7. **end if**
8. $e \leftarrow next(e, f)$
9. **until** $e = e_{start}$
10. **repeat**
11. **if** $e = e_{entry}$ **then**
12. $entry[e] \leftarrow true$
13. **else**
14. $entry[e] \leftarrow false$
15. **end if**
16. $e \leftarrow next(e, f)$
17. **until** $e = e_{start}$

Figure 2.18: The entry edge computation algorithm used by MARK ENTRY EDGE FACE TREE.

not have any of its edges as entry edge for the opposite faces then the entry edge for f is still the last element in Q. This implies that this face is a leaf node in the face tree and should not be visited again. This check is made while returning to the parent face of f through the entry edge for f, and if it is true then this edge is dequeued from Q and added to a set S also maintained in the packet. The membership of an entry edge of f in the set S indicates that the subtree rooted at f has been completely traversed. Entry edges for the opposite face are enqueued in Q only if they are not present in S. Following the algorithm as described above ensures that the faces of the face tree are traversed in breadth first order. The algorithm stops if the destination is found enroute or if Q is empty. Figure 2.19 gives the pseudocode for the BFS FACE TREE algorithm.

An example of the algorithm showing up to two levels of traversal of the face tree is given in Figure 2.20. Here, the start edge is $e1$ and Q at ei denotes the state of Q while traversing edge ei. The state of the queue is shown only for the entry edges since the queue does not change while traversing other edges. The algorithm begins by traversing face 1 once and enqueuing edges $e1$ and $e2$. The packet moves to edge $e3$ on face 2 while traversing face 1 the second time. This is because, now $e1$ is the first element in Q. Before starting the traversal of face 2, edge $e1$ is dequeued and enqueued again. The first traversal of face 2 adds edges $e3$ and $e4$ to the queue. The packet returns to the parent face 1 through edge $e1$. This process is followed until reaching face 5 in Step 15. Face 5 is a leaf in the face tree and does not have any of its edges as an entry edge for the opposite faces. Hence, in Step 16., edge $e4$ is all together dequeued from Q and added to S. Because of this adding of $e4$ in S, face 5 will not be visited again when the packet comes back to face 2 to visit faces 4 and 8, which are at level 3 of the face tree.

Note that at any point of time, there are no duplicate edges in the queue. Hence the bound on the size of the queue is $O(F)$, where F is the number of faces in the graph. However, for planar graphs in the worst case this can be $O(n)$, where n is the number of nodes in the network. The time complexity of the BFS FACE TREE algorithm is $O(n^3)$.

Note that although we discuss the BFS FACE TREE algorithm in terms of edges, our actual implementation involves nodes and packets. We give the pseudocode for the implemented protocol in Appendix A.5.

BFS FACE TREE **Algorithm**

1. $S \leftarrow \emptyset$
2. $Q \leftarrow null$
3. $f \leftarrow face\_of(s')$
4. $e \leftarrow$ first clockwise edge from $\overline{qp}$ incident on q //where q is the entry node
5. **repeat**
6. **if** e intersects the uncertainty zone **then**
7. **if** $e = first(Q)$ **then**
8. $enqueue(Q, dequeue(Q))$
9. $f \leftarrow opposite(e, f)$ //visit child of f
10. **else if** $e = entry(f, s')$ **then**
11. **if** $e = last(Q)$ **then**
12. $S \leftarrow S \cup \{dequeue(Q)\}$
13. **if** $empty(Q) = true$ **then**
14. **return**
15. **end if**
16. **end if**
17. $f \leftarrow opposite(e, f)$ //return to parent of f
18. **else if** $e = entry(opposite(e, f), s')$ **then**
19. **if** $e \notin S$ **then**
20. $enqueue(Q, e)$
21. **end if**
22. **end if**
23. **end if**
24. $e \leftarrow next(e, f)$
25. **until** $destination =$ one of the end-points of e

Figure 2.19: The BFS FACE TREE algorithm.

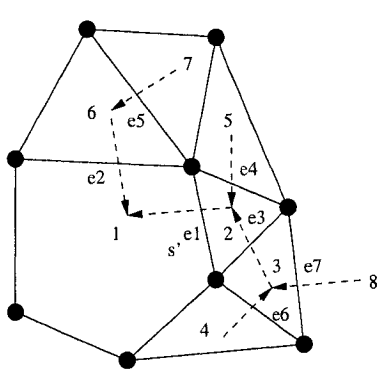
Finally, the characteristics of BFS FACE TREE can be summarized as follows:

Stretch factor : $O(n^3)$

Memory at node : *Memoryless*

Memory in packet : $O(n \log n)$

Duplicate packets : *No*



Step 1. Q at e1: e1
Step 2. Q at e2: e1, e2
Step 3. Q at e1: e2, e1
Step 4. Q at e3: e2, e1, e3
Step 5. Q at e4: e2, e1, e3, e4
Step 6. Q at e1: e2, e1, e3, e4
Step 7. Q at e2: e1, e3, e4, e2
Step 8. Q at e5: e1, e3, e4, e2, e5
Step 9. Q at e2: e1, e3, e4, e2, e5
Step 10. Q at e1: e3, e4, e2, e5, e1
Step 11. Q at e3: e4, e2, e5, e1, e3
Step 12. Q at e6: e4, e2, e5, e1, e3, e6
Step 13. Q at e7: e4, e2, e5, e1, e3, e6, e7
Step 14. Q at e3: e4, e2, e5, e1, e3, e6, e7
Step 15. Q at e4: e2, e5, e1, e3, e6, e7, e4
Step 16. Q at e4: e2, e5, e1, e3, e6, e7 S: e4
Step 17. Q at e1: e2, e5, e1, e3, e6, e7
Step 18. Q at e2: e5, e1, e3, e6, e7, e2
Step 19. Q at e5: e1, e3, e6, e7, e5
Step 20. Q at e5: e1, e3, e6, e7 S: e4,e5
Step 21. Q at e2: e1, e3, e6, e7
.....

Figure 2.20: An example showing the execution of BFS FACE TREE up to two levels of the face tree.

|| Face Tree Algorithm

This is a parallelized version of the Face Tree algorithm. While traversing a face, if a given edge is the entry edge for the opposite face, then another copy of the packet is spawned to explore that face. Each packet continues the traversal until either finding the destination or until completing the traversal of a face. Figure 2.21 gives the pseudocode for the algorithm. The algorithm is memoryless and has a time complexity of $O(n)$. For the example in Figure 2.13, first face 1 is traversed, then faces 2 and 6 in parallel, then faces 3 and 5 in parallel, and finally faces 7, 4 and 8 in parallel. Unlike all other algorithms in this chapter, this algorithm involves many duplicate packets in the network to solve the same routing problem. The number of concurrent duplicate packets is $O(f)$ where f is the number of faces in the graph. The packet format and the pseudocode for the || FACE TREE protocol are given in Appendix A.6.

Finally, the characteristics of || FACE TREE can be summarized as follows:

Stretch factor : $O(n)$
Memory at node : *Memoryless*
Memory in packet : $O(\log n)$
Duplicate packets : *Yes*

|| FACE TREE **Algorithm**

1. $f \leftarrow face\_of(s')$
2. $e \leftarrow$ first clockwise edge from $\overline{qp}$ incident on q //where q is the entry node
3. VISIT(f, e)

VISIT(f, e)

1. **repeat**
2. **if** e intersects the uncertainty zone **then**
3. **if** $e = entry(f, s')$ **then**
4. drop packet
5. **else if** $e = entry(opposite(e, f), s')$ **then** //visit child of f
6. spawn a new packet to solve VISIT$(opposite(e, f), e)$
7. **end if**
8. **end if**
9. $e \leftarrow next(e, f)$
10. **until** $destination =$ one of the end-points of e

Figure 2.21: The || FACE TREE algorithm.

Chapter 3

Flooding Based Protocols

The following chapter discusses algorithms to solve the ROUTE-U(s,d,p,r) problem that are based on simple flooding of the uncertainty zone. The algorithms discussed in Chapter 2, except the || FACE TREE algorithm, have only a single copy of the packet traversing the network at any point of time. On the other hand, flooding-based algorithms generate a vast number of duplicate packets that traverse the network concurrently thereby increasing the energy cost of such algorithms. Flooding algorithms are simple and easy to implement, however they fail to guarantee delivery. In the following sections we present the simple flooding algorithm. We also present three variants of simple flooding which attempt to improve the delivery rate and reduce the number of duplicate packets in the network. In all the algorithms, we follow the same path as GFG ROUTING on the problem ROUTE(s,d,p), until reaching the entry node. In the following sections, we assume that the packet has already reached the entry node within the uncertainty zone and we focus on the algorithm starting at the entry node.

3.1 All-Neighbor (AN) Flooding

This is the simple flooding algorithm. Here, to start with, an entry node broadcasts the packet to all its 1-hop neighbors. Any node receiving this packet, broadcasts it if and only if it lies inside the region. Any further copies of the packet are ignored. The packet format and protocol pseudocode for AN FLOODING are given in Appendix A.7.

In all the flooding-based approaches, a unique identifier corresponding to the packet to be flooded is stored at each node, to prevent re-transmission of the same packet *ad*

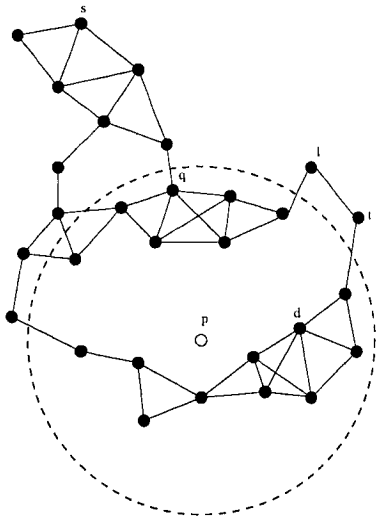


Figure 3.1: An example where flooding within the uncertainty zone fails

infinitum. This requires a per packet constant memory of routing state to be maintained at the nodes. The time complexity of the flooding-based algorithms is $O(n)$, which is the diameter of the network in the worst case. Also, in flooding-based algorithms, unlike the face tree traversal-based algorithms, all the edges in the unit disk graph are traversed.

As mentioned earlier, flooding within the uncertainty zone does not guarantee delivery of packet to the destination. This is because all paths from the entry node to the destination may go through nodes outside the uncertainty zone. Figure 3.1 shows an example of such a case.

Finally, the characteristics of AN FLOODING can be summarized as follows:

Stretch factor : $O(n)$
Memory at node : $O(1)$
Memory in packet : $O(\log n)$
Duplicate packets : *Yes*

GREEDY NEIGHBOR SUBSET **Algorithm**

1. $v \leftarrow$ current node
2. $R \leftarrow N_2(v)$
3. $S \leftarrow \emptyset$
4. **while** $R \neq \emptyset$ **do**
5. select an $x \in N_1(v)$ that maximizes $| N_1(x) \cap R |$
6. $R \leftarrow R - N_1(x)$
7. $S \leftarrow S \cup \{x\}$
8. **end while**
9. **output** S

Figure 3.2: Greedy algorithm to choose minimum subset of 1-hop neighbors that cover all 2-hop neighbors. $N_1(v)$ is the set of 1-hop neighbors of a node v that lie within the uncertainty zone and $N_2(v)$ is the set of 2-hop neighbors of a node v that lie within the uncertainty zone.

3.2 Subset-Neighbor (SN) Flooding

The difference between Subset Neighbor (SN) Flooding and AN Flooding is that nodes inside the region, instead of broadcasting the packet to all 1-hop neighbors, forward it to only a subset of 1-hop neighbors whose neighbors in turn include all 2-hop neighbors of the original node. Computing a minimal subset of such 1-hop neighbors is known to be an NP-complete problem for arbitrary graphs, but its complexity for unit disk graphs is unknown [CMWZ04]. Hence we use a greedy algorithm to compute the subset: we iteratively select a 1-hop neighbor that covers the maximum number of 2-hop neighbors not yet covered, and terminates when all 2-hop neighbors have been covered. Note that only the 2-hop neighbors inside the uncertainty region are considered in the algorithm. The pseudocode for the algorithm to compute the subset is illustrated in Figure 3.2.

The SN FLOODING algorithm reduces the transmission cost by restricting flooding of packets, however, it fails to deliver the packet in some situations where AN FLOODING succeeds. As seen in Figure 3.3, the packet for the problem ROUTE-U(s,d,p,r) is not sent to t, the 2-hop neighbor of r, since only 2-hop neighbors inside the uncertainty region are considered by the SN FLOODING algorithm. Hence, the packet is not delivered to d. Note that in all the face tree traversal-based and flooding-based algorithms, a packet is directly delivered to the destination if the destination is a 1-hop neighbor of the current node. Now, in contrast to SN FLOODING , in AN FLOODING all nodes inside the region

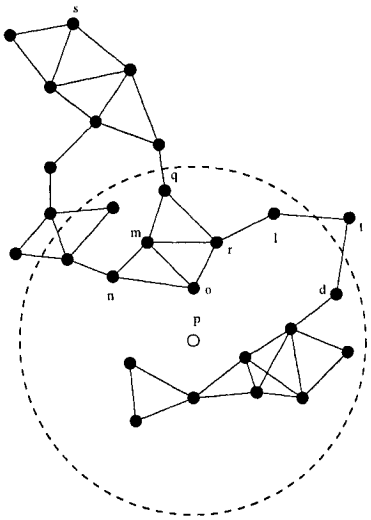


Figure 3.3: An example where SN FLOODING within the uncertainty circle fails but AN FLOODING succeeds.

flood to all their neighbors. Hence, the packet will be sent to t, which in turn will deliver it to d, its 1-hop neighbor.

The packet format and protocol pseudocode for SN FLOODING are given in Appendix A.8.

Finally, the characteristics of AN FLOODING can be summarized as follows:

Stretch factor: $O(n)$
Memory at node: $O(1)$
Memory in packet: $O(\log n)$
Duplicate packets: *Yes*

3.3 Extended AN Flooding

In order to discover paths to the destination, we extend the region of flooding by a constant value in this algorithm. EXTENDED AN FLOODING on the problem ROUTE-U(s,d,p,r) is identical to flooding on the problem ROUTE-U$(s,d,p,r+\lambda)$ where λ is the transmission radius of nodes. Therefore, in Figure 3.1, the packet will be sent to nodes l and t thereby reach the destination d.

The characteristics of EXTENDED AN FLOODING can be summarized as follows:

Stretch factor : $O(n)$
Memory at node : $O(1)$
Memory in packet : $O(\log n)$
Duplicate packets : *Yes*

3.4 Extended SN Flooding

In this algorithm, we extend the uncertainty radius of SN FLOODING . That is, EXTENDED SN FLOODING on the problem ROUTE-U(s,d,p,r) is the same as SN FLOODING on the problem ROUTE-U$(s,d,p,r+\lambda)$.

The extended flooding algorithms are expected to have higher transmission cost because of the increased flooding area, however they also have higher delivery rate. This increase in delivery rate is achieved without compromising the stretch factor.

The characteristics of EXTENDED SN FLOODING can be summarized as follows:

Stretch factor : $O(n)$
Memory at node : $O(1)$
Memory in packet : $O(\log n)$
Duplicate packets : *Yes*

Chapter 4

Simulation Results and Discussion

To compare the performance of the algorithms given in Chapters 2 and 3, we simulate them on a large number of static network topologies. We also study the effect of the number of nodes and the size of the uncertainty radius on delivery rate, stretch factor, and the transmission cost. In this chapter, we present the results of our simulations. Section 4.1 gives the details of our simulation environment. We discuss the results of face tree traversal-based algorithms in Section 4.2. In Section 4.3, we discuss the results of flooding-based algorithms. Finally, in Section 4.4 we compare and contrast the two classes of algorithms.

4.1 Simulation Environment

In the simulation experiments, a set S of n points (where $n \in \{75, 100, 125\}$) is randomly generated on a rectangle of 800m by 700m. For the transmission range λ of nodes, we use 120m. We implement a beaconing protocol for nodes to discover their neighbor's address and location. A single run of the beaconing protocol generates the unit disk graph $G = UDG(S)$. Note that the density of the network is 6, 8 and 10 nodes per unit disk for $n = 75$, $n = 100$ and $n = 125$ respectively. After generating the UDG, a source and destination node is randomly chosen. If there is no path from s to d in $UDG(S)$, the graph is discarded; otherwise, all routing algorithms are applied on G. This process is repeated for 100 node-pairs on each graph, and then for 1000 graphs. We do this for increasing uncertainty radius starting from zero to 10λ, in which case the whole network is contained within the uncertainty zone. Recall that the uncertainty radius $r = v(t_1 - t_0)$. Here, we assume that the network latency l is negligible as compared to $t_1 - t_0$. However, if that is

not the case then a better estimation of the uncertainty radius r would be $v(t_1 + l - t_0)$.

In our experiments we measure delivery rate, stretch factor and transmission cost. The delivery rate is the percentage of packets that get transmitted successfully to the destination on a valid graph. The stretch factor is the number of hops taken by a packet compared to the minimum hop path available in the network, averaged over all successfully delivered packets. The transmission cost is the ratio of total number of times that copies of the packet get transmitted in the course of successful delivery of the packet to the number of transmissions in the minimum hop path, averaged over all successfully delivered packets. It is a measure of the energy costs of the algorithm.

4.2 Face tree traversal-based algorithms

The graphs in Figure 4.1 show the stretch factor for face tree traversal-based algorithms for 75, 100, and 125 nodes, respectively. The stretch factor of || FACE TREE is best followed by BFS FACE TREE, DFS FACE TREE, MARK ENTRY EDGE FACE TREE, which all have similar performance followed by DOUBLING FACE TREE which has the worst performance. While all others are quite close, the doubling algorithm's performance is significantly worse. The worst case time complexity of DOUBLING FACE TREE is better than DFS FACE TREE , however, as mentioned above, DFS FACE TREE has better stretch factor than DOUBLING FACE TREE. The reason is that DFS FACE TREE can sometimes end up encountering the actual destination while simply doing its entry edge computation, which involves traversing the entire face. Note that when used for geocasting, DFS FACE TREE can never have a better performance than DOUBLING FACE TREE .

The stretch factor of all algorithms increases with increasing uncertainty radius, especially the non-parallelized algorithms. This is because as the uncertainty radius increases, more and more of the graph is being searched using the face tree traversal method. In particular, there is a greater chance of the exterior face being traversed.

Somewhat surprisingly, using marked bits does not help in reducing the stretch factor for smaller values of uncertainty radius. At higher values of uncertainty radius, the MARK ENTRY EDGE FACE TREE implementation starts to outperform all face tree traversal-based algorithms except || FACE TREE.

The graphs in Figure 4.2 show the transmission cost for face tree traversal-based algorithms for 75, 100, and 125 nodes, respectively. The transmission cost of BFS FACE

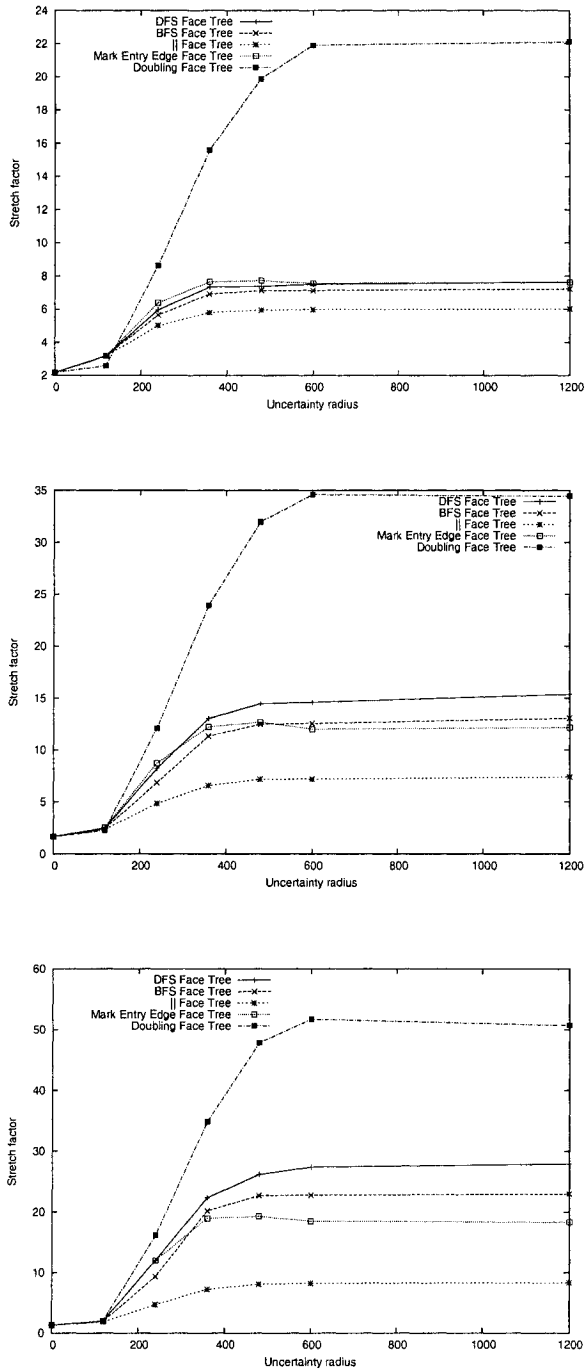


Figure 4.1: Stretch factor for varying uncertainty radius for 75, 100, and 125 nodes respectively. Simulation field - 800 x 700. Transmission radius - 120

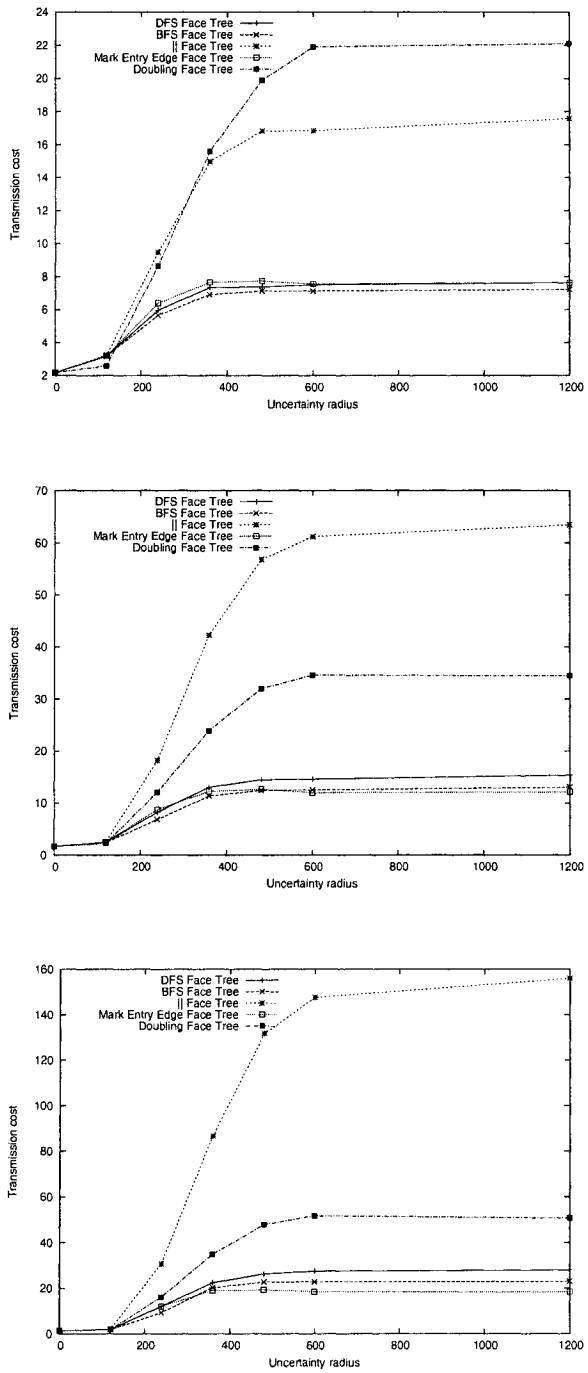


Figure 4.2: Transmission cost for varying uncertainty radius for 75, 100, and 125 nodes respectively. Simulation field - 800 x 700. Transmission radius - 120

TREE, MARK ENTRY EDGE FACE TREE, and DFS FACE TREE is much lower than that of || FACE TREE or DOUBLING FACE TREE. For all algorithms, as the number of nodes increases, the stretch factor and transmission cost both increase. In particular, the transmission cost of || FACE TREE becomes intolerable. || FACE TREE uses more energy because there is no way of stopping the spawned packets that are exploring other faces while the destination has been already found. We expect that if the destination node does not lie within the uncertainty zone then || FACE TREE will have the same energy cost as other algorithms.

The delivery rate is 1 for all face tree traversal-based algorithms and is therefore not shown here.

We conclude that the performance of DFS FACE TREE, BFS FACE TREE, and MARK ENTRY EDGE FACE TREE are all similar, and they use the least energy, while || FACE TREE achieves the best stretch factor.

4.3 Flooding-based algorithms

The graphs in Figure 4.3 show the delivery rate for flooding-based algorithms for 75, 100 and 125 nodes, respectively. It is interesting to note that all flooding-based algorithms have a dip in the delivery rate at around $r = 2\lambda$ where λ is the transmission radius of nodes. This can be explained by the following observation. When $r \leq \lambda/2$, all nodes in the uncertainty circle are directly connected to each other, therefore flooding must succeed (this is confirmed by the experiments). Similarly, when the uncertainty radius is large enough that the entire field is contained in the uncertainty circle, flooding is guaranteed to succeed, since we only use connected graphs. However, as the uncertainty radius increases from $\lambda/2$, the chance of having two disconnected components inside the uncertainty circle first decreases, and then again increases as the number of nodes inside the zone increases. At what radius is the probability of having two or more disconnected components in the uncertainty zone highest? Is it when r is close to 2λ or does it depend on the size of the simulation area?

To answer these questions, we carried out simulations with large number of nodes for the AN FLOODING algorithm. The transmission radius is the same, however the length and width of the simulation area are now chosen to be 30λ. We pick the number of nodes such that the density is 10 nodes per unit disk. Figure 4.4 shows the results of our simulations.

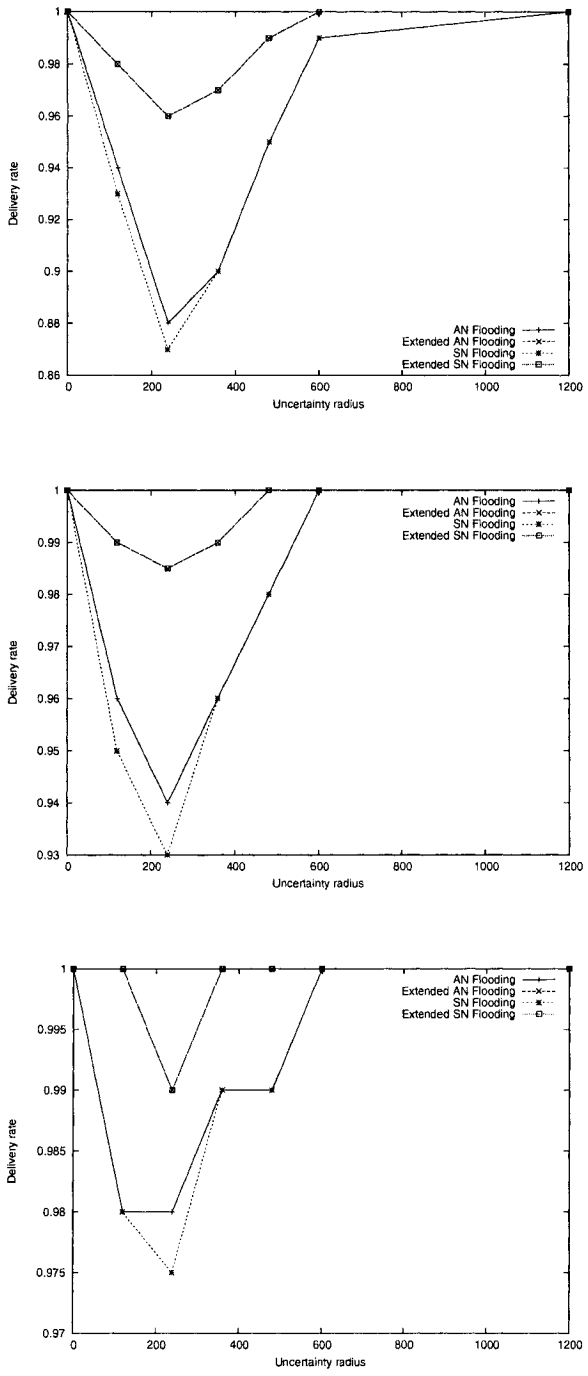


Figure 4.3: Delivery rate for varying uncertainty radius for 75, 100, and 125 nodes respectively. Simulation field - 800 x 700. Transmission radius - 120

As we can see, AN FLOODING follows the same trend as in the simulations with smaller number of nodes, that is, the delivery rate first decreases and then increases as the number of nodes within the zone increases. Also, the dip occurs around $r = 2\lambda$, which is the same as earlier simulations.

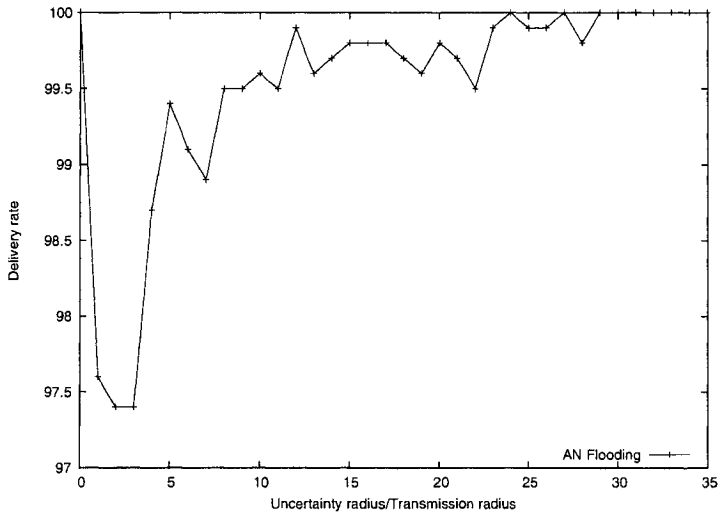


Figure 4.4: Delivery rate of AN FLOODING for increasing r/λ. Simulation field - 3600 x 3600. Number of nodes - 2866. Transmission radius - 120

EXTENDED AN FLOODING improves the delivery rate, but cannot guarantee delivery for all values of uncertainty radius, though always over 95% for all values studied.

The graphs in Figure 4.5 show the stretch factor for flooding-based algorithms for 75, 100 and 125 nodes, respectively. The stretch factor is almost the same for all flooding-based approaches. The stretch factor is in general very good, always less than 2.2, and exactly 1 for uncertainty radius $>= 5\lambda$. Note that the stretch factor of the route found in the second phase is exactly 1 for AN FLOODING and EXTENDED AN FLOODING; the shortest path may not be found in the first phase where GFG is being used. As the uncertainty radius increases, the part of the path that is constructed in the first phase is proportionally smaller; this explains why the stretch factor decreases as the uncertainty radius increases. Also, the stretch factor decreases as the number of nodes increases; this is in line with the well-known behavior of GFG ROUTING.

The graphs in Figure 4.6 show the transmission cost for flooding-based algorithms for

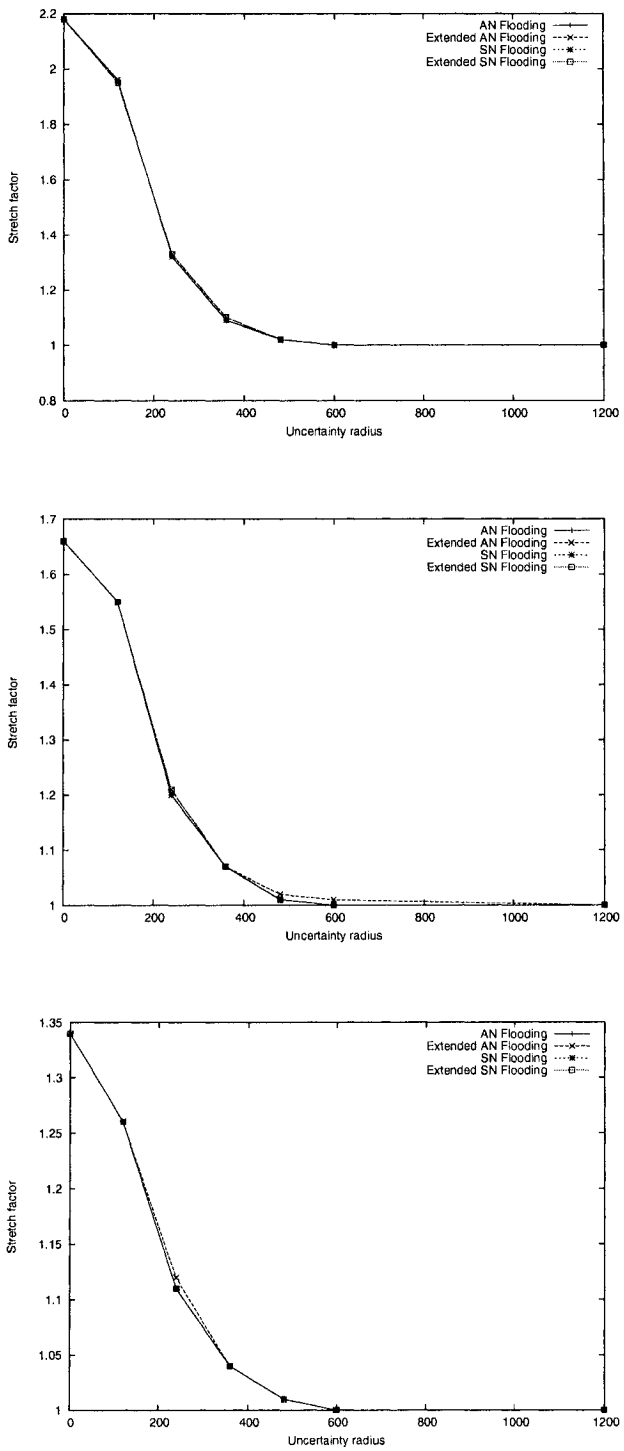


Figure 4.5: Stretch factor for varying uncertainty radius for 75, 100, and 125 nodes respectively. Simulation field - 800 x 700. Transmission radius - 120

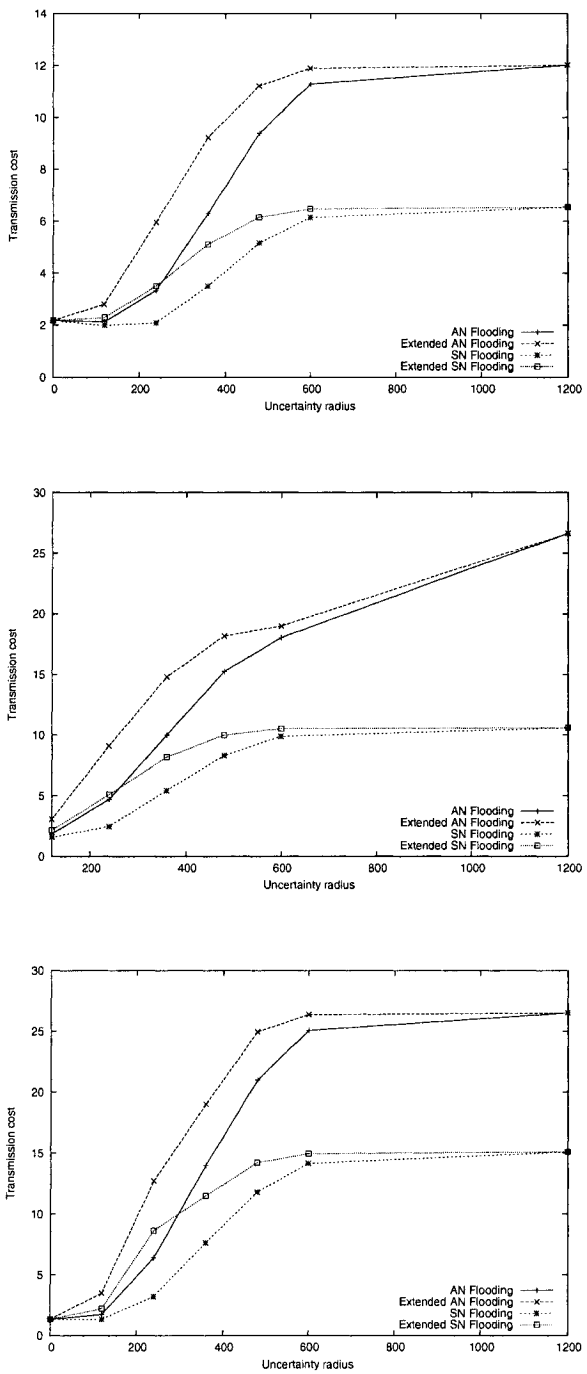


Figure 4.6: Transmission cost for varying uncertainty radius for 75, 100, and 125 nodes respectively. Simulation field - 800 x 700. Transmission radius - 120

75, 100 and 125 nodes, respectively.The transmission cost of SN FLOODING and EXTENDED SN FLOODING are similar and much lower than other flooding-based approaches. The cost of AN FLOODING is higher, and the cost of EXTENDED AN FLOODING is even higher.

EXTENDED SN FLOODING appears to achieve a good balance between delivery rate and transmission cost.

4.4 Comparison between two approaches

It is clear that the flooding-based approaches have much better stretch factor than the face-tree approaches. However, many of the face tree traversal-based algorithms examined here are memoryless, and as such are not comparable to the flooding-based algorithms, which all require routing state to be maintained at nodes. The interesting finding is that incorporating routing state into face tree traversal as in MARK ENTRY EDGE FACE TREE does not seem to improve the stretch factor except when the uncertainty is very high.

Flooding is known to be resource-inefficient. However, our experiments show that the transmission cost of AN FLOODING, the most expensive algorithm in the flooding-based class, while worse than most of the face tree traversal-based approaches, is not significantly higher, and indeed, is better than the || FACE TREE and DOUBLING FACE TREE algorithms. At the other end, the transmission costs of SN FLOODING and EXTENDED SN FLOODING are lower than the cheapest face tree traversal-based algorithm. Thus, making minor adjustments to the basic flooding algorithm results in greatly reduced transmission cost while not sacrificing the delivery rate or the stretch factor. Meanwhile, the high stretch factor of the face tree traversal-based approaches also translates to a high transmission cost.

Put another way, while flooding uses duplicate packets and seems to explore all possible edges in an unintelligent manner, the maximum number of edge traversals performed by flooding is $2e$, since each end-point of an edge will send a packet across the edge at most once. In face tree traversal-based algorithms edges should be traversed at most once, but in fact, the entry edge computation often imply that an edge is traversed many times.

Chapter 5

Conclusion

With the slogan 'anytime anywhere', today's communication world demands information access on the move, in many cases without any previously established infrastructure. Ad hoc networking is the technology that realizes this vision. An ad hoc network is a collection of wireless devices, called nodes, that do not need to rely on any predefined infrastructure to keep the network connected. In the absence of infrastructure, designing routing protocols for such networks is a challenging task, especially in the face of changing topologies. To this end, several authors have proposed the use of host *location information* [KSU99, BMSU99, KK00, KV98, BBC+01]. In such protocols, called position-based routing protocols, every node in the network is assumed to know the locations of itself and its neighbors. Also it is assumed that the position of the destination host is known to the source node. It is this assumption that we question in this thesis.

We study the problem of routing in an ad hoc network where the source node is uncertain about the exact current position of the destination but knows only an approximate position of the destination node. We formulate the problem as routing from a source node s to a destination node d contained in the circle of radius r centered at position p and call it the ROUTE-U(s,d,p,r) problem. We call the circle the uncertainty zone and the radius the uncertainty radius. The position p can be understood as the last known location of the destination node.

To solve this problem, we divide it into two phases. The first phase involves reaching a node inside the uncertainty zone, while the second phase involves searching for the destination node within the uncertainty zone. For the first phase, we use the GFG ROUTING algorithm as proposed in [BMSU99] and [KK00]. For the second phase, we investigate

two classes of algorithms: one, based on a traversal of a tree of the faces of a planar subgraph of the graph representing the network, and the second, based on flooding of the uncertainty zone. We discuss the face tree traversal algorithm proposed by Bose and Morin [BM02, Mor01] (we call it the DOUBLING FACE TREE algorithm) and propose four variants: DFS FACE TREE, BFS FACE TREE , MARK ENTRY EDGE FACE TREE and || FACE TREE . The face tree traversal-based algorithms guarantee delivery. We also propose some algorithms based on flooding the uncertainty zone. Flooding does not guarantee delivery, hence, we propose EXTENDED AN FLOODING , which results in an increase in the delivery rate while SN FLOODING and EXTENDED SN FLOODING aim to reduce the transmission cost. Table 1 gives a comparison of the algorithms. We also give detailed pseudocode for the corresponding protocol implementation for each of the algorithms in the appendices. This can serve as a guide for anyone wanting to implement them.

Table 1: A comparison of the algorithms. Here, n is the number of nodes in the network and d is the maximum degree of a node. All memory requirements are in terms of number of bits. The symbol – means the algorithm is memoryless.

Algorithm	**Guaranteed delivery**	**Worst case complexity**	**Memory at node**	**Memory in packet**	**Duplicate packets**
DOUBLING FACE TREE [BMSU99, Mor01]	Yes	$O(n \log n)$	–	$O(\log n)$	No
DFS FACE TREE [dBvOO96]	Yes	$O(n^2)$	–	$O(\log n)$	No
BFS FACE TREE	Yes	$O(n^3)$	–	$O(n \log n)$	No
MARK ENTRY EDGE FACE TREE	Yes	$O(n)$	$O(d)$	$O(\log n)$	No
|| FACE TREE	Yes	$O(n)$	–	$O(\log n)$	Yes
AN FLOODING[SH04]	No	$O(n)$	$O(1)$	$O(\log n)$	Yes
SN FLOODING	No	$O(n)$	$O(1)$	$O(\log n)$	Yes
EXTENDED AN FLOODING	No	$O(n)$	$O(1)$	$O(\log n)$	Yes
EXTENDED SN FLOODING	No	$O(n)$	$O(1)$	$O(\log n)$	Yes

In order to compare performance, we carried out experiments by simulating the protocols for large number of static network topologies. We study the effect of the number of nodes and the size of the uncertainty radius on the delivery rate, stretch factor, and the transmission cost of the protocols. We observe that flooding-based algorithms show an interesting behavior whereby the delivery rate first decreases and then increases as the uncertainty radius increases. Also, the EXTENDED SN FLOODING algorithm achieves very high delivery rate, at the same time as achieving very low stretch factor, and a drastically reduced transmission cost. Our results show that some variations of the face tree traversal approach, including the version given in [BMSU99], have as high transmission cost as the flooding-based approaches. Indeed, there is considerable overlap between the transmission cost profiles of the two approaches. In particular, the cheapest algorithms among the ones studied are EXTENDED SN FLOODING and SN FLOODING, while the two most expensive algorithms are face tree traversal-based algorithms. We note that the difference between geocasting and ROUTE-U is highlighted by the fact that DOUBLING FACE TREE algorithm, which provably improves the performance for geocasting, appears to *degrade* the performance for ROUTE-U. Finally, we conclude that if marked bits are not practical, or if guaranteed delivery is required, then DFS FACE TREE would seem to be the best approach, but otherwise, EXTENDED SN FLOODING would be best.

As shown in our results, the DFS FACE TREE algorithm outperforms the DOUBLING FACE TREE algorithm, though the worst case time complexities of the two algorithms suggest otherwise. Worst case analysis is more indicative of the time required for geocast, where every node in the uncertainty zone needs to be reached. However, in our case, we are looking for a specific node in the uncertainty zone. Therefore, we believe that an average case analysis of the algorithms would be more instructive for the ROUTE-U problem and would be a useful avenue for further investigation. Also, as we have seen in Chapter 3, the flooding approach can sometimes fail to deliver packets to the destination. An interesting question to ask here is: what is the likelihood that flooding fails? In this regard, we have seen in our results that the delivery rate of flooding-based algorithms first decreases and then increases with increase in the uncertainty radius. A theoretical analysis of this interesting behavior displayed by flooding should be useful.

In all our simulations the network nodes are generated uniformly at random, hence, our results are valid only for networks with such a distribution of nodes. Simulating the protocols for non-uniform distribution of the network nodes may give different results,

specifically, we expect a reduction in the delivery rate of flooding-based approaches. In this thesis, whenever needed, we use the Gabriel graph algorithm for planarization of the UDG. However, there exist other planar subgraphs of the UDG such as the *Relative Neighborhood Graph* for which there exist localized extraction algorithms [Tou80]. Though the worst-case complexity of all the algorithms would remain the same, irrespective of the planarization algorithms used, in practice, using a different planar subgraph might give improved results.

Bibliography

[ASSC02] I. F. Akyildiz, Weilian Su, Y. Sankarasubramaniam, and E. Cayirci. A survey on sensor networks. *IEEE Communications Magazine*, 40(8):102– 114, 2002.

[AWW05] I. F. Akyildiz, X. Wang, and W. Wang. Wireless Mesh Networks: A Survey. *Computer Networks Journal (Elsevier)*, 47(4):445–487, 2005.

[BBC+01] L. Blazevic, L. Buttyan, S. Capkun, S. Giordano, J. Hubaux, and J. Le Boudec. Self-organization in Mobile Ad Hoc Networks: The Approach of Terminodes. *IEEE Communications Magazine*, 39(6):166– 174, June 2001.

[BM76] J.A. Bondy and U.S.R. Murty. *Graph Theory with applications*. Elsevier North Holland, 1976.

[BM02] P. Bose and P. Morin. An improved algorithm for subdivision traversal without extra storage. *International Journal of Computational Geometry and Applications*, 12(4):297–308, 2002. Special issue of selected papers from the *11th Annual International Symposium on Algorithms and Computation (ISAAC 2000)*.

[BMSU99] P. Bose, P. Morin, I. Stojmenovic, and J. Urrutia. Routing with guaranteed delivery in ad hoc wireless networks. In *Proc. of 3rd ACM Int. Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications (DIALM99)*, pages 48–55, August 1999.

[Car79] Bernard Carré. *Graph and Networks*. Clarendon Press, 1979.

[CBW02] T. Camp, J. Boleng, and L. Wilcox. Location information services in mobile ad hoc networks. In *Proceedings of the IEEE International Conference on Communications (ICC '02)*, pages 3318–3324, 2002.

[CeA+03] T. Clausen, P. Jacquet (editors), C. Adjih, A. Laouiti, P. Minet, P. Muhlethaler, A. Qayyum, and L.Viennot. Optimized link state routing protocol (olsr). RFC 3626, October 2003. Network Working Group.

[CMWZ04] G. Calinescu, I. Mandoiu, P. J. Wan, and A. Z. Zelikovsky. Selecting forwarding neighbors in wireless ad hoc networks. *Mobile Networks and Applications*, 9(2):101–111, 2004.

[DAR] DARPA Home Page. http://www.darpa.mil.

[dBvOO96] Mark de Berg, Ren van Oostrum, and Mark Overmars. Simple traversal of a subdivision without extra storage. In *SCG '96: Proceedings of the twelfth annual symposium on Computational geometry*, pages 405–406, 1996.

[Fin87] G. G. Finn. Routing and addressing problems in large metropolitan-scale internetworks. In *ISI res. rep. ISU/RR-*, pages 87–180, 1987.

[FL01] James A. Freebersyser and Barry Leiner. *Ad Hoc Networking*, pages 29–51. Addison-Wesley, 2001.

[GS69] K. Gabriel and R. Sokal. A new statistical approach to geographic variation analysis. *Systematic Zoology*, 18:259–278, 1969.

[GS04] S. Giordano and I. Stojmenovic. Position based routing algorithms for ad hoc networks: A taxonomy. *Ad Hoc Wireless Networking, X. Cheng, X. Huang and D.Z. Du (eds.)*, pages 103–136, 2004.

[IET] IETF MANET Working Group Information. http://www.ietf.org/html.charters/manet-charter.html.

[JM96] David B. Johnson and David A. Maltz. Dynamic source routing in ad hoc wireless networks. In Tomasz Imielinski, editor, *Mobile Computing*, chapter 5, pages 153–181. Kluwer Academic Publishers, 1996.

[KK00] B. Karp and H.T. Kung. Greedy perimeter stateless routing for wireless networks. In *Proceedings of the Sixth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom 2000)*, pages 243–254, August 2000.

[Kle03] L. Kleinrock. An Internet vision: the invisible global infrastructure. *Ad Hoc Networks*, 1(1):3–11, July 2003.

[KSU99] E. Kranakis, H. Singh, and J. Urrutia. Compass routing on geometric networks. In *Proceedings of 11th Canadian Conference on Computational Geometry*, 1999.

[KV98] Young-Bae Ko and Nitin H. Vaidya. Location-aided routing (lar) in mobile ad hoc networks. In *MobiCom '98: Proceedings of the 4th annual ACM/IEEE international conference on Mobile computing and networking*, pages 66–75, 1998.

[KV99] Young-Bae Ko and Nitin H. Vaidya. Geocasting in mobile ad hoc networks: Location-based multicast algorithms. In *IEEE Workshop on Mobile Computing Systems and Applications (WMCSA)*, pages 101–110, 1999.

[MM04] C. Siva Ram Murthy and B. S. Manoj. *Ad Hoc Wireless Networks: Architectures and Protocols*, pages 191–225. Prentice Hall, 2004.

[Mor01] P. Morin. *Online Routing in Geometric Graphs*. PhD thesis, School of Computer Science, Carleton University, January 2001.

[PB94] Charles E. Perkins and Pravin Bhagwat. Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. In *Proceedings of the SIGCOMM 94 Conference on Communications Architectures, Protocols and Applications*, pages 234–244, August 1994.

[PR99] Charles E. Perkins and Elizabeth M. Royer. Ad hoc on-demand distance vector routing. In *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*, pages 90–100, February 1999.

[SH04] K. Saeda and A. Helmy. Efficient geocasting with perfect delivery in wireless networks. In *Proceedings of IEEE Wireless Communications and Networking*, 2004.

[SL01] Ivan Stojmenovic and Xu Lin. Loop-free hybrid single-path/flooding routing algorithms with guaranteed delivery for wireless networks. *IEEE Trans. Parallel Distrib. Syst.*, 12(10):1023–1032, 2001.

[Sto04] I. Stojmenovic. Geocasting with guaranteed delivery in sensor networks. *IEEE Wireless Communications Magazine*, 11(6):29–37, December 2004.

[Tan02] Andrew Tanenbaum. *Computer Networks*. Prentice Hall, 2002.

[Tou80] G. Toussaint. The relative neighborhood graph of a finite planar set. *Pattern Recognition*, 12(4):261–268, 1980.

[Tug69] Thomas Tugend. UCLA to be first station in nationwide computer network, UCLA Press Release. http://www.lk.cs.ucla.edu/LK/Bib/REPORT/press.html, July 1969.

[WQDT01] Hongyi Wu, Chunming Qiao, Swades De, and Ozan Tonguz. Integrated cellular and ad hoc relaying systems: iCAR. *IEEE Journal on Selected Areas in Communications*, 19(10):2105 – 2115, 2001.

Appendix A

Protocol Packet Formats and Pseudocodes

A.1 GFG Routing Packet Format and Protocol Pseudocode

GFG ROUTING **Packet format**

type ← a numeric value indicating the protocol this packet belongs to
srcId ← source address
dstId ← destination address
nextHop ← next hop address
prevHop ← previous hop address
hopCount ← number of hops the packet has traveled so far
ttl ← maximum number of hops the packet can travel
dstLoc ← destination location
distance ← distance to the destination of the last node chosen by GREEDY ROUTING
t ← the intersection point in FACE ROUTING

GFG ROUTING **Protocol**

Initialization of the packet *pkt*:
srcId ← *s* // source address
dstId ← *d* // destination address

$nextHop \leftarrow null$ // next hop address

$prevHop \leftarrow s$ // previous hop address

$hopCount \leftarrow 0$ // number of hops the packet has traveled so far

$ttl \leftarrow MAXVAL$ // maximum number of hops the packet can travel

$dstLoc \leftarrow location(d)$ // destination location

$distance \leftarrow MAXVAL$ // distance to the destination of the last node chosen by GREEDY ROUTING

$t \leftarrow null$ // the intersection point in FACE ROUTING

GFG-ROUTING(pkt, c) // c is the address of the current node

1. $doFace \leftarrow true$
2. $GG(c) \leftarrow \emptyset$ // this set stores the one-hop neighbors of c in the gabriel graph
3. $N(c) \leftarrow$ set of one-hop neighbors of c in the UDG // a node discovers its neighbors by running the neighbor discovery protocol
4. **if** $c = pkt.d$ **then**
5. $\quad deliver(pkt)$
6. $\quad$ **return**
7. **end if**
8. **if** $pkt.hopCount > pkt.ttl$ **then**
9. $\quad drop(pkt)$
10. $\quad$ **return**
11. **end if**
12. **if** $pkt.d \in N(c)$ **then**
13. $\quad pkt.nextHop \leftarrow pkt.d$
14. $\quad pkt.prevHop \leftarrow c$
15. $\quad pkt.hopCount \leftarrow pkt.hopCount + 1$
16. $\quad send(pkt)$
17. $\quad$ **return**
18. **end if**
19. **if** $distance(location(c), pkt.dstLoc) < pkt.distance$ **then**
20. $\quad pkt.distance \leftarrow distance(location(c), pkt.dstLoc)$
21. $\quad$ **for** each $u \in N(c)$ **do**
22. $\quad\quad$ **if** $distance(location(u), pkt.dstLoc) < pkt.distance$ **then**

```
23.                 pkt.distance ← distance(location(u), pkt.dstLoc)
24.                 pkt.nextHop ← u
25.                 doFace ← false
26.             end if
27.         end for
28.     end if
29.     if doFace = true then
30.         changeFace ← false
31.         if pkt.t = null then
32.             pkt.t ← location(c)
33.             changeFace ← true
34.         end if
35.         GG(c) ← GABRIEL-GRAPH(c)
36.         do
37.             if changeFace = true then
38.                 if location(c) = pkt.t then
39.                     pkt.nextHop ← w ∈ GG(c) such that [c, w] is the first clockwise
                        edge from td̄ incident on c
40.                 else
41.                     pkt.nextHop ← w ∈ GG(c) such that [c, w] is the first clockwise
                        edge from [c, pkt.nextHop] incident on c
42.                 end if
43.             else
44.                 pkt.nextHop ← w ∈ GG(c) such that [c, w] is the first clockwise
                    edge from [pkt.prevHop, c] incident on c
45.             end if
46.             if [c, pkt.nextHop] intersects td̄ at point x
                and distance(x, pkt.dstLoc) < distance(t, pkt.dstLoc) then
47.                 if d is to the right of [c, pkt.nextHop] then
48.                     changeFace ← true
49.                     pkt.t ← x
50.                 end if
51.             else
```

52. $changeFace \leftarrow false$
53. **end if**
54. **while** $changeFace = true$
55. **end if**
56. $pkt.hopCount = pkt.hopCount + 1$
57. $pkt.prevHop = c$
58. $send(pkt)$

GABRIEL-GRAPH(v)

1. $GG(v) \leftarrow \emptyset$
2. **for** each $u \in N(v)$ **do**
3. **if** $disk(u,v) \cap (N(v) \setminus \{u,v\}) = \emptyset$ **then**
4. $GG(v) \leftarrow GG(v) \cup \{u\}$
5. **end if**
6. **end for**
7. **return** $GG(v)$

A.2 Doubling Face Tree Routing Packet Format and Protocol Pseudocode

DOUBLING FACE TREE **Packet format**

$type \leftarrow$ a numeric value indicating the protocol this packet belongs to
$srcId \leftarrow$ source address
$dstId \leftarrow$ destination address
$nextHop \leftarrow$ next hop address
$prevHop \leftarrow$ previous hop address
$hopCount \leftarrow$ number of hops the packet has travelled so far
$ttl \leftarrow$ maximum number of hops the packet can travel
$dstLoc \leftarrow$ destination location, this is the center of the uncertainty zone
$radius \leftarrow$ radius of the uncertainty zone
$s' \leftarrow$ reference point for the face tree

$e_{start} \leftarrow$ start edge

$e \leftarrow$ current edge

$ent \leftarrow$ entry edge for the face

$lastNode \leftarrow$ used for doubling entry edge computation

$startNode \leftarrow$ used for doubling entry edge computation

$k \leftarrow$ used for doubling entry edge computation

$j \leftarrow$ used for doubling entry edge computation

$isentry \leftarrow$ used for doubling entry edge computation

$direction \leftarrow$ used for doubling entry edge computation

$thisface \leftarrow$ used for doubling entry edge computation

$stop \leftarrow$ used for doubling entry edge computation

$l \leftarrow$ used for doubling entry edge computation

DOUBLING FACE TREE **Protocol**

Initialization of the packet pkt:

$srcId \leftarrow s$ // source address

$dstId \leftarrow d$ // destination address

$nextHop \leftarrow null$ // next hop address

$prevHop \leftarrow null$ // previous hop address

$hopCount \leftarrow 0$ // number of hops the packet has traveled so far

$ttl \leftarrow MAXVAL$ // maximum number of hops the packet can travel

$dstLoc \leftarrow p$ // the old location of destination, this is the center of the uncertainty zone

$radius \leftarrow$ radius of the uncertainty zone

$e_{start} \leftarrow [q, w]$ where q is the entry node and $w \in$ GABRIEL-GRAPH(q) such that $[q, w]$ is the first clockwise edge from $[q, pkt.dstLoc]$ incident on q

$e \leftarrow e_{start}$

$ent \leftarrow null$

$s' \leftarrow$ a point close to the mid of e_{start} on the left side

$lastNode \leftarrow null$

$startNode \leftarrow null$

$k \leftarrow 0$

$j \leftarrow 0$

$isentry \leftarrow -1$

$direction \leftarrow true$

$thisface \leftarrow true$

$stop \leftarrow false$

$l \leftarrow null$

DOUBLING-FACE-TREE-ROUTING(pkt, c) // c is the address of the current node

1. $change \leftarrow false$
2. $GG(c) \leftarrow \emptyset$ // this set stores the one-hop neighbors of c in the gabriel graph
3. $N(c) \leftarrow$ set of one-hop neighbors of c in the UDG // a node discovers its neighbors by running the neighbor discovery protocol
4. **if** $c = pkt.d$ **then**
5. $\quad deliver(pkt)$
6. $\quad$ **return**
7. **end if**
8. **if** $pkt.hopCount > pkt.ttl$ **then**
9. $\quad drop(pkt)$
10. $\quad$ **return**
11. **end if**
12. **if** $pkt.d \in N(c)$ **then**
13. $\quad pkt.nextHop \leftarrow pkt.d$
14. $\quad pkt.prevHop \leftarrow c$
15. $\quad pkt.hopCount \leftarrow pkt.hopCount + 1$
16. $\quad send(pkt)$
17. $\quad$ **return**
18. **end if**
19. $GG(c) \leftarrow$ GABRIEL-GRAPH(c)
20. **do**
21. $\quad change \leftarrow false$
22. $\quad$ **if** $pkt.nextHop = null$ **then**
23. $\quad\quad pkt.nextHop \leftarrow w$ such that $e_{start} = [c, w]$
24. $\quad$ **else**

25. $pkt.nextHop \leftarrow w \in GG(c)$ such that $[c,w]$ is the first clockwise edge from $[pkt.prevHop, c]$ incident on c
26. **if** $pkt.startNode \neq null$ **then**
27. $isentry(pkt, true)$
28. **return**
29. **else**
30. $pkt.e \leftarrow [c, pkt.nextHop]$
31. **if** $pkt.e = pkt.e_{start}$ **then**
32. $drop(pkt)$
33. **return**
34. **end if**
35. **end if**
36. **end if**
37. **if** $intersects([c, pkt.nextHop], pkt.dstLoc, pkt.radius)$ **then**
38. **if** s' not contained in the face with $pkt.e$ on its boundary **and** $((pkt.ent \neq null$ **and** $pkt.e = pkt.ent)$ **or** $(pkt.ent = null$ **and** $isentry(pkt, true)))$ **and** $pkt.stop = false$ **then**
39. $pkt.e \leftarrow [pkt.nextHop, c]$
40. $pkt.prevHop \leftarrow pkt.nextHop$
41. $change \leftarrow true$
42. $pkt.ent \leftarrow null$
43. **else if** s' not contained in the face with $pkt.e$ on its boundary **and** $pkt.stop = false$ **and** $isentry(pkt, false)$ **then**
44. $pkt.e \leftarrow [pkt.nextHop, c]$
45. $pkt.prevHop \leftarrow pkt.nextHop$
46. $change \leftarrow true$
47. $pkt.ent \leftarrow pkt.e$
48. **end if**
49. **if** $pkt.stop = true$ **then**
50. **return**
51. **end if**
52. **end if**
53. **while** $changeFace = true$

54. $pkt.hopCount = pkt.hopCount + 1$
55. $pkt.prevHop = c$
56. $send(pkt)$

ISENTRY$(pkt, thisface)$

1. $change \leftarrow false$
2. **do**
3. $change \leftarrow false$
4. **if** $pkt.startNode \neq null$ **and** $pkt.startNode = c$ **then**
5. **if** $pkt.isentry = 0$ **then**
6. $pkt.startNode \leftarrow null$
7. $pkt.nextHop \leftarrow w$ such that $pkt.e = [c, w]$
8. $pkt.prevHop \leftarrow w \in GG(c)$ such that $[c, w]$ is the first counterclockwise edge from $[c, pkt.nextHop]$ incident on c
9. **return** false
10. **else if** $pkt.isentry = 1$ **then**
11. $pkt.startNode \leftarrow null$
12. $pkt.nextHop \leftarrow w$ such that $pkt.e = [c, w]$
13. $pkt.prevHop \leftarrow w \in GG(c)$ such that $[c, w]$ is the first counterclockwise edge from $[c, pkt.nextHop]$ incident on c
14. **return** true
15. **end if**
16. $pkt.k \leftarrow 2 * pkt.k$
17. **if** $pkt.thisface = true$ **then**
18. **if** $pkt.direction = true$ **then**
19. $pkt.j \leftarrow -1$
20. **else**
21. $pkt.j \leftarrow 0$
22. **end if**
23. **else**
24. **if** $pkt.direction = true$ **then**
25. $pkt.j \leftarrow 0$
26. **else**

27. $pkt.j \leftarrow -1$
28. **end if**
29. **end if**
30. **else if** $pkt.startNode = null$ **then**
31. **if** $|GG(c)| = 1$ **then**
32. **return** false
33. **end if**
34. $pkt.lastNode \leftarrow null$
35. $pkt.isentry \leftarrow -1$
36. $pkt.direction \leftarrow true$
37. $pkt.startNode \leftarrow c$
38. **if** $thisface \leftarrow true$
39. $pkt.l \leftarrow pkt.e$
40. $pkt.k \leftarrow 1$
41. $pkt.j \leftarrow -1$
42. $pkt.prevHop \leftarrow w \in GG(c)$ such that $[c,w]$ is the first counterclockwise edge from $[c, pkt.nextHop]$ incident on c
43. $pkt.thisface \leftarrow true$
44. **else**
45. $pkt.l \leftarrow [x,w]$ such that $[w,x] = pkt.e$
46. $pkt.k \leftarrow 1$
47. $pkt.j \leftarrow 0$
48. $pkt.prevHop \leftarrow x$ such that $[c,x] = pkt.e$
49. $pkt.thisface \leftarrow false$
50. **end if**
51. **end if**
52. **if** $pkt.direction = true$ **then**
53. $pkt.nextHop \leftarrow w \in GG(c)$ such that $[c,w]$ is the first clockwise edge from $[c, pkt.nextHop]$ incident on c
54. **else**
55. $pkt.nextHop \leftarrow w \in GG(c)$ such that $[c,w]$ is the first counterclockwise edge from $[c, pkt.nextHop]$ incident on c
56. **end if**

```
57.     if pkt.j ≠ −100 then
58.         pkt.j ← pkt.j + 1
59.         if pkt.direction ← true then
60.             tempe ← [c, pkt.nextHop]
61.         else
62.             tempe ← [pkt.nextHop, c]
63.         end if
64.         if key_s'(tempe) ≺ key_s'(pkt.l) then
65.             pkt.isentry ← 0
66.             if pkt.startNode = c then
67.                 pkt.startNode ← null
68.                 pkt.nextHop ← w such that [c, w] = pkt.e
69.                 pkt.prevHop ← w ∈ GG(c) such that [c, w] is the first
                    counterclockwise edge from [c, pkt.nextHop] incident on c
70.                 return false
71.             end if
72.             t ← pkt.nextHop
73.             pkt.nextHop ← pkt.prevHop
74.             pkt.prevHop ← t
75.             pkt.direction ←!pkt.direction
76.             pkt.j ← −100
77.         else if pkt.lastNode = pkt.nextHop then
78.             pkt.isentry ← 1
79.             if pkt.startNode = c then
80.                 pkt.startNode ← null
81.                 pkt.nextHop ← w such that [c, w] = pkt.e
82.                 pkt.prevHop ← w ∈ GG(c) such that [c, w] is the first counterclockwise
                edge from [c, pkt.nextHop] incident on c
83.                 return false
84.             end if
85.             t ← pkt.nextHop
86.             pkt.nextHop ← pkt.prevHop
87.             pkt.prevHop ← t
```

```
88.                 pkt.direction ← !pkt.direction
89.                 pkt.j ← −100
90.             else if pkt.j = pkt.k then
91.                 pkt.lastNode ← pkt.nextHop
92.                 t ← pkt.nextHop
93.                 pkt.nextHop ← pkt.prevHop
94.                 pkt.prevHop ← t
95.                 pkt.direction ← !pkt.direction
96.                 pkt.j ← −100
97.                 if pkt.startNode = c then
98.                     change ← true
99.                 end if
100.            end if
101.        while changeFace = true
102.        pkt.hopCount = pkt.hopCount + 1
103.        pkt.prevHop = c
104.        send(pkt)
```

GABRIEL-GRAPH(v)

```
1.  GG(v) ← ∅
2.  for each u ∈ N(v) do
3.      if disk(u, v) ∩ (N(v)\{u, v}) = ∅ then
4.          GG(v) ← GG(v) ∪ {u}
5.      end if
6.  end for
7.  return GG(v)
```

A.3 DFS Face Tree Routing Packet Format and Protocol Pseudocode

DFS FACE TREE **Packet format**

$type \leftarrow$ a numeric value indicating the protocol this packet belongs to

$srcId \leftarrow$ source address

$dstId \leftarrow$ destination address

$nextHop \leftarrow$ next hop address

$prevHop \leftarrow$ previous hop address

$hopCount \leftarrow$ number of hops the packet has travelled so far

$ttl \leftarrow$ maximum number of hops the packet can travel

$dstLoc \leftarrow$ destination location, this is the center of the uncertainty zone

$radius \leftarrow$ radius of the uncertainty zone

$s' \leftarrow$ reference point for the face tree

$e_{start} \leftarrow$ start edge

$e \leftarrow$ current edge

$e' \leftarrow$ starting edge for traversal of face during entry edge computation

$ent \leftarrow$ entry edge for the face

$oent \leftarrow$ entry edge for the opposite face

$mode \leftarrow$ used during entry edge computation

DFS FACE TREE **Protocol**

Initialization of the packet pkt:

$srcId \leftarrow s$ // source address

$dstId \leftarrow d$ // destination address

$nextHop \leftarrow null$ // next hop address

$prevHop \leftarrow null$ // previous hop address

$hopCount \leftarrow 0$ // number of hops the packet has traveled so far

$ttl \leftarrow MAXVAL$ // maximum number of hops the packet can travel

$dstLoc \leftarrow p$ // the old location of destination, this is the center of the uncertainty zone

$radius \leftarrow$ radius of the uncertainty zone

$e_{start} \leftarrow [q, w]$ where q is the entry node and $w \in$ GABRIEL-GRAPH(q) such that $[q, w]$ is the first clockwise edge from $[q, pkt.dstLoc]$ incident on q

$e \leftarrow e_{start}$

$e' \leftarrow null$

$ent \leftarrow null$

$oent \leftarrow null$

$mode \leftarrow 0$

$s' \leftarrow$ a point close to the mid of e_{start} on the left side

DFS-FACE-TREE-ROUTING(pkt, c) // c is the address of the current node

1. $change \leftarrow false$
2. $GG(c) \leftarrow \emptyset$ // this set stores the one-hop neighbors of c in the gabriel graph
3. $N(c) \leftarrow$ set of one-hop neighbors of c in the UDG // a node discovers its neighbors by running the neighbor discovery protocol
4. **if** $c = pkt.d$ **then**
5. $\quad deliver(pkt)$
6. $\quad$ **return**
7. **end if**
8. **if** $pkt.hopCount > pkt.ttl$ **then**
9. $\quad drop(pkt)$
10. $\quad$ **return**
11. **end if**
12. **if** $pkt.d \in N(c)$ **then**
13. $\quad pkt.nextHop \leftarrow pkt.d$
14. $\quad pkt.prevHop \leftarrow c$
15. $\quad pkt.hopCount \leftarrow pkt.hopCount + 1$
16. $\quad send(pkt)$
17. $\quad$ **return**
18. **end if**
19. $GG(c) \leftarrow$ GABRIEL-GRAPH(c)
20. **do**
21. $\quad change \leftarrow false$
22. $\quad$ **if** $pkt.nextHop = null$ **then**
23. $\quad\quad pkt.nextHop \leftarrow w$ such that $e_{start} = [c, w]$
24. $\quad$ **else**
25. $\quad\quad pkt.nextHop \leftarrow w \in GG(c)$ such that $[c, w]$ is the first clockwise edge from $[pkt.prevHop, c]$ incident on c

```
26.         if pkt.mode = 0 then
27.             pkt.e ← [c, pkt.nextHop]
28.             if pkt.ent ≠ null and pkt.oent ≠ null and e = e_start then
29.                 drop(pkt)
30.                 return
31.             end if
32.             if intersects([c, pkt.nextHop], pkt.dstLoc, pkt.radius) and
                |GG(c)| > 1 then
33.                 pkt.oent ← null
34.             end if
35.         end if
36.     end if
37.     if pkt.ent = null and pkt.mode = 0 then
38.         pkt.mode ← 1
39.         e' ← [c, pkt.nextHop]
40.         pkt.ent ← [c, pkt.nextHop]
41.     else
42.         if pkt.oent = null and pkt.mode = 0 then
43.             pkt.mode ← −1
44.             pkt.prevHop ← pkt.nextHop
45.             e' ← [pkt.prevHop, c]
46.             pkt.oent ← [pkt.prevHop, c]
47.             pkt.nextHop ← w ∈ GG(c) such that [c, w] is the first clockwise
                edge from [pkt.prevHop, c] incident on c
48.         else if pkt.e' ≠ null then
49.             if (pkt.mode = 1 and pkt.e' = [c, pkt.nextHop]) or
                (pkt.mode = −1 and pkt.e' = [pkt.prevHop, c]) then
50.                 pkt.e' ← null
51.                 pkt.mode ← 0
52.                 pkt.nextHop ← w such that pkt.e = [x, w]
53.                 if pkt.oent = null then
54.                     change ← true
55.                     continue
```

```
56.                 end if
57.             end if
58.         else if pkt.mode = 1 then
59.             if key_{s'}([c, pkt.nextHop]) ≺ key_{s'}(pkt.ent) then
60.                 pkt.ent ← [c, pkt.nextHop]
61.             end if
62.         else if pkt.mode = −1 then
63.             if key_{s'}([pkt.prevtHop, c]) ≺ key_{s'}(pkt.oent) then
64.                 pkt.oent ← [pkt.prevHop, c]
65.             end if
66.         end if
67.     end if
68.     if pkt.e' = null then
69.         if intersects([c, pkt.nextHop], pkt.dstLoc, pkt.radius) then
70.             if s' not contained in the face with pkt.e on its boundary
                and pkt.e = pkt.ent then
71.                 pkt.e ← [pkt.nextHop, c]
72.                 pkt.prevHop ← pkt.nextHop
73.                 change ← true
74.                 temp ← pkt.ent
75.                 pkt.ent ← pkt.oent
76.                 pkt.oent ← temp
77.             end if
78.             else if s' not contained in the face with pkt.e on its boundary
                and [x, w] = pkt.oent such that pkt.e = [w, x] then
79.                 pkt.e ← [pkt.nextHop, c]
80.                 pkt.prevHop ← pkt.nextHop
81.                 change ← true
82.                 temp ← pkt.ent
83.                 pkt.ent ← pkt.oent
84.                 pkt.oent ← temp
85.             end if
86.         end if
```

```
87.         end if
88.   while changeFace = true
89.   pkt.hopCount = pkt.hopCount + 1
90.   pkt.prevHop = c
91.   send(pkt)
```

GABRIEL-GRAPH(v)

```
1.   GG(v) ← ∅
2.   for each u ∈ N(v) do
3.         if disk(u,v) ∩ (N(v)\{u,v}) = ∅ then
4.               GG(v) ← GG(v) ∪ {u}
5.         end if
6.   end for
7.   return GG(v)
```

A.4 Mark Entry Edge Face Tree Routing Packet Format and Protocol Pseudocode

MARK ENTRY EDGE FACE TREE **Packet format**

$type \leftarrow$ a numeric value indicating the protocol this packet belongs to
$srcId \leftarrow$ source address
$dstId \leftarrow$ destination address
$nextHop \leftarrow$ next hop address
$prevHop \leftarrow$ previous hop address
$hopCount \leftarrow$ number of hops the packet has travelled so far
$ttl \leftarrow$ maximum number of hops the packet can travel
$dstLoc \leftarrow$ destination location, this is the center of the uncertainty zone
$radius \leftarrow$ radius of the uncertainty zone
$s' \leftarrow$ reference point for the face tree
$e_{start} \leftarrow$ start edge
$e \leftarrow$ current edge

$e' \leftarrow$ starting edge for traversal of face during entry edge computation

$ent \leftarrow$ entry edge for the face

$oent \leftarrow$ entry edge for the opposite face

$mode \leftarrow$ used during entry edge computation

MARK ENTRY EDGE FACE TREE **Protocol**

Initialization of the packet pkt:

$srcId \leftarrow s$ // source address

$dstId \leftarrow d$ // destination address

$nextHop \leftarrow null$ // next hop address

$prevHop \leftarrow null$ // previous hop address

$hopCount \leftarrow 0$ // number of hops the packet has traveled so far

$ttl \leftarrow MAXVAL$ // maximum number of hops the packet can travel

$dstLoc \leftarrow p$ // the old location of destination, this is the center of the uncertainty zone

$radius \leftarrow$ radius of the uncertainty zone

$e_{start} \leftarrow [q, w]$ where q is the entry node and $w \in$ GABRIEL-GRAPH(q) such that $[q, w]$ is the first clockwise edge from $[q, pkt.dstLoc]$ incident on q

$e \leftarrow e_{start}$

$e' \leftarrow null$

$ent \leftarrow null$

$oent \leftarrow null$

$mode \leftarrow 0$

$s' \leftarrow$ a point close to the mid of e_{start} on the left side

MARK-ENTRY-EDGE-FACE-TREE-ROUTING(pkt, c) // c is the address of the current node

1. $change \leftarrow false$
2. $GG(c) \leftarrow \emptyset$ // this set stores the one-hop neighbors of c in the gabriel graph
3. $N(c) \leftarrow$ set of one-hop neighbors of c in the UDG // a node discovers its neighbors by running the neighbor discovery protocol
4. $marked[v] \leftarrow null \; \forall \; v \in GG(c)$
4. **if** $c = pkt.d$ **then**

5. $deliver(pkt)$
6. **return**
7. **end if**
8. **if** $pkt.hopCount > pkt.ttl$ **then**
9. $drop(pkt)$
10. **return**
11. **end if**
12. **if** $pkt.d \in N(c)$ **then**
13. $pkt.nextHop \leftarrow pkt.d$
14. $pkt.prevHop \leftarrow c$
15. $pkt.hopCount \leftarrow pkt.hopCount + 1$
16. $send(pkt)$
17. **return**
18. **end if**
19. $GG(c) \leftarrow$ GABRIEL-GRAPH(c)
20. **do**
21. $change \leftarrow false$
22. **if** $pkt.nextHop = null$ **then**
23. $pkt.nextHop \leftarrow w$ such that $e_{start} = [c, w]$
24. **else**
25. $pkt.nextHop \leftarrow w \in GG(c)$ such that $[c, w]$ is the first clockwise edge from $[pkt.prevHop, c]$ incident on c
26. **if** $pkt.mode = 0$ **then**
27. $pkt.e \leftarrow [c, pkt.nextHop]$
28. **if** $pkt.ent \neq null$ **and** $pkt.oent \neq null$ **and** $e = e_{start}$ **then**
29. $drop(pkt)$
30. **return**
31. **end if**
32. **if** $intersects([c, pkt.nextHop], pkt.dstLoc, pkt.radius)$ **and** $|GG(c)| > 1$ **then**
33. $pkt.oent \leftarrow null$
34. **end if**
35. **end if**

```
36.     end if
37.     if marked[[c, pkt.nextHop]] = null and pkt.ent = null and pkt.mode = 0 then
38.         pkt.mode ← 1
39.         e' ← [c, pkt.nextHop]
40.         pkt.ent ← [c, pkt.nextHop]
41.     else
42.         if marked[[c, pkt.nextHop]] = null and pkt.oent = null and
            pkt.mode = 0 then
43.             pkt.mode ← −1
44.             pkt.prevHop ← pkt.nextHop
45.             e' ← [pkt.prevHop, c]
46.             pkt.oent ← [pkt.prevHop, c]
47.             pkt.nextHop ← w ∈ GG(c) such that [c, w] is the first clockwise
                edge from [pkt.prevHop, c] incident on c
48.         else if pkt.e' ≠ null then
49.             if (pkt.mode = 1 and pkt.e' = [c, pkt.nextHop]) or
                (pkt.mode = −1 and pkt.e' = [pkt.prevHop, c]) then
50.                 if pkt.mode = 1 then
51.                     pkt.mode ← 2
52.                 else
53.                     pkt.mode ← −2
54.                 end if
55.                 if pkt.ent ≠ null then
56.                     if pkt.ent = [c, pkt.nextHop] then
55.                         marked[[c, pkt.nextHop]] ← true
56.                     else
55.                         marked[[c, pkt.nextHop]] ← false
56.                     end if
55.                 end if
55.                 if pkt.oent ≠ null then
56.                     if pkt.oent = [c, pkt.nextHop] then
55.                         marked[[c, pkt.nextHop]] ← true
56.                     else
```

```
55.                         marked[[c, pkt.nextHop]] ← false
56.                     end if
55.                 end if
49.             else if pkt.mode = 2 and pkt.e' = [c, pkt.nextHop]) or
                (pkt.mode = −2 and pkt.e' = [pkt.prevHop, c]) then
50.                 pkt.e' ← null
51.                 pkt.mode ← 0
52.                 pkt.nextHop ← w such that pkt.e = [x, w]
53.                 if pkt.oent = null then
54.                     change ← true
55.                     continue
56.                 end if
49.             else if pkt.mode = 2 then
55.                 if pkt.ent ≠ null then
56.                     if pkt.ent = [c, pkt.nextHop] then
55.                         marked[[c, pkt.nextHop]] ← true
56.                     else
55.                         marked[[c, pkt.nextHop]] ← false
56.                     end if
55.                 end if
49.             else if pkt.mode = −2 then
55.                 if pkt.oent ≠ null then
56.                     if pkt.oent = [c, pkt.nextHop] then
55.                         marked[[c, pkt.nextHop]] ← true
56.                     else
55.                         marked[[c, pkt.nextHop]] ← false
56.                     end if
55.                 end if
58.             else if pkt.mode = 1 then
59.                 if key_s'([c, pkt.nextHop]) ≺ key_s'(pkt.ent) then
60.                     pkt.ent ← [c, pkt.nextHop]
61.                 end if
62.             else if pkt.mode = −1 then
```

```
63.                     if key_{s'}([pkt.prevtHop, c]) ≺ key_{s'}(pkt.oent) then
64.                         pkt.oent ← [pkt.prevHop, c]
65.                     end if
66.                 end if
67.             end if
67.         end if
68.         if pkt.e' = null then
69.             if intersects([c, pkt.nextHop], pkt.dstLoc, pkt.radius) then
70.                 if s' not contained in the face with pkt.e on its boundary
                    and marked[pkt.e] = true then
71.                     pkt.e ← [pkt.nextHop, c]
72.                     pkt.prevHop ← pkt.nextHop
73.                     change ← true
74.                     temp ← pkt.ent
75.                     pkt.ent ← pkt.oent
76.                     pkt.oent ← temp
77.                 end if
78.                 else if s' not contained in the face with pkt.e on its boundary
                    and marked[[x, w]] = true such that pkt.e = [w, x] then
79.                     pkt.e ← [pkt.nextHop, c]
80.                     pkt.prevHop ← pkt.nextHop
81.                     change ← true
82.                     temp ← pkt.ent
83.                     pkt.ent ← pkt.oent
84.                     pkt.oent ← temp
85.                 end if
86.             end if
87.         end if
88.     while changeFace = true
89.     pkt.hopCount = pkt.hopCount + 1
90.     pkt.prevHop = c
91.     send(pkt)
```

GABRIEL-GRAPH(v)

1. $GG(v) \leftarrow \emptyset$
2. **for** each $u \in N(v)$ **do**
3. **if** $disk(u,v) \cap (N(v) \setminus \{u,v\}) = \emptyset$ **then**
4. $GG(v) \leftarrow GG(v) \cup \{u\}$
5. **end if**
6. **end for**
7. **return** $GG(v)$

A.5 BFS Face Tree Routing Packet Format and Protocol Pseudocode

BFS FACE TREE **Packet format**

type ← a numeric value indicating the protocol this packet belongs to
srcId ← source address
dstId ← destination address
nextHop ← next hop address
prevHop ← previous hop address
hopCount ← number of hops the packet has travelled so far
ttl ← maximum number of hops the packet can travel
dstLoc ← destination location, this is the center of the uncertainty zone
radius ← radius of the uncertainty zone
s' ← reference point for the face tree
e_{start} ← start edge
e ← current edge
e' ← starting edge for traversal of face during entry edge computation
ent ← entry edge for the face
oent ← entry edge for the opposite face
mode ← used during entry edge computation
Q ← bfs queue, maintains a queue of entry edges to be visited
S ← a set of entry edges

BFS FACE TREE **Protocol**

Initialization of the packet pkt:

$srcId \leftarrow s$ // source address

$dstId \leftarrow d$ // destination address

$nextHop \leftarrow null$ // next hop address

$prevHop \leftarrow null$ // previous hop address

$hopCount \leftarrow 0$ // number of hops the packet has traveled so far

$ttl \leftarrow MAXVAL$ // maximum number of hops the packet can travel

$dstLoc \leftarrow p$ // the old location of destination, this is the center of the uncertainty zone

$radius \leftarrow$ radius of the uncertainty zone

$e_{start} \leftarrow [q, w]$ where q is the entry node and $w \in$ GABRIEL-GRAPH(q) such that $[q, w]$ is the first clockwise edge from $[q, pkt.dstLoc]$ incident on q

$e \leftarrow e_{start}$

$e' \leftarrow null$

$ent \leftarrow null$

$oent \leftarrow null$

$mode \leftarrow 0$

$s' \leftarrow$ a point close to the mid of e_{start} on the left side

$Q \leftarrow \emptyset$

$S \leftarrow \emptyset$

BFS-FACE-TREE-ROUTING(pkt, c) // c is the address of the current node

1. $change \leftarrow false$
2. $GG(c) \leftarrow \emptyset$ // this set stores the one-hop neighbors of c in the gabriel graph
3. $N(c) \leftarrow$ set of one-hop neighbors of c in the UDG // a node discovers its neighbors by running the neighbor discovery protocol
4. **if** $c = pkt.d$ **then**
5. $\quad deliver(pkt)$
6. $\quad$ **return**
7. **end if**

8. **if** $pkt.hopCount > pkt.ttl$ **then**
9. $drop(pkt)$
10. **return**
11. **end if**
12. **if** $pkt.d \in N(c)$ **then**
13. $pkt.nextHop \leftarrow pkt.d$
14. $pkt.prevHop \leftarrow c$
15. $pkt.hopCount \leftarrow pkt.hopCount + 1$
16. $send(pkt)$
17. **return**
18. **end if**
19. $GG(c) \leftarrow$ GABRIEL-GRAPH(c)
20. **do**
21. $change \leftarrow false$
22. **if** $pkt.nextHop = null$ **then**
23. $pkt.nextHop \leftarrow w$ such that $e_{start} = [c, w]$
24. **else**
25. $pkt.nextHop \leftarrow w \in GG(c)$ such that $[c, w]$ is the first clockwise edge from $[pkt.prevHop, c]$ incident on c
26. **if** $pkt.mode = 0$ **then**
27. $pkt.e \leftarrow [c, pkt.nextHop]$
28. **if** $intersects([c, pkt.nextHop], pkt.dstLoc, pkt.radius)$ **and** $|GG(c)| > 1$ **then**
29. $pkt.oent \leftarrow null$
30. **end if**
31. **end if**
32. **end if**
33. **if** $pkt.ent = null$ **and** $pkt.mode = 0$ **then**
34. $pkt.mode \leftarrow 1$
35. $e' \leftarrow [c, pkt.nextHop]$
36. $pkt.ent \leftarrow [c, pkt.nextHop]$
37. **else**
38. **if** $pkt.oent = null$ **and** $pkt.mode = 0$ **then**

39. $pkt.mode \leftarrow -1$
40. $pkt.prevHop \leftarrow pkt.nextHop$
41. $e' \leftarrow [pkt.prevHop, c]$
42. $pkt.oent \leftarrow [pkt.prevHop, c]$
43. $pkt.nextHop \leftarrow w \in GG(c)$ such that $[c, w]$ is the first clockwise edge from $[pkt.prevHop, c]$ incident on c
44. **else if** $pkt.e' \neq null$ **then**
45. **if** $(pkt.mode = 1$ **and** $pkt.e' = [c, pkt.nextHop])$ **or** $(pkt.mode = -1$ **and** $pkt.e' = [pkt.prevHop, c])$ **then**
46. $pkt.e' \leftarrow null$
47. $pkt.mode \leftarrow 0$
48. $pkt.nextHop \leftarrow w$ such that $pkt.e = [x, w]$
49. **if** $pkt.oent = null$ **then**
50. $change \leftarrow true$
51. **continue**
52. **end if**
53. **end if**
54. **else if** $pkt.mode = 1$ **then**
55. **if** $key_{s'}([c, pkt.nextHop]) \prec key_{s'}(pkt.ent)$ **then**
56. $pkt.ent \leftarrow [c, pkt.nextHop]$
57. **end if**
58. **else if** $pkt.mode = -1$ **then**
59. **if** $key_{s'}([pkt.prevtHop, c]) \prec key_{s'}(pkt.oent)$ **then**
60. $pkt.oent \leftarrow [pkt.prevHop, c]$
61. **end if**
62. **end if**
63. **end if**
64. **if** $pkt.e' = null$ **then**
65. **if** $intersects([c, pkt.nextHop], pkt.dstLoc, pkt.radius)$ **then**
66. **if** $pkt.Q \neq \emptyset$ **and** $e = first(pkt.Q)$ **then**
67. $enqueue(pkt.Q, dequeue(pkt.Q))$
68. $pkt.e \leftarrow [pkt.nextHop, c]$
69. $pkt.prevHop \leftarrow pkt.nextHop$

```
70.                     change ← true
71.                     temp ← pkt.ent
72.                     pkt.ent ← pkt.oent
73.                     pkt.oent ← temp
74.                 end if
75.                 if s' not contained in the face with pkt.e on its boundary
                    and pkt.e = pkt.ent then
76.                     if pkt.e = last(pkt.Q) then
77.                         pkt.S ← pkt.S ∪ {dequeue(pkt.Q)}
78.                         if pkt.Q = ∅ then
79.                             drop(pkt)
80.                             return
81.                         end if
82.                     end if
83.                     pkt.e ← [pkt.nextHop, c]
84.                     pkt.prevHop ← pkt.nextHop
85.                     change ← true
86.                     temp ← pkt.ent
87.                     pkt.ent ← pkt.oent
88.                     pkt.oent ← temp
89.                 end if
90.                 else if s' not contained in the face with pkt.e on its boundary
                    and [x, w] = pkt.oent such that pkt.e = [w, x] then
91.                     if pkt.e ∉ S then
92.                         enqueue(pkt.Q, pkt.e)
93.                     end if
94.                 end if
95.             end if
96.         end if
97.  while changeFace = true
98.  pkt.hopCount = pkt.hopCount + 1
99.  pkt.prevHop = c
100. send(pkt)
```

GABRIEL-GRAPH(v)

1. $GG(v) \leftarrow \emptyset$
2. **for** each $u \in N(v)$ **do**
3. **if** $disk(u,v) \cap (N(v) \setminus \{u,v\}) = \emptyset$ **then**
4. $GG(v) \leftarrow GG(v) \cup \{u\}$
5. **end if**
6. **end for**
7. **return** $GG(v)$

A.6 || Face Tree Routing Packet Format and Protocol Pseudocode

|| FACE TREE **Packet format**

type ← a numeric value indicating the protocol this packet belongs to
srcId ← source address
dstId ← destination address
nextHop ← next hop address
prevHop ← previous hop address
hopCount ← number of hops the packet has travelled so far
ttl ← maximum number of hops the packet can travel
dstLoc ← destination location, this is the center of the uncertainty zone
radius ← radius of the uncertainty zone
s' ← reference point for the face tree
e_{start} ← start edge
e ← current edge
e' ← starting edge for traversal of face during entry edge computation
ent ← entry edge for the face
oent ← entry edge for the opposite face
mode ← used during entry edge computation

|| FACE TREE **Protocol**

Initialization of the packet *pkt*:

srcId ← *s* // source address

dstId ← *d* // destination address

nextHop ← *null* // next hop address

prevHop ← *null* // previous hop address

hopCount ← 0 // number of hops the packet has traveled so far

ttl ← *MAXVAL* // maximum number of hops the packet can travel

dstLoc ← *p* // the old location of destination, this is the center of the uncertainty zone

radius ← radius of the uncertainty zone

e_{start} ← $[q, w]$ where q is the entry node and $w \in$ GABRIEL-GRAPH(q) such that $[q, w]$ is the first clockwise edge from $[q, pkt.dstLoc]$ incident on q

$e \leftarrow e_{start}$

$e' \leftarrow null$

ent ← *null*

oent ← *null*

mode ← 0

s' ← a point close to the mid of e_{start} on the left side

||-FACE-TREE-ROUTING(*pkt*, *c*) // *c* is the address of the current node

1. *change* ← *false*
2. $GG(c) \leftarrow \emptyset$ // this set stores the one-hop neighbors of c in the gabriel graph
3. $N(c)$ ← set of one-hop neighbors of c in the UDG // a node discovers its neighbors by running the neighbor discovery protocol
4. **if** $c = pkt.d$ **then**
5. $deliver(pkt)$
6. **return**
7. **end if**
8. **if** $pkt.hopCount > pkt.ttl$ **then**
9. $drop(pkt)$
10. **return**

```
11. end if
12. if pkt.d ∈ N(c) then
13.     pkt.nextHop ← pkt.d
14.     pkt.prevHop ← c
15.     pkt.hopCount ← pkt.hopCount + 1
16.     send(pkt)
17.     return
18. end if
19. GG(c) ← GABRIEL-GRAPH(c)
20. do
21.     change ← false
22.     if pkt.nextHop = null then
23.         pkt.nextHop ← w such that e_start = [c, w]
24.     else
25.         pkt.nextHop ← w ∈ GG(c) such that [c, w] is the first clockwise
            edge from [pkt.prevHop, c] incident on c
26.         if pkt.mode = 0 then
27.             pkt.e ← [c, pkt.nextHop]
28.             if pkt.ent ≠ null and pkt.oent ≠ null and e = e_start then
29.                 drop(pkt)
30.                 return
31.             end if
32.             if intersects([c, pkt.nextHop], pkt.dstLoc, pkt.radius) and
                |GG(c)| > 1 then
33.                 pkt.oent ← null
34.             end if
35.         end if
36.     end if
37.     if pkt.ent = null and pkt.mode = 0 then
38.         pkt.mode ← 1
39.         e' ← [c, pkt.nextHop]
40.         pkt.ent ← [c, pkt.nextHop]
41.     else
```

42. **if** $pkt.oent = null$ **and** $pkt.mode = 0$ **then**
43. $pkt.mode \leftarrow -1$
44. $pkt.prevHop \leftarrow pkt.nextHop$
45. $e' \leftarrow [pkt.prevHop, c]$
46. $pkt.oent \leftarrow [pkt.prevHop, c]$
47. $pkt.nextHop \leftarrow w \in GG(c)$ such that $[c, w]$ is the first clockwise edge from $[pkt.prevHop, c]$ incident on c
48. **else if** $pkt.e' \neq null$ **then**
49. **if** $(pkt.mode = 1$ **and** $pkt.e' = [c, pkt.nextHop])$ **or** $(pkt.mode = -1$ **and** $pkt.e' = [pkt.prevHop, c])$ **then**
50. $pkt.e' \leftarrow null$
51. $pkt.mode \leftarrow 0$
52. $pkt.nextHop \leftarrow w$ such that $pkt.e = [x, w]$
53. **if** $pkt.oent = null$ **then**
54. $change \leftarrow true$
55. **continue**
56. **end if**
57. **end if**
58. **else if** $pkt.mode = 1$ **then**
59. **if** $key_{s'}([c, pkt.nextHop]) \prec key_{s'}(pkt.ent)$ **then**
60. $pkt.ent \leftarrow [c, pkt.nextHop]$
61. **end if**
62. **else if** $pkt.mode = -1$ **then**
63. **if** $key_{s'}([pkt.prevtHop, c]) \prec key_{s'}(pkt.oent)$ **then**
64. $pkt.oent \leftarrow [pkt.prevHop, c]$
65. **end if**
66. **end if**
67. **end if**
68. **if** $pkt.e' = null$ **then**
69. **if** $intersects([c, pkt.nextHop], pkt.dstLoc, pkt.radius)$ **then**
70. **if** s' not contained in the face with $pkt.e$ on its boundary **and** $pkt.e = pkt.ent$ **then**
71. **return**

72. **end if**
73. **else if** s' not contained in the face with $pkt.e$ on its boundary **and** $[x,w] = pkt.oent$ such that $pkt.e = [w,x]$ **then**
74. make a copy of pkt and send //the packet is sent to pkt.nextHop
75. $pkt.e \leftarrow [pkt.nextHop, c]$
76. $pkt.prevHop \leftarrow pkt.nextHop$
77. $change \leftarrow true$
78. $temp \leftarrow pkt.ent$
79. $pkt.ent \leftarrow pkt.oent$
80. $pkt.oent \leftarrow temp$
81. **end if**
82. **end if**
83. **end if**
84. **while** $changeFace = true$
85. $pkt.hopCount = pkt.hopCount + 1$
86. $pkt.prevHop = c$
87. $send(pkt)$

GABRIEL-GRAPH(v)

1. $GG(v) \leftarrow \emptyset$
2. **for** each $u \in N(v)$ **do**
3. **if** $disk(u,v) \cap (N(v) \setminus \{u,v\}) = \emptyset$ **then**
4. $GG(v) \leftarrow GG(v) \cup \{u\}$
5. **end if**
6. **end for**
7. **return** $GG(v)$

A.7 AN Flooding Routing Packet Format and Protocol Pseudocode

AN FLOODING **Packet format**

type ← a numeric value indicating the protocol this packet belongs to
srcId ← source address
dstId ← destination address
nextHop ← next hop address
prevHop ← previous hop address
hopCount ← number of hops the packet has travelled so far
ttl ← maximum number of hops the packet can travel
dstLoc ← destination location, this is the center of the uncertainty zone
radius ← radius of the uncertainty zone
seqno ← a sequence number to stop broadcasting of duplicate packets

AN FLOODING **Protocol**

Initialization of the packet *pkt*:

srcId ← *s* // source address
dstId ← *d* // destination address
nextHop ← *null* // next hop address
prevHop ← *null* // previous hop address
hopCount ← 0 // number of hops the packet has traveled so far
ttl ← *MAXVAL* // maximum number of hops the packet can travel
dstLoc ← *p* // the old location of destination, this is the center of the uncertainty zone
radius ← radius of the uncertainty zone
seqno ← a sequence number to stop broadcasting of duplicate packets

AN-FLOODING-ROUTING(*pkt*, *c*) // *c* is the address of the current node

1. $S \leftarrow \emptyset$
2. *seqno* ← 0
3. $N(c)$ ← set of one-hop neighbors of *c* in the UDG // a node discovers its neighbors by running the neighbor discovery protocol
4. **if** $c = pkt.d$ **then**
5. *deliver*(*pkt*)
6. **return**

```
7.  end if
8.  if pkt.hopCount > pkt.ttl then
9.      drop(pkt)
10.     return
11. end if
12. if pkt.d ∈ N(c) then
13.     pkt.nextHop ← pkt.d
14.     pkt.prevHop ← c
15.     pkt.hopCount ← pkt.hopCount + 1
16.     send(pkt)
17.     return
18. end if
19. if pkt.srcId = c then
20.     pkt.seqno ← seqno
21.     seqno ← seqno + 1
22. end if
23. if distance(location(c), pkt.dstLoc) ≤ pkt.radius then
24.     if {pkt.srcId + pkt.dstId + pkt.seqno} ∈ S then
25.         drop(pkt)
26.         return
27.     else
28.         pkt.hopCount = pkt.hopCount + 1
29.         pkt.prevHop = c
30.         for all w ∈ N(c) do
31.             pkt.nextHop ← w
32.             make a copy of pkt and send //the packet is sent to pkt.nextHop
33.         end for
34.         S ← S ∪ {pkt.srcId + pkt.dstId + pkt.seqno}
35.     end if
36. else
37.     drop(pkt)
38.     return
39. end if
```

A.8 SN Flooding Routing Packet Format and Protocol Pseudocode

SN FLOODING **Packet format**

type ← a numeric value indicating the protocol this packet belongs to
srcId ← source address
dstId ← destination address
nextHop ← next hop address
prevHop ← previous hop address
hopCount ← number of hops the packet has travelled so far
ttl ← maximum number of hops the packet can travel
dstLoc ← destination location, this is the center of the uncertainty zone
radius ← radius of the uncertainty zone
seqno ← a sequence number to stop broadcasting of duplicate packets

SN FLOODING **Protocol**

Initialization of the packet *pkt*:
srcId ← *s* // source address
dstId ← *d* // destination address
nextHop ← *null* // next hop address
prevHop ← *null* // previous hop address
hopCount ← 0 // number of hops the packet has traveled so far
ttl ← *MAXVAL* // maximum number of hops the packet can travel
dstLoc ← *p* // the old location of destination, this is the center of
the uncertainty zone
radius ← radius of the uncertainty zone
seqno ← a sequence number to stop broadcasting of duplicate packets

SN-FLOODING-ROUTING(*pkt*, *c*) // *c* is the address of the current node

1. $S \leftarrow \emptyset$
2. $seqno \leftarrow 0$
3. $N(c) \leftarrow$ set of one-hop neighbors of c in the UDG // a node discovers its neighbors by running the neighbor discovery protocol
4. **if** $c = pkt.d$ **then**
5. $\quad deliver(pkt)$
6. $\quad$ **return**
7. **end if**
8. **if** $pkt.hopCount > pkt.ttl$ **then**
9. $\quad drop(pkt)$
10. $\quad$ **return**
11. **end if**
12. **if** $pkt.d \in N(c)$ **then**
13. $\quad pkt.nextHop \leftarrow pkt.d$
14. $\quad pkt.prevHop \leftarrow c$
15. $\quad pkt.hopCount \leftarrow pkt.hopCount + 1$
16. $\quad send(pkt)$
17. $\quad$ **return**
18. **end if**
19. **if** $pkt.srcId = c$ **then**
20. $\quad pkt.seqno \leftarrow seqno$
21. $\quad seqno \leftarrow seqno + 1$
22. **end if**
23. **if** $distance(location(c), pkt.dstLoc) \leq pkt.radius$ **then**
24. $\quad$ **if** $\{pkt.srcId + pkt.dstId + pkt.seqno\} \in S$ **then**
25. $\quad\quad drop(pkt)$
26. $\quad\quad$ **return**
27. $\quad$ **else**
28. $\quad\quad pkt.hopCount = pkt.hopCount + 1$
29. $\quad\quad pkt.prevHop = c$
30. $\quad\quad$ **for all** $w \in$ GREEDY-NEIGHBOR-SUBSET(c) **do**
31. $\quad\quad\quad pkt.nextHop \leftarrow w$
32. $\quad\quad\quad$ make a copy of pkt and send //the packet is sent to pkt.nextHop

33. **end for**
34. $S \leftarrow S \cup \{pkt.srcId + pkt.dstId + pkt.seqno\}$
35. **end if**
36. **else**
37. $drop(pkt)$
38. **return**
39. **end if**

GREEDY-NEIGHBOR-SUBSET(v)

1. $v \leftarrow$ current node
2. $R \leftarrow N_2(v)$//set of two-hop neighbors of v
3. $S \leftarrow \emptyset$
4. **while** $R \neq \emptyset$ **do**
5. select an $x \in N_1(v)$ that maximizes $| N_1(x) \cap R |$// N_1 is set of one-hop neighbors of v
6. $R \leftarrow R - N_1(x)$
7. $S \leftarrow S \cup \{x\}$
8. **end while**
9. **return** S