Topologies for Ad-Hoc Networks Utilizing Directional Antennas with Restricted Fields of View

by
Brian C. Anderson

B.S. Electrical Engineering and Computer Science
Massachusetts Institute of Technology, 2004

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

Master of Engineering in Electrical Engineering and Computer Science
at the
Massachusetts Institute of Technology

September 2005

Signature of Author:_____

Brian C. Anderson
Department of Electrical Engineering and Computer Science
October 6, 2005

Certified by: _____

Steven G. Finn
Principle Research Scientist, Deptartment of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by:_____

Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Topologies for Ad-Hoc Networks
Utilizing Directional Antennas with
Restricted Fields of View

by
Brian C. Anderson

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

September 2005

ABSTRACT

ORCLE (Optical/RF Combined Link Experiment), is an airborne network in which aircraft have multiple directional antennas that are restricted in their pointing direction. A pair of aircraft in ORCLE can be linked if they both have an antenna pointing at each other. Four topology algorithms, which coordinate the pointing of the antennas and attempt to maximize a connectedness metric, are presented and analyzed using a custom 2D simulation platform. Three of the algorithms are based on the Relative Neighbor Graph (RNG): the first constrains the RNG to requirements of the ORCLE network, the second augments the constrained RNG with edges from the Delaunay Triangulation, and the third algorithm tries to improve on the second by adding edges to reduce the diameter. The final algorithm uses a novel concept of overlapping sets of nested convex hulls to select the links of the network. All algorithms are stateless and interface with a Target Transition Layer, which gradually migrates topologies to prevent a large number of edges from being lost simultaneously. Scenes with varying node density, number of terminals per node, fields of view, and re-targeting delays are used to test the algorithms against a wide range of possible situations.

Table of Contents

Index of Tables

Index of Figures

1. Introduction

As part of the ORCLE (Optical & RF Combined Link Experiment) project [1] [2] , this thesis focuses on the problem of creating topology algorithms that interconnect aircraft to form an airborne network.

Each aircraft in ORCLE has one or more antennas, or terminals, that can be directed at a terminal of another aircraft to form a direct point-to-point communication link. The terminals are range limited and are hybrids of highly directional Free Space Optical (FSO) transmitters and Radio Frequency (RF) transmitters. The FSO transmitters have a higher bandwidth and longer range in comparison to the RF transmitters. RF transmitters, however, are more immune to outages due to atmospheric obstructions such as clouds, fog, or the wake of a jet engine. Integrating the strengths of both transmitter types is the premise of ORCLE. Unfortunately, neither transmitter type can propagate its signal through the body of the plane, restricting the field of view in which a terminal can effectively point.

Networks like ORCLE have a number of benefits when compared to conventional broadcast ad-hoc networks. These benefits include a decrease in power usage and an increase in range and bandwidth gained by focusing the communication signal into a beam. Also, the resilience to jamming and eavesdropping of directional links may be useful for sensitive military applications.

These benefits come with additional complexity however, since the ORCLE network presents a combination of topology issues that are not typical of most wireless networks. These issues include the high mobility of aircraft nodes, the directionality and long range of the terminals, the restricted fields of view of terminals, the limited number of terminals per node, and link failures due to atmospheric conditions.

In this thesis we assume all nodes know the positions and velocities of all other nodes through a separate communication medium. Using this information, each node is to run a topology algorithm that determines what nodes each of its terminals should target. To ensure that nodes will arrive at the same global topology independently, the topology algorithms considered in this thesis are deterministic.

1.1. Thesis Goal

Even though ORCLE has various potential applications, ranging from a self-contained network to a backbone network that relays data between external users, this thesis focuses on forming a self-contained network in which aircraft communicate amongst themselves.

Within this context, the goal of this thesis is to create point-to-point topology algorithms that maximize a connectedness metric of all the nodes, while at the same time, attempting to minimize the number of terminals required per node. Reducing the number of terminals per node will help reduce production costs, and forming well-connected topologies will hopefully result in robust networks that can survive link outages. However, since these are competing goals, trade-offs must be made.

To objectively measure how effectively the topology algorithms create well-connected topologies, we will define two intermediate topological graphs: the link graph and the functional graph. The link graph contains edges between nodes that are within range and have terminals pointing at each other, and the functional graph has edges from the link graph removed to simulate random outages. Our connectedness metric is defined as the average fraction of node pairs connected, either directly or indirectly, in the functional graph. It is calculated as the average number of connected node pairs in the functional graph over the course of a simulation divided by the total possible number of node pairs. A more detailed description of the link graph, functional graph, and connectedness metric can be found in section 2.5.

1.2. Previous Work

Most ad-hoc wireless topology research has assumed that nodes have omni-directional/broadcast transmitters, where nodes can communicate with all other nodes within a radius. A communication link in ORCLE, however, is point-to-point and limited by line of sight. Also, previous work in ad-hoc networks has focused on energy efficient transmission strategies intended for small battery powered nodes [3][4]. These algorithms normally favor short links, but ORCLE will not run on nodes with this energy constraint, so longer links may be favorable in some cases.

Some work has been done on topologies for networks with directional antennas, but not much is applicable to ORCLE. For example, Gurumohan and Hui [5] consider a few topologies for degree constrained free-space optical networks, but do not consider field of view constraints. Similarly, in [6], Liu, Vishkin, and Milner propose a method to initially connect a degree constrained free-space optical network, but also do not account for field of view constraints.

The Relative Neighborhood Graph (RNG) [7] has been used as a link topology strategy for various types of ad-hoc networks [8][9][10][11], and is also applicable to the ORCLE network. For the simplified model of ORCLE described in section 2, the RNG is guaranteed to generate a fully connected link graph if all aircraft in the network have 5 terminals symetrically placed, each with at least 120° fields of view. RNG is therefore one of the algorithms analyzed and is used as a baseline for comparing all other algorithms presented.

Also used extensively in this thesis is the Delaunay Triangulation (DT), which is known to maximize the minimum interior angle of all triangles over the set of all possible triangulations [12]. (i.e. they tend to avoid "skinny" triangles.) The DT has been used in other ad-hoc networks before, and is also applicable to the ORCLE network.

Appendix B of this thesis explains the RNG and the DT and their properties in more detail.

1.3. Thesis Organization

Chapter 2 describes a simplified model of the ORCLE network used in this thesis. Chapter 3 provides an analysis of potential topologies, given the simplified model, and outlines the methodology used to test proposed topology algorithms. Chapter 4 describes the four topology algorithms proposed in this thesis. Results and discussion of all the simulations is presented in Chapter 5. Chapter 6 concludes the thesis and proposes algorithm improvements and directions for future research. Further information on the simulator, the RNG, the DT, and algorithm implementations can be found in the appendices.

2. Simplified Thesis Model

In this section, we describe the simplified model of the ORCLE system, which we believe to be illustrative of many key issues in a real airborne ORCLE network. The physical world is modeled as a square scene with the parameters listed in Table 2.a, each of which will be described in this section. For this work we assume all aircraft are flying around the same altitude and can be represented using a 2 dimensional model.

Table 2.a: Model Parameters

Parameter	***Description***
D	width and height of the scene in kilometers
N	number of nodes in the scene
N_i	denotes the i^{th} node in the scene (i = [0,N-1])
T	number of terminals per node
$T_{i,j}$	denotes the j^{th} terminal on N_i (j = [0, T-1])
R	range in kilometers of all terminals
$P(N_i,t)$	denotes the position of N_i at time t
$V(N_i,t)$	denotes the velocity vector of N_i at time t. (dx/dt, dy/dt)
$FOV_{i,j}$	angular field of view of $T_{i,j}$ in degrees
$nDir(N_i,t)$	denotes the facing direction of the velocity of N_i at time t
$fDir(FOV_{i,j})$	denotes the facing direction of $T_{i,j}$ relative to $nDir(N_i,t)$. bisects $FOV_{i,j}$
$target(T_{i,j}, t)$	denotes the node at which $T_{i,j}$ is pointing at time t.
W	delay in seconds before a terminal establishes a new link
$lockedOn(T_{i,j})$	indicates whether $T_{i,j}$ has had the same target for > W seconds
$dist(N_A,N_B,t)$	$\|\|P(N_A,t)-P(N_B,t)\|\|$ = the distance between N_A and N_B at time t.
$PR_L(N_A,N_B,t)$	probability that a link between N_A and N_B is functional at time t

2.1. Scene Model

To test the proposed topology algorithms, we consider series of both static and dynamic scenes. Static scenes are snapshots of randomly placed and rotated nodes and are meant to test the basic functionality of the algorithms. The dynamic scenes change over time. Half the nodes fly in random circular paths and the other half fly in random straight line paths. The dynamic scene is used to test how well algorithms adapt their topologies over time. Excessive re-targeting of the terminals can adversely affect connectivity due to re-targeting delays described in Section 2.3.

2.1.1. Static Scenes

In the static scenes, nodes have uniformly distributed stationary positions within the scene and uniformly distributed rotational offsets between 0 and 2π.

2.1.2. Dynamic Scenes

In the dynamic scenes, half the nodes follow circular paths and the other half follow straight-line paths from one end of the scene to another. All nodes are created such that they fly at random speeds between 160 kmph and 640 kmph.

The straight flying nodes have a constant direction and velocity. Once a straight flying node exits the scene, it immediately re-enters from a random location along the perimeter of the scene and is assigned a random new velocity.

For the circular paths, each node, N_i, follows a random circular path within the scene dimensions such that:

$$P(N_i,t)=\{\ x0_i+r_i\cos(\omega_i t+\alpha_i);\quad y0_i+r_i\sin(\omega_i t+\alpha_i)\ \}$$

The velocity is implicitly defined by the derivative of $P_i(t)$ with respect to t.

$$V(N_i,t)=\{\ -\omega_i r_i\sin(\omega_i t+\alpha_i);\quad \omega_i r_i\cos(\omega_i t+\alpha_i)\ \}$$

The parameters in the equation are defined and initialized to random numbers that are uniformly selected from the ranges specified in Table 2.1.a.

Table 2.1.a: Circular Path Parameters

Symbol	***Range***	***Description***
r_i	[10, D/4]	radius of the path for N_i
$x0_i$	[r_i, D-r_i]	x coordinate of the origin of the circular path for N_i
$y0_i$	[r_i, D-r_i]	y coordinate of the origin of the circular path for N_i
ω_i	[160/r, 640/r] / 3600	angular velocity of N_i in radians/second. linear velocity = [160 kmph, 640 kmph] or ~[100 mph, 400 mph]
α_i	[0, 2π]	angular offset at time t=0

These scene types tend to be uniform clusters of nodes within a square. Imbalanced scenes which are oddly shaped and have clusters, protrusions, concavities, or holes are not addressed in this thesis.

2.2. Node and Terminal Model

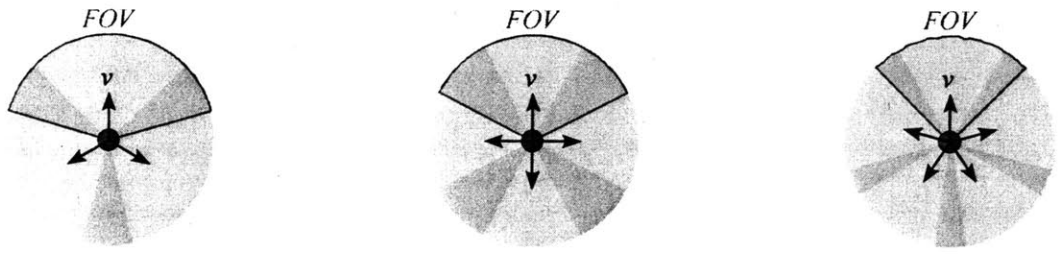


Figure 2.2.a: Examples of Facing Directions and Fields of View for T = 3,4, 5 where v = nDir($N_{i,0}$)

Aircraft nodes are modeled as zero-dimensional point objects where nDir(N_i,t) is defined as the direction in which the nose is facing. For dynamic scenes, nDir(N_i,t) is assumed to be equal to the direction of the node's velocity.

Each terminal, $T_{i,j}$, is located in the same position as the its aircraft node and can point in any direction

within its field of view, $FOV_{i,j}$. Example FOVs are depicted as shaded sectors in Figure 2.2.a, with the first terminal's FOV outlined. Each FOV is defined by an angle which is bisected by a facing direction, $fDir(T_{i,j},t)$. $fDir(T_{i,j},t)$ is relative to $nDir(N_i,t)$ and is defined for all i,j as $fDir(T_{i,j},t) = nDir(N_i,t) + 360*j/T$. This means that all terminals face equally apart such that adjacent terminals are separated by 360/T degrees.

The zero-dimensional model for aircraft and terminals uniformly distributed around the aircraft is a highly simplified compared to the real world. The terminals in ORCLE are restricted from placement on many parts of the plane due to physical or mechanical design constraints. For example, it is unlikely that terminals can be placed on the wings. Obstructions due to the plane's body can restrict a terminal's FOV in drastically different ways, depending on the shape of the body and the placement of the terminal.

Although it may not be feasible for terminals to be placed facing equally apart on a real aircraft, Figure 2.2.b demonstrates 3 terminal placement schemes for T=4 that would result in facing directions and FOVs that closely resemble the model.

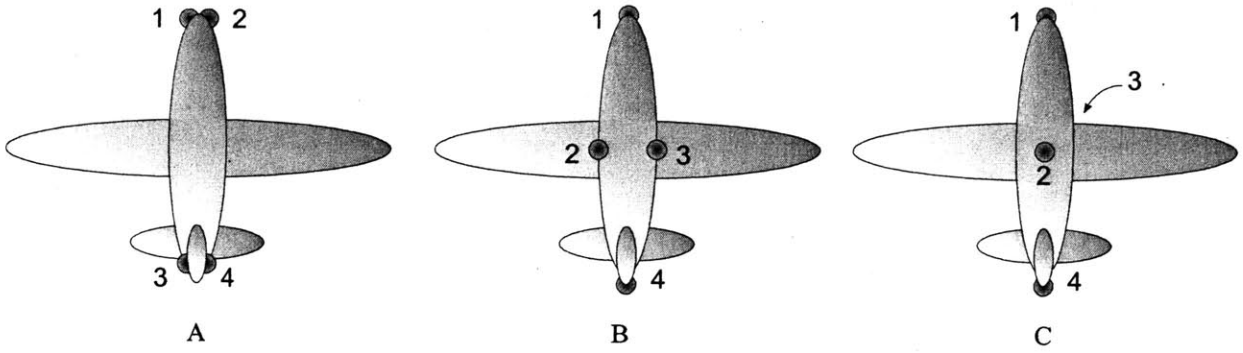


Figure 2.2.b: 3 Terminal Placement Options for T=4

In scheme A, terminals 1 and 2 are fairly unrestricted. Terminals 3 and 4, however, have FOVs that are significantly restricted due to their placement by the tail wing.

In scheme B, terminals 2 and 3 are placed high on the body of the plane so that their FOVs are less obstructed by the main wings. This scheme works until the plane banks left or right, in which case either terminal 2 or 3 will become significantly obstructed in the horizontal plane by the wing and/or body.

In scheme C, terminals 2 and 3 are placed on the top and bottom of the body. This scheme only works if the terminals have FOVs greater than 180°, such that terminals 2 and 3 can point along the horizontal plane even though their facing direction is up/down. The highlight of this scheme is that as the plane banks, terminal 2 will cover the side of the horizontal plane that is obstructed for terminal 3, and vice versa. In this way, terminals 2 and 3 alternate being the "left" and "right" terminal in the model. Therefore, regardless of the plane's orientation, the terminals provide 360°-coverage of the area around the plane.

2.3. Link Model

Even though ORCLE uses hybrid FSO/RF terminals, our model assumes terminals only have a single radio type, and therefore a single range. This is done to simplify the problem while still keeping many of the key attributes.

A **link** is formed between two nodes when they are within **range**, R, and each has a terminal pointing at the other. We will refer to such nodes as linked. Communication between nodes is not necessarily

guaranteed if they are linked, however. **Link failures** affect whether a link is functional or not functional. Linked nodes can only send data to each other over the link if it is up.

let $PR_L(N_A, N_B, t)$ be the probability that a link between N_A and N_B is functional.

Although, link outages may be correlated in reality, this thesis models them as probabilistically independent events.

Two nodes can also be linked indirectly by a **path**, which is a continuous route of links that has the two nodes as it's endpoints. A path is functional if all links along the path are functional.

To determine where terminals should point each node runs a **topology algorithm**. It is assumed all nodes have accurate location and velocity information of every other node. This allows all nodes to independently arrive at the same topology. The topology algorithm sets the pointing direction of a terminal by indicating what node it should target.

let target($T_{A,i}$, t) = N_B when $T_{A,i}$ is attempting to point at N_B at time t.

Once two terminals have started pointing at each other, however, a link is not established immediately. There may be a **targeting delay** due to terminal repositioning and discovery.

let W be the targeting delay in seconds.

let lockedOn($T_{A,i}$) indicate whether $T_{A,i}$ has had the same target for > W seconds.

2.4. Link Graph and Functional Graph

To help analyze how well pointing algorithms connect the nodes, we will introduce the concept of a **link graph** and a **functional graph**. The link graph is the a graph of the linked nodes, and the functional graph is a graph of nodes and their functional links. For this thesis, we are primarily concerned with the connectedness of the functional graph.

The link graph will have N nodes, and undirected edges between nodes that are within each other's FOV and range R, and whose terminals have been targeting each other for more than W seconds. At time $t=0$, it is assumed that all terminals have been pointing in the same direction for greater than W seconds.

The functional graph will be a sub-graph of the link graph: all nodes identical, but with some of the edges removed to model link failures. In other words, only functional links will be edges in the functional graph.

Links transition into up and down states according to a Bernoulli process that transitions from down to up with probability PR_L and transitions from up to down with probability (1-PR_L) every simulation time step. This means the average time a link stays up or down depends on the time step length.

2.5. Connectedness Metric

The connectedness metric is what we are trying to maximize. It is the average number of connected node pairs in the functional graph over the maximum possible number of node pairs. It is defined explicitly as follows:

For a given simulation of N nodes over a time period of τ:

let *ConnectedPairs(t)* be the number of pairs of nodes between which there exists a functional path at time t.

$$AverageConnectedPairs = \lim_{\tau \to \infty} \frac{1}{\tau} \int_0^{\tau} ConnectedPairs(t)\,dt$$

let *MaxConnectedPairs* be the number of pairs which would exist if the nodes were fully connected.

$$MaxConnectedPairs = \frac{1}{2} N(N-1)$$

Connectedness is then the ratio of *AverageConnectedPairs* over *MaxConnectedPairs*.

$$Connectedness = \lim_{\tau \to \infty} \frac{2}{\tau \cdot N(N-1)} \int_0^{\tau} ConnectedPairs(t)\,dt$$

3. Topology Algorithm Considerations

Here, we outline some of the desirable properties in a topology and present some methods for analytically determining if a topology can be formed. Section 3.1 introduces concepts of topology diameter and stability, which may affect the connectedness metric we are trying to maximize. Section 3.2 establishes rules for determining if a set of links can be made by a node, and is used in later sections to show certain properties of the algorithms.

3.1. Topology Stability and Diameter

Although not measured directly as metrics for an algorithm's quality, topology diameter and stability may indirectly affect the connectedness metric.

The diameter is the worst-case minimum number of hops between two nodes. Decreasing the diameter should help increase connectedness, since the number of hops between nodes will be reduced, lowering the probability that any one path has an outage.

We would like to stay away from unstable topology algorithms that change all terminal targets rapidly, since the terminals can end up spending more time trying to establish links than being linked. Constantly changing topologies might also result in hard to maintain routing tables, which could make routing inefficient.

The effects on connectivity may still be felt in less extreme cases. For example, even if the frequency of topology changes is low, but the number of simultaneous target changes is high, there will be moments where many links will be out as terminals try to establish new links, potentially reducing the connectedness drastically.

The effect of many simultaneous target changes may be compounded in real world situations, where position information could become unsynchronized across nodes. Unsynchronized position information can result in some nodes calculating a different topology than the rest. If the resulting topologies vary drastically, many terminals could end up targeting aircraft that are not pointing back. Addressing unsynchronized node information, however, is beyond the scope of this thesis.

In considering and developing topology algorithms, we explored the effects of stable topologies and topologies with smaller diameters and found that they can improve performance.

3.2. Rotationally Independent Links

Rotationally independent links are a set of links for a given node that can be formed independently of that node's orientation. Rules for rotationally independent links are used in this thesis to analytically show whether or not a given topology can be formed. We start by defining three variables: Δ, α, and β.

let Δ be the degrees the FOV falls short of having all areas visible by two terminals.

$$\Delta = 720/T - FOV$$

let β be the maximum angle between radially adjacent links in a set.

let $\alpha = 360 - \beta$.

Figure 3.2.a illustrates some example α's and β's. The two edges forming α and β will be referred to as the critical edges.

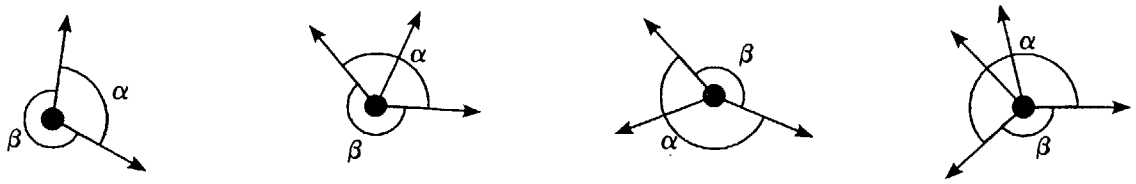


Figure 3.2.a: Example α's and β's

There is a critical minimum angle for α which guarantees that both critical edges can be made simultaneously, given that the *n-2* other links can also be made. This critical angle is the angle of the sector which is not visible by any terminals when *n-1* adjacent terminals are removed. Using recursion, we can determine if all links are likable or not. Table 3.2.a lists the critical angles of α in terms of Δ.

Table 3.2.a: Minimum α to Guarantee Linkable Critical Edges

#Links in Subset	***T = 3, Δ = 240-FOV°***	***T = 4, Δ = 180-FOV°***	***T = 5, Δ = 144-FOV°***
2	Δ°	Δ°	Δ°
3	120+Δ°	90+Δ°	72+Δ°.
4	N/A	180+Δ°	144+Δ°
5	N/A	N/A	216+Δ°

Assuming a node's rotation is uniformly distributed between 0 and 2π, we can also express the probability that a set of links for a given node has of being made. Table 3.2.b gives the probabilities that both links forming α can be made if the minimum α is not met. The expressions are derived from the percentage of the rotational period (360/T°) over which a sector α° wide overlaps at least *n* separate FOVs, where *n* is the number of links in the subset.

Table 3.2.b: Probabilities of Linking Critical Edges if Minimum α is not met.

#Links in Subset	***T = 3, Δ = 240-FOV°***	***T = 4, Δ = 180-FOV°***	***T = 5, Δ = 144-FOV°***
2	α/Δ	α/Δ	α/Δ
3	max(0, (α-Δ)/120)	max(0, (α-Δ)/90)	max(0, (α-Δ)/72)
4	0	max(0, (α-Δ-90)/90)	max(0, (α-Δ-72)/72)
5	0	0	max(0, (α-Δ-144)/72)

4. The Topology Algorithms

Although it is possible to enumerate all possible topologies and pick an optimal solution, the computation time of such an algorithm is impractical as N grows large. The four topology algorithms we consider thus rely on heuristics to generate their preferred topologies.

The first algorithm is a derivative of a geometric topology algorithm called the Relative Neighborhood Graph (RNG) [7]. Two other algorithms are extensions of the RNG, and the last one is based on nested convex hulls [13].

Sections 4.1 through 4.4 describe each of the 4 topology algorithms proposed in this thesis in a general fashion. Implementation details for each algorithm can be found in Apendix D.

Section 4.5 describes a target transition layer, which is used with all the algorithms in this thesis. Its purpose is to help improve connectivity by preventing nodes from re-targeting too many terminals simultaneously.

4.1. The Constrained RNG Topology (CRNG)

For our first algorithm, we will start off with the RNG and remove edges to meet the range and field of view constraints. By starting with the RNG topology, the hope is to localize topology changes due to node translation and node entries/exits that will be encountered in the dynamic scenes. Also, since RNG does not take into account node orientation, we also start off with a base of rotationally independent links. For a more in-depth explanation of RNG and it's properties, see Appendix B.

We use an algorithm similar to the one described by Supowit in [14] to calculate the RNG topology, which starts with all the edges of the DT and removes edges that violate RNG rules. The DT is calculated using a method described in [15], by first projecting the set of 2D positions (x,y) onto a 3D parabola (x, y, x^2 + y^2). After taking the 3D convex hull of the resulting projection, the DT is composed of the edges of the faces whose normals have a negative z component.

Each edge of the DT is then considered for removal. A naiive method would test each edge with every node for intersection with the edge's area of exclusion. To speed things up, the nodes are sorted in ascending order of their x value, and only nodes within a plausible interval of x are tested to see if they are within the RNG area of exclusion. A maximum and minimum y value is also calculated for quick pruning of nodes above or below the area of exclusion. The left and right values of the x interval and the top and bottom values of the y pruning variables are calculated as follows:

> let d = distance between the endpoint nodes of the edge being considered for removal
> let xmin, xmax, ymin, ymax = the corresponding values of the endpoint node positions
> left = xmin – d; right = xmax + d; top = ymin-d; bottom = ymax+d;

The runtime of this algorithm is dominated by the 3D convex hull step, which can be calculated optimally in O(NlogN) time using either the quick-hull or divide and conquer [16] methods. For simplicity, however, we construct the 3D convex hull using an incremental method [16] which takes $O(N^2)$ time.

4.1.1. Constraining the RNG Topology

The RNG by itself nearly meets the constraints of our model for many parameterizations, and is in fact guaranteed to create valid topologies in our model if $T \geq 5$ and $FOV \geq 120$ according to the rotational independence rules from section 3.2. (Each link would be separated by at least 60 degrees.) However, for $T<5$ and/or $FOV<120$ some edges may have to be removed to ensure no edge violates degree or FOV constraints.

To remove edges from the RNG that violate constraints, we first consider edges that are between two violating nodes (nodes that cannot assign terminals to all their edges). If removing the edge fixes both nodes, it is removed. We then consider all violating nodes from highest degree to smallest degree. For each node, we remove the edge connected to the highest degree node on the other end that will make the node meet constraints. If removing any single edge cannot make the node meet constraints, we remove the edge connected to the highest degree node. All ties are arbitrarily broken by choosing the edge with the node that is farthest from the center of the scene. The result may not leave a connected graph, however in our work this is generally not the case.

4.1.2. RNG Validation

Visual inspection of the RNG output and comparison with previous work was performed. For scenes with 1000 nodes, the average node degree hovered around 2.5, and the maximum node degree rarely exceeded 4, which matches the results reported in [4].

Furthermore, versions of the RNG and DT algorithms which use enumeration were implemented for comparison. These algorithms consider all possible node pairs / triplets for edge addition by testing to see if any node is within the area of exclusion. Both the enumeration algorithms and algorithms used in this thesis produce identical results for more than 1000 randomly generated graphs of 100 nodes. The enumeration RNG c++ code is included here for reader verification.

```
void RNGnaive(Scene &scene, fsoGraph &graph)
{
    graph.removeAllEdges();

    for( fsoGraph::node_iterator n1 = graph.nodesBegin(); n1 != graph.nodesEnd(); n1++ )
    for( fsoGraph::node_iterator n2 = n1; n2 != graph.nodesEnd(); n2++ )
    {
        if( *n1==*n2 ) continue;

        //calculate distance between n1 and n2 squared
        Vector2d x1( (*n1)->n()->x ), x2( (*n2)->n()->x );
        double d2 = (x1-x2).lengthSquared();

        //test to see if any nodes are within the luna
        bool fNodeInLuna = false;
        for( fsoGraph::node_iterator n3 = graph.nodesBegin(); n3 != graph.nodesEnd();  n3++ )
        {
            if( *n1==*n3 || *n2==*n3 ) continue;

            Vector2d &x = (*n3)->n()->x;
            if( d2 > ( x - x1 ).lengthSquared() &&
                d2 > ( x - x2 ).lengthSquared()  )
            {
                fNodeInLuna=true;
                break;
            }
        }

        if(!fNodeInLuna) graph.newEdge(*n1,*n2);
    }
}
```

4.2. The Augmented RNG Topology (ARNG)

RNG is a simple and well established algorithm which may not meet the degree constraints of ORCLE. While CRNG does meet the constraints, it can leave many terminals unused. To utilize these unused terminals, a hybrid of the CRNG and DT is proposed. We first start with the CRNG topology and iteratively add edges from the DT that do not violate constraints. The method by which they are added is described in 5.5.2, however we will first discuss other existing methods that produced degree constrained graphs from the DT and why they are not suitable for the requirements of our model.

4.2.1. Previous methods of reducing node degrees of the DT

Other papers have proposed using the Yao structure [3] to reduce the node degree of the DT, which, in this context, would mean splitting the entire 360 FOV of a node into T equal sectors. Each node choses the shortest edge of the DT in each sector, and an edge remains only if it is chosen by both it's endpoint nodes. The Yao structure will produce valid graphs since each terminal can only be assigned one edge within a sector it is known to be able to point in. However, it is more restrictive than our model allows, since it may be possible for links to be formed within the same sector, assuming the FOV of the terminals are greater than 360/T.

Another method used in [5] attempts to make all node degrees approach identical values by iteratively adding and removing edges in the DT. Nodes that exceed constraints have a random edge removed and nodes that do not exceed constraints have a random edge added, if possible. A node removing or adding one of it's edges will affect the degree of the node it is connecting to, possibly causing it to exceed or fall under the constraints. In the limit, all nodes should approach identical degrees. This method produces unstable topologies, which require significant re-targeting of terminals, and is not used in this thesis.

4.2.2. Proposed method

The ARNG starts off with the edges of the CRNG. Nodes are then considered for edge addition in sorted order from lowest degree to highest degree, to give low degree nodes priority. Ties for nodes with identical degree will be broken by prioritizing nodes farther from the center of the scene. The edge in the DT of the node being considered that does not violate any constraints of its endpoint nodes is added. (Constraints are tested using the algorithm from 3.2) If there is more than one non-violating edge, the edge connected to the lowest degree node on the other end is given priority. Furthermore, if there is a tie for lowest degree node, we arbitrarily select the edge with the node farthest from the center of the scene.

4.3. The Augmented RNG Topology with Cross Links (ARNGx)

The CRNG and ARNG algorithms are based on the RNG and DT graphs which have a $O(\sqrt{n})$ diameter for n randomly placed nodes [17]. Graphs of degree constrained nodes, however, can ideally have logarithmic diameters [18]. There are a number of topologies for degree constrained nodes which achieve logarithmic diameters, such as DeBruijn [19], Shuffle-Ring [20], and Caley [21] graphs. Unfortunately, there is no obvious way to map any of these graph types to nodes in the ORCLE model while still meeting range and field of view constraints. Although we may not be able to achieve logarithmic diameters, the ARNGx algorithm at least attempts to make improvements on the diameter of the ARNG.

4.3.1. The Algorithm

The ARNGx algorithm adds mostly non-planar cross links to connect nodes that are many hops apart in the ARNG graph using a 3 step algorithm.

First, the algorithm takes the CRNG and attempts to make all nodes at least degree 2 by adding edges from the DT to nodes of degree 1. The next step involves adding the cross links before the algorithm finishes up by augmenting the graph with edges from the DT. Making all nodes degree 2 from the start prevents nodes from being stranded with only one edge when we add the cross links, which was a problem with early iterations of the algorithm.

The cross links are added subdividing the graph into clusters of nodes that are within a constant number of hops from each other and then interconnecting adjacent clusters. The following sections explain the method we used to subdivide and interconnect distant clusters.

4.3.2. Recursive Area Subdivision

There are numerous ways of dividing the nodes. We can blindly split the scene recursively into 4 equal quadrants, but this may not handle unbalanced scenes well, where one quadrant has significantly more nodes than the other quadrants, resulting in unbalanced trees or divisions with unequal number of nodes.

Using an idea borrowed from artificial intelligence called K-D trees [22], we can take such imbalance into account. K-D trees are binary trees that split the graph in half such that there are an equal number of nodes on each side of the tree. The splits alternate horizontally and vertically. In this thesis, we will refer to a horizontal and vertical split pair as a single level of the K-D subdivision. The nodes are split until all subdivisions are below the threshold number of nodes, which we set to be 6.

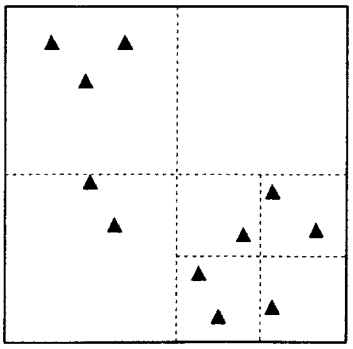


Figure 4.3.a: Quad tree Subdivision

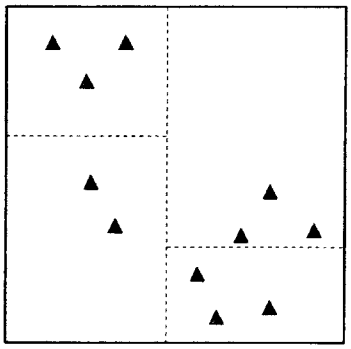


Figure 4.3.b: K-D Subdivision

4.3.3. Adding the Cross Links

For each level of the K-D subdivision, horizontally and vertically adjacent subdivisions are interconnected by adding edges between the farthest possible nodes along the x-axis and y-axis respectively. For the horizontal edges, this is done by sorting the nodes in each subdivision according to their x position, and then iterating through all node pairs consisting of one node from the left subdivision, starting with the smallest x-value, and one node from the right subdivision, starting with the largest x-value. An edge is added between the first pair that can have an edge added. The steps are similar for adding vertical cross links. Every adjacent subdivision in each level has only one cross link added.

4.4. Nested Overlapping Convex Hull Topology (NOCH)

The convex hull based topology for creating degree constrained connected graphs, is somewhat unique in the world of ad-hoc networks. This topology is based on starting off with concentric rings of edges, and then interconnecting those rings with another set of overlapping rings. Figures 4.4.a, 4.4.b, and 4.4.c show one set of concentric rings and an example of how they may be connected.

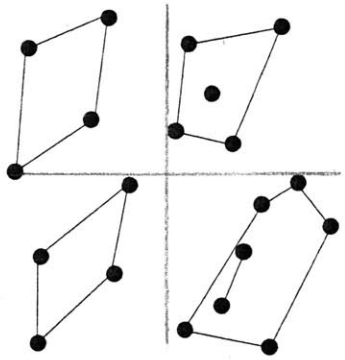


Figure 4.4.a: Original Set of Nested Convex Hulls

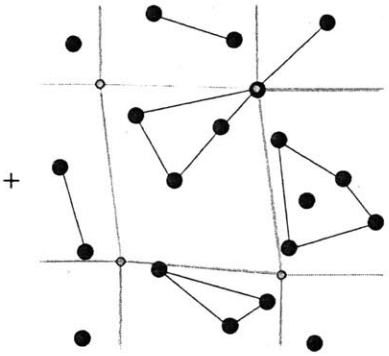


Figure 4.4.b: Overlapping Set of Nested Convex Hulls

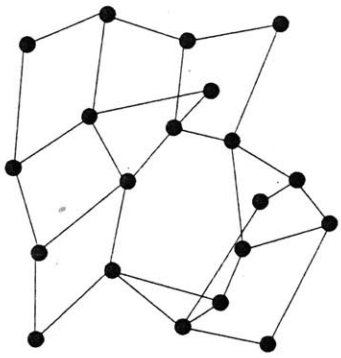


Figure 4.4.c: Combined Topology

4.4.1. Nested convex hulls

Formally, the convex hull is the minimum convex subset of points that encompass all the points in a set. Intuitively, if you had a set of pegs in a board and stretched a rubber band around the outside pegs, the convex hull would be the set of pegs the rubber band is touching.

Nested convex hulls are obtained by taking the the convex hull, CH_0, of a set S_0 and subtracting the nodes in CH_0 from S_0 to arrive at a new set S_1. The next convex hull, CH_1, is of the set S_1. These steps are repeated until S is empty and we have *n* nested convex hulls CH_0 to CH_n.

There exists well known and efficient algorithms for calculating single convex hulls in 2D [16]. Although Chazelle describes an algorithm in [23] to compute each of the nested convex hulls simultaneously with a total runtime of O(n log n), we use the incremental method described in [16] for simplicity, which has an $O(n^2)$ runtime for per hull.

4.4.2. Properties of the of nested convex hulls

There are many advantages to starting off the topology with nested convex hulls, but there are a few limitations as well.

We will see that the advantage of starting with concentric rings is two-fold. The first motivation is geometric. Rings of nodes can usually be formed by using opposite facing terminals on every node, which frees terminals facing towards the center of the ring as well as outwards (when T>3). Theoretically, since the rings are concentric, inner rings should be able to use their outward facing terminals to connect with the inward facing terminals of the outer rings. The second motivation is the 2-connectedness of rings, which is a desirable property when we are focusing on reliability. To be effective, however, T must be greater than 3, which is the major drawback of this algorithm.

Using the rotational independence rules in section 3.2, we can analyze how edges can be added to the rings.

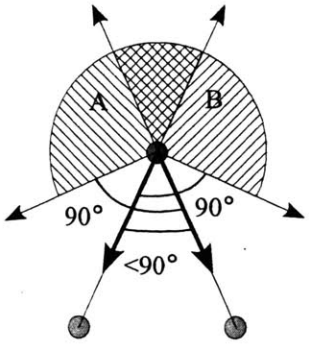


Figure 4.4.d: Angle <90°

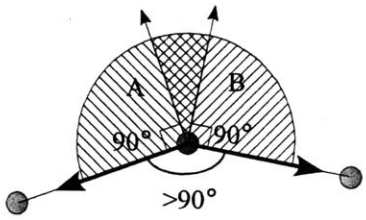


Figure 4.4.e: Angle >90°; 2 exterior links

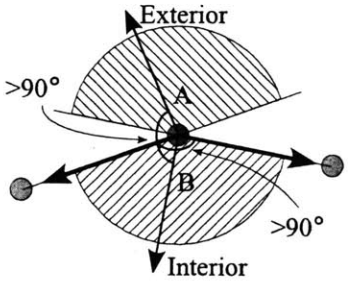


Figure 4.4.f: Angle >90°; 1 exterior, 1 interior

For example, when T=4 FOV=180°, we start by taking the angle formed by the 2 given links. If the angle is less than 90 degrees, we know the new links must both be within the exterior angle of the given links, one in sector A and one in sector B as depicted in Figure 4.4.a. If the angle is greater than 90 degrees, both new links may also be within the exterior angle (Figure 4.4.b) or one can be within the exterior angle and the other in the interior angle (Figure 4.4.c). If both an interior and exterior link are made, the angle they form must be greater than 90°.

4.4.3. Properties of the Interior Links

To ensure we have a enough interior links, it is helpful to calculate how many internal links we are guaranteed to have, given a convex hull with n nodes.

To find the number of guaranteed interior links given a convex hull with n nodes, we consider the case which minimizes the number of nodes with a possible interior link, which is identical to the case which maximizes the number of nodes without a possible interior link. This maximum is obtained by having as many nodes just below the critical angle (which guarantees an interior link) as possible.

Again, assuming T=4 and FOV=180°, the critical angle is 90°. The sum of the interior angles is equal to 180*(n-2)°, so for n=3, we can have at most a single 90° node, leaving the 2 other nodes to average 45°. Unfortunately, the other nodes are still below the critical angle, meaning that it cannot be guaranteed to have an interior link. In fact, if a convex hull for n=3 can guarantee an interior link, it can have no more than one since a triangle can have no more than one angle greater than 90°.

For $n \geq 4$, we can essentially have at most 3 90° nodes, leaving the remaining nodes to have interior facing terminals. We can therefore guarantee at least n-3 inward facing terminals for a convex hull with n nodes. Even though we can only guarantee n-3 free terminals, there are usually more because the probability of all nodes being oriented in such a way is low.

More generally, for T=4 and FOV = 180 – Δ°, the critical angle for guaranteeing an interior link becomes 90+Δ° and the maximum number of critical angles possible becomes floor(360 / (90 – Δ)). This leaves n - floor(360 / (90 – Δ)) guaranteed interior links.

The main thing to gather from all this number crunching is that for $T \geq 4$ and for FOV near double coverage, we can be sure that we have a good number on interior links.

4.4.4. Stability of Nested Convex Hulls

Rotational stability of the hulls is inherent, since the they remain the same regardless of node orientation. Translational stability and entry/exit stability are less certain.

Considering translational stability, the complete set of nested convex hulls will remain stable, unless one

of two things happen. The first is a node crossing and joining an outer hull (case A in Figure 4.4.g). The second is a node crossing it's own hull and joining an inside hull (case B in Figure 4.4.h). In both cases, the hulls outside of the two hulls involved in the crossover are not affected. However, all hulls inside are susceptible to propagating changes if the inner hull involved in the crossover has nodes removed from itself or nodes added from an inside hull. If nodes are removed from a hull, it is because the nodes become part of an inner hull, thus propagating changes. This only happens in cases like B. Nodes are possibly added from an inside hull only in case A. By observation, the changes are usually limited to the side of each hull nearest to the crossover.

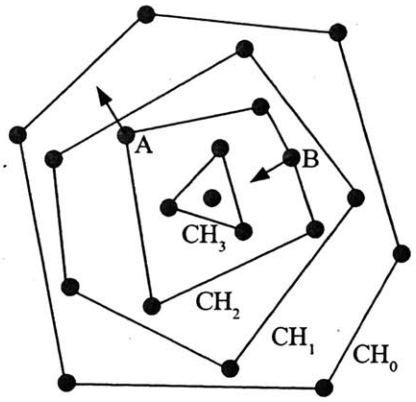


Figure 4.4.g: Nested Convex Hulls

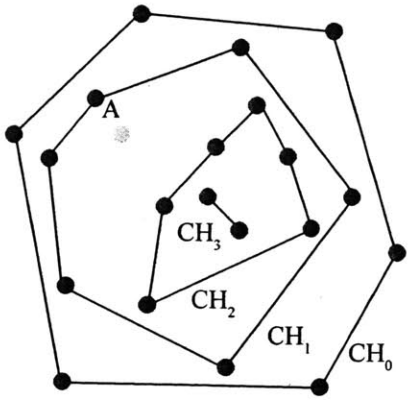


Figure 4.4.h: Change Case A

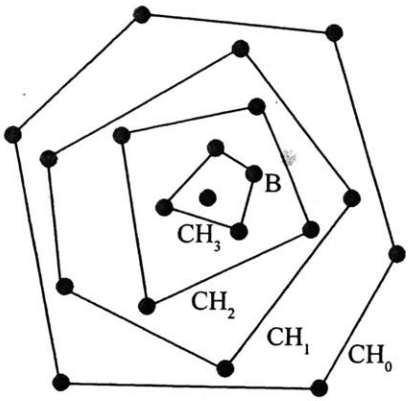


Figure 4.4.i: Change Case B

Entry/exit stability is similar, since a node entering or exiting will affect the outside hull, propagating changes inward if all nodes previously included in the outside hull are not included in the new outside hull. Unfortunately, looking at the results in section 5, it appears the changes in topology are not localized as well as we would like, even though the propagation of change is at least somewhat limited.

4.4.5. Choosing the Convex Hulls

To guarantee that nodes can be linked as convex hulls and interconnected, the scene is subdivided into sets of nodes which are all within range of each other. The subdivisions are individually connected as nested convex hulls.

Two simple and regular ways of subdividing the scene are shown, in Figure 4.4.j using a square grid, and in Figure 4.4.k using a hexagonal honeycomb structure. Using other regular convex shapes for subdivision are impractical since they cannot be tiled regularly. The honeycomb subdivision seems more likely to produce "well-shaped" convex hulls that have more inward facing terminals. Hexagon subdivision is more difficult to implement, so we use square grids in this thesis.

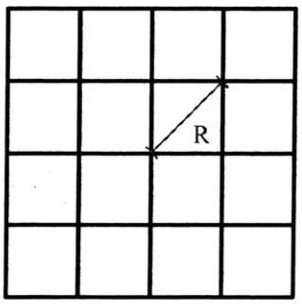


Figure 4.4.j: Grid Subdivision

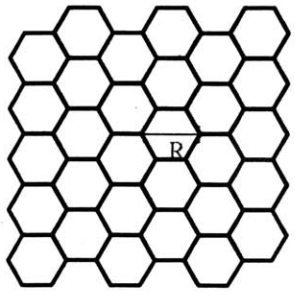


Figure 4.4.k: Honeycomb Subdivision

Figure 4.4.l: Overlapping Grid Subdivision

4.4.6. Interconnecting the Rings

The easiest way to interconnect the rings is with another set of overlapping rings. We can use the exact same nested convex hulls algorithm, but this time apply it to subdivisions that overlap the original ones. The corners of the new partitions are the centers of the inner-most convex hull of the old partition, as depicted in Figure 4.4.l, where the blue dots represent the centers of the gray partitions. Previous partitions without any nodes use the center of their partitions instead, which is the case for all border partitions.

Even though this way of partitioning may lead to sets of nodes that are not within range, it is overcome by the reduction in number of repeat edges in comparison to simply using another simple grid partition. After all the rings are formed, every node should be of degree 4 if no two edges overlap between the two sets of convex hulls. If there is overlap, some nodes may be of degree 3 or 2. This ensures the maximum degree of any node is 4. This does not ensure the FOV constraints are also met though, so we use the algorithm in Appendix C to choose a maximal subset of linkable edges.

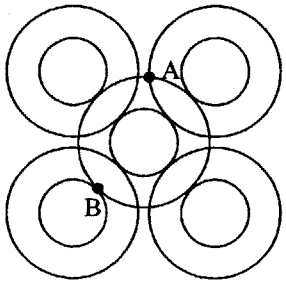
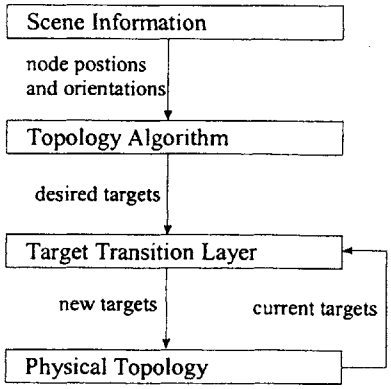


Figure 4.4.m: Idealized overlapping convex hulls

Although FOV constraints are not always met, the simulations and analysis show that the FOV constraints are met often. If we look at how overlapping convex hulls intersect, which is where nodes would be, we see that they range from perpendicular (area A in Figure 4.4.m) to tangent (area B in Figure 4.4.m). This means we never have edges which all point in the same direction. Instead, the edges either point in all different directions (perpendicular) or they will point in sets of two in opposite directions (tangent).

Border partitions have particularly bad overlap since they share the same outward most hulls. This is a significant problem that is not addressed in this thesis.

4.5. Target Transition Layer



During preliminary testing, it was obvious that these algorithms were transitioning many terminals at once, reducing the overall connectedness of their graphs when the terminal search delay, W, was greater than 0. To address this issue, we introduced the target transition layer, which gradually moves the current physical topology to the desired topology specified by the topology algorithm.

Instead of setting targets directly, each topology algorithm will create a set of desired targets

let desiredTarget($T_{A,i}$, t) be defined as
the desired target of $T_{A,i}$ at time t.

If W = 0, there is no terminal downtime if the transition layer points all terminals to their desired targets immediately. However, if W $\geq$0, we do not want the transition layer to change all the terminal targets at the same time since it often results in many terminals being unlinked simultaneously. The following algorithm attempts to address this problem by only unlinking one terminal per node per time-step for re-targeting.

Unused terminals or redundant terminals (pointing at the same node as another terminal) are automatically re-targetted to their desired targets. After this step, if no terminals are re-targeting, then the remaining terminals are re-targeted one at a time, prioritizing terminals that have desired targets that are already pointing back.

This algorithm only considers a node's own target transitions and does not attempt to optimize target transitions of the scene as a whole, which may improve performance. For example, no concessions are made in this algorithm to prevent a node from performing a target transition that would disconnect the graph.

5. Results

The performance of the algorithms will be compared under various parameterizations detailed in section 5.1. These values were chosen to encompass a wide range of situations and to help determine what trade offs exist for varying values of T, FOV, and node density.

In the results, we concentrate on the effect FOV has on connectedness by sampling the FOV at a high resolution for a set of simulations, and we concentrate on the effect node density has on connectedness by sampling node density at a high resolution for a set of simulations. In addition to seeing the relationship between connectedness and FOV and node density, we also gain insight into what effects the terminal delay and the link uptime probability have on each of the algorithms. Statistics concerning terminal target changes are provided to illustrate the re-targeting rates of terminals. Section 5.2 contains the graphs of all these results.

In preliminary simulations, we found that the static scenes resulted in a lower overall connectedness than the dynamic scenes by a few percent. This discrepancy is apparently due to the effect the nodes with circular paths have on node distribution within the scene. Therefore, to make sure all results are comparable, only results from dynamic scenes are presented.

The dynamic scenes are run for 2 time steps before measurements are taken to prevent initial terminal target delays from affecting results. Also, to diversify the scenes that algorithms are tested against, scenes are re-initialized to another random dynamic scene every 20 time steps. This prevents any one scene, which may favor one algorithm over another, from controlling results.

Results are simulated with a 95% confidence level to be within 1% of their simulated value. For connectedness values > 95%, results are 95% confident to be within 0.1%. All simulations were run for a minimum of 500 iterations with a time-step of 2 seconds between each iteration.

5.1. Simulation Parameters

The results are simulated with parameters that take on the matrix of values specified in Table 5.1.a.

Table 5.1.a: Parameterizations

Parameter	***Values***	***Description***
D	1000km	Scene dimensions are constant for all simulations
R	200km	Range is 200km for all simulations, since it is the target range for the ORCLE project[1].
N, n	n={5,10,20,30,50,75,100} → N={41,81,160,240,399,598,797}	n=node density; N is a function of n. *(see 5.1.1)*
T	3, 4, 5	Number of terminals
FOV	{90,180,270,360,450,540,630,720}/T	Field of view
W	0, 2 seconds	Link establishment delay
$PR_L(T_A,T_B,t)$	k=.95; c={ 3, 20 }	Link uptime probability function *(see 5.1.2)*
Δt	2 seconds	All simulations are run with a 2 second time-step.

5.1.1. Node Density

We define node density, n, as the average number of nodes in range of a single node. To set the value of n to the desired values, we change the value of N using a function of D, R, and n. If we ignore edge cases where a node's range area is not completely within the scene bounds

$$n=(N-1)\cdot\left(\frac{\pi R^2}{D^2}\right); \qquad \text{where } \left(\frac{\pi R^2}{D^2}\right) \text{ is the fraction of the scene visible by a single node.}$$

Solving for n we get: $$N=\frac{D^2 n}{R^2 \pi}-1$$

5.1.2. Link Uptime Probability Function

The Link Uptime probability will be defined with the following function:

$$PR_L(T_A, T_B, t) = k\left[1-\left(\frac{dist(N_A, N_B, t)}{R}\right)^c\right] \qquad \text{if dist(A,B,t)} < R;$$

$$= 0 \qquad \text{otherwise;}$$

We will use the values of k=.95 and c={ 3, 20 }, which results in the graph of PR_L's depicted in Figure 5.1.a. These values were chosen to simulate link uptime probabilities that fall off sharply with distance (c=20) and that fall off gradually with distance (c=3).

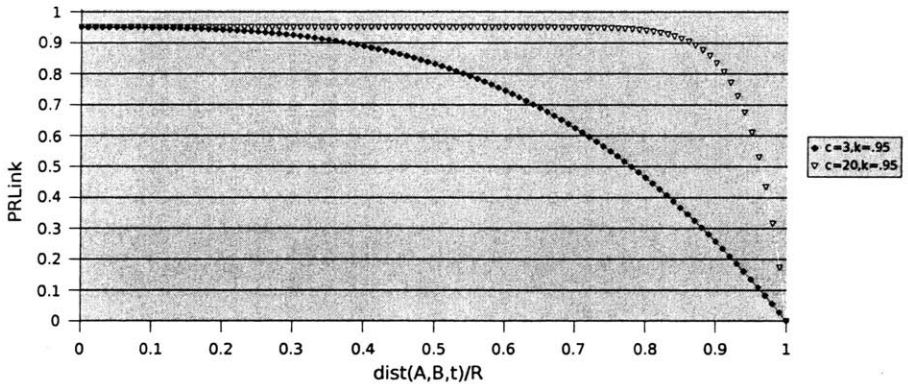


Figure 5.1.a: Link Uptime Probability Functions

5.2. Graphs

The next five pages contain tables of the graphs of all the results. Tables 5.2.a – 5.2.d contain the graphs for the results sampled with the high FOV resolution. Tables 5.2.e – 5.2.h contain the graphs for the results sampled with the high node density resolution. Tables 5.2.i and 5.2.j contain statistics on the number of terminal target changes per terminal per second. Section 5.3 contains discussion on these results.

Table 5.2.a: Connectedness vs FOV; D=1000, R=200, Δt=2, k=.95, c=20, W=0

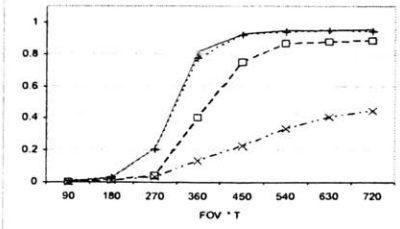
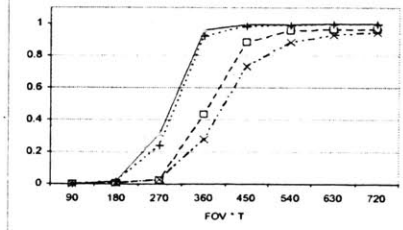
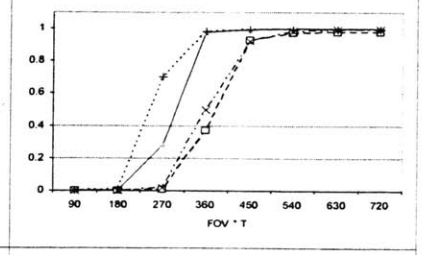
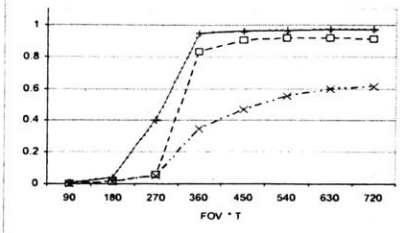
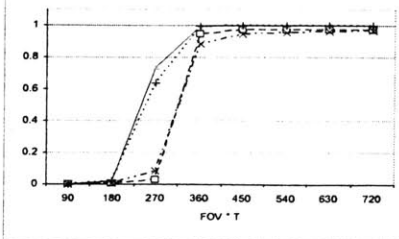
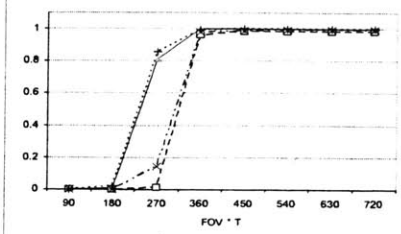
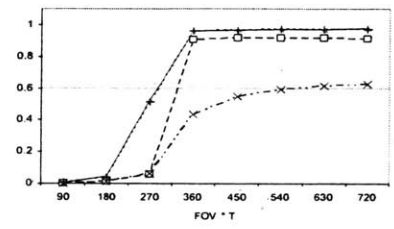
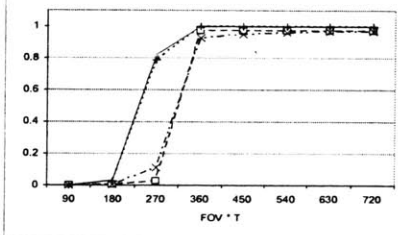
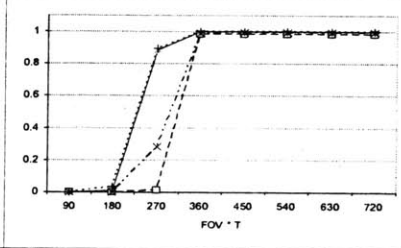
T	n=10; (N=81)	n=30; (N=241)	n=100; (N=797)
3			
4			
5			

Table 5.2.b: Connectedness vs FOV; D=1000, R=200, Δt=2, k=.95, c=20, W=2

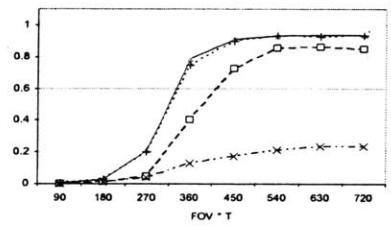
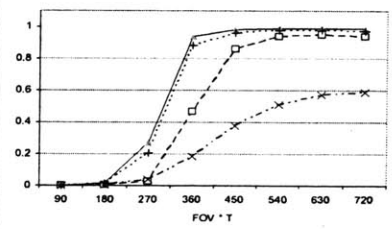
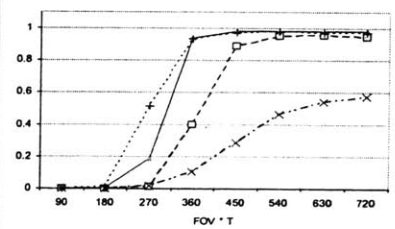
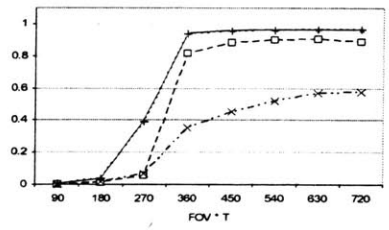
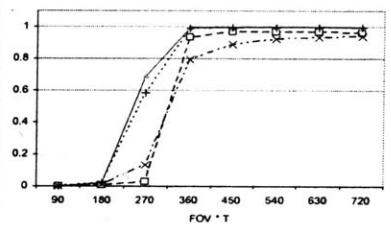
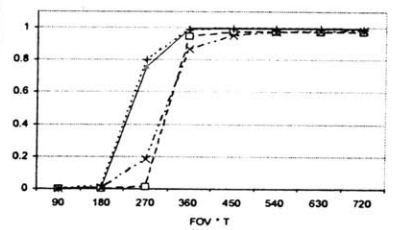
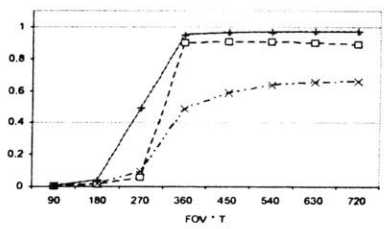
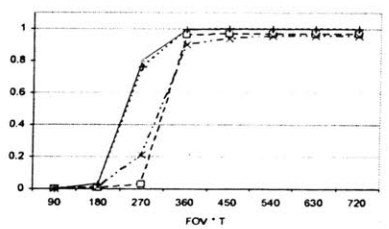
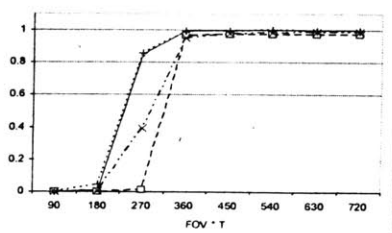
T	n=10; (N=81)	n=30; (N=241)	n=100; (N=797)
3			
4			
5			

Table 5.2.c: Connectedness vs FOV; D=1000, R=200, Δt=2, k=.95, c=3, W=0

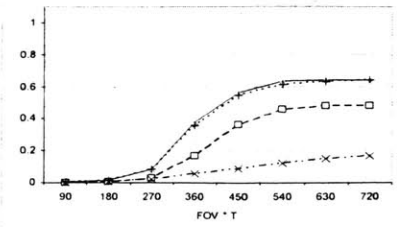
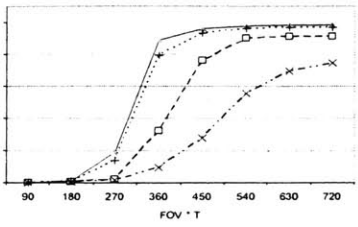
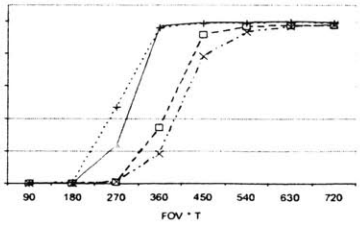
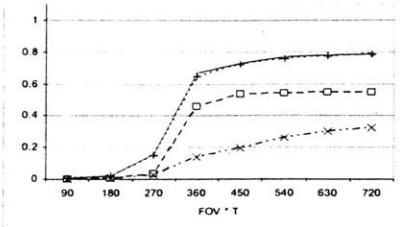
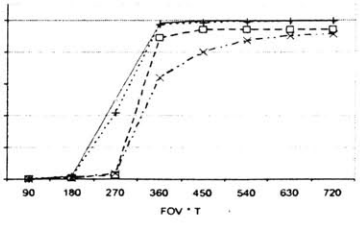
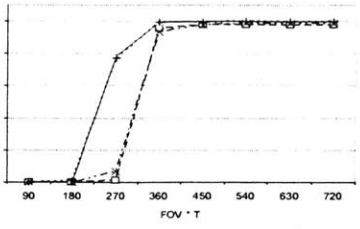
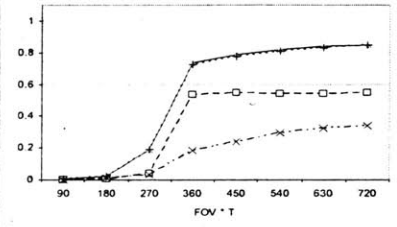
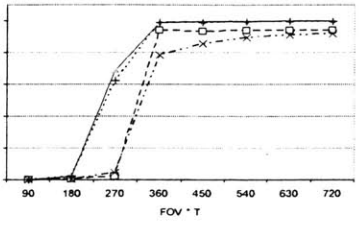
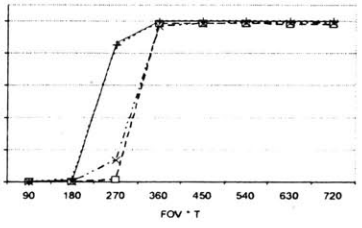
T	n=10; (N=81)	n=30; (N=241)	n=100; (N=797)
3			
4			
5			

Table 5.2.d: Connectedness vs FOV; D=1000, R=200, Δt=2, k=.95, c=3, W=2

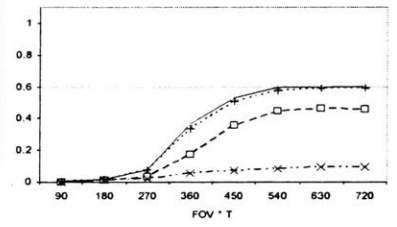
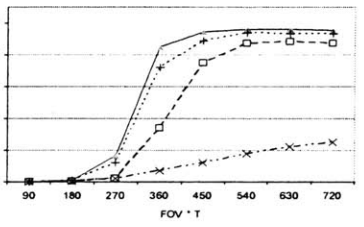
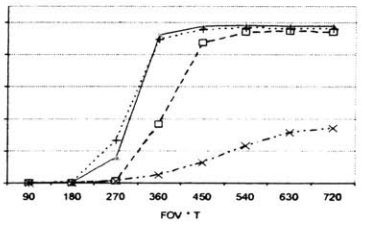
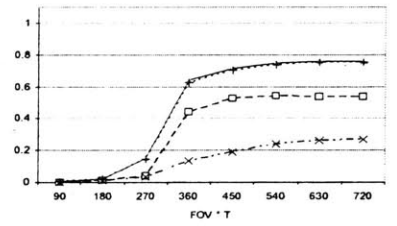
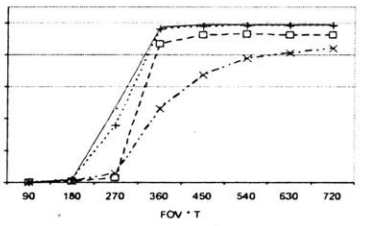
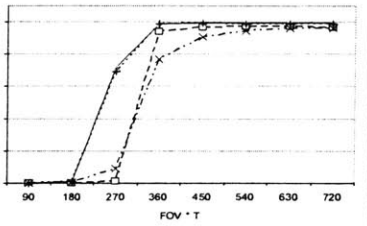
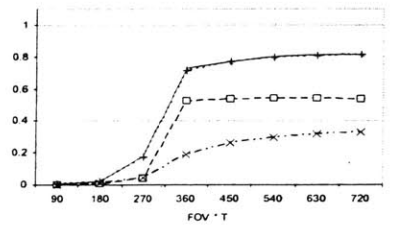
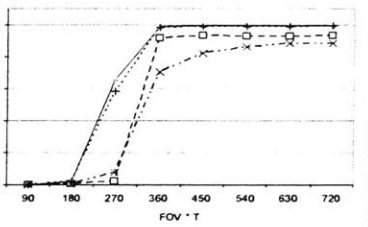
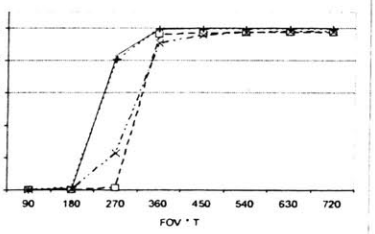
T	n=10; (N=81)	n=30; (N=241)	n=100; (N=797)
3			
4			
5			

Table 5.2.e: Connectedness vs n; D=1000, R=200, Δt=2, k=.95, c=20, W=0

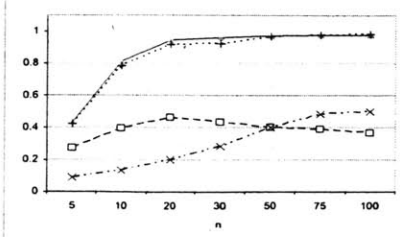
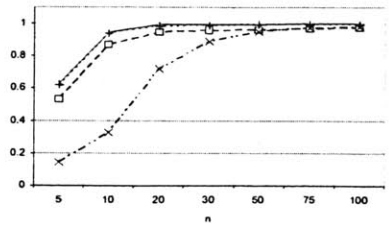
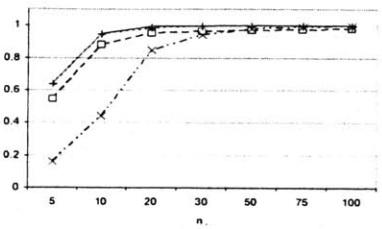
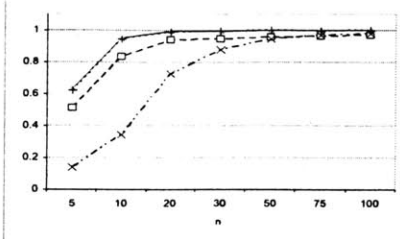
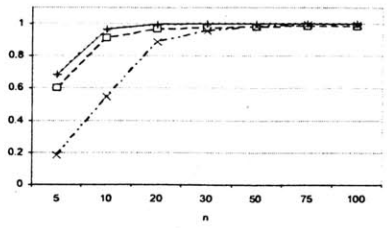
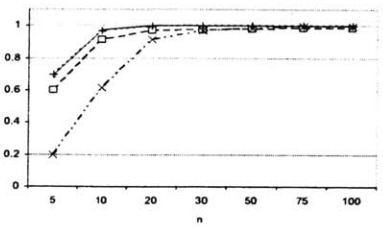
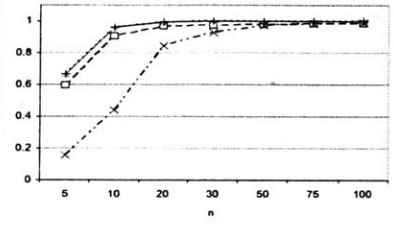
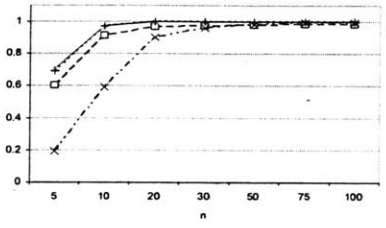
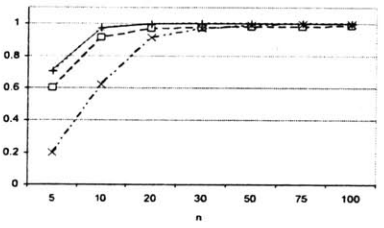
T	FOV=360/T	FOV=540/T	FOV=720/T
3			
4			
5			

Table 5.2.f: Connectedness vs n; D=1000, R=200, Δt=2, k=.95, c=20, W=2

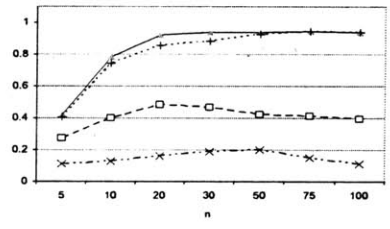
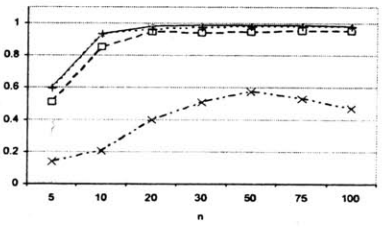
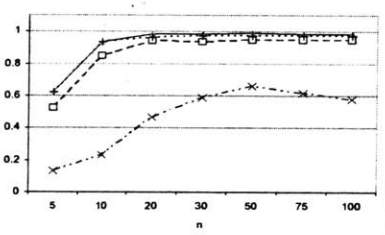
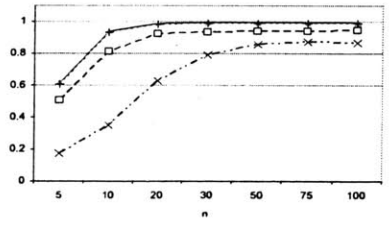
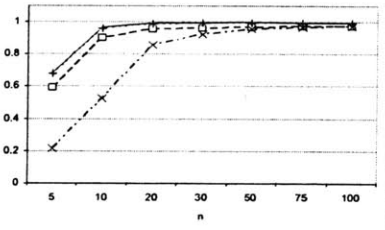
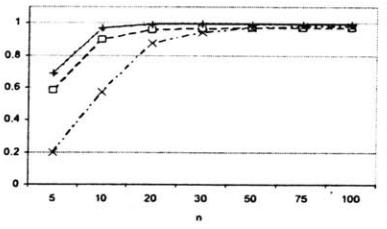
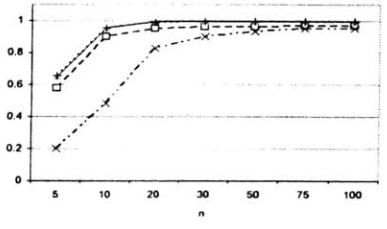
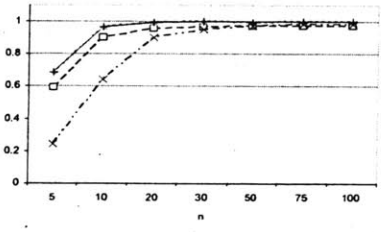
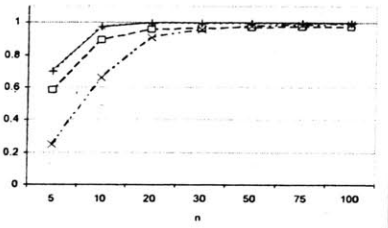
T	FOV=360/T	FOV=540/T	FOV=720/T
3			
4			
5			

Table 5.2.g: Connectedness vs n; $D=1000$, $R=200$, $\Delta t=2$, $k=.95$, $c=3$, $W=0$

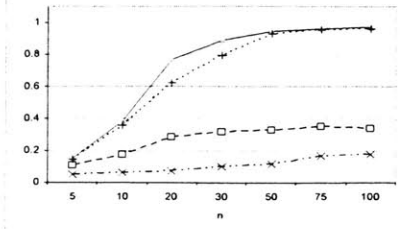
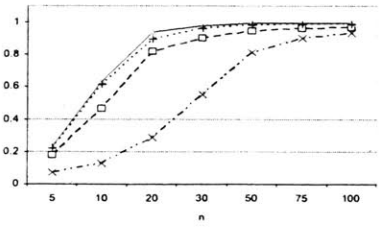
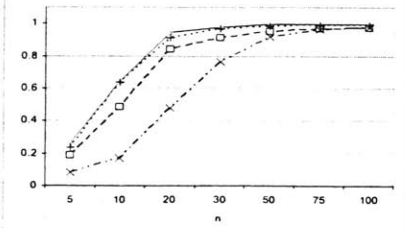
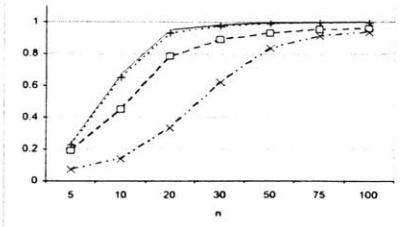
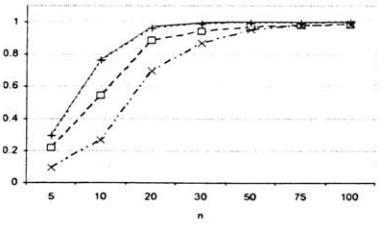
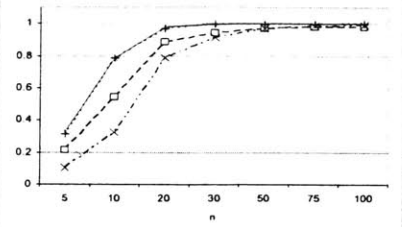
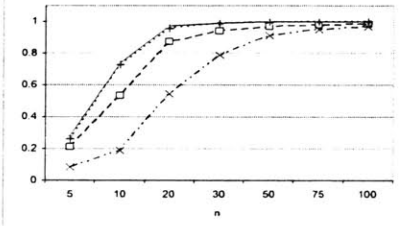
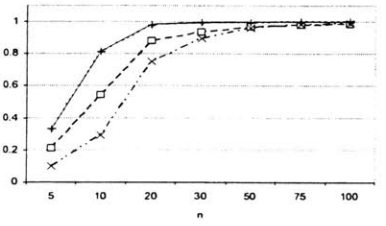
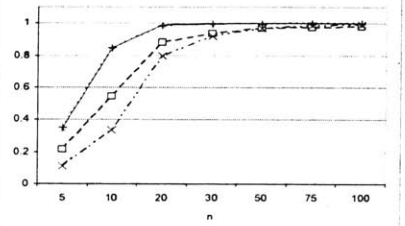
T	FOV=360/T	FOV=540/T	FOV=720/T
3			
4			
5			

Table 5.2.h: Connectedness vs n; $D=1000$, $R=200$, $\Delta t=2$, $k=.95$, $c=3$, $W=2$

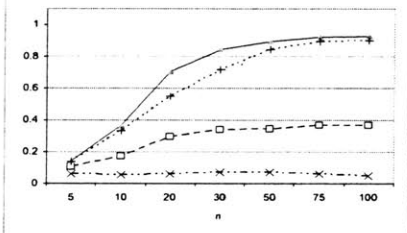
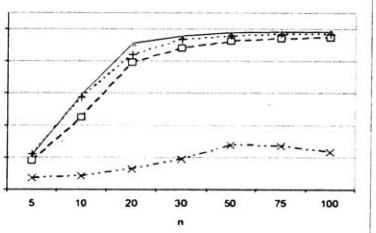
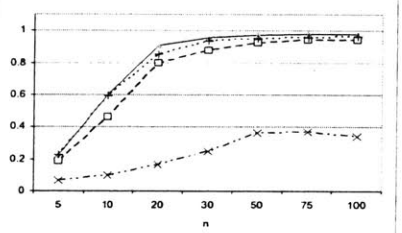
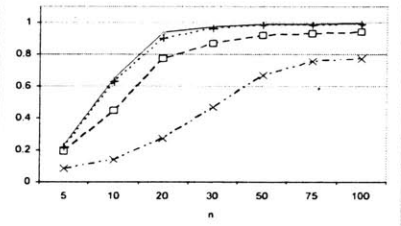
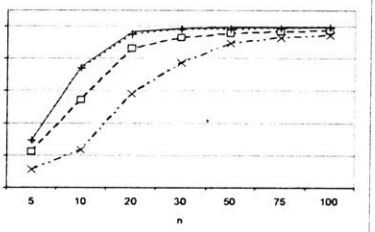
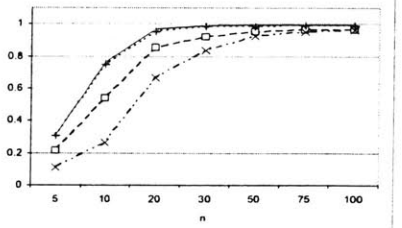
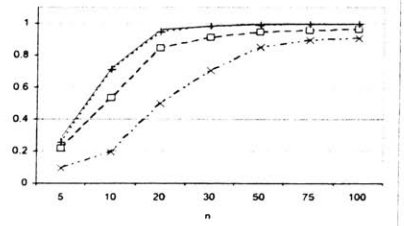
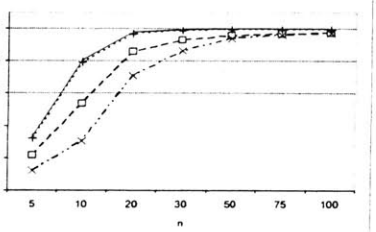
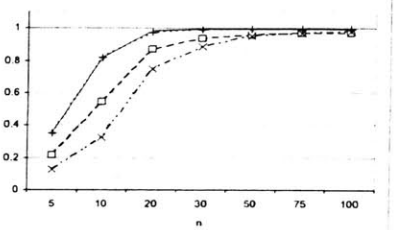
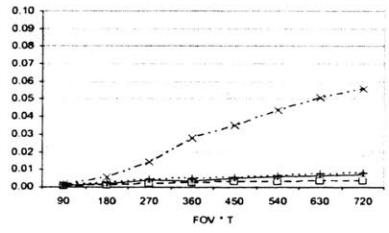
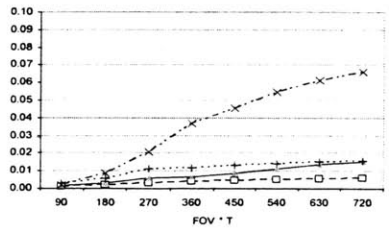
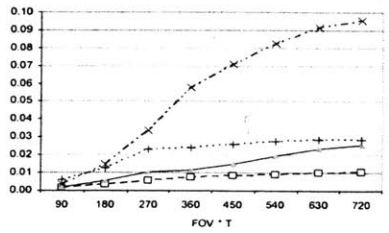
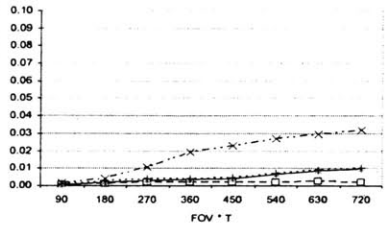
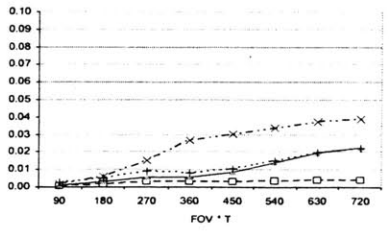
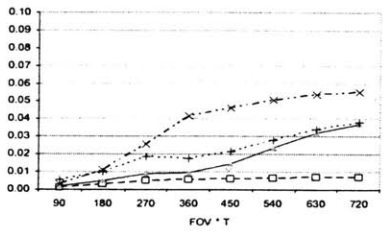
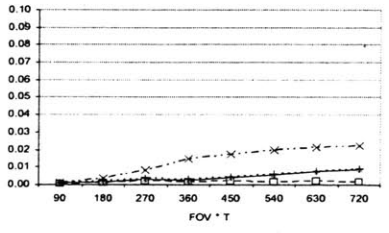
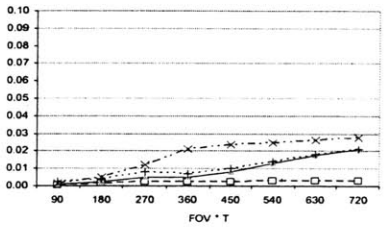
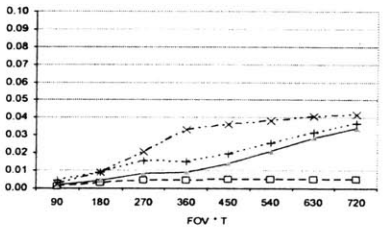
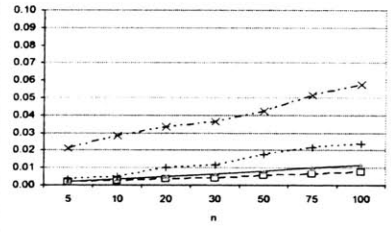
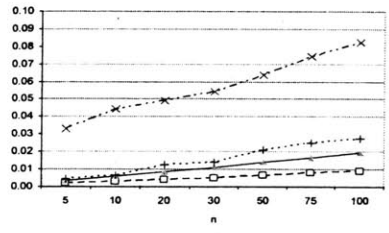
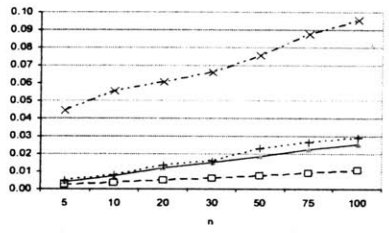
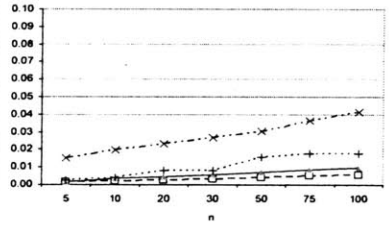
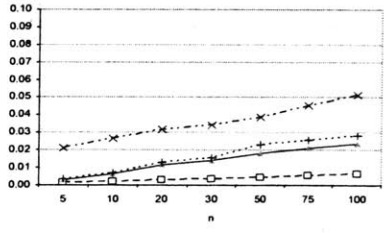
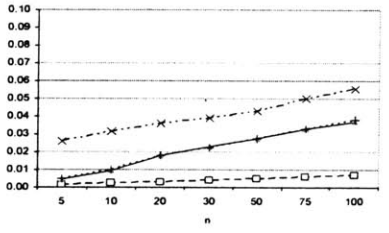
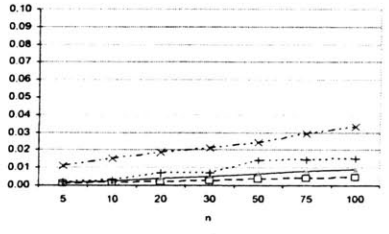
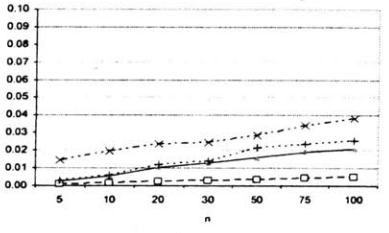
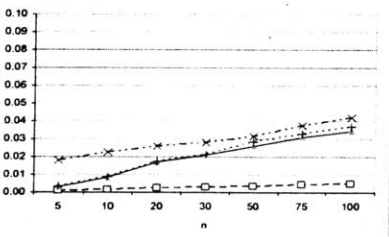
T	FOV=360/T	FOV=540/T	FOV=720/T
3			
4			
5			

Table 5.2.i: Target changes per terminal per second vs. FOV; D=1000, R=200, Δt=2, k=.95, c=3, W=2

T	n=10	n=30	n=100
3			
4			
5			

CRNG ARNG ARNGx NOCH

Table 5.2.j: Target changes per terminal per second vs. n; D=1000, R=200, Δt=2, k=.95, c=3, W=2

T	FOV=360/T	FOV=540/T	FOV=720/T
3			
4			
5			

5.3. Discussion

5.3.1. Effects of FOV

The set of graphs in tables 5.2.a – 5.2.d show one of the most significant findings, which is the gains in connectedness due to an increase in the FOV are marginal past a certain point. In all cases, the ARNG and ARNGx algorithms are within about 5% of their maximum values when FOV=540/T, and in many cases when FOV=450/T. Decreasing the FOV past these points, however, begins to greatly affect the connectivity of the network, which is indicated by the large slopes of the graphs in their periods of transition.

Results are similar for NOCH, however increasing the FOV helps more in the higher node densities, since in many cases, ARNG and ARNGx have already almost reached a connectedness of 1 by the time FOV is 450/T, while NOCH still has a relatively low connectedness.

5.3.2. Effects of Node Density

In the set of tables 5.2.e-5.2.h, all algorithms initially show an increase in connectedness as node density increases, however, after a certain point when k=.95, c=20, and W=2, some algorithms actually begin to perform worse! It is very apparent for NOCH when n=3 and, although it is difficult to see in the graphs, the ARNG consistently performs between .001 to .003 worse when going from n=50 to n=100. Numbers of this size may be negligible, but are worth noting none-the-less.

At first this seems counter-intuitive: if there are more nodes within range, shouldn't it be easier to create a connected graph? Indeed this is the case for the results, when W=0 or c=3. However, when c=20 and W=2, the reduced connectivity can be attributed to the higher number of terminal target changes, as indicated by the graphs in Table 5.2.j. This is not likely a problem when c=3, because the quality of shorter links as n increases outweighs the effect of the increased terminal target changes.

5.3.3. Effects of Link Probability Falloff

The link uptime probability that falls off quickly with distance (k=.95, c=3) does not have much of an effect, compared to when k=.95 and c=20, when the node densities are high because the link lengths are shorter. For lower node densities (n=10), however, the low link uptime probability greatly affects all algorithms such that none perform better than a .8 connectedness, even when T=5 and FOV=720/T. If link uptime probabilities fall off sharply with distance, node densities must be significant enough for an airborne network to be reliable.

5.3.4. Effects of Targeting Delay

Comparing the sets of graphs where W=2, to the graphs where W=0, we can see that the terminal delay did not affect the RNG based algorithms significantly. In general, we also see that changing W from 0 to 2 significantly hinders each algorithm's performance when T=3 more than when T=4 or 5.

A surprising result is that for low node density (n=10) and low link uptime (c=3), the connectedness is actually better when there is a targeting penalty (W=2) as opposed to when there is not (W=0), all other things equal (see section 5.4). This may seem contradictory, but if we remember that the target transition layer merely forwards target requests when W=0, we realize the discrepancy comes from when algorithms take links down that can still be used. In such cases the target transition layer would keep the previous links without removing them. This effect is noticed only in low node densities because so much of the graph is already disconnected that any extra edges help dramatically. This is a clear sign that the algorithms have room for improvement.

5.3.5. Effects of T

As expected, it is painfully obvious that NOCH only performs well for T>=4, but even then, NOCH does not outperform ARNG or ARNG in any of the simulations.

For the RNG based algorithms, however, it is interesting to note that reducing T to 3 does not affect the connectedness drastically for higher node densities. For lower node densities, however, every terminal counts.

6. Conclusion

Across the board, the ARNG algorithm keeps the nodes connected the best and is the most versatile in handling the gamut of conditions thrown at it. Only in an extremely few cases does the ARNGx barely beat out ARNG.

While the NOCH algorithm was based on a novel concept, it's applications are limited to high node densities and T>=4, and even then, it fails to beat out either the ARNG or ARNGx algorithm in any single simulation.

We've seen that node density is the greatest factor affecting connectedness. Unfortunately, we are unlikely to control this parameter, so we must instead rely on increasing T and FOV. Although increasing the FOV of each terminal helps, the marginal gains in connectedness begin to diminish past single coverage. Far greater improvements in the connectedness can be achieved by increasing T. Depending on the requirements of the network and the desired connectedness, however, it may be inefficient to have many terminals if the node density is known to be high.

6.1. Algorithm Improvements

The algorithms in this thesis are far from perfect, and in many cases we know what issues need to be addressed.

For the ARNGx, different ways of subdividing a scene and interconnecting adjacent areas can be experimented with, using more or less cross links. Also, methods to keep cross links from preventing edge addition from the DT could be investigated.

For the NOCH, the sets of rings that share edges result in unused terminals, which should be targeted somehow. Also, different ways of interconnecting the first set of convex hulls with shorter edges can be experimented with, since convex hulls tend to use longer edges.

When constraining the edges chosen by the topology algorithms, instead of only considering edges locally on a per-node basis, it may help to consider the effect on the connectedness of the scene as a whole. Similarly, the target transition layer could be made to optimize for the entire scene, attempting to keep the graph fully connected.

6.2. Future Work

There were many simplifications to ORCLE that we made in this thesis. The real test of these algorithms will come when run under more realistic environments, where the terminals cannot be placed facing equally apart, and the nodes are not flying all at the same altitude. We expect the requirements for T and FOV to only increase once the move to 3D is made.

Hybrid optical/rf terminals, which are an integral aspect of the ORCLE network, were ignored in this thesis and should be considered in the future. Since the ARNG algorithm already favors short links, it is likely to be more applicable to the hybrid network than the ARNGx and NOCH algorithms, which favor longer edges. This, however, remains to be seen.

Other aspect to address include methods of connecting the network of airplanes to ground stations and satellites, and interconnecting different-degree nodes within the same scene.

Despite the improvements that can be made with the algorithms and the issues that must still be addressed under real conditions, we have an optimistic outlook for ORCLE topology algorithms, as long as node density is high or link uptime probabilities do not diminish too much with distance.

7. Appendix A: The Simulator

The simulator used in this thesis is programmed in C++, and implements the model described in the section 2. It uses the QT API to visually display node positions and velocities, the link graph, the functional graph, and the sectors of each terminal.

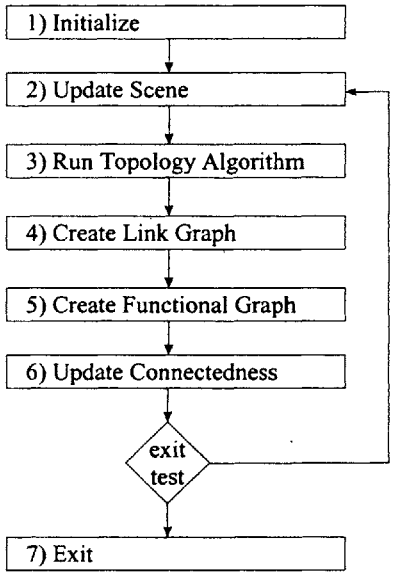
A brief description of the simulator is provided as background for the reader and methods used to validate the simulator are also described to ensure the reader of the simulator's correctness.

7.1. Flow Chart

At startup/initialization, the simulator takes arguments for values of the parameters in Table 7.1.a.

Table 7.1.a: Simulation Inputs

Parameters	***Description***
N	Number of nodes
T	Number of terminals
FOV	Angular field of view of all terminals
D	Dimension of square scene
R	Range of all the terminals
W	Terminal targeting delay
c, k	PR_L function parameters
random number seed	Determines what random sequence of scenes is used.
algorithm #	Specifies which algorithm to use. 0=CRNG; 1=ARNG; 2=ARNGx; 3=NOCH
logfile	Filename to concatenate the results



The simulation then enters the main loop which takes a constant time step after each iteration. If W>0, this time step is equal to W, otherwise the time step defaults to 2 seconds. The loop starts off by updating the scene. Next the topology algorithm sets the terminal targets.

The link graph is created from the terminal targets and the current state of the scene. This is in comparison to having the topology algorithm create the link graph itself. Creating the link graph separately is a sanity check to ensure algorithms are targeting the terminals correctly.

The functional graph is created by randomly removing edges of the link graph according to the link probability function, PR_L.

The connectedness is calculated by first calculating the number of nodes in each of the connected sub-graphs within the functional graph. If we let N_i be the number of nodes in subgraph i, each connected subgraph has $N_i * (N_i-1) / 2$ connected node pairs. Since the maximum possible node pairs is $N * (N-1) / 2$, the connectedness of the j^{th} step in the simulation is calculated as:

$$C_j = \sum_{i=0}^{S} \frac{N_i \cdot (N_i - 1)}{N \cdot (N-1)} \qquad where\, S = number\, of\, subgraphs$$

A list of all C_j's are stored to calculate the average and standard deviation of C_j's. Using the standard deviation, we can determine when we've simulated enough steps. The simulation is run until we are at least 95% confident that the average connectedness of the j iterations is within .1% of the average connectedness at the limit. This is done using the method described in [24] to calculate the confidence interval of an estimated mean when the standard deviation of the sample is known.

7.2. Random Number Generation

To generate random numbers, we use the Mersenne Twister 19937 C functions from [25]. The functions are encapsulated into C++ classes so we can have independently running instances of the Mersenne twister. This allows the random number generation in scene creation and in the topology algorithm to be decoupled, which allows the same randomly created scene to be reproduced for different algorithms.

7.3. Simulation Validation

Visualization of the nodes, terminals, and topologies was a simple way of cursorily validating that topologies are within degree and FOV constraints (Figure 7.3.a). In addition to the visualization, three tests were created in order to more rigorously validate critical aspects of the simulator.

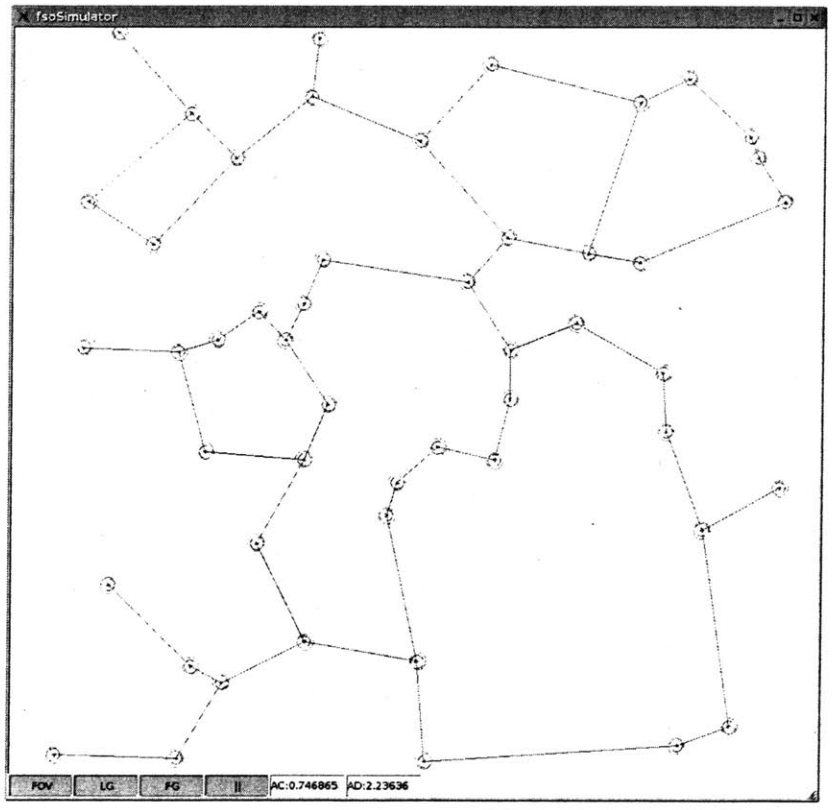


Figure 7.3.a: Simulator Visualization N=50, T=3, FOV=180

One of the most important functions of the simulator is its ability to ensure that only links that meet all constraints exist in the link graph. There are 2 pivotal criteria used to determine if a link can be formed: the within FOV test and within range test. The range test is trivial, but the within FOV is more complicated. The following section validates the within FOV test.

7.3.1. Validating the Within FOV Test

The within FOV test is a key software function that determines if the node N_k is within the FOV of terminal $T_{i,j}$. To validate this function works properly, we compare simulation results to theoretical results assuming there are no range constraints.

We define a statistic we call the average aggregate visible nodes, or AAVN. The AAVN is the sum of the number of nodes visible by *each* terminal of a node in the scene averaged over a series of S scenes.

$$AAVN = \frac{1}{S \cdot N} \sum_{scene=0}^{S} \sum_{n=0}^{N} \sum_{t=0}^{T} \sum_{i=0}^{N} \begin{Bmatrix} 1 & if\ N_i\ is\ within\ the\ FOV\ of\ T_{n,t} \\ 0 & otherwise \end{Bmatrix}$$

A single terminal is expected to see a fraction of the N-1 nodes equal to FOV/360, and each node has T terminals, so given a set of N nodes, the expected value of the AAVN is:

$$E(AAVN) = \frac{T \cdot (N-1) \cdot FOV}{360}$$

We ran the simulator for 1000 iterations an calculated the AAVN for several values of N, T, and FOV and compared them to the expected values. The results, which show good agreement of the simulator to theory, are presented in the table below.

Table 7.3.a: Table of AAVNs

N	*T*	*FOV*	*E(AAVN)*	*Simulated AAVN*
100	4	90	99	99
100	4	180	198	198
100	4	120	132	131.98
100	3	120	99	99
100	3	180	148.5	148.502
100	5	100	137.5	137.501
100	5	90	123.75	123.751

7.3.2. Validating the RNG is always Formable when T=5 and FOV=120

Previously, we made the statement that the RNG can always be realized by targeting the terminals in ORCLE correctly if T=5, FOV=120, and range is infinite. This test is designed to make sure this statement is true for our simulator. The vanilla RNG (not run through the simulator) is compared to the resulting link graph of the CRNG topology algorithm. For over 1000 different iterations of static scene graphs with N=100, the two resulting graphs were identical.

We can be certain that for T=5, FOV=120, and N=100 the simulator does not incorrectly constrain any links and that the targeting algorithm in Appendix C works. To a lesser degree, it also provides confidence that the RNG algorithms do what they are supposed to do.

7.3.3. A Specific Test Scene

To test the simulator even further, we created a test scene with a predictable connectedness. The scene contains four nodes arranged at the corners of a square that is 200km by 200km as depicted in Figure 7.3.b. The range, R, is set to be 210km, limiting nodes to forming links horizontally and vertically. The PR_L is set using the equation from 5.1 such that k=.95 and c=20.

Since k=.95 and c=20, any link on the sides of the square has a $PR_L = .95\left[1-\left(\frac{200}{210}\right)^{20}\right] = 0.59195...$

We target the terminals by attempting to create all the links that are the side of the square. Nodes that can only choose to link with one of their neighbors, choses so randomly and with equal probability.

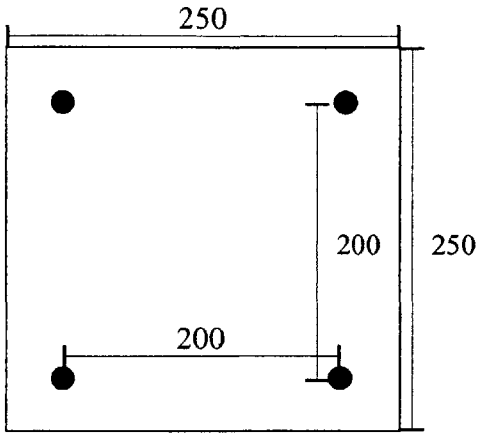


Figure 7.3.b: Test Scene Setup

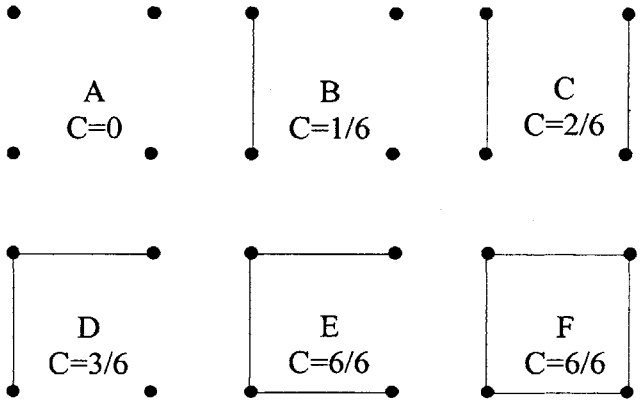


Figure 7.3.c: Possible Functional Graphs and Connectedness.

If we let PR_S be the probability that a link is formed on any given side of the square, then the corresponding edge exists in the functional graph with probability $PR_F = PR_S * PR_L$. If both nodes of an edge can form two links, then this edge can be linked with probability 1. Otherwise, if either node can only form one link, the edge can only be linked with probability $\frac{1}{2}$. These probabilities are reflected in the following equation for PR_S:

let A = the event that a node can form links with both it's adjacent nodes.
let B = the event that a node can only form a link with one of it's adjacent nodes.

$$PR_S = PR(A)^2 + \frac{1}{2}\cdot[2\cdot PR(A)PR(B) + PR(B)^2]$$

if we assume PR(A) = 1 – PR(B):

$$PR_S = PR(A)^2 + \frac{1}{2}[1 - PR(A)^2]$$

Figure 7.3.c enumerates all possible functional graphs and their connectedness. Using combinatorics, we can calculate the expected connectedness as:

$$E(\text{Connectedness}) = \frac{0}{6}\cdot PR_F^0\cdot(1-PR_F)^4 + 4\cdot\frac{1}{6}\cdot PR_F\cdot(1-PR_F)^3 + 2\cdot\frac{2}{6}\cdot PR_F^2\cdot(1-PR_F)^2 + 4\cdot\frac{3}{6}\cdot PR_F^2\cdot(1-PR_F)^2 + 4\cdot\frac{6}{6}\cdot PR_F^3\cdot(1-PR_F)^1 + \frac{6}{6}\cdot PR_F^4\cdot(1-PR_F)^0$$

For different values for T and FOV, we can calculate the actual value of PR_S and substitute it into the equations to predict the connectedness of the resulting functional graph and compare it to simulated results. Table 7.3.b presents various values for T and FOV, and their corresponding PR_S, PR_F, E(Connectedness), and simulated Connectedness over 1000 iterations. The simulated and theoretical values agree, indicating that many parts of the simulator all work to arrive at the correct answer. These parts include the randomness of the scene, the internal PR_L calculation, the process of link failures, the field of view constraints, and the method used to calculate the connectedness, amongst other things.

Table 7.3.b: Expected vs. Simulated Connectedness

T	*FOV*	PR_S	PR_F	*E(Conn.)*	*Simulated Conn.*
2	180	.625	.37	.35296	.357095
3	120	.78125	.4625	.471136	.471087
4	90	1	.592	.643801	.644899
5	72	1	.592	.643801	.644899

8. Appendix B: RNG & DT Background

Minimum Spanning Trees (MST), Relative Neighborhood Graphs (RNG), Gabriel Graphs (GG), and Delaunay Triangulations (DT) are graphs that have been used extensively in ad-hoc networks [8][9][11]. RNG, in particular, is important because a node can determine its links by only communicating with its nearest neighbors [10]. These topologies also tend to minimize power consumption when using omni-directional antenna, since nodes tend to communicate with nearer nodes. For directional antenna however, we are interested in these topologies since they inherently provide stable topologies where topology changes in one area of the graph does not cause topology changes in the other.

Although, only the RNG and DT are used in this thesis, the MST and GG are provided for completeness. The following figure gives a quick visual overview of what each of the resulting graphs look like. Each successive graph type is a superset of the previous.

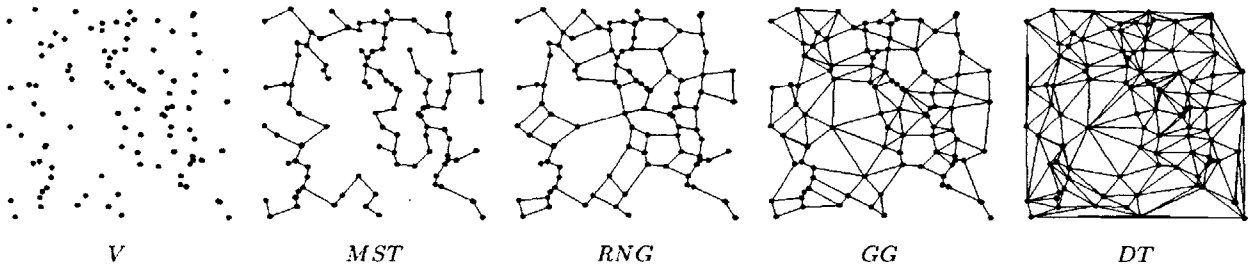


Figure 8.a: MST, RNG, GG, and DT of the points in V[1]

8.1.1. MST

The euclidean minimum spanning tree (EMST) is the set of edges which fully connect the nodes in a graph using the minimum cumulative length of edges possible. When the points lie on a euclidean plane, the maximum degree of any node in the EMST is 5 and all links of a given node are guaranteed to be separated by at least 60 degrees [4]. Although the EMST ensures a connected link graph, redundant links are non-existent, which will result in a poorly connected functional graph.

8.1.2. RNG, GG, and DT

In both the RNG and GG, two nodes are linked when there are no other nodes within their area of exclusion. The area of exclusion for RNG (Figure 8.b) is defined by the luna of two circles centered at each of the two nodes whose radii are equal to the distance of the two nodes. The area of exclusion for Gabriel Graphs (Figure 8.c) is the circle whose diameter is defined by the edge connecting the two nodes.

The Delaunay triangulation is similar, but uses the circle which circumscribes three nodes as it's area of exclusion (Figure 8.d). If no other nodes are within the circumscribed circle, the three nodes are linked as a triangle.

1 Image taken from [8]

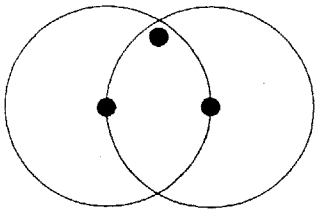


Figure 8.b: RNG Area of Exclusion

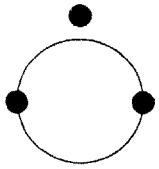


Figure 8.c: Gabriel Area of Exclusion

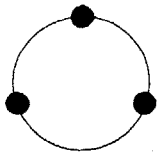


Figure 8.d: Delaunay Area of Exclusion

Similar to the EMST, the RNG's maximum possible node degree is 5 and all edges of a node are separated by at least 60 degrees. Unlike the EMST, however, some redundant local links are allowed, resulting in an average node degree of about 2.5 [4]. Although the maximum possible node degree is 5, the average maximum node degree for randomly located nodes is only 4.1 [4]. This makes RNG a good candidate for the constraints of our model, and thus is used as the baseline when comparing topologies in this thesis. The altered RNG graph used as the baseline is described in the following section.

The Gabriel Graph (GG) is a superset of the RNG, with the primary difference being that the Gabriel Graph does not guarantee degree bounded nodes. Despite this lack of guarantee, node degrees for random graphs remain somewhat low by inspection.

The Delaunay Triangulation (DT), in turn, is a superset of the GG, adding even more edges, such that each edge is a side of a triangle. The DT is known to maximize the minimum interior angle of all triangles over the set of all possible triangulations [12]. (i.e. they tend to avoid "skinny" triangles.)

9. Appendix D: Selecting a Subset of Edges to Link

The topology algorithms in this paper will often come up with a set of edges between nodes that may or may not all be converted into links between nodes. This will depend on the constraints of the parameterizations and the edges chosen.

To simplify discussion, if an edge is converted into a link, it is said to be linked.

Given a set of edges E for a single node, we would like to determine if a node can link all edges in E. And if all edges cannot be linked, we would also like to select a maximal subset of E that can all be linked. For a node with terminals that have overlapping FOVs, there may be multiple ways to link edges, since each terminal can potentially link with overlapping subsets of the E edges. Therefore, care is taken to ensure edge linking is deterministic.

These algorithms will be used to determine final terminal target assignments, but will also be useful for topology algorithms in removing edges from its intermediate graphs and for knowing what terminals can be free for additional edges.

9.1.1. Determining if a Node can Link all Edges

```
TestAllLinkable( i, E )

let E be the the set of edges to be tested for node Ni
let f = the first edge in E within or to the right of  Ti,0's FOV.
sort E radially clockwise such that the first element is f

while(true)
{
    let L be a list of edges set equal to the current E

    for j=0 to j<T
    {
        let e be the first edge in L
        if Ti,j can form a link with e
            pop the first edge in L.
    }

    //success
    if L is empty, then return true.

    //otherwise, failure: rotate E to try again
    move the first edge in E to the back of E

    //base cases: all rotations of E have failed
        if the first link in E = f OR
           Ti,0 cannot form a link with the first edge in E:
        return false;
}
```

9.1.2. Selecting a Maximal Subset of Edges to Link

```
GetMaximalLinkableSubset( i, E )

let E be the the set of edges to be tested for node Ni
let f = the first edge in E within or to the right of  Ti,0's FOV.
sort E radially clockwise such that the first element is f

while(true)
{
    let L be a list of edges set equal to the current E
    j = 0

    do{
        let e be the first edge in L
        if Ti,j can form a link with e
            Target( Ti,j , t ) = opposite node on e.
            pop the first edge in L.
            j = j + 1;
        else if e is to the left of Ti,j's facing direction
            pop the first edge in L.
        else j = j+1;
    }
    while( j<T );

    degree = |E| - |L|
    associate target set with degree

    move the first edge in E to the back of E

    //base case: all rotations of E have been tried
    if the first link in E = f OR
       Ti,0 cannot link the first edge in E
    {
        return target set with most targets used
    }
}
```

9.1.3. Explanation

If the first terminal can link with an edge in E, linking it can only make assigning the rest of the terminals easier. The algorithm iterates through all possible edges the first terminal can link, including the possibility of none at all, starting with the left most edge. Since both the edges and terminals are sorted radially in the same direction, assigning the edges and terminals in order will result in the use of the maximum terminals possible, given that the first terminal must link a specified edge. If a terminal is assigned to an edge later in the list, a later terminal might not be assignable to the first edge, when the first terminal could have. Likewise, if the first link is assigned to a later terminal, it may not be possible for the current terminal to be assigned to a later edge, when the later terminal could have.

What this does not prove, however, is the maximum terminal utilization in the entire scene, if this algorithm is run on nodes consecutively. This is because edges that are linked for one node, may not be linked by other nodes.

10. Appendix D: Algorithm Implementations

10.1. Excerpts from fsoAlgorithmRNG.cpp

```
#include "fsoAlgorithm.h"
#include "linear.h"
#include "mesh3d.h"


//#define DEBUG_ALG

#ifdef DEBUG_ALG
#define DEBUG(x) x
#else
#define DEBUG(x)
#endif


bool gNodeXCompare(const gNode *n1, const gNode *n2)
{
    return n1->n()->x.x() < n2->n()->x.x();
}

bool gNodeYCompare(const gNode *n1, const gNode *n2)
{
    return n1->n()->x.y() < n2->n()->x.y();
}

bool NodeXCompare(const Node *n1, const Node *n2)
{
    return n1->x.x() < n2->x.x();
}

bool NodeYCompare(const Node *n1, const Node *n2)
{
    return n1->x.y() < n2->x.y();
}

//breaks ties using the distance from the center (closer < farther)
struct gNodeDegreeCompare
{
    bool operator()(const gNode *n1, const gNode *n2)
    {
        int c1 = n1->nEdges();
        int c2 = n2->nEdges();

        if(c1!=c2) return c1 < c2;
        else return n1->n()->x.lengthSquared() < n2->n()->x.lengthSquared();
    }
};


//=============================================================================
// Delaunay Triangulation Stuff
// uses the 3d convex hull of (x,y,x*x+y*y) projected onto the xy plane
//=============================================================================

//wrapper class for 2d->3d
class DelaunayVertex : public Vertex
{
public:
    gNode *node;

public:
    DelaunayVertex(gNode *node)
    : Vertex(), node(node)
    {
        double x = node->n()->x.x();
        double y = node->n()->x.y();
```

```
        set(x,y,x*x+y*y);
    }
};

void DT(Scene &scene, fsoGraph &graph)
{
    graph.removeAllEdges();

    int nNodes = graph.nNodes();

    if(nNodes==0) return;
    if(nNodes==1) return;
    if(nNodes==2)
    {
        fsoGraph::node_iterator n1 = graph.nodesBegin();
        fsoGraph::node_iterator n2 = n1++;
        graph.newEdge( *n1, *n2);
        return;
    }
    if(nNodes==3)
    {
        fsoGraph::node_iterator n1 = graph.nodesBegin();
        fsoGraph::node_iterator n2 = n1++;
        fsoGraph::node_iterator n3 = n1++;
        graph.newEdge( *n1, *n2);
        graph.newEdge( *n1, *n3);
        graph.newEdge( *n2, *n3);
        return;
    }

    vector<Vertex*> vertices( nNodes );

    int i=0;
    for( fsoGraph::node_iterator n = graph.nodesBegin();
            n != graph.nodesEnd(); n++ )
    {
        vertices[i] = new DelaunayVertex(*n);
        i++;
    }

    Mesh *mesh3d = new Mesh();
    mesh3d->convexHull( vertices );

    //only add edges of faces that are facing in the
    // projection direction, otherwise we would add "back edges"
    Vector3d projection(0,0,-1);
    for( Mesh::const_face_iterator f = mesh3d->facesBegin();
            f != mesh3d->facesEnd(); f++ )
    {
        //skip "back edges"
        if( (*f)->normal.dot(projection) < 0 ) continue;

        //add all the edges of face
        HalfEdge *e = (*f)->firstEdge;
        do{
            graph.newEdge( ((DelaunayVertex*)e->p1)->node,
                           ((DelaunayVertex*)e->p2)->node );
            e = e->next;
        }while( e != (*f)->firstEdge );
    }

    delete mesh3d;
    for( i=0; i<nNodes; i++ )
        delete vertices[i];

    return;
}


//============================================================================
// RNG
// Relative Neighbor Graph
// Removes edges from the Delauny Triangulation that do not meet constraints.
//============================================================================
void RNG(Scene &scene, fsoGraph &graph, fsoGraph *superset)
```

```
{
    graph.removeAllEdges();

    if(superset!=NULL)
        graph.addEdges(*superset);
    else DT(scene, graph);

    vector<gNode*> nodes(graph.nodesBegin(), graph.nodesEnd());
    sort(nodes.begin(), nodes.end(), gNodeXCompare);

    vector<gEdge*> edges(graph.edgesBegin(), graph.edgesEnd());

    for( vector<gEdge*>::iterator e = edges.begin();
            e != edges.end(); e++ )
    {
        bool remove = false;

        gNode *n1 = (*e)->n1();
        gNode *n2 = (*e)->n2();

        //make sure n1 is to the left of n2
        if( n1->n()->x.x() > n2->n()->x.x() )
            swap(n1,n2);

        Vector2d x1( n1->n()->x ), x2( n2->n()->x );
        Vector2d midpoint = (x1+x2)/2;
        double d2 = (x1-x2).lengthSquared();
        double r2 = (x1-midpoint).lengthSquared();

        //some boundary conditions for quick cuttoffs
        double d = sqrt(d2);
        double minX = x1.x() - d;
        double maxX = x2.x() + d;
        double minY = x1.y(), maxY = x2.y();
        if(minY>maxY) swap(minY,maxY);
        minY-=d;
        maxY+=d;

        //iterate between n1 and n2
        vector<gNode*>::iterator i1, i2;
        i1 = i2 = upper_bound(nodes.begin(), nodes.end(), n1, gNodeXCompare);

        for( /*nada*/; (*i2) != n2 && i2!=nodes.end(); i2++ )
        {
            Vector2d &x = (*i2)->n()->x;
            if(x.y()<minY || x.y()>maxY) continue;

            //if in luna...
            if( d2 > ( x - x1 ).lengthSquared() &&
                 d2 > ( x - x2 ).lengthSquared()  )
            {
                remove=true;
                break;
            }
        }

        if(i2!=nodes.end()) i2++;       //dont consider n2
        if(i1==nodes.begin()) i1=nodes.end();
        else i1--;

        //iterate alternating left and right
        if(!remove)
        while( i1!=nodes.end() || i2!=nodes.end() )
        {
            if( i1!=nodes.end() )
            {
                Vector2d &x = (*i1)->n()->x;
                //if in luna...
                if( (x.y()>minY && x.y()<maxY) &&
                     d2 > ( x - x1 ).lengthSquared() &&
                     d2 > ( x - x2 ).lengthSquared()  )
                {
                    remove=true;
                    break;
                }
```

```
                if(i1==nodes.begin()) i1=nodes.end();
                else i1--;
            }

            if( i2!=nodes.end() )
            {
                Vector2d &x = (*i2)->n()->x;
                //if in luna...
                if( (x.y()>minY && x.y()<maxY) &&
                     d2 > ( x - x1 ).lengthSquared() &&
                     d2 > ( x - x2 ).lengthSquared()  )
                {
                    remove=true;
                    break;
                }

                i2++;
            }
        }
        if(remove) graph.removeEdge(*e);
    }

    int counter=0;
    for( fsoGraph::node_iterator n = graph.nodesBegin();
         n != graph.nodesEnd(); n++                          )
    {
        if( !(*n)->testAllEdgesLinkable() )
            counter++;
    }
    //cout << "[RNG:" << counter << "]"; cout.flush();

    return;
}


//================================================================================
// Augmented RNG + GG + DT
// Relative Neighbor Graph + Grabriel Graph + Delauny Triangulation
//================================================================================
void ARNG(Scene &scene, fsoGraph &graph, fsoGraph *superset)
{
//*
    graph.removeAllEdges();

    fsoGraph extra(&scene);             //to hold edges to be considered for addition

    if(superset!=NULL)
        extra.addEdges(*superset);
    else DT(scene, extra);

    extra.removeEdgesGT( scene.R2 );

    RNG(scene, graph, &extra);
    graph.constrain();
    extra.removeEdgesOfGraph(graph);        //only consider adding edges in rng not in rng...

    set<gNode*,gNodeDegreeCompare> nodes( graph.nodesBegin(), graph.nodesEnd() );

    for( set<gNode*>::iterator n = nodes.begin();
            n != nodes.end();  )
    {
        set<gNode*, gNodeDegreeCompare> targets;

        //add edges from extra that meet constraints on both sides
        gNode *node = extra.getNode( (*n)->n() );
        for( gNode::edge_iterator e = node->edgesBegin();
                e != node->edgesEnd(); e++ )
        {
            gNode *otherNode = graph.getNode( (*e)->otherNode(node)->n() );


            if( (*n)->testAllEdgesLinkablePlus( otherNode->n()->x ) &&  //constraints of n1
                 otherNode->testAllEdgesLinkablePlus( (*n)->n()->x )  ) //constraints of n2
            {
```

```
                targets.insert(
                    graph.getNode( (*e)->otherNode(node)->n() ) );
            }
        }

        //choose the "best" target
        if( targets.begin() != targets.end() )
        {
            gNode *nA = *n;
            gNode *nB = *targets.begin();

            n++;
            if( nB == (*n) ) n++; //make sure we do not invalidate the iterator

            nodes.erase( nB );
            nodes.erase( nA );

            graph.newEdge( nA, nB );

            nodes.insert(nA);
            nodes.insert(nB);
        }
        else n++;
    }
//*/
    return;
}

//============================================================================
// Augmented RNG + GG + DT + Overlay
// Relative Neighbor Graph + Grabriel Graph + Delauny Triangulation + Layered
//============================================================================

//creates RNG overlay for a single kd split
// nodes should be sorted accordingly in xsorted and ysorted
void ARNGxOverlaysH( Scene &scene, fsoGraph &graph,
                     Matrix< vector<gNode*> > &xsorted,
                     Matrix< vector<gNode*> > &ysorted )
{
    const int nThreshold = 6;  //number of nodes in a cell to qualify for adding a node
    const int maxLanes = 1;

    int nRows = xsorted.rows()*2;
    int nCols = xsorted.cols()*2;
    Matrix< vector<gNode*> > xsorted2( nRows, nCols );
    Matrix< vector<gNode*> > ysorted2( nRows, nCols );

    //add 1 node to overlay per cell if cell is < R/2 in the cross section
    for( int row=0; row<xsorted.rows(); row++ )
    for( int col=0; col<xsorted.cols(); col++ )
    {
        double xMedian, yMedian;
        vector<gNode*> &xs = xsorted(row, col);
        vector<gNode*> &ys = ysorted(row, col);

        if(xs.size() < nThreshold) continue;

        //left + right side
        vector<gNode*> xs1( xs.begin(), xs.begin()+xs.size()/2 );
        vector<gNode*> xs2( xs.begin()+xs.size()/2, xs.end() );
        vector<gNode*> ys1, ys2;

        xMedian = (*xs2.begin())->n()->x.x();
        for( int i=0; i<ys.size(); i++)
        {
            if( ys[i]->n()->x.x() < xMedian )
                ys1.push_back(ys[i]);
            else ys2.push_back(ys[i]);
        }

        //left side, top + bottom
        vector<gNode*> &ytl = ysorted2(row*2,col*2);
        vector<gNode*> &ybl = ysorted2(row*2+1, col*2);
        vector<gNode*> &xtl = xsorted2(row*2,col*2);
        vector<gNode*> &xbl = xsorted2(row*2+1, col*2);
```

```
        ytl.insert(ytl.begin(), ys1.begin(), ys1.begin() + ys1.size()/2 );
        ybl.insert(ybl.begin(), ys1.begin() + ys1.size()/2, ys1.end() );

        yMedian = (*(ys1.begin() + ys1.size()/2))->n()->x.y();
        for( int i=0; i<xs1.size(); i++)
        {
            if( xs1[i]->n()->x.y() < yMedian )
                xtl.push_back(xs1[i]);
            else xbl.push_back(xs1[i]);
        }

        //right side, top + bottom
        vector<gNode*> &ytr = ysorted2(row*2,col*2+1);
        vector<gNode*> &ybr = ysorted2(row*2+1, col*2+1);
        vector<gNode*> &xtr = xsorted2(row*2,col*2+1);
        vector<gNode*> &xbr = xsorted2(row*2+1, col*2+1);

        ytr.insert(ytr.begin(), ys2.begin(), ys2.begin() + ys2.size()/2 );
        ybr.insert(ybr.begin(), ys2.begin() + ys2.size()/2, ys2.end() );

        yMedian = (*(ys2.begin() + ys2.size()/2))->n()->x.y();
        for( int i=0; i<xs2.size(); i++)
        {
            if( xs2[i]->n()->x.y() < yMedian )
                xtr.push_back(xs2[i]);
            else xbr.push_back(xs2[i]);

    }

    bool subdivideFlag = false;

    //create left/right "lanes" and add those edges to the final graph
    for( int row=0; row<xsorted2.rows(); row++ )
    {
        for( int col=0; col<xsorted2.cols()-1; col++ )
        {
            vector<gNode*> &nodes1 = xsorted2(row, col);
            vector<gNode*> &nodes2 = xsorted2(row, col+1);
            if(nodes1.size() < nThreshold || nodes2.size() < nThreshold) continue;
            subdivideFlag = true;

            int nLanes = 0;
            int minNodes = min(nodes1.size(), nodes2.size());

            for(int i=0; i<minNodes; i++)
            {
                for(int j=0; j<=i; j++)
                {
                    //left side constant right side shifting
                    if( nodes1[i]->testAllEdgesLinkablePlus( nodes2[nodes2.size()-j-1]->n()->x )
&&
                        nodes2[nodes2.size()-j-1]->testAllEdgesLinkablePlus( nodes1[i]->n()->x
)   )
                    {
                        graph.newEdge( nodes1[i], nodes2[nodes2.size()-j-1] );
                        nLanes++;
                        if(nLanes>=maxLanes) break;
                    }

                    //right side constant left side shifting
                    if(i!=j)
                    if( nodes1[j]->testAllEdgesLinkablePlus( nodes2[nodes2.size()-i-1]->n()->x )
&&
                        nodes2[nodes2.size()-i-1]->testAllEdgesLinkablePlus( nodes1[j]->n()->x
)   )
                    {
                        graph.newEdge( nodes1[j], nodes2[nodes2.size()-i-1] );
                        nLanes++;
                        if(nLanes>=maxLanes) break;
                    }
                }
                if( nLanes>=maxLanes ) break;
            }
        }
```

```
    }

    //create left/right "lanes" and add those edges to the final graph
    for( int col=0; col<ysorted2.cols(); col++ )
    {
        for( int row=0; row<ysorted2.rows()-1; row++ )
        {
            vector<gNode*> &nodes1 = ysorted2(row, col);
            vector<gNode*> &nodes2 = ysorted2(row+1, col);
            if(nodes1.size() < nThreshold || nodes2.size() < nThreshold) continue;
            subdivideFlag = true;

            int nLanes = 0;
            int minNodes = min(nodes1.size(), nodes2.size());

            for(int i=0; i<minNodes; i++)
            {
                for(int j=0; j<=i; j++)
                {
                    //left side constant right side shifting
                    if( nodes1[i]->testAllEdgesLinkablePlus( nodes2[nodes2.size()-j-1]->n()->x )
&&
                        nodes2[nodes2.size()-j-1]->testAllEdgesLinkablePlus( nodes1[i]->n()->x
)   )
                    {
                        graph.newEdge( nodes1[i], nodes2[nodes2.size()-j-1] );
                        nLanes++;
                        if(nLanes>=maxLanes) break;
                    }

                    //right side constant left side shifting
                    if(i!=j)
                    if( nodes1[j]->testAllEdgesLinkablePlus( nodes2[nodes2.size()-i-1]->n()->x )
&&
                        nodes2[nodes2.size()-i-1]->testAllEdgesLinkablePlus( nodes1[j]->n()->x
)   )
                    {
                        graph.newEdge( nodes1[j], nodes2[nodes2.size()-i-1] );
                        nLanes++;
                        if(nLanes>=maxLanes) break;
                    }
                }
                if( nLanes>=maxLanes ) break;
            }
        }
    }

    if( subdivideFlag )
        ARNGxOverlaysH( scene, graph, xsorted2, ysorted2 );

    return;
}

void ARNGxOverlays(Scene &scene, fsoGraph &graph)
{
    Matrix< vector<gNode*> > xsorted(1,1);
    Matrix< vector<gNode*> > ysorted(1,1);

    vector<gNode*> &xs = xsorted(0,0);
    vector<gNode*> &ys = ysorted(0,0);
    xs.insert( xs.begin(), graph.nodesBegin(), graph.nodesEnd() );
    ys.insert( ys.begin(), graph.nodesBegin(), graph.nodesEnd() );
    sort(xs.begin(), xs.end(), gNodeXCompare);
    sort(ys.begin(), ys.end(), gNodeYCompare);

    ARNGxOverlaysH( scene, graph, xsorted, ysorted );
    return;
}


void ARNGx(Scene &scene, fsoGraph &graph, fsoGraph *superset)
{
//*
    graph.removeAllEdges();
```

```
fsoGraph extra(&scene);              //to hold edges to be considered for addition

if(superset!=NULL)
    extra.addEdges(*superset);
else DT(scene, extra);

extra.removeEdgesGT( scene.R2 );

RNG(scene, graph, &extra);
graph.constrain();
extra.removeEdgesOfGraph(graph);        //only consider adding edges to rng not in rng...

//make all nodes degree 2
for( fsoGraph::node_iterator n = graph.nodesBegin();
        n != graph.nodesEnd(); n++  )
{
    if( (*n)->nEdges() > 1 ) continue;
    set<gNode*, gNodeDegreeCompare> targets;

    //add edges from extra that meet constraints on both sides
    gNode *node = extra.getNode( (*n)->n() );
    for( gNode::edge_iterator e = node->edgesBegin();
            e != node->edgesEnd(); e++ )
    {
        gNode *otherNode = graph.getNode( (*e)->otherNode(node)->n() );


        if( (*n)->testAllEdgesLinkablePlus( otherNode->n()->x ) &&  //constraints of n1
             otherNode->testAllEdgesLinkablePlus( (*n)->n()->x )  ) //constraints of n2
        {
            targets.insert(
                graph.getNode( (*e)->otherNode(node)->n() ) );
        }
    }

    //choose the "best" target
    if( targets.begin() != targets.end() )
    {
        gNode *nA = *n;
        gNode *nB = *targets.begin();
        graph.newEdge( nA, nB );
    }
}

ARNGxOverlays(scene,graph);


set<gNode*,gNodeDegreeCompare> nodes( graph.nodesBegin(), graph.nodesEnd() );
for( set<gNode*>::iterator n = nodes.begin();
        n != nodes.end();  )
{
    set<gNode*, gNodeDegreeCompare> targets;

    //add edges from extra that meet constraints on both sides
    gNode *node = extra.getNode( (*n)->n() );
    for( gNode::edge_iterator e = node->edgesBegin();
            e != node->edgesEnd(); e++ )
    {
        gNode *otherNode = graph.getNode( (*e)->otherNode(node)->n() );


        if( (*n)->testAllEdgesLinkablePlus( otherNode->n()->x ) &&  //constraints of n1
             otherNode->testAllEdgesLinkablePlus( (*n)->n()->x )  ) //constraints of n2
        {
            targets.insert(
                graph.getNode( (*e)->otherNode(node)->n() ) );
        }
    }

    //choose the "best" target
    if( targets.begin() != targets.end() )
    {
        gNode *nA = *n;
        gNode *nB = *targets.begin();
```

```
            n++;
            if( nB == (*n) ) n++; //make sure we do not invalidate the iterator

            nodes.erase( nB );
            nodes.erase( nA );

            graph.newEdge( nA, nB );

            nodes.insert(nA);
            nodes.insert(nB);
        }
        else n++;
    }

    return;
}
```

10.2. Excerpts from fsoAlgorithmNOCH.cpp

```
#include <math.h>

#include "fsoGraph.h"
#include "fsoAlgorithm.h"
#include "linear.h"

bool NodeCompareVTCount(Node *n1, Node *n2)
{
    return n1->nVisibleTerminals < n2->nVisibleTerminals;
}

//#define DEBUG_ALG

#ifdef DEBUG_ALG
#define DEBUG(x) x
#else
#define DEBUG(x)
#endif

//============================================================================
// Single planar convex hull, storing all nodes of the hull in hull.
//============================================================================
void ConvexHull(fsoGraph &graph, list<gNode*> &hull)
{
    if( graph.nodesEmpty() ) return;

    //at least 1 node
    fsoGraph::node_iterator n = graph.nodesBegin();
    gNode *nA = *n;
    if( ++n == graph.nodesEnd() )
    {
        hull.push_back( nA );
        return;
    }

    //at least 2 nodes
    gNode *nB = *n;
    assert( nB != nA );
    gEdge *e1 = graph.newEdge( nA, nB );
    if( ++n == graph.nodesEnd() )
    {
        hull.push_back(nA);
        hull.push_back(nB);
        return;
    }

    //3 or more nodes...
    //make sure normals match
    if( e1->onRightSide( (*n)->n()->x - e1->n1()->n()->x ) )
        { e1->swap(); }
    assert( (*n) != e1->n1() );
    assert( (*n) != e1->n2() );
    graph.newEdge( (*n), e1->n1() );
```

```
    graph.newEdge( e1->n2(), (*n) );

    //return;
    int i=0;
    for( ++n; n != graph.nodesEnd(); ++n )
    {
        list<gEdge*> edgesToRemove;

        for( fsoGraph::edge_iterator e = graph.edgesBegin();
             e != graph.edgesEnd(); e++  )
        {
            assert( (*e)->n1() != (*e)->n2() );

            if( (*e)->onRightSide( (*n)->n()->x - (*e)->n1()->n()->x ) )
                edgesToRemove.push_back(*e);
        }

        if( edgesToRemove.empty() ) continue;

        //endpoints should end up with the nodes that only appear once
        // in the edges to remove
        hash_set<gNode*> endpoints;
        endpoints.clear();

        for( list<gEdge*>::iterator e = edgesToRemove.begin();
             e != edgesToRemove.end(); e++  )
        {
            assert( (*e)->n1() != (*e)->n2() );

            //for node1, remove if it exists, else add it
            hash_set<gNode*>::iterator n1 = endpoints.find( (*e)->n1() );
            if( n1 != endpoints.end() )
                endpoints.erase(n1);
            else endpoints.insert( (*e)->n1() );

            //same for node 2
            hash_set<gNode*>::iterator n2 = endpoints.find( (*e)->n2() );
            if( n2 != endpoints.end() )
                endpoints.erase(n2);
            else endpoints.insert( (*e)->n2() );

            //might as well remove edges here
            graph.removeEdge(*e);
        }

        assert( !endpoints.empty() );

        hash_set<gNode*>::iterator twoNodes = endpoints.begin();
        gNode *n1 = *twoNodes;
        gNode *n2 = *(++twoNodes);

        assert(n1!=n2);
        gEdge tEdge( n1, n2 );
        if( tEdge.onRightSide( (*n)->n()->x - n1->n()->x ) )
            tEdge.swap();

        graph.newEdge( *n, tEdge.n1() );
        graph.newEdge( tEdge.n2(), *n );
    }

    for( fsoGraph::edge_iterator e = graph.edgesBegin();
         e != graph.edgesEnd(); e++ )
    {
        hull.push_back( (*e)->n1() );
    }
}


//============================================================================
// Nested Convex Hulls
//============================================================================
//returns center of convex hulls
Vector2d NestedConvexHulls(fsoGraph &graph, vector<fsoGraph> *hulls = NULL )
{
    fsoGraph tmp;
```

```
    tmp.addNodes(graph);

    while( !tmp.nodesEmpty() )// && g.nNodes()!=1 )
    {
        //calculate outer most hull
        list<gNode*> hull;
        ConvexHull(tmp, hull);

        //add outer most hull to entire graph
        graph.addEdges(tmp);

        //add hull to hulls
        if(hulls != NULL)
        {
            fsoGraph hullgraph;
            for( list<gNode*>::iterator n = hull.begin();
                    n != hull.end(); n++ )
            {
                hullgraph.newNode( (*n)->n() );
            }
            hullgraph.addEdges(tmp);
            hulls->push_back(hullgraph);
        }

        //last hull, so calculate center and return
        if( hull.size() == tmp.nNodes() )
        {
            Vector2d center(0,0);
            for( list<gNode*>::iterator n = hull.begin();
                    n != hull.end(); n++ )
            {
                center += (*n)->n()->x;
            }
            center /= tmp.nNodes();
            tmp.removeAllNodes();
            return center;
        }

        //remove nodes on outer most hull, for next convex hull
        while( !hull.empty() )
        {
            tmp.removeNode( hull.front() );
            hull.pop_front();
        }
    }

    return Vector2d(0,0);       //should never get here
}

//=============================================================================
// Partitioning
//=============================================================================
class Partition
{
public:
    vector<Vector2d> v;         //vertices
    vector<Vector2d> n;         //normals

    //make sure to add vertices counter clockwise!
    Partition(const vector<Vector2d> &v) : v(v), n(v.size())
    {
        n[0] = v[0] - v[v.size()-1];
        n[0].set( -n[0].y(), n[0].x() );
        for(int i=1; i<v.size(); i++)
        {
            n[i] = v[i] - v[i-1];
            n[i].set( -n[i].y(), n[i].x() );
        }
    }

    bool inBounds(const Vector2d &p)
    {
        for(int i=0; i<v.size(); i++)
        {
            if( n[i].dot( p - v[i] ) < 0 )
```

```
                return false;
            }
            return true;
        }
};

//=============================================================================
// Square Partitioning
//=============================================================================
void SquareDimensions(Scene &scene, int &rows, int &cols)
{
    rows = ceil( scene.Y / scene.R ) + 4;
    cols = ceil( scene.X / scene.R ) + 4;
}

Vector2d SquareCenter(Scene &scene, int row, int col)
{
    int rows, cols;
    SquareDimensions(scene, rows, cols);

    int mRow = rows/2;
    int mCol = cols/2;

    double x = (col - mCol) * scene.R;
    double y = (row - mRow) * scene.R;
    if( 0 == rows%2 ) y+=scene.R/2;
    if( 0 == cols%2 ) x+=scene.R/2;

    return Vector2d(x,y);
}

void SquarePartition1(Scene &scene, Matrix<fsoGraph> &cells )
{
    int rows, cols;
    SquareDimensions(scene, rows, cols);

    int mRow = rows/2;
    int mCol = cols/2;

    int offsetX = 0, offsetY = 0;
    if( 0 == rows%2 ) offsetY=-scene.R/2;
    if( 0 == cols%2 ) offsetX=-scene.R/2;

    for( Scene::node_iterator n = scene.nodesBegin();
            n != scene.nodesEnd(); n++              )
    {
        int row = floor( ((*n)->x.y() + offsetY) / scene.R );
        int col = floor( ((*n)->x.x() + offsetX) / scene.R );

        cells(row+mRow,col+mCol).newNode( *n );
    }
}

void SquarePartition2( Scene &scene, Matrix<fsoGraph> &cells,
                                                    Matrix<fsoGraph> &prevCells,
                                                    Matrix<Vector2d> &centers )
{
    Matrix<Partition*> partitions( cells.rows(), cells.cols() );

    for( int row=0; row<cells.rows()-1; row++ )
    for( int col=0; col<cells.cols()-1; col++ )
    {
        //make sure to add vertices counter clockwise!
        vector<Vector2d> v(4);
        v[0] = centers(row, col);
        v[1] = centers(row, col+1);
        v[2] = centers(row+1, col+1);
        v[3] = centers(row+1, col);
        partitions(row,col) = new Partition(v);
    }

    /*
    for( Scene::node_iterator n = scene.nodesBegin();
            n != scene.nodesEnd(); n++              )
    for( int row=0; row<cells.rows()-1; row++ )
```

```
    for( int col=0; col<cells.cols()-1; col++ )
    {
        if( partitions(row,col)->inBounds( (*n)->x ) )
            cells(row,col).newNode( (*n) );
    }
    //*/

    //*
    for( int row=1; row<cells.rows(); row++ )
    for( int col=1; col<cells.cols(); col++ )
    {
        fsoGraph &g = prevCells(row,col);

        for( fsoGraph::node_iterator n = g.nodesBegin();
            n != g.nodesEnd(); n++              )
        {
            bool p1 = partitions(row-1,col-1)->inBounds( (*n)->n()->x );
            bool p2 = partitions(row,col-1)->inBounds( (*n)->n()->x );
            bool p3 = partitions(row,col)->inBounds( (*n)->n()->x );
            bool p4 = partitions(row-1,col)->inBounds( (*n)->n()->x );

            if( p1 && p2 && p3 && p4 )
            {
                cells(row-1,col-1).newNode( (*n)->n() );
                cells(row,col).newNode( (*n)->n() );
            }
            else if(p1)
                cells(row-1,col-1).newNode( (*n)->n() );
            else if(p2)
                cells(row,col-1).newNode( (*n)->n() );
            else if(p3)
                cells(row,col).newNode( (*n)->n() );
            else if(p4)
                cells(row-1,col).newNode( (*n)->n() );
            else
                cout << "*";
        }
    }
    //*/

    for( int row=0; row<cells.rows()-1; row++ )
    for( int col=0; col<cells.cols()-1; col++ )
        delete partitions(row,col);

    return;
}


//============================================================================
// Nested + Overlaping Convex Hulls: Square Partition
//============================================================================
int NOCH4(Scene &scene, fsoGraph &graph1, fsoGraph &graph2, int h)
{
    graph1.removeAllEdges();
    graph2.removeAllEdges();

    int rows, cols;
    SquareDimensions(scene, rows, cols);

    Matrix<fsoGraph> cells1(rows, cols);
    Matrix<fsoGraph> cells2(rows, cols);
    Matrix<Vector2d> centers(rows, cols);            //centers of cells in cells1

    SquarePartition1( scene, cells1 );

    for( int row=0; row<rows; row++ )
    for( int col=0; col<cols; col++ )
    {
        if( !cells1(row,col).nodesEmpty() )
        {
            centers(row,col) = NestedConvexHulls( cells1(row,col) );
            cells1(row,col).constrain();
            graph1.addEdges( cells1(row,col) );
        }
        else centers(row,col) = SquareCenter(scene, row, col);
```

```
    }

    SquarePartition2( scene, cells2, cells1, centers );

    for( int row=0; row<rows; row++ )
    for( int col=0; col<cols; col++ )
    {
        NestedConvexHulls( cells2(row,col) );
        cells2(row,col).constrain();
        graph2.addEdges( cells2(row,col) );
    }

    /*
    int counter=0;
    for( fsoGraph::node_iterator n = graph.nodesBegin();
          n != graph.nodesEnd(); n++                    )
    {
        if( !(*n)->testAllEdgesLinkable() )
            counter++;
    }
    */
}
```

10.3. Excerpts from fsoGraph.cpp

```
//=============================================================================
// fsoGraph
//=============================================================================
//breaks ties using the distance from the center (closer < farther)
struct gNodeDegreeCompareA
{
    bool operator()(const gNode *n1, const gNode *n2)
    {
        int c1 = n1->nEdges();
        int c2 = n2->nEdges();

        if(c1!=c2) return c1 > c2;
        else return n1->n()->x.lengthSquared() < n2->n()->x.lengthSquared();
    }
};


void fsoGraph::constrain()
{
    list<gEdge*> rEdges;        //edges to remove

    //Handle edges that are connected to two bad nodes first
    for( edge_iterator e = edgesBegin();
            e != edgesEnd(); e++ )
    {
        if( !(*e)->n1()->testAllEdgesLinkable() &&
             !(*e)->n2()->testAllEdgesLinkable() &&
             (*e)->n1()->testAllEdgesLinkableMinus(*e) &&
             (*e)->n2()->testAllEdgesLinkableMinus(*e) )
        {
            rEdges.push_back(*e);
        }
    }

    for( list<gEdge*>::iterator e = rEdges.begin();
            e != rEdges.end(); e++ )
    {
        removeEdge(*e);
    }

    //Handle the rest of the nodes in order
    set<gNode*, gNodeDegreeCompareA> orderedNodes( nodesBegin(), nodesEnd() );

    for( set<gNode*>::iterator n = orderedNodes.begin();
            n != orderedNodes.end();  )
    {
        if( (*n)->testAllEdgesLinkable() )
```

```
            { n++; continue; }

        set<gNode*, gNodeDegreeCompareA> targets;
        set<gNode*, gNodeDegreeCompareA> allTargets;

        for( gNode::edge_iterator e = (*n)->edgesBegin();
             e != (*n)->edgesEnd(); e++ )
        {
             allTargets.insert( (*e)->otherNode(*n) );

             if( (*n)->testAllEdgesLinkableMinus(*e) )
                  targets.insert( (*e)->otherNode(*n) );
        }

        gNode *nA = *n;
        gNode *nB = targets.empty() ? *allTargets.begin() : *targets.begin();

        n++;
        if( nB == (*n) ) n++;

        orderedNodes.erase( nB );
        orderedNodes.erase( nA );

        removeEdge( getEdge(nA,nB) );

        orderedNodes.insert( nA );
        orderedNodes.insert( nB );
    }
}

vector<int> fsoGraph::getSizeOfIslands()
{
    vector<int> result;
    hash_set<gNode*> rNodes( nodes );   //remaining nodes
    list<gNode*> pNodes;                          //pending nodes

    while( !rNodes.empty() )
    {
        int size = 0;
        pNodes.push_back( *rNodes.begin() );
        rNodes.erase( rNodes.begin() );

        while( !pNodes.empty() )
        {
            gNode *n1 = pNodes.front();
            pNodes.pop_front();

            gNode::edge_iterator e = n1->edgesBegin();
            while( e != n1->edgesEnd() )
            {
                gNode *n2 = (*e)->otherNode( n1 );
                if( rNodes.find(n2) != rNodes.end() )
                {
                    rNodes.erase( n2 );
                    pNodes.push_back( n2 );
                }
                ++e;
            }

            size++;
        }

        result.push_back(size);
    }
    return result;
}

double fsoGraph::connectedness()
{
    vector<int> sizes = getSizeOfIslands();

    int connectedPairs = 0;

    for(int i=sizes.size()-1; i>=0; i--)
    {
```

```
        int s = sizes[i];
        connectedPairs += s*(s-1)/2;
    }

    int m = nNodes();
    int maxPairs = m*(m-1)/2;
    return ((double)connectedPairs) / maxPairs;
}


//============================================================================
// fsoNode
//============================================================================

bool gNode::testAllEdgesLinkable() const
{
    return testAllEdgesLinkableMinus(NULL);
}

bool gNode::testAllEdgesLinkableMinus(gEdge *minusEdge) const
{
    if( edges.size() > node->nTerminals ) return false;
    if( edges.size() <= 0 ) return true;

    list<double> angles;

    for( edge_iterator e = edgesBegin();
         e!=edgesEnd(); e++ )
    {
        if( *e == minusEdge ) continue;
        Vector2d u( (*e)->otherNode(this)->node->x - node->x );
        //cout << u; cout.flush();
        double angle = node->vAngle(u);
        if( angle > node->terminals[0]->a1 )
            angle -= 2*M_PI;
        angles.push_back( angle );
    }

    angles.sort();
    /*
    cout << "\n";
    for_each( angles.begin(), angles.end(), print<double>(cout));
    //*/

    list<double>::iterator a = angles.begin();
    while( (*a)<0 ) { (*a)+=2*M_PI; a++; }

    /*
    cout << "\n";
    for_each( angles.begin(), angles.end(), print<double>(cout));
    //*/

    int triesLeft = nEdges();
    do {
        int t=0;
        a = angles.begin();

        while( t<node->nTerminals && a!=angles.end() )
        {
            if( node->terminals[t]->inFOV( *a ) )    //match!
            {
                a++;
                t++;
            }
            else if( (*a) > node->terminals[t]->a )  //a ahead of t
            {
                t++;
            }
            else //t ahead of a = fail!!!!
            {
                break;
                //a++;
            }
        }
```

```
        //all angles have been assigned a terminal
        if( a==angles.end() ) return true;

        angles.push_back( angles.front() );
        angles.pop_front();
    }while( --triesLeft > 0 &&
              node->terminals[0]->inFOV( angles.front() ) );

    return false;
}


bool gNode::testAllEdgesLinkablePlus(const Vector2d &x) const
{
    if( edges.size() > node->nTerminals ) return false;

    list<double> angles;

    //add angle to new node
    Vector2d u( x - node->x );
    double angle = node->vAngle(u);
    if( angle > node->terminals[0]->a1 )
            angle -= 2*M_PI;
    angles.push_back( angle );

    //add all edges' angles
    for( edge_iterator e = edgesBegin();
          e!=edgesEnd(); e++ )
    {
        Vector2d u( (*e)->otherNode(this)->node->x - node->x );
        //cout << u; cout.flush();
        double angle = node->vAngle(u);
        if( angle > node->terminals[0]->a1 )
            angle -= 2*M_PI;
        angles.push_back( angle );
    }

    angles.sort();
    /*
    cout << "\n";
    for_each( angles.begin(), angles.end(), print<double>(cout));
    //*/

    list<double>::iterator a = angles.begin();
    while( (*a)<0 ) { (*a)+=2*M_PI; a++; }

    /*
    cout << "\n";
    for_each( angles.begin(), angles.end(), print<double>(cout));
    //*/

    int triesLeft = nEdges();
    do {
        int t=0;
        a = angles.begin();

        while( t<node->nTerminals && a!=angles.end() )
        {
            if( node->terminals[t]->inFOV( *a ) )    //match!
            {
                a++;
                t++;
            }
            else if( (*a) > node->terminals[t]->a )  //a ahead of t
            {
                t++;
            }
            else //t ahead of a = fail!!!!
            {
                break;
                //a++;
            }
        }
```

```
        //all angles have been assigned a terminal
        if( a==angles.end() ) return true;

        angles.push_back( angles.front() );
        angles.pop_front();
    }while( --triesLeft > 0 &&
              node->terminals[0]->inFOV( angles.front() ) );

    return false;
}

bool gNode::updateTerminalTargets(double time)
{
    list<double> angles;
    hash_map<double,Node*> nodeMap;

    //add all edges' angles
    for( edge_iterator e = edgesBegin();
         e!=edgesEnd(); e++ )
    {
        Node *oNode = (*e)->otherNode(this)->node;
        Vector2d u( oNode->x - node->x );
        //cout << u; cout.flush();
        double angle = node->vAngle(u);
        nodeMap[angle] = oNode;
        if( angle > node->terminals[0]->a1 )
            angle -= 2*M_PI;
        angles.push_back( angle );
    }

    angles.sort();

    list<double>::iterator a = angles.begin();
    while( (*a)<0 ) { (*a)+=2*M_PI; a++;}

    vector< vector<Node*> > allTargets;
    int maxTargets = -1, bestTargetSet = -1;

    int currentTry = 0;
    int triesLeft = nEdges();

    do {
        int t=0;
        a = angles.begin();

        assert( &angles.front() == &*a );

        vector<Node*> targets;
        targets.clear();
        int nTargets = 0;

        while( t<node->nTerminals && a!=angles.end() )
        {
            if( node->terminals[t]->inFOV( *a ) )    //match!
            {
                targets.push_back( nodeMap[*a] );
                nTargets++;
                a++;
                t++;
            }
            else if( (*a) > node->terminals[t]->a )  //a ahead of t
            {
                t++;
                targets.push_back( NULL );
            }
            else //t ahead of a
            {
                a++;
            }
        }

        //best target set so far!
        if( nTargets > maxTargets )
        {
            maxTargets = nTargets;
```

```
            bestTargetSet = currentTry;
        }

        allTargets.push_back(targets);

        angles.push_back( angles.front() );
        angles.pop_front();
        currentTry++;

    }while( --triesLeft > 0 &&
            node->terminals[0]->inFOV( angles.front() ) );

   if( nEdges() > maxTargets )
        {cout << " nEdges>maxTargets! "; cout.flush();}

    //assign best target set
    vector<Node*> &bestSet = allTargets[bestTargetSet];
    for(int t=0; t<node->nTerminals; t++)
    {
        if(t<bestSet.size())
            node->terminals[t]->queueTarget( bestSet[t] );
        else
            node->terminals[t]->queueTarget( NULL );
    }

    return false;
}
```

10.4. Excerpts from fsoScene.cpp

```
void Node::update(double t, double dt)
{
   path->update(t, dt, x, v);
    rv.set( v(1), -v(0) );      //rotate v 90 degrees clockwise

    bool lockOnPending = false;

    for(int i=0; i<nTerminals; i++)
    {
        terminals[i]->update(t,dt);
        if( !terminals[i]->lockedOn ) lockOnPending = true;
    }

    //re-target unused or redundant terminals automatically
    for(int i=0; i<nTerminals; i++)
    {
        // currently no target or currently targeting a node not pointing back
        if( terminals[i]->qTarget != NULL )
        if( ( terminals[i]->target == NULL ) ||
                ( (terminals[i]->target != terminals[i]->qTarget)  &&
                    ( !terminals[i]->target->isTargeting( this ) ||
                      isRedundant(terminals[i])  )   )    )
        {
            terminals[i]->setTarget( terminals[i]->qTarget, t );
            lockOnPending = true;
        }
    }

    //stager re-targeting of used terminals.
    if(!lockOnPending)
    {
        //prioritize waiting terminals
        bool nodeIsWaiting = false;
        for(int i=0; i<nTerminals; i++)
        {
            if( terminals[i]->qTarget != NULL &&
                 terminals[i]->qTarget != terminals[i]->target &&
             terminals[i]->qTarget->isTargeting(this) )
            {
                terminals[i]->setTarget( terminals[i]->qTarget, t );
                nodeIsWaiting = true;
```

```
                    break;
                }
            }

            if(!nodeIsWaiting)
            for(int i=0; i<nTerminals; i++)
            {
                if( terminals[i]->qTarget != NULL &&
                     terminals[i]->qTarget != terminals[i]->target &&
                 !terminals[i]->target->isRetargeting() )
                {
                    terminals[i]->setTarget( terminals[i]->qTarget, t );
                    break;
                }
            }
        }
    }
```

Bibliography

[1] http://www.darpa.mil/ato/solicit/orcle/; "*ORCLE Webpage*," .

[2] http://www.darpa.mil/ato/solicit/ORCLE/19novorcle.pdf; "*ORCLE Project Proposal Document*," .

[3] Xiang-Yang Li, Yu Wang, Peng-Jun Wan, Wen-Zhan Song, and Ophir Frieder; "*Localized Low-Weight Graph and Its Applications in Wireless Ad Hoc Networks*," 2004.

[4] J. Cartigny, D. Simplot and I. Stojmenovic; "*Localized minimum-energy broadcasting in ad-hoc networks*," In Proc. IEEE INFOCOM'2003.

[5] Prabhanjan C Gurumohan, Joseph Hui; "*Topology Design for Free Space Optical Networks*," IEE ICCCN, 2003, 576-579.

[6] Fang Liu, Uzi Vishkin, Stuart Milner; "*Bootstrapping Free-Space Optical Networks*," IEEE IPDPS, 2005.

[7] Toussaint, G.T.; "*The relative neighborhood graph of a finite planar set*," Pattern Recognition 12, 1980, 261-268.

[8] Yu Wang, Ivan Stojmenovic, Xiang-Yang Li; "*Bluetooth Scatternet Formation for Single-hop Ad Hoc Networks Based on Virtual Positions*," Draft, 2004.

[9] Ivan Stojmenovic, Jie Wu; "*Broadccasting and Activity-Scheduling in Ad Hoc Networks*," 2004.

[10] Gaurav Srivastava, Paul Boustead, Joe F. Chicharo; "*Comparison of Topology Control Algorithms for Ad-hoc Networks*," 2003.

[11] Ivan Stjmenovic and Jie Wu; "*Broadcasting and Activity-Scheduling in Ad Hoc Networks*," 2004.

[12] http://en.wikipedia.org/wiki/Delaunay_triangulation; "*Delaunay Triangulation*," .

[13] Bernard Chazelle; "*On the Convex Layers of a Planar Set*," IEEE Transactions on Information Theory, Vol. IT-31, No. 4, July 1985.

[14] Kenneth J. Supowit; "*The relative neighborhood graph, with an application to minimum spanning trees*," Journal of the ACM, vol. 30, no. 3, pp. 428--448, 1983.

[15] http://www.cse.unsw.edu.au/~lambert/java/3d/delaunay.html; "*Delaunay Triangulation Algorithms*," .

[16] Lambert; "*Convex Hull Algorithms*," http://www.cse.unsw.edu.au/~lambert/java/3d/hull.html, 1998.

[17] C. Georgiou, E. Kranakis, R. Marcelin-Jimenez, S. Rajsbaum, J. Urrutia; "*Distributed Dynamic Storage in Wireless Networks*," .

[18] Kemal Efe and Antonio Fernandez; "*Products of Networks with Logarithmic Diameter and Fixed Degree*," IEEE Transactions on Parallel and istributed Systems, Vol. 6, No. 9, September 1995.

[19] D. Coudert, A. Ferreira, and S. Perennes; "*De Bruijn Isomorphisms and Free Space Optical Networks*," IEEE, 2004.

[20] Guihai Chen and Francis C.M. Lau; "*Shuffle-Ring: Overcoming the Increasing Degree of Hypercube*," IEEE, 1996.

[21] Premkumar Vadapalli and Pradip K. Srimani; "*A New Family of Cayley Graph Interconnection*

Networks of Constant Degree Four," IEEE Transactions on Parallel & Distributed Systems, Vol. 7, No. 1, January 1996.

[22] Jon Louis Bentley; "*Multidimensional Binary Search Tree Used for Associative Searching*," Communications of the ACM, Vol. 18, No. 9, September 1975.

[23] Bernard Chazelle; "*On the Convex Layers of a Planar Set*," IEE Transations on Information Theory, Vol. IT-31, No. 4, July 1985.

[24] David M. Lane; "*Confidence interval for μ, standard deviation known*," http://davidmlane.com/hyperstat/B7281.html.

[25] Makoto Matsumoto; "*Mersenne Twister Homepage*," http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html.