Graph Theory Algorithms for Mobile Ad Hoc Networks

Natarajan Meghanathan
Department of Computer Science, Jackson State University
Jackson, MS 39217, USA
E-mail: natarajan.meghanathan@jsums.edu

Keywords: mobile ad hoc networks, routing protocols, graph algorithms, unicast, multi-path, multicast and broadcast

Received: April 22, 2011

Various simulators (e.g., ns-2 and GloMoSim) are available to implement and study the behavior of the routing protocols for mobile ad hoc networks (MANETs). But, students and investigators who are new to this area often get perplexed in the complexity of these simulators and lose the focus in designing and analyzing the characteristics of the network and the protocol. Most of the time would be spent in learning the existing code modules of the simulator and the logical flow between the different code modules. The purpose of this paper would be to illustrate the applications of Graph Theory algorithms to study, analyze and simulate the behavior of routing protocols for MANETs. Specifically, we focus on the applications of Graph Theory algorithms to determine paths, trees and connected dominating sets for simulating and analyzing respectively unicast (single-path and multi-path), multicast and broadcast communication in mobile ad hoc networks (MANETs). We will discuss the (i) Dijkstra's shortest path algorithm and its modifications for finding stable paths and bottleneck paths; (ii) Prim's minimum spanning tree algorithm and its modification for finding all pairs smallest and largest bottleneck paths; (iii) Minimum Steiner tree algorithm to connect a source node to all the receivers of a multicast group; (iv) A node-degree based algorithm to construct an approximate minimum connected dominating set (CDS) for sending information from one node to all other nodes in the network; and (v) Algorithms to find a sequence of link-disjoint, node-disjoint and zone-disjoint multi-path routes in MANETs.

Povzetek: Prispevek opisuje algoritme za mobilna omrežja.

1 Introduction

A Mobile Ad hoc Network (MANET) is a dynamically changing infrastructureless and resource-constrained network of wireless nodes that may move arbitrarily, independent of each other. The transmission range of the wireless nodes is often limited, necessitating multi-hop routing to be a common phenomenon for communication between any two nodes in a MANET. Various routing protocols for unicast, multicast, multi-path and broadcast communication have been proposed for MANETs. The communication structures that are often determined include: a path (for unicast – single-path and multi-path routing), a tree (for multicast routing) and a connected dominating set – CDS (for broadcast routing). Within a particular class, it is almost impossible to find a single routing protocol that yields an optimal communication structure with respect to different route selection metrics and operating conditions.

Various simulators such as ns-2 [5] and GloMoSim [20] are available to implement and study the behavior of the routing protocols. But, students and investigators who are new to this area often get perplexed in the complexity of these simulators and lose the focus in designing and analyzing the characteristics of the network and the protocol. Most of the time would be spent in learning the existing code modules of the simulator and the logical flow between the different code modules. The purpose of this paper would be to illustrate the applications of Graph Theory algorithms to study, analyze and simulate the behavior of routing protocols for MANETs. We will discuss the applications of Graph Theory algorithms for unicast (single-path and multi-path), multicast and broadcast communication in MANETs.

An ad hoc network is often approximated as a unit disk graph [10]. In this graph, the vertices represent the wireless nodes and an edge exists between two vertices u and v if the normalized Euclidean distance (i.e., the physical Euclidean distance divided by the transmission range) between u and v is at most 1. Two nodes can communicate only if each node lies within (or on the edge of) the unit disk of the other node. The unit disk graph model neatly captures the behavior of many practical ad hoc networks and would be used in the rest of this paper for discussing the algorithms to simulate the MANET routing protocols.

Most of the contemporary routing protocols proposed in the MANET literature adopt a Least Overhead Routing Approach (LORA) according to which a communication structure (route, tree or CDS) discovered through a global flooding procedure would be used as long as the communication structure exist, irrespective of the structure becoming sub-optimal since the time of its discovery in the MANET. We will also

adopt a similar strategy and focus only on discovering a communication structure on a particular network graph taken as a snapshot during the functioning of the MANET. Such a graph snapshot would be hereafter referred to as a 'Static Graph' and a sequence of such static graphs over the duration of the MANET simulation session would be called a 'Mobile Graph'. A communication structure determined on a particular static graph would be then validated for its existence in the subsequent static graphs and once the structure breaks, the appropriate graph algorithm can be invoked on the static graph corresponding to that particular time instant and the above procedure would be continued for the rest of the static graphs in the mobile graph. We use the big-O notation to express the theoretical worst-case run-time complexity of the algorithms discussed in this paper. Given a problem size x, where x is usually the number of items, we say $f(x) = O(g(x))$, when there exists positive constants c and k such that $0 \leq f(x) \leq cg(x)$, for all $x \geq k$ [4].

The rest of this paper is organized as follows: Section 2 reviews related work on unicast, multicast, broadcast and multi-path communication in MANETs. In the subsequent sections, we discuss graph theory algorithms for unicast communication (Section 3), the tree-based algorithms for multicast communication (Section 4), a maximum density-based CDS algorithm for broadcast communication (Section 5) and multi-path algorithms for determining link-disjoint, node-disjoint and zone-disjoint routes (Section 6) in MANETs. Section 7 concludes the paper and Section 8 discuss future research directions in this area. Throughout the paper, the terms 'route' and 'path', 'link' and 'edge', 'message' and 'packet' are used interchangeably. They mean the same.

2 Background Work

2.1 Unicast Communication in MANETs

There are two broad classifications of unicast routing protocols: minimum-weight based routing and stability-based routing. Routing protocols under the minimum-weight category have been primarily designed to optimize the hop count of source-destination (s-d) routes. Some of the well-known minimum-hop based routing protocols include the Dynamic Source Routing (DSR) protocol [8] and the Ad hoc On-demand Distance Vector (AODV) routing protocol [16]. The stability-based routing protocols aim to minimize the number of route failures and in turn reduce the number of flooding-based route discoveries. Some of the well-known stability-based routing protocols include the Flow-Oriented Routing Protocol [18] and the Node Velocity-based Stable Path (NVSP) routing protocol [12]. In [13] and [14], it was observed that there exists a stability-hop count tradeoff and it is not possible to simultaneously optimize both the hop count as well as the number of route discoveries.

The DSR protocol is a source routing protocol that requires the entire route information to be included in the header of every data packet. However, because of this feature, intermediate nodes do not need to store up-to-date routing information in their routing tables. Route discovery is by means of the broadcast query-reply cycle. The Route Request (RREQ) packet reaching a node contains the list of intermediate nodes through which it has propagated from the source node. After receiving the first RREQ packet, the destination node waits for a short time period for any more RREQ packets, then chooses a path with the minimum hop count and sends a Route Reply (RREP) along the selected path. Later, if any new RREQ is received through a path with hop count less than that of the selected path, another RREP would be sent on the latest minimum hop path discovered.

The AODV protocol, like DSR, is also a shortest path based routing protocol. However, it is table-driven. Upon receiving an unseen RREQ packet (with the highest sequence number seen so far), an intermediate node records the upstream node (sender) of the RREQ packet in its routing table entry for the source-destination route. The intermediate node then forwards the RREQ packet by incrementing the hop count of the path from the source node. The destination node receives RREQ packets on several routes and selects that RREQ packet that traversed on the minimum-hop path to the destination node. The RREP packet is then sent on the reverse of this minimum-hop path towards the source node. The destination node includes the upstream node from which the RREQ was received as the downstream node on the path from the destination node to the source node. An intermediate node upon receiving the RREP packet will check whether it has been listed as the downstream node ID. In that case, the intermediate node processes the RREP packet and completes its routing table by including the sender of the RREP packet as the next hop node on the path from the source node towards the destination node. The intermediate node then replaces its own ID in the RREP downstream node entry with the ID of the upstream node that it has in its routing table for the path from the source node to the destination node.

The FORP protocol has been observed to discover the sequence of most stable routes among the contemporary stable path routing protocols [13]. FORP utilizes the mobility and location information of the nodes to approximately predict the expiration time (LET) of a wireless link. The minimum of LET values of all wireless links on a path is termed as the route expiration time (RET). The route with the maximum RET value is selected as the desired route. Each node is assumed to be able to predict the LET values of the links with its neighboring nodes based on the information regarding the current position of the nodes, velocity, the direction of movement, and transmission range. FORP assumes the availability of location-update mechanisms like Global Positioning System (GPS) [6] to identify the location of the nodes and also requires each node to periodically broadcast its location and mobility information to its neighbors through beacons.

The NVSP protocol is the only beaconless routing protocol that can discover long-living stable routes without significant increase in the hop count per path.

FORP discovers routes that have a significantly larger hop count than the minimum value. NVSP only requires each intermediate node to include its velocity in the RREQ packets propagated via flooding from the source node to the destination node. With flooding, each intermediate node forwards the RREQ packet exactly once, the first time the node sees the packet as part of a particular route discovery session. The destination node receives the RREQ packets through several paths and determines the bottleneck velocity of each of those paths. The bottleneck velocity of a path is the maximum among the velocities of the intermediate nodes on the path. The destination node chooses the path with the minimum bottleneck velocity and sends a RREP packet along that path. In case of a tie, the destination node chooses the path with the lowest hop count and if the tie could not be still broken, the destination node chooses an arbitrary path among the contending paths.

2.2 Multicast Communication in MANETs

The Multicast communication refers to sending messages from one source node to a set of receiver nodes in a network. The receiver nodes form the multicast group and we typically find a tree that connects the source node to the multicast group members such that there is exactly one path from the source node to each receiver node. The tree could be constructed based on either one of the following two objectives: (i) Shortest path tree – the tree would have the minimum hop count paths from the source node to each receiver node and (ii) Steiner tree – the tree would have the minimum number of links spanning the source node and the multicast group members. Both these trees cannot be simultaneously built and there would always be a tradeoff between the above two objectives [14]. The Multicast Extension of the Ad hoc On-demand Distance Vector (MAODV) protocol and the Bandwidth Efficient Multicast Routing Protocol (BEMRP) are respectively examples of the minimum hop and minimum link based multicast protocols.

MAODV [15] is the multicast extension of the AODV unicast routing protocol. Here, a receiver node joins the multicast tree through a member node that lies on the minimum-hop path to the source node. A potential receiver node wishing to join the multicast group broadcasts a RREQ message. If a node receives the RREQ message and is not part of the multicast tree, the node broadcasts the message in its neighborhood and also establishes the reverse path by storing the state information consisting of the group address, requesting node id and the sender node id in a temporary cache. If a node receiving the RREQ message is a member of the multicast tree and has not seen the RREQ message earlier, the node waits to receive several RREQ messages and sends back a RREP message on the shortest path to the receiver node. The member node also informs in the RREP message, the number of hops from itself to the source node. The potential receiver node receives several RREP messages and selects the member node which lies on the shortest path to the source node. The receiver node sends a Multicast Activation (MACT) message to the selected member node along the chosen route. The route from the source node to the receiver node is set up when the member node and all the intermediate nodes in the chosen path update their multicast table with state information from the temporary cache.

According to BEMRP [17], a newly joining node to the multicast group opts for the nearest forwarding node in the existing tree, rather than choosing a minimum-hop count path from the source node of the multicast group. As a result, the number of links in the multicast tree is reduced leading to savings in the network bandwidth. Multicast tree construction is receiver-initiated. When a node wishes to join the multicast group as a receiver node, it initiates the flooding of Join control packets targeted towards the nodes that are currently members of the multicast tree. On receiving the first Join control packet, the member node waits for a certain time before sending a Reply packet. The member node sends a Reply packet on the path, traversed by the Join control packet, with the minimum number of intermediate forwarding nodes. The newly joining receiver node collects the Reply packets from different member nodes and would send a Reserve packet on the path that has the minimum number of forwarding nodes from the member node to itself.

2.3 Broadcast Communication in MANETs

Broadcast communication refers to sending a message from one node to all the other nodes in the network. Since MANET topology is not fully connected as nodes operate with a limited transmission range, multi-hop communication is a common phenomenon in routing. As a result, a message has to be broadcast by more than one node (in its neighborhood) so that the message can reach all the nodes in the network. An extreme case of broadcasting is called flooding wherein each node broadcasts the message among its neighbors, exactly once, when the message is seen for the first time. This ensures that the message is received by all the nodes in the network. However, flooding would cause unnecessary retransmissions, exhausting the network bandwidth and the energy reserves at the nodes.

Connected Dominating Sets (CDS) are considered to be very efficient for broadcasting a message from one node to all the nodes in the network. A CDS is a sub graph of a given undirected connected graph such that all nodes in the graph are included in the CDS or directly attached to a node (i.e., covered by a node) in the CDS. A Minimum Connected Dominating Set (MCDS) is the smallest CDS (in terms of the number of nodes in the CDS) for the entire graph. For a virtual backbone-based route discovery, the smaller the size of the CDS, the smaller is the number of unnecessary retransmissions. If the RREQ packets of a broadcast route discovery process get forwarded only by the nodes in the MCDS, we will have the minimum number of retransmissions. Unfortunately, the problem of determining the MCDS in an undirected graph, like that of the unit disk graph considered for modeling MANETs, is NP-complete. In

[1], [2] and [3], efficient algorithms have been proposed to approximate the MCDS for wireless ad hoc networks. A common thread among these algorithms is to give preference to nodes with high neighborhood density (i.e., a larger number of uncovered neighbors) for inclusion in the MCDS.

2.4 Multi-path Communication in MANETs

MANET routing protocols incur high route discovery latency and also incur frequent route discoveries in the presence of a dynamically changing topology. Recent research has started to focus on multi-path routing protocols for fault tolerance and load balancing. Multi-path on-demand routing protocols tend to compute multiple paths, at both the traffic sources as well as at intermediary nodes, in a single route discovery attempt. This reduces both the route discovery latency and the control overhead as a route discovery is needed only when all the discovered paths fail. Spreading the traffic along several routes could alleviate congestion and bottlenecks. Multi-path routing also provides a higher aggregate bandwidth and effective load balancing as the data forwarding load can be distributed over all the paths.

Multi-paths can be of three types: link-disjoint, node-disjoint and zone-disjoint. For a given source node s and destination node d, the set of link-disjoint s-d routes comprises of paths that have no link present in more than one constituent s-d path. Similarly, the set of node-disjoint s-d routes comprises of paths that have no node (other than the source node and destination node) present in more than one constituent s-d path. A set of zone-disjoint s-d routes comprises of paths such that an intermediate node in one path is not a neighbor node of an intermediate node in another path. Multi-path on-demand routing protocols tend to compute multiple paths between a source-destination (s-d) pair, in a single route discovery attempt. A new network-wide route discovery operation is initiated only when all the s-d paths fail. The Split Multi-path Routing (SMR) protocol [11], the AODV-Multi-path (AODVM) protocol [19] and the Zone-Disjoint multi-path extension to the DSR (ZD-DSR) protocol [7] are respectively well-known examples for link-disjoint, node-disjoint and zone-disjoint multi-path routing protocols.

In SMR, the intermediate nodes forward RREQs that are received along a different link and with a hop count not larger than the first received RREQ. The destination node selects the route on which it received the first RREQ packet (which will be a shortest delay path), and then waits to receive more RREQs. The destination node then selects the path which is maximally disjoint from the shortest delay path. If more than one maximally disjoint path exists, the tie is broken by choosing the path with the shortest hop count.

In AODVM, an intermediate node does not discard duplicate RREQ packets and records them in a RREQ table. The destination node responds with an RREP for each RREQ packet received. An intermediate node, on receiving the RREP, checks its RREQ table and forwards the packet to the neighbor that lies on the shortest path to the source node. The neighbor entry is then removed from the RREQ table. Also, whenever a node hears a neighbor node forwarding the RREP packet, the node removes the entry for the neighbor node in its RREQ table.

The Zone-Disjoint Multi-path extension of the Dynamic Source Routing (ZD-MPDSR) protocol proposed for an omni-directional system works as follows: Whenever a source node has no route to send data to a destination node, the source node initiates broadcast of the RREQ messages. The number of active neighbors for a node indicates the number of neighbor nodes that have received and forwarded the RREQ message during a route discovery process. The RREQ message has an ActiveNeighborCount field and it is updated by each intermediate node before broadcasting the message in the neighborhood. When an intermediate node receives the RREQ message, it broadcasts a 1-hop RREQ-query message in its neighborhood to determine the number of neighbors who have also seen the RREQ message. The number of RREQ-query-replies received from the nodes in the neighborhood is the value of the ActiveNeighborCount field updated by a node in the RREQ message. The destination node receives several RREQ messages and selects the node-disjoint paths with lower ActiveNeighborCount values and sends the RREP messages to the source node along these paths. Even though the selection of the zone-disjoint paths with lower number of active neighbors will lead to reduction in the end-to-end delay per data packet, the route acquisition phase will incur a significantly longer delay as RREQ-query messages are broadcast at every hop (in addition to the regular RREQ message) and the intermediate nodes have to wait to receive the RREQ-query and reply messages from their neighbors. This will significantly increase the control overhead in the network.

3 Graph Theory Algorithms for Unicast Communication in MANETs

In a graph theoretic context, we illustrate that the minimum-weight (minimum-hop) based routing protocols could be simulated by running the shortest-path Dijkstra algorithm [4] on a mobile graph (i.e. a sequence of static graphs). We then illustrate that the NVSP and FORP protocols could be simulated by respectively solving the smallest bottleneck and the largest bottleneck path problems – each of which could be implemented as a slight variation of the shortest path Dijkstra algorithm. In addition, we also illustrate that the Prim's minimum spanning tree algorithm and its modification to compute the maximum spanning tree can be respectively used to determine the 'All Pairs Smallest Bottleneck Paths' and 'All Pairs Largest Bottleneck Paths' in a weighted network graph.

3.1 Shortest Path Problem

Given a weighted graph G = (V, E), where V is the set of vertices and E is the set of weighted edges, the shortest path problem is to determine a minimum-weight path between any two nodes (identified as source node s and destination node d) in the graph. The execution of the Dijkstra algorithm (pseudo code in Figure 1) on a weighted graph starting at the source node s results in a shortest path tree rooted at s. In other words, the Dijkstra algorithm will actually return the minimum-weight paths from the source vertex s to every other vertex in the weighted graph. If all the edge weights are 1, then the minimum-weight paths are nothing but minimum-hop paths.

```
Begin Algorithm Dijkstra-Shortest-Path (G, s)
1   For each vertex v ∈ V
2    weight [v] ← ∞ // an estimate of the minimum-
                       weight path from s to v
3   End For
4   weight [s] ← 0
5   S ← Φ // set of nodes for which we know the
            minimum-weight path from s
6   Q ← V // set of nodes for which we know estimate of
            the minimum-weight path from s
7   While Q ≠ Φ
8      u ← EXTRACT-MIN (Q)
9      S ← S U {u}
10     For each vertex v such that (u, v) ∈ E
11        If weight [v] > weight [u] + weight (u, v) then
12          weight [v] ← weight [u] + weight (u, v)
13          Predecessor (v) ← u
14        End If
15     End For
16  End While
17 End Dijkstra-Shortest-Path
```

Figure 1: Pseudo Code for Dijkstra's Shortest Path Algorithm.

Dijkstra algorithm proceeds in iterations. To begin with, the weights of the minimum-weight paths from the source vertex to every other vertex is assumed to be $+\infty$ (as estimate value, indicating that the paths are actually not known) and from the source vertex to itself is assumed to be 0. During each iteration, we determine the shortest path from the source vertex s to a particular vertex u, which would be the vertex with the minimum weight among the vertices that have been not yet optimized (i.e. for which the shortest path has not been yet determined). We then explore the neighbors of u and determine whether we can reach any of the neighbor vertices, say v, from s through u on a path with weight less than the estimated weight of the current path we know from s to v. If we could find such a neighbor v, then we set the predecessor of v to be vertex u on the shortest path from s to v. This step is called the relaxation step and is repeated over all iterations. The darkened edges shown in the working example of Figure 2 are the edges that are part of the shortest-path tree rooted at the source vertex s. The run-time complexity of the Dijkstra's shortest path algorithm is $O(|V|^2)$.

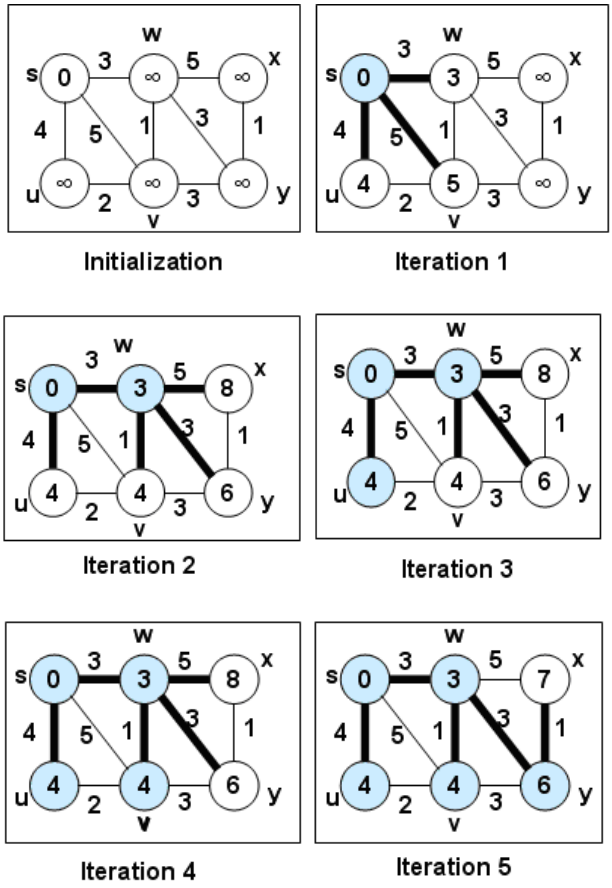


Figure 2: Example to Illustrate the Working of the Dijsktra's Shortest Path Algorithm.

3.2 Smallest Bottleneck Path Problem

In the context of the smallest bottleneck path problem, we define the bottleneck weight of a path p to be the maximum of the weights of the constituent edges, $e \in p$. Given the set of all loop-free paths P between a source node s and destination node d, the smallest bottleneck path is the path with the smallest bottleneck weight. We can express this mathematically as $\underset{p \in P}{Min}\left[\underset{\forall e \in p}{Max}\left(weight(e)\right)\right]$.

The

```
Begin Algorithm Modified-Dijkstra-Smallest-
                Bottleneck-Path (G, s)
1   For each vertex v ∈ V
2      weight [v] ← +∞ // an estimate of the smallest
                         bottleneck weight path from s to v
3   End For
4   weight [s] ← - ∞
5   S ← Φ // set of nodes for which we know the
            smallest bottleneck weight path from s
```

```
6   Q ← V // set of nodes for which we know an
          estimate of the smallest bottleneck weight
          path from s
7   While Q ≠ Φ
8     u ← EXTRACT-MIN (Q)
9     S ← S U {u}
10    For each vertex v such that (u, v) Є E
11      If weight[v] > Max(weight [u], weight (u, v)) then
12          weight [v] ← Max (weight [u], weight (u, v))
13          Predecessor (v) ← u
14      End If
15    End For
16  End While
17 End Modified-Dijkstra-Smallest-Bottleneck-Path
```

Figure 3: Pseudo Code for the Modified Dijkstra Smallest Bottleneck Path Algorithm.

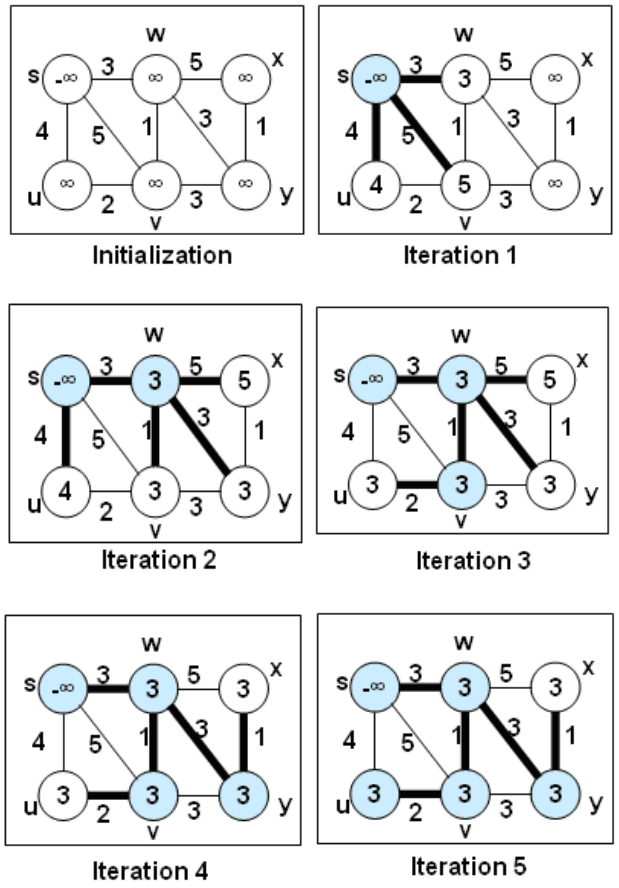


Figure 4**:** Example for the Smallest Bottleneck Path Problem.

NVSP protocol can be implemented in a graph theoretic context through a modified version of the Dijkstra's algorithm (pseudo code in Figure 3) that solves the smallest bottleneck path problem. Accordingly, the weight of a link from node u to node v is the velocity of the downstream node v. To start with, the weight of the smallest bottleneck path from the source vertex s to every other vertex is estimated to be +∞; whereas the weight of the smallest bottleneck path from the source vertex s to itself is set to - ∞. At the beginning of an iteration, the vertex (say u) with the smallest bottleneck weight among the vertices that have been not yet optimized is now considered to be optimized. As part of the relaxation step, we check whether the current weight of the smallest bottleneck path to any non-optimized neighbor v, i.e. weight[v], is greater than the maximum of the weight of the recently optimized largest bottleneck path from s to u, i.e. weight[u] and the weight of the edge (u, v). If this relaxation condition evaluates to true, then the bottleneck weight of the path from s to v is correspondingly updated (i.e., weight[v] = Max (weight[u], weight(u, v)) and the predecessor of v is set to be u for the path from s to v. This step is repeated over all iterations. A working example is presented in Figure 4. The run-time complexity of the modified Dijkstra algorithm for the smallest bottleneck path problem is the same as that of the original Dijkstra algorithm.

3.3 All Pairs Smallest Bottleneck Paths Problem

In this section, we show that the smallest bottleneck path between any two vertices u and $v \in V$ in an undirected weighted graph $G = (V, E)$ is the path between u and v in the minimum spanning tree of G. The Prim's algorithm [4] is a well-known algorithm to determine the minimum spanning tree of weighted graphs and its pseudo code is illustrated in Figure 5. The Prim's algorithm is very similar to the Dijkstra algorithm – the major difference is in the relaxation step.

```
Begin Algorithm Prim (G, s); s – is any arbitrarily
                              chosen starting vertex
1   For each vertex v Є V
2       weight [v] ← ∞
3   End For
4   weight [s] ← 0
5   S ← Φ // set of nodes whose bottleneck weights will
          not change further
6   Q ← V // set of nodes whose bottleneck weights are
          only estimates; final weight could change
7   While Q ≠ Φ
8     u ← EXTRACT-MIN (Q)
9     S ← S U {u}
10    For each vertex v such that (u, v) Є E
11      If weight [v] > weight (u, v) then
12        weight [v] ← weight (u, v)
13        Predecessor (v) ← u
14      End If
15    End For
16  End While
17 End Prim
```

Figure 5: Pseudo Code for the Prim's Algorithm to Determine a Minimum Spanning Tree.

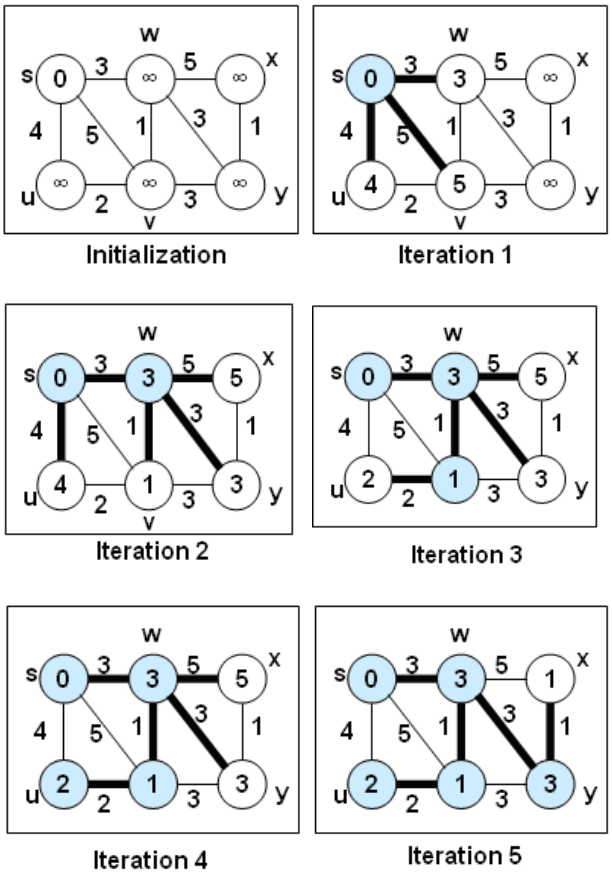


Figure 6: Example for the Minimum Spanning Tree – All Pairs Smallest Bottleneck Paths.

The Prim's algorithm work as follows: The starting vertex is any arbitrarily chosen vertex (say s) in the given undirected weighted graph G. To begin with, the weights of the smallest bottleneck paths from the starting vertex to every other vertex is assumed to be $+\infty$ (as estimate value, indicating that the paths are actually not known) and the path from the starting vertex to itself is assumed to be 0. During every iteration, we determine the smallest bottleneck path from the starting vertex s to a particular vertex u, which would be the vertex with the minimum weight among the vertices that have been not yet optimized (i.e. for which the smallest bottleneck path has not been yet determined). We then explore the neighbors of u and determine whether we can reach any of the neighbor vertex, say v, from s through u on a path with weight less than the estimated bottleneck weight of the current path we know from s to v. If we could find such a neighbor v as part of the relaxation step, we set the new estimated bottleneck weight of vertex v to the weight of the edge (u, v) and also set the predecessor of v to be vertex u on the smallest bottleneck path from s to v. The darkened edges shown in the working example of Figure 6 are the edges that are part of the smallest bottleneck path tree rooted at the starting vertex s. The path between any two vertices in this smallest bottleneck path tree is the smallest bottleneck path between the two vertices in the original graph. The run-time complexity of the Prim's minimum spanning tree algorithm is $O(|V|^2)$.

Note that in both Figures 4 and 6, we start with the same initial graph. Since, the relaxation step of the modified Dijkstra algorithm and the Prim's algorithm are different, the sequence of vertices that are optimized in each algorithm is different from one another. However, the final tree rooted at the starting vertex s is the same in both the figures. This example vindicates our argument that the minimum spanning tree contains the smallest bottleneck paths between any two vertices in the original graph. We now formally prove this argument (refer Figure 7 for an illustration of the example) below through the method of Proof by Contradiction.

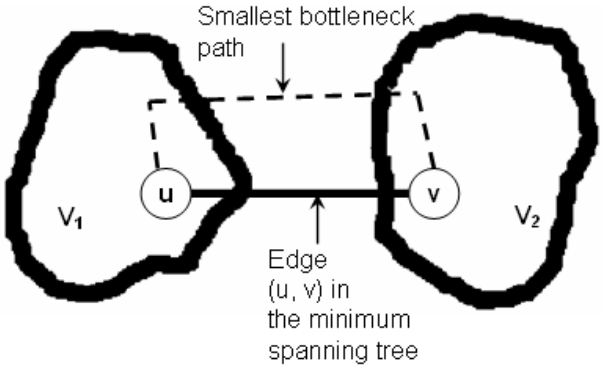


Figure 7: Proof by Contradiction: Minimum Spanning Tree with All Pairs Smallest Bottleneck Paths.

Let there be a pair of vertices $u \in V_1$ and $v \in V_2$ in $G = (V, E)$ such that the edge $(u, v) \in E$, $V_1 \cup V_2 = V$ and $V_1 \cap V_2 = \Phi$. Assume the edge (u, v) belongs to the minimum spanning tree T of G; but the edge is not part of the smallest bottleneck path from u to v. Let there exist an alternate path from u to v that is the smallest bottleneck path. Since u and v are in two disjoint vertex partitions, there should be at least one edge (call it e) in the path from u to v with endpoint vertices in each partition. But, by definition of a minimum spanning tree, the weight$(u, v) \leq$ weight(edge e); otherwise, a cheaper tree could be obtained by replacing (u, v) with the edge e and T containing (u, v) would not be a minimum spanning tree. Hence, edge e could be replaced by edge (u, v) without increasing the weight of any smallest bottleneck path. Likewise, we can prove that every edge in T would be part of a smallest bottleneck path. Since T is a minimum spanning tree, all its edges constitute the all pairs smallest bottleneck paths for the entire graph.

3.4 Largest Bottleneck Paths Problem

In the context of the largest bottleneck path problem, we define the bottleneck weight of a path p to be the minimum of the weights of the constituent edges, $e \in p$. Given the set of all loop-free paths P between a source node s and destination node d, the largest bottleneck path is the path with the largest bottleneck weight. We can express this mathematically as $\underset{p \in P}{Max}\left[\underset{\forall e \in p}{Min}\left(weight(e)\right)\right]$.

Begin Algorithm *Modified-Dijkstra-Largest-Bottleneck-Path* (*G*, *s)*
1 **For** each vertex $v \in V$
2 *weight* [*v*] ← - ∞ // an estimate of the largest bottleneck weight path from *s* to *v*
3 **End For**
4 *weight* [*s*] ← + ∞
5 *S* ← *Φ* // set of nodes for which we know the largest bottleneck weight path from *s*
6 *Q* ← *V* // set of nodes for which we know an estimate of the largest bottleneck weight path from *s*
7 **While** $Q \neq \Phi$
8 *u* ← EXTRACT-MAX (*Q*)
9 *S* ← *S* U {*u*}
10 **For** each vertex *v* such that $(u, v) \in E$
11 **If** *weight*[*v*] < Min (*weight* [*u*], *weight* (*u*, *v*)) then
12 *weight* [*v*] ← Min (*weight* [*u*], *weight* (*u*, *v*))
13 Predecessor (*v*) ← *u*
14 **End If**
15 **End For**
16 **End While**
17 **End** *Modified-Dijkstra-Largest-Bottleneck-Path*

Figure 8: Pseudo Code for the Modified Dijkstra's Algorithm for the Largest Bottleneck Path Problem.

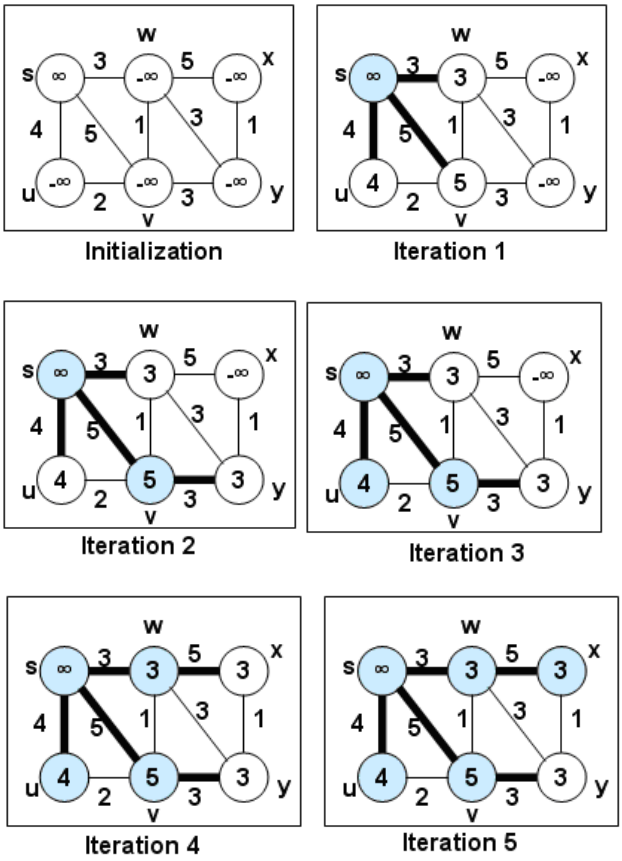


Figure 9: Example for the Largest Bottleneck Path Problem.

The FORP protocol can be simulated using a modified version of the Dijsktra's algorithm (pseudo code in Figure 8) that solves the Largest Bottleneck Path problem on a static graph. The edge weights correspond to the predicted LET values for the corresponding links. To start with, the weight of the largest bottleneck path from the source vertex *s* to every other vertex is estimated to be - ∞; whereas the weight of the largest bottleneck path from the source vertex *s* to itself is set to +∞. At the beginning of an iteration, the vertex (say *u*) with the largest bottleneck weight among the vertices that have been not yet optimized is now considered to be optimized (i.e., the largest bottleneck path from the source vertex *s* to the vertex *u* is considered to have been determined by now). As part of the relaxation step, we check whether the current weight of the largest bottleneck path to any non-optimized neighbor *v*, i.e. *weight*[*v*], is lower than the minimum of the weight of the recently optimized largest bottleneck path from *s* to *u*, i.e. *weight*[*u*] and the weight of the edge (*u*, *v*). If this relaxation condition evaluates to true, then the bottleneck weight of the path from *s* to *v* is correspondingly updated (i.e., *weight*[*v*] = Min(*weight*[*u*], *weight*(*u*, *v*)) and the predecessor of *v* is set to be *u* for the path from *s* to *v*. This step is repeated over all iterations. A working example is presented in Figure 9. The run-time complexity of the modified Dijkstra algorithm for the largest bottleneck path problem is the same as that of the original algorithm for the shortest path problem.

3.5 All Pairs Largest Bottleneck Paths Problem

In this section, we show that the largest bottleneck path between any two vertices *u* and $v \in V$ in an undirected weighted graph $G = (V, E)$ is the path between *u* and *v* in the maximum spanning tree of *G*. The maximum spanning tree of a graph can be determined using a slightly modified version of the Prim's algorithm – the modification is in the initialization step and the relaxation condition. In the original Prim's algorithm, the initial weight of all the vertices other than the starting vertex is set to +∞; whereas in the modified Prim's algorithm for the all pairs largest bottleneck path problem, the initial weight of all the vertices other than the starting vertex is set to – ∞ (an initial estimate for the largest bottleneck paths, which are actually not know to start with). The weight of the starting vertex, *s*, in both algorithms is 0. The pseudo code of the modified Prim's algorithm is given in Figure 10.

Begin Algorithm *Modified-Prim* (*G*, *s)*; *s* – is any arbitrarily chosen starting vertex
1 **For** each vertex $v \in V$
2 *weight* [*v*] ← – ∞
3 **End For**
4 *weight* [*s*] ← 0
5 *S* ← *Φ* // set of nodes whose bottleneck weights will not change further
6 *Q* ← *V* // set of nodes whose bottleneck weights

```
                are only estimates and the final weight
                could change
7    While Q ≠ Φ
8       u ← EXTRACT-MAX (Q)
9       S ← S U {u}
10      For each vertex v such that (u, v) Є E
11         If weight [v] < weight (u, v) then
12         weight [v] ← weight (u, v)
13         Predecessor (v) ← u
14         End If
15      End For
16   End While
17 End Modified-Prim
```

Figure 10: Pseudo Code for the Modified Prim's Algorithm to Determine a Maximum Spanning Tree

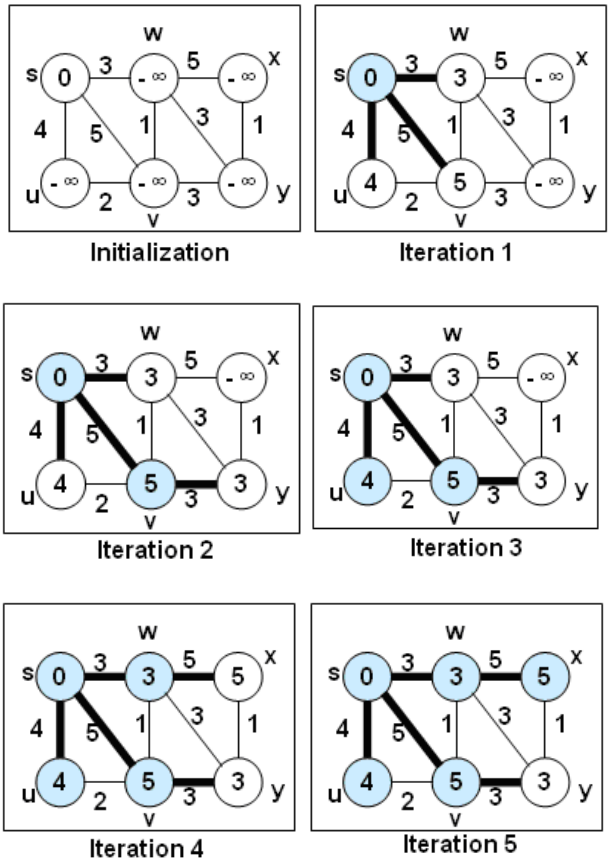


Figure 11: Example for the Maximum Spanning Tree – All Pairs Largest Bottleneck Paths.

During every iteration, we determine the largest bottleneck path from the starting vertex s to a particular vertex u, which would be the vertex with the maximum weight among the vertices that have been not yet optimized (i.e. for which the largest bottleneck path has not been yet determined). We then explore the neighbors of u and determine whether we can reach any of the neighbor vertices, say v, from s through u on a path with weight greater than the estimated bottleneck weight of the current path we know from s to v. If we could find such a neighbor v as part of the relaxation step, we set the new estimated bottleneck weight of vertex v to the weight of the edge (u, v) and also set the predecessor of v to be vertex u on the largest bottleneck path from s to v. The darkened edges shown in the working example of Figure 11 are the edges that are part of the largest bottleneck path tree rooted at the starting vertex s. The path between any two vertices in this largest bottleneck path tree is the largest bottleneck path between the two vertices in the original graph. The run-time complexity of the modified Prim's algorithm to determine maximum spanning tree is $O(|V|^2)$, the same as the original Prim's algorithm for minimum spanning trees. The correctness of the modified Prim's algorithm for the all pairs largest bottleneck path problem can be proved using the same logic used to prove the correctness of the Prim's algorithm for the all pairs smallest bottleneck path problem.

4 Graph Theory Algorithms for Multicast Communication in MANETs

The original Dijkstra shortest path algorithm described previously can be used to determine shortest path trees with minimum hop count per source-receiver path. The problem of determining a multicast tree with the minimum number of links is a NP-complete problem for which there is no polynomial-time algorithm yielding an optimistic solution. Hence, algorithms have been proposed in the literature to approximate such multicast trees. The Steiner tree algorithm is the best-known algorithm in the literature to approximate multicast trees with the minimum number of links connecting a source node to the receiver nodes of the multicast group.

Given a static graph, $G = (V, E)$, where V is the set of vertices, E is the set of edges and a subset of vertices (called the multicast group or Steiner points) $MG \subseteq V$, the multicast Steiner tree is the tree with the least number of edges required to connect all the vertices in MG. In this paper, we use a well-known $O(|V||MG|^2)$ algorithm of Kou et al [9], where $|V|$ is the number of nodes in the network graph and $|MG|$ is the size of the multicast group comprising of the source nodes and the receiver nodes, to approximate the minimum edge Steiner tree in graphs representing snapshots of the network topology. In unit disk graphs such as the static graphs used in our research, Step 5 of the algorithm (pseudo code in Figure 12) is not needed and the minimal spanning tree T_{MG} obtained at the end of Step 4 could be considered as the minimum edge Steiner tree. One could use the Prim's algorithm to find the minimum spanning trees.

Input: A Static Graph $G = (V, E)$
Multicast Group $MG \subseteq V$
Output: A *MG-Steiner-tree* for the set $MG \subseteq V$

Begin Kou et al Algorithm (G, MG)

Step 1: Construct a complete undirected weighted graph $G_C = (MG, E_C)$ from G and MG where $\forall (v_i, v_j) \in E_C$, v_i and v_j are in MG, and the weight of edge (v_i, v_j) is the length of the shortest path from v_i to v_j in G.
Step 2: Find the minimum weight spanning tree T_C in G_C (If more than one minimal spanning tree exists, pick an arbitrary one).
Step 3: Construct the sub graph G_{MG} of G, by replacing each edge in T_C with the corresponding shortest path from G (In case of any tie, between an arbitrary shortest path between the two vertices).
Step 4: Find the minimal spanning tree T_{MG} in G_{MG} with unit edge weights. If more than one minimal spanning tree exists, pick an arbitrary one).
return T_{MG} as the *MG-Steiner-tree*

End *Kou et al Algorithm*

Figure 12: Kou et al's Algorithm to Find an Approximate Minimum Edge Steiner Tree.

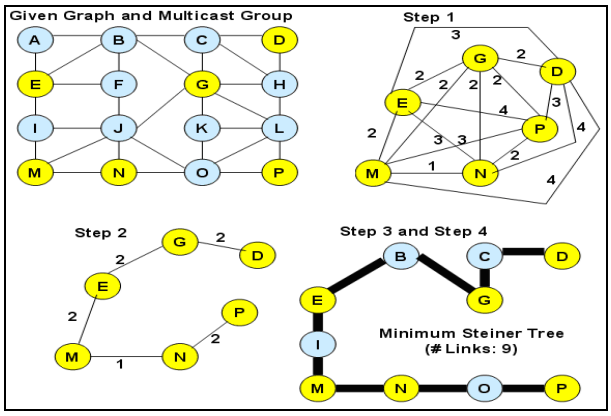


Figure 13: Construction of a Minimum Steiner Tree using Kou et al.'s Algorithm.

We now illustrate the working of the Kou et al's algorithm through an example in Figure 13. The vertices {D, G, E, M, N, P} form the multicast group in the vertex set {A, B … P}. As observed in the example, the sub graph G_{MG} obtained in Step 3 is nothing but the minimal spanning tree T_{MG}, which is the output of Step 4. In general, for unit disk graphs, like the static graphs we are working with, the outputs of both Steps 3 and 4 are the same and it is enough that we stop at Step 3 and output the MG-Steiner-tree.

5 Graph Theory Algorithms for Broadcast Communication in MANETs

In this section, we describe a maximum-density based CDS (MaxD-CDS) graph theoretic algorithm to approximate a Minimum Connected Dominating Set (MCDS) in a static graph, taken as a snapshot of a MANET topology.

5.1 Data Structures used by the MaxD-CDS Algorithm and Breadth First Search

We use the following principal data structures for the MaxD-CDS algorithm:

(i) *CDS-Node-List* – includes all nodes that are members of the CDS
(ii) *Covered-Nodes-List* – includes all nodes that are in the CDS-Node-List and all nodes that are adjacent to at least one member of the CDS-Node-List.

Before we run the CDS formation algorithm, we make sure the underlying network graph is connected by running the Breadth First Search (BFS) algorithm [4]; because, if the underlying network graph is not connected, we would not be able to find a CDS that will cover all the nodes in the network. We run BFS, starting with an arbitrarily chosen node in the network graph. If we are able to visit all the vertices in the graph, then the corresponding network is said to be connected. If the graph is not connected, we simply continue with the static graph (snapshot of the network topology) collected at the next time instant and start with the BFS test. The pseudo code for BFS is shown in Figure 14. The run-time complexity of BFS is O(|*V*|+|*E*|).

Input: Graph $G = (V, E)$
Auxiliary Variables/Initialization:
Nodes-Explored = Φ, *FIFO-Queue* = Φ
Begin Algorithm *BFS* (*G*, *s)*
root-node = randomly chosen vertex in *V*
Nodes-Explored = *Nodes-Explored* U {*root-node*}
FIFO-Queue = *FIFO-Queue* U {*root-node*}
while (|*FIFO-Queue*| > 0) **do**
front-node u = Dequeue(*FIFO-Queue*) // extract the first node
for (every edge (*u*, *v*)) **do** // i.e. every neighbor *v* of node *u*
if (*v* ∉ *Nodes-Explored*) **then**
Nodes-Explored = *Nodes-Explored* U {*v*}
FIFO-Queue = *FIFO-Queue* U {*v*}
Parent (*v*) = *u*
end if
end for
end while
if (| *Nodes-Explored* | = | *V* |) **then**
return Connected Graph - true
else return Connected Graph - false
end if
End Algorithm *BFS*

Figure 14: Pseudo Code for the BFS Algorithm to Determine Network Connectivity.

5.2 Maximum Density-based Algorithm to Approximate a MCDS

The idea of the maximum density (MaxD)-based CDS formation algorithm is to select nodes with larger number of uncovered neighbors for inclusion in the CDS. The algorithm forms and outputs a CDS based on a given input graph representing a snapshot of the MANET at a particular time instant. Specifically, the algorithm outputs a list (*CDS-Node-List)* of all nodes that are part of the CDS formed based on the given MANET. The first node to be included in the *CDS-Node-List* is the node with the maximum number of uncovered neighbors (any ties are broken arbitrarily). A CDS member is considered to be "covered", so a CDS member is additionally added to the *Covered-Nodes-List* when it is included in the *CDS-Node-List*. All nodes that are adjacent to a CDS member are also said to be covered, so the uncovered neighbors of a CDS member are also added to the *Covered-Nodes-List* as the member is added to the *CDS-Node-List*. To determine the next node to be added to the *CDS-Node-List*, we must select the node with the largest density amongst the nodes that meet the criteria for inclusion into the CDS.

Input: Graph $G = (V, E)$, where V is the vertex set and E is the edge set.
Source vertex, s – vertex with the largest number of uncovered neighbors in V.
Auxiliary Variables and Functions: *CDS-Node-List*, *Covered-Nodes-List, Neighbors*(v), $\forall v \in V$.
Output: *CDS-Node-List*
Initialization: *Covered-Nodes-List* = $\{s\}$, *CDS-Node-List* = Φ
Begin Construction of *MaxD-CDS*
  while (|*Covered-Nodes-List*| < |V|) **do**
    Select a vertex $r \in$ *Covered-Nodes-List* and $r \notin$ *CDS-Node-List* such that r has the largest number of uncovered neighbors that are not in *Covered-Nodes-List*

    CDS-Node-List = *CDS-Node-List* U $\{r\}$

    for all $u \in$ *Neighbors*(r) and $u \notin$ *Covered-Nodes-List*
      Covered-Nodes-List = *Covered-Nodes-List* U $\{u\}$
    end for
end while
return *CDS-Node-List*
End Construction of *MaxD-CDS*

Figure 15: Pseudo Code for the Algorithm to Construct the Maximum Density (MaxD)-based CDS.

The criteria for CDS membership selection are the following: the node should not already be a part of the CDS (*CDS-Node-List),* the node must be in the *Covered-Nodes-List*, and the node must have at least one uncovered neighbor (at least one neighbor that is not in the *Covered-Nodes-List*). Amongst the nodes that meet these criteria for CDS membership inclusion, we select the node with the largest density (i.e., the largest number of uncovered neighbors) to be the next member of the CDS. Ties are broken arbitrarily. This process is repeated until all nodes in the network are included in the *Covered-Nodes-List*. Once all nodes in the network are considered to be "covered", the CDS is formed and the algorithm returns a list of nodes in the resulting MaxD-CDS (nodes in the *CDS-Node-List*). The run-time complexity of the MaxD-CDS algorithm is $O(|V|^2+|E|)$ since there would be at most $|V|$ iterations – it would take $O(|V|)$ time to determine the node with the largest number of uncovered neighbors in each iteration and across all these iterations, we would be visiting $|E|$ edges totally. The pseudo code for the MaxD-CDS algorithm is given in Figure 15 and a working example of the algorithm is illustrated in Figure 16.

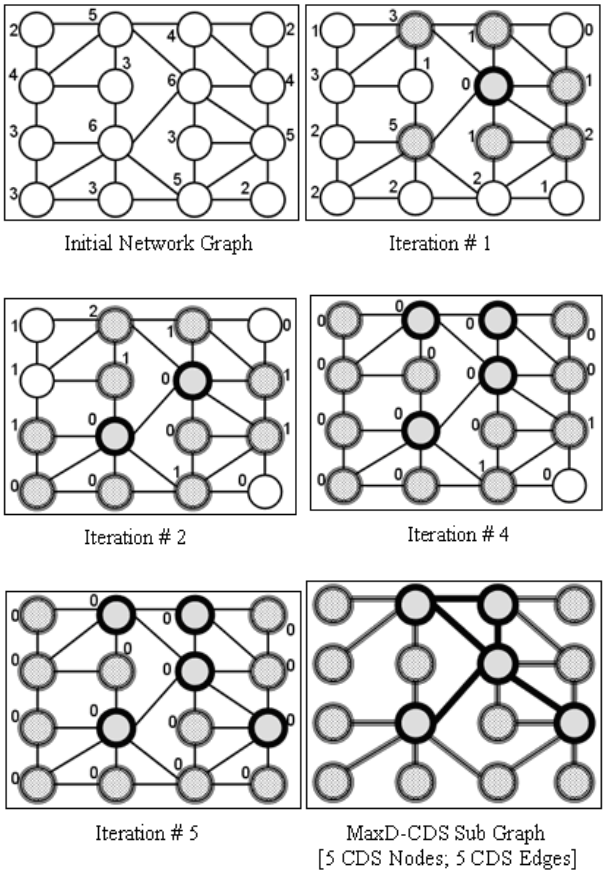


Figure 16: Example to Illustrate the Construction of a Maximum Density (MaxD)-based CDS

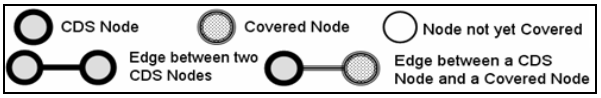


Figure 17: Legend for Figure 16

6 Graph Theory Algorithms for Multi-path Communication in MANETs

Let P_L, P_N and P_Z be the set of link-disjoint, node-disjoint and zone-disjoint *s-d* routes respectively. We use the Dijkstra $O(|V|^2)$ algorithm to determine the minimum hop *s-d* path in a graph of *n* nodes. We assume the *s-d* routes in a multi-path set are used in the increasing order of the hop count. In other words, the *s-d* route with the least hop count is used as long as it exists, then the *s-d* route with the next highest hop count is used as long as it exists and so on. We thus persist with the determined multi-path set of *s-d* routes as long as at least one path in the set exists.

6.1 Algorithm to Determine Link-Disjoint Paths

To determine the set of link-disjoint paths, P_L, (refer Figure 18), we remove all the links that were part of *p* from the graph *G* to obtain a modified graph G^L (V, E^L). We then determine the minimum hop *s-d* path in the modified graph *G'*, add it to the set P_L and remove the links that were part of this path to get a new updated G^L (V, E^L). We repeat this procedure until there exists no more *s-d* paths in the network. The set P_L is now said to have the link-disjoint *s-d* paths in the original network graph *G* at the given time instant. Figure 20 illustrates a working-example of the algorithm to find the set of link-disjoint paths on a static graph.

Input: Graph $G(V, E)$, source vertex *s* and destination vertex *d*
Output: Set of link-disjoint paths P_L
Auxiliary Variables: Graph $G^L(V, E^L)$
Initialization: $G^L(V, E^L) \leftarrow G(V, E)$, $P_L \leftarrow \Phi$
Begin *Algorithm Link-Disjoint-Paths*
1 **while** ($\exists$ at least one *s-d* path in G^L)
2 $p \leftarrow$ Minimum hop *s-d* path in G^L.
3 $P_L \leftarrow P_L \cup \{p\}$
4 $\forall_{edge, e \in p}$ $G^L(V, E^L) \leftarrow G^L(V, E^L - \{e\})$
5 **end While**
6 **return** P_L
End *Algorithm Link-Disjoint-Paths*

Figure 18: Algorithm to Determine the Set of Link-Disjoint *s-d* Paths in a Network Graph.

6.2 Algorithm to Determine Node-Disjoint Paths

To determine the set of node-disjoint paths, P_N, (refer Figure 19), we remove all the intermediate nodes (nodes other than the source vertex *s* and destination vertex *d*) that were part of the minimum hop *s-d* path *p* in the original graph *G* to obtain the modified graph, G^N (V^N, E^N). We determine the minimum hop *s-d* path in the modified graph G^N (V^N, E^N), add it to the set P_N and remove the intermediate nodes that were part of this *s-d* path to get a new updated G^N (V^N, E^N). We then repeat this procedure until there exists no more *s-d* paths in the network. The set P_N is now said to contain the node-disjoint *s-d* paths in the original network graph *G*. Figure 21 illustrates a working example of the algorithm to find the set of node-disjoint paths on a static graph.

Input: Graph $G(V, E)$, source vertex *s* and destination vertex *d*
Output: Set of node-disjoint paths P_N
Auxiliary Variables: Graph $G^N(V^N, E^N)$
Initialization: $G^N(V^N, E^N) \leftarrow G(V, E)$, $P_N \leftarrow \Phi$
Begin *Algorithm Node-Disjoint-Paths*
1 **While** ($\exists$ at least one *s-d* path in G^N)
2 $p \leftarrow$ Minimum hop *s-d* path in G^N.
3 $P_N \leftarrow P_N \cup \{p\}$
4 $\forall_{vertex, v \in p, v \neq s,d, edge, e \in Adj-list(v)}$ $G^N(V^N, E^N) \leftarrow G^N(V^N - \{v\}, E^N - \{e\})$
5 **end While**
6 **return** P_N
End *Algorithm Node-Disjoint-Paths*

Figure 19: Algorithm to Determine the Set of Node-Disjoint *s-d* Paths in a Network Graph.

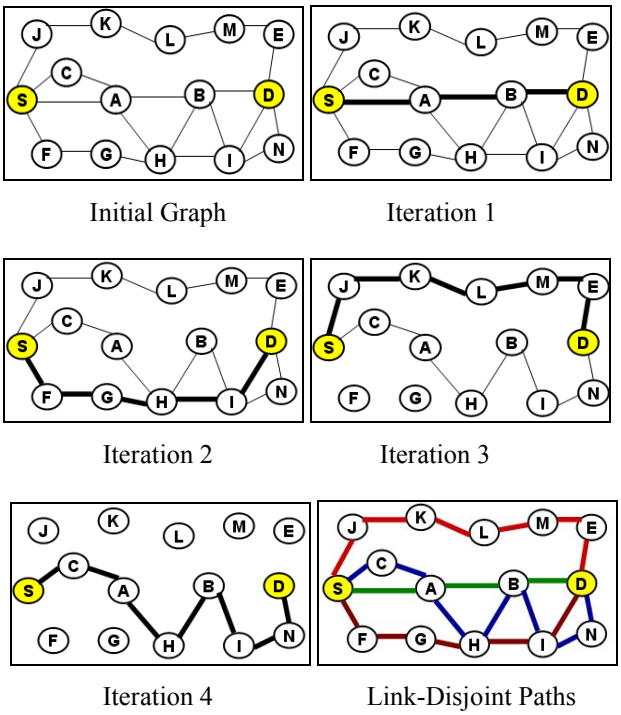
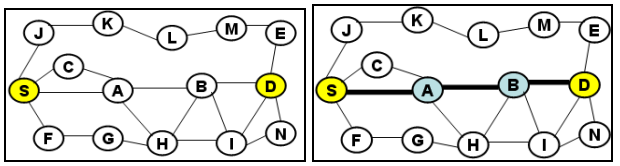
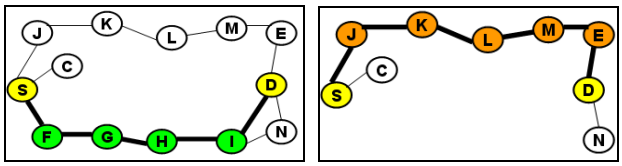


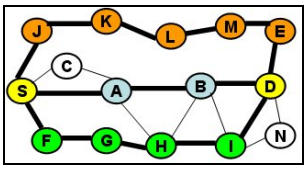
Figure 20: Example to Illustrate the Working of the Algorithm to Find Link-Disjoint Paths.



Initial Graph Iteration 1



Iteration 2 Iteration 3



Node-Disjoint Paths

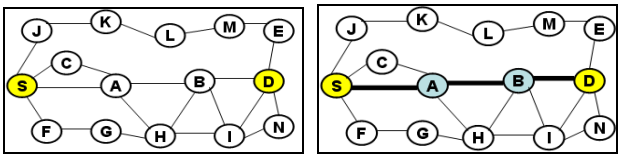
Figure 21: Example to Illustrate the Working of the Algorithm to Find Node-Disjoint Paths.

6.3 Algorithm to Determine Zone-Disjoint Paths

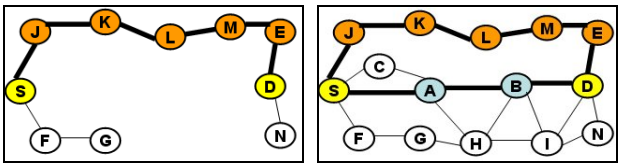
To determine the set of zone-disjoint paths, P_Z, (refer Figure 22), we remove all the intermediate nodes (nodes other than the source vertex s and destination vertex d) that were part of the minimum hop s-d path p and also all their neighbor nodes from the original graph G to obtain the modified graph G^Z (V^Z, E^Z). We determine the minimum hop s-d path in the modified graph G^Z, add it to the set P_Z and remove the intermediate nodes that were part of this s-d path and all their neighbor nodes to obtain a new updated graph G^Z (V^Z, E^Z). We then repeat this procedure until there exists no more s-d paths in the network. The set P_Z is now said to contain the set of zone-disjoint s-d paths in the original network graph G. Note that when we remove a node v from a network graph, we also remove all the links associated with the node (i.e., links belonging to the adjacency list *Adj-list*(v)) whereas when we remove a link from a graph, no change occurs in the vertex set of the graph. Figure 23 illustrates a working example of the algorithm to find the set of zone-disjoint paths on a static graph.

Input: Graph G (V, E), Source vertex s and Destination vertex d
Output: Set of Zone-Disjoint Paths P_Z
Auxiliary Variables: Graph G^Z (V^Z, E^Z)
Initialization: G^Z (V^Z, E^Z) ← G (V, E), P_Z ← Φ
Begin *Algorithm Zone-Disjoint-Paths*
1 **While** (∃ at least one s-d path in G^Z)
2 p ← Minimum hop s-d path in G^Z
3 P_Z ← P_Z U {p}
4 $\forall_{vertex,u\in p,u\neq s,d;\ edge,e\in Adj-list(u)}$ G^Z (V^Z, E^Z)← G^Z (V^Z – {u}, E^Z – {e})
5 $\forall_{vertex,u\in p,u\neq s,d;\ v\in Neighbor(u),v\neq s,d;\ edge,e'\in Adj-list(v)}$ G^Z (V^Z, E^Z)←G^Z(V^Z–{v}, E^Z– {e'})
6 **end While**
7 **return** P_Z
End *Algorithm Zone-Disjoint-Paths*

Figure 22: Algorithm to Determine the Set of Zone-Disjoint s-d Paths in a Network Graph.



Initial Graph Iteration 1



Iteration 2 Zone-Disjoint Paths

Figure 23: Example to Illustrate the Working of the Algorithm to Find Zone-Disjoint Paths

7 Conclusions

The high-level contribution of this paper is the idea of using traditional graph theory algorithms, which have been taught in academic institutions at undergraduate and graduate level, to simulate and study the behavior of the complex routing protocols for unicast, multicast, broadcast and multi-path communication in MANETs. In the Section on Background work, we provided an exhaustive set of background information on the routing protocols that have been proposed for the above different communication problems. In the subsequent sections, we described one or more graph theoretic algorithms for studying each of these communication problems. We chose the Dijkstra algorithm for shortest path routing as the core algorithm and meticulously modified it and/or adopted it to (i) find a solution for the largest bottleneck path and smallest bottleneck path problems, which could be used to determine a sequence of stable routes as well as to (ii) find a set of link-disjoint, node-disjoint or zone-disjoint routes for multi-path communication. We illustrate the use of Prim's algorithm for minimum spanning tree to determine the 'All Pairs Smallest Bottleneck Paths' and also show how the modified version of the Prim's algorithm to determine maximum spanning tree can be used to solve the 'All Pairs Largest Bottleneck Paths' problem. We prove that the path

between any two nodes in the minimum spanning tree is the smallest bottleneck path between those two nodes in the original graph. We also discussed a maximum density-based algorithm to approximate minimum connected dominating sets (MCDS) for MANETs such that the MCDS could be used as a backbone towards broadcast communication for route discoveries and other global communication needs. In addition, we discussed the Steiner tree problem and the Kou et al.'s algorithm to approximate a multicast tree that has the minimum number of links connecting the source nodes to the members of a multicast group. Each of the graph theoretic algorithms discussed in this paper presents the optimal solutions or the best approximations for the appropriate problems they have been suggested for and all of them could be implemented in the most efficient manner using the pseudo code presented and executed in a polynomial run-time.

As a concluding remark, we would like to state that the proposed idea could go a long way in avoiding the significant loss of time faced by student researchers to understand and modify the simulation code for even conducting simple experimental studies. This idea could be adopted to facilitate undergraduate student research in the area of MANETs without requiring the students to directly work on the complex discrete-event simulators. A successful implementation of this idea is the Research Experiences for Undergraduates (REU) site in the areas of Wireless Ad hoc Networks and Sensor Networks, hosted by the Department of Computer Science at Jackson State University, Jackson, MS, USA. The REU site is currently being funded by the U.S. National Science Foundation (NSF) and is accessible through http://www.jsums.edu/cms/reu.

8 Future Research Directions

Graph theory algorithms form the backbone for research on communication protocols for wireless ad hoc networks and sensor networks. This paper lays the foundation for use of several simplistic graph theoretic algorithms (taught at the undergraduate and graduate level) to simulate the behavior of the complex MANET routing protocols. The next step of research in this direction would involve implementing these graph theoretic algorithms in a centralized environment using offline traces of the mobility profiles of the nodes (under a particular mobility model) to generate the mobile graph (i.e., sequence of static graphs representing snapshots of the network topology at different time instants) and compare the performance metrics obtained for the communication structures with that of those obtained for the actual routing protocols when simulated in a discrete-event simulator such as ns-2, GloMoSim and etc. Some of the performance metrics that could be directly compared are the hop count per source-destination path (for unicasting), hop count per source-receiver path (for multicasting), number of links per multicast tree, lifetime per path, lifetime per multicast tree, time between two consecutive route discoveries for link-disjoint, node-disjoint and zone-disjoint routes, number of nodes per CDS, hop count of a source-destination path per CDS and etc. We conjecture that the results obtained for the above performance metrics from the centralized graph theory implementations will serve as the optimal benchmarks to which the results obtained from the actual routing protocols in a discrete-event simulator environment would be actually bounded under. This is because the centralized implementations would assume an ideal medium-access control (MAC) layer that would not offer any interference to constrain the communication.

If the simulations could be conducted in more than one discrete-event simulator, then the results for the performance metrics obtained from the different simulators could be compared to the optimal benchmarks obtained with our theoretical algorithms and could be helpful in identifying the simulator that gives performance closest to the optimum for a particular communication problem (unicast, multicast, broadcast, multi-path) under specific operating conditions. Our proposed approach of using graph theory algorithms to study the MANET routing protocols could also be extended to wireless sensor networks, wherein we can use the tree and CDS construction algorithms to study the data gathering protocols.

Acknowledgments

Research was sponsored by the U. S. National Science Foundation grant (CNS-0851646) entitled: "REU Site: Undergraduate Research Program in Wireless Ad hoc Networks and Sensor Networks. The material for this journal paper evolved from the tutorial slides developed by the author to simulate the routing protocols for mobile ad hoc networks with centralized and distributed implementations of the appropriate graph theory algorithms. The views and conclusions in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the funding agency.

References

[1] K. M. Alzoubi, P.-J Wan and O. Frieder, "Distributed Heuristics for Connected Dominating Set in Wireless Ad Hoc Networks," *IEEE / KICS Journal on Communication Networks*, Vol. 4, No. 1, pp. 22-29, 2002.

[2] S. Butenko, X. Cheng, D.-Z. Du and P. M. Paradlos, "On the Construction of Virtual Backbone for Ad Hoc Wireless Networks," *Cooperative Control: Models, Applications and Algorithms*, pp. 43-54, Kluwer Academic Publishers, 2002.

[3] S. Butenko, X. Cheng, C. Oliviera and P. M. Paradlos, "A New Heuristic for the Minimum Connected Dominating Set Problem on Ad Hoc Wireless Networks," *Recent Developments in Cooperative Control and Optimization*, pp. 61-73, Kluwer Academic Publishers, 2004.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, "Introduction to Algorithms," 2nd Edition, MIT Press/ McGraw-Hill, September 2001.

[5] K. Fall and K. Varadhan, "ns notes and documentation," The VINT Project at LBL, Xerox PARC, UCB, and USC/ISI, http://www.isi.edu/nsnam/ns, August 2001.

[6] B. Hofmann-Wellenhof, H. Lichtenegger and J. Collins, *Global Positioning System: Theory and Practice*, 5th ed., Springer, September 2004.

[7] N. T. Javan and M. Dehghan, "Reducing End-to-End Delay in Multi-path Routing Algorithms for Mobile Ad hoc Networks," *Proceedings of the International Conference on Mobile Ad hoc and Sensor Networks* (MSN 2007), Lecture Notes in Computer Science (LNCS) 4864, pp. 703 – 712, December 2007.

[8] D. B. Johnson, D. A. Maltz and J. Broch, "DSR: The Dynamic Source Routing Protocol for Multihop Wireless Ad Hoc Networks," *Ad Hoc Networking*, edited by Charles E. Perkins, Chapter 5, pp. 139 – 172, Addison Wesley, 2001.

[9] L. Kou, G. Markowsky and L. Berman, "A Fast Algorithm for Steiner Trees," *Acta Informatica*, vol. 15, no. 2, pp. 141-145, 1981.

[10] F. Kuhn, T. Moscibroda and R. Wattenhofer, "Unit Disk Graph Approximation," *Proceedings of the Joint Workshop on Foundations of Mobile Computing*, pp. 17-23, October 2004.

[11] S. Lee and M. Gerla, "Split Multipath Routing with Maximally Disjoint Paths in Ad Hoc Networks," *Proceedings of the IEEE International Conference on Communications*, Vol. 10, pp. 3201-3205, 2001.

[12] N. Meghanathan, "A Beaconless Node Velocity-based Stable Path Routing Protocol for Mobile Ad hoc Networks," *Proceedings of the IEEE Sarnoff Symposium Conference*, Princeton, NJ, March 30-April 1, 2009.

[13] N. Meghanathan, "Exploring the Stability-Energy Consumption-Delay-Network Lifetime Tradeoff of Mobile Ad Hoc Network Routing Protocols," *Journal of Networks*, vol. 3, no. 2, pp. 17 – 28, February 2008.

[14] N. Meghanathan, "Benchmarks and Tradeoffs for Minimum Hop, Minimum Edge and Maximum Lifetime per Multicast Tree in Mobile Ad hoc Networks," *International Journal of Advancements in Technology*, vol. 1, no. 2, pp. 234-251, October 2010.

[15] E. Royer and C. E. Perkins, "Multicast Operation of the Ad-hoc On-demand Distance Vector Routing Protocol," *Proceedings of the 5th ACM/IEEE Annual Conference on Mobile Computing and Networking*, pp. 207-218, Seattle, USA, August 1999.

[16] C. E. Perkins and E. M. Royer, "Ad Hoc On-demand Distance Vector Routing," *Proceedings of the 2nd Annual IEEE International Workshop on Mobile Computing Systems and Applications*, pp. 90 – 100, February 1999.

[17] T. Ozaki, J-B. Kim and T. Suda, "Bandwidth-Efficient Multicast Routing for Multihop, Ad hoc Wireless Networks," *Proceedings of the IEEE INFOCOM Conference*, vol. 2, pp. 1182-1192, Anchorage, USA, April 2001.

[18] W. Su, S-J. Lee and M. Gerla, "Mobility Prediction and Routing in Ad hoc Wireless Networks," *International Journal of Network Management*, vol. 11, no. 1, pp. 3-30, 2001.

[19] Z. Ye, S. V. Krishnamurthy and S. K. Tripathi, "A Framework for Reliable Routing in Mobile Ad Hoc Networks," *Proceedings of the IEEE International Conference on Computer Communications*, 2003.

[20] X. Zeng, R. Bagrodia and M. Gerla, "GloMoSim: A Library for Parallel Simulation of Large-Scale Wireless Networks," *Proceedings of the 12th Workshop on Parallel and Distributed Simulations*, Banff, Alberta, Canada, May 26-29, 1998.