3-2007

An Efficient Metaheuristic for Dynamic Network Design and Message Routing

Robert B. Hartlage

AN EFFICIENT METAHEURISTIC FOR DYNAMIC NETWORK DESIGN AND MESSAGE ROUTING

THESIS

Robert B. Hartlage, Captain, USAF

AFIT/GOR/ENS/07-10

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the United States Government.

AFIT/GOR/ENS/07-10

AN EFFICIENT METAHEURISTIC FOR DYNAMIC NETWORK DESIGN AND MESSAGE ROUTING

THESIS

Presented to the Faculty

Department of Operational Sciences

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Operations Research

Robert B. Hartlage, MS

Captain, USAF

March 2007

AFIT/GOR/ENS/07-10

AN EFFICIENT METAHEURISTIC FOR DYNAMIC NETWORK DESIGN AND MESSAGE ROUTING

Robert B. Hartlage, MS
Captain, USAF

Approved:

Gary W. Kinney Jr, Maj, USAF (Chairman)	date
Scott R. Graham, Maj, USAF (Member)	date
Dr. Kenneth M. Hopkinson (Member)	date

Abstract

The implementation of Net-centric warfare presents major challenges in terms of effectively and efficiently delivering critical information across the Global Information Grid. In many cases, the amount of information requested will exceed the capabilities of the network. One challenge is to dynamically design the network (assign transceivers) to maximize the amount of required information that can be transmitted and the quality of service for those transmissions – to best implement the communications tasking order. The problem is as follows: given a list of required message traffic, to include source, destination, size, and priority, design the network to maximize the delivery of the message traffic based on message priority and quality of service. Once the network is designed, the routing for the messages must be determined. Due to the dynamic nature of the problem and the combinatorial explosion in size as new network nodes are added, a quick-running heuristic approach is needed. In this research, a metaheuristic is developed to dynamically design the network based on the projected message traffic requirements and efficiently route the required messages on the network, based on priority, maximizing the number of messages successfully delivered and the quality of service of the delivery. The meta-heuristic is tested against previous efforts and is shown to generate high quality solutions in a very short amount of time relative to methods in the current literature.

To my Wife & Daughter

Acknowledgments

I would first like to recognize the sacrifices made by my family during the time we've spent here at AFIT. I am especially appreciative of my wife who has endured my absence at our dinner table on more than one occasion. Attending many church, family, and social events alone has, I'm sure, at times caused her to feel like a widow. It is to her great credit and my esteem that she was willing to fill-in on my behalf during those absences...thanks Babe for sticking with me, I love you! Although at the time this is being written she is yet too young to realize it, my daughter has also been a source of encouragement and support. Her ever sunny disposition and smiling face have lifted my spirits each evening when I finally find my way home…daddy love's you! To my parents and siblings I'd like to express my appreciation for your continuing love, advice, and support. Having a refuge from the slings and arrows of grad school was truly priceless in more than just a "MasterCard" kind of way. I'd also like to express my deepest appreciation to my thesis advisor Maj. Gary Kinney, for his endless patience, leadership, and encouragement through this tedious process; you're the best, Sir! Thanks to Maj. Laura Suzuki for the hours she spent aiding me in debugging the MATLAB "mess" in Appendix A. To my classmates, thanks for silently enduring my loud and frequent "soapbox moments" about school, general injustices, and neighborly relations, I'll be seeing many of you again, much to your chagrin! A special thanks is due to Travis for contributing to my expanding waistline through his exquisite baked goods. Finally, to the fine folks at EPF I bestow the dubious honor of teaching me, through many a poignant counterexample, that there <u>are</u> endeavors in life much more important than onfxrgonyy, such as NGNPGPP. Ben Shrode would most certainly agree…

Table of Contents

List of Figures

Page

List of Tables

Page

AN EFFICIENT METAHEURISTIC FOR DYNAMIC NETWORK DESIGN AND MESSAGE ROUTING

I. Introduction

Background

In the near future, military operations will rely heavily on network-centric warfare (NCW) capabilities for communications and in establishing and maintaining information superiority (Department of Defense 2000:8). In order for these capabilities to be fully realized, methods for efficiently routing communications through a dynamic network with limited bandwidth must be developed. As Erwin describes, the requirements placed on military networks are unique (Erwin 2006:1). Communication requests are likely to be very time sensitive and a majority of network nodes may be in motion. In such a network, large segments could be disabled unless a method to quickly and dynamically reconnect the network is developed.

Erwin's research defined several concepts and methods that this research will build upon. The goal of this research is to provide a theoretical construct and proof-of-concept for the implementation of wireless network optimization within a military NCW context. A software prototype will be created using MATLAB. The development of an optimization tool is critical if the DoD is to transition to NCW on schedule as stated in JV2020 (Department of Defense 2000:35).

Statement of the Problem

An integral part of intelligent preparation of the battlespace (IPB) is ensuring the communication connectivity of all units in the campaign. The DoD recognizes that current technology is capable of supporting this objective, but that robust doctrine for NCW must be developed. An essential step in the development of this doctrine is in creating a communications tasking order (CTO). For readers already familiar with the concept of an air tasking order (ATO), the CTO is easily understood by recognizing the natural parallels existing between the ATO and CTO. For instance, communications are analogous to air assets, sorties flown are analogous to the Quality of Service (QoS), sorties needed are analogous to the commodity flow requests, etc. CTO development is done by the policy-makers in much the same way as the ATO.

It is assumed in any given period there are more commodity requests than the communication network bandwidth can support. If the network were able to support all communications requirements, then the problem would be reduced to maximizing QoS for high priority message traffic.

Unfortunately, in many instances security concerns prevent the use of omni-directional broadcast radio frequency (RF) and satellite communications. This research considers two, more secure methods of transmission: directional RF and high-bandwidth directional laser. These methods reduce the probability of communication interception by the enemy. The drawback to using these methods is that they are directional; in order for the communications to be effectively transmitted, both the transmitting unit and receiving unit must be "pointed" at each other. Another disadvantage is transmission distance. In order for two units to communicate, each must be within the others effective radius or the signal might be too weak to be received.

Research Approach

This work will improve upon earlier results by making progress toward solving realistic sized network instances in an acceptable amount of time. Additionally, several methodological improvements over earlier work are introduced to more closely resemble a realistic scenario. This research will present an updated objective function that strengthens the connection between true objectives and the model representation. The impact of strengthening this relationship is that the objective value of a given solution is a better indicator of the true quality of the solution.

This research will also implement a more sophisticated metaheuristic search that is significantly faster than the LP based method used by Erwin. The heuristic is tested on a variety of network instances of varying size with varying characteristics for arc capacity, message bandwidth, and average number of transceivers at each vertex. Two versions of the metaheuristic are tested, each using different embedded shortest path heuristics, to determine which method performs better for each set of problem characteristics. Additionally, several metrics are developed based on upper bounds on the network characteristics and average QoS of the network in order to provide a basis for comparison.

Scope & Limitations

The NDP is known to be NP-complete (Wong 1978:3). This research is intended to address problem instances that are likely to arise in a typical military scenario. Several assumptions have been made here regarding the characteristics of the network instances that are considered here.

First, the upper bound on transceivers to be located at a single vertex is nine. It is also assumed that there are two possible transceiver types and that a single vertex may

have any combination of transceivers as long as the total number does not exceed nine transceivers. Two transceivers are adjacent if they are the endpoints of a common edge. Each transceiver may connect to at most one other transceiver.

The vertices in the network only communicate through transceivers. Two vertices are adjacent only if they are able to communicate through an adjacent pair of transceivers. Vertices may be adjacent to one another through more than one pair of transceivers. If two vertices are adjacent through more than one set of transceivers than the capacity available between the set of vertices is equivalent to the sum of the capacities of the arcs connecting them.

Each arc has associated with it both a capacity and a QoS. The commodity that is being flowed through the networks in this research is communications requests. For this research it is assumed that both arc capacities and message bandwidth to include packets assume integer values. Each communication request has a unique ID, an assigned priority, and a positive bandwidth. The priority scale used is integer values in the set [1, 5] with 5 being the highest priority. A sample arc is presented below along with a notional four vertex example of a network addressed by this research. Both graphics were originally presented by Erwin (Erwin 2006:22-23).

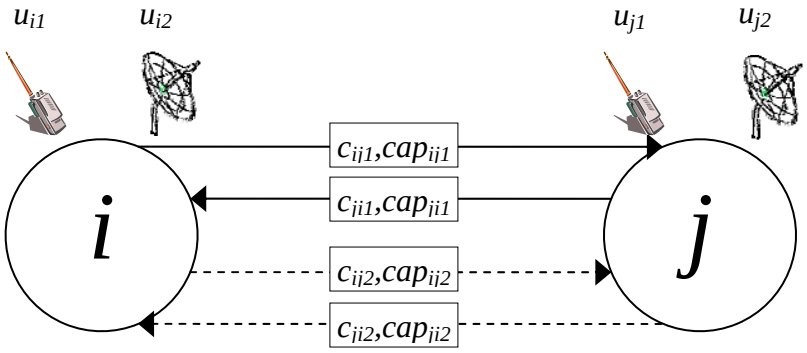


Figure 1: A sample arc for a network addressed by this research.

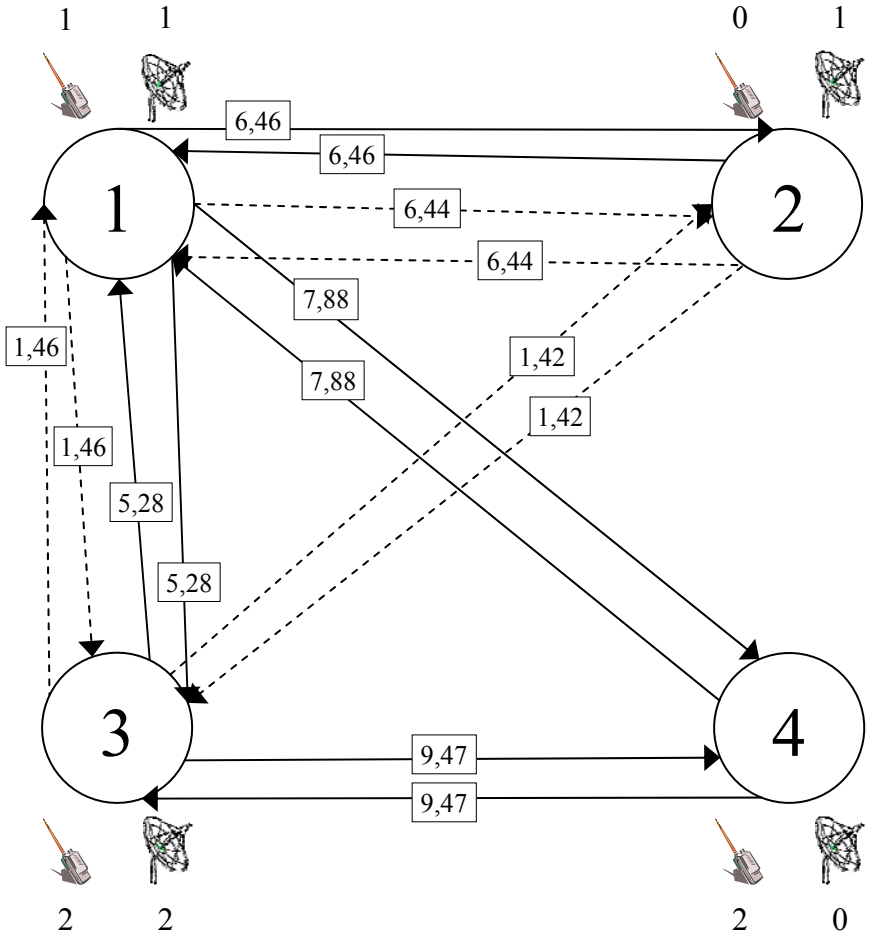


Figure 2: A notional communication network with four vertices.

Summary

The preceding sections present the motivation for this research. With these ideas in place, the following chapter will provide several concepts and relevant formulations that are either used directly in this research or have proven to be conceptually significant in the methodology upon which this work is based.

Chapter 3 will then describe the detailed methodology developed by this research and discuss several issues with transforming the methodology into prototype code. Chapter 4 develops a test plan and discusses the relevant test parameters. A majority of this chapter is dedicated to discussing appropriate heuristic testing procedures used and

how these procedures are implemented. Finally, Chapter 5 concludes with a brief summary of the research and several suggestions regarding potential areas of future research.

II. Literature Review

Introduction

Before delving into the formulations applied to the network structures in this problem, it is important to first characterize the communications networks used in this research. Following this review, several network formulations are presented. Each formulation has aspects that are useful either conceptually or directly within the special Network Design Problem (NDP) presented in this research. Next promising solution methods for the NDP are explored. This is followed by a general description of meta-heuristics. Finally, network metrics are explored to determine which are most suitable for measuring the quality of directional hybrid wireless networks.

Definition of Terms

In order to discuss network concepts it is necessary to provide formal definitions for the tenant graph theory concepts that are foundational to defining network structures. This section is provided as a quick reference for the reader to refer to in the following sections. For a more thorough treatment of any topic listed here the reader will find the source for these definitions to be instructive (West 2001).

Definition 1: A **graph** G is a triple consisting of a **vertex set** $V(G)$, an **edge set** $E(G)$, and a relation that associates with each edge two (not necessarily distinct) vertices called its **endpoints**.

Definition 2: Let G be a loopless graph with vertex set $V(G) = \{v_1,\ldots,v_n\}$ and edge set $E(G) = \{e_1,\ldots,e_m\}$. The **incidence matrix** $M(G)$ is the n-by-m matrix in which entry $m_{i,j}$ is 1 if v_i is an endpoint of e_j and otherwise is 0. If vertex v is an endpoint of edge e, then v and e are **incident**. The **degree** of a vertex v (in a loopless graph) is the number of incident edges.

Definition 3: When u and v are the endpoints of an edge, they are **adjacent** and are **neighbors**.

Definition 4: An **adjacency matrix** of G, written $A(G)$, is the n-by-n matrix in which entry $a_{i,j}$ is the number of edges in G with endpoints $\{v_i, v_j\}$.

Definition 5: A **matching** in a graph G is a set of non-loop edges with no shared endpoints.

Definition 6: A **maximal matching** in a graph is a matching that cannot be enlarged by adding an edge.

Definition 7: In general, a **network** is a digraph with a non-negative capacity $c(e)$ on each edge e and a distinguished source vertex s and sink vertex t. However, for the purposes of this paper a network is treated as an undirected graph.

Definition 8: The **connectivity** of a graph G is the minimum size of a vertex set S, such that $(G - S)$ is disconnected or has only one vertex. In the context of this research, connectivity is used as a measure of a particular networks' "robustness."

Definition 9: A **flow** f assigns a value $f(e)$ to each edge e.

Definition 10: A **walk** in a graph G, is a list $v_0, e_1, v_1, ..., e_k, v_k$ of vertices and edges such that for $1 \leq i \leq k$ the edge e_i has endpoints v_{i-1} and v_i.

Definition 11: A **path** in a graph G, is a walk with no repeated vertices.

In order to formulate a communications network several conventions regarding vertices and edges and how they relate to the physical network structures in question are adopted. In the context of this research, a vertex in a network is defined as a group of

transceivers with the ability to send, receive and route communications to other adjacent vertices. An edge in the network represents a transceiver pairing. Each edge has a commodity flow capacity. If two transceivers are connected by an edge then they are able to send and receive communications to one another. Each vertex may have multiple transceiver types and/or multiple units of the same transceiver as such vertices may share several common edges. The degree of a vertex is the number of transceivers at that vertex. In other words, it is the number of possible connections that can be made to/from the vertex. The upper bound on total transceiver units at each vertex is 20 implying the degree of every vertex in the network is constrained to be ≤ 20.

For an edge to connect two transceivers neither can be outside of the radius of communication of the other. The radius of communication, *r*, is the effective range of the transceiver. Although the communications are directional, the direction of the connection may be chosen as needed where the possible choices are in the degree range $\{0^0,\ 360^0\}$. Now a formal definition of communication radius is provided.

Let $d(i, j)$ = distance from vertex i to vertex j
Let $r(i, t_k)$ = maximum radial communication distance from vertex i of transceiver type k

Two vertices may be connected by an edge *iff*:

$$d(i, j) \leq \max_k (r(i, t_k), r(j, t_k))$$

Now that the defining characteristics of the real-world network have been given, it is appropriate to focus attention on the commodities that flow across the network. In this network instance, the commodity of interest is communication or rather the data streams that are transmitted through the network in order to relay communication. Each desired communication is referred to as a *commodity flow request* or simply a *request*.

This research assumes, at any given instance in time, commodity flow requests exceed network flow capacity. The requests are assigned a priority to indicate relative importance. Each request has a required bandwidth. Any request may be parsed into data streams (not necessarily of uniform bandwidth) that flow separately through the network from origin to destination. However, all streams of a single request must reach the destination or the request is considered "dropped" and the communication is not transmitted.

For each request, a path of maximum Quality of Service (QoS) is preferred. QoS has many definitions depending upon the application and context of the network. For the purposes of this research QoS is not defined but is characterized as an edge rating ranging from 0 (worst) to 1 (best). The QoS realized for a successfully transmitted request is equal to the lowest QoS edge among all paths carrying some fraction of the request.

Telecommunication Networks

A telecommunication network is a graph having the properties listed in definition 7 above. However, in a telecommunication network the flow assigned to each edge is information flow. Flow is assigned to a path based on the need to relay information from some original vertex called the source to some destination vertex called the sink. With many contexts, including military applications, there exist the potential to have multiple source vertices and multiple sink vertices. A natural question at this point is "What is the best design for the network?" The best design depends on the goals and needs of the end user.

Network Design Problem (NDP)

A network design involves choosing which vertices to connect in order to satisfy the information flow requirements. Ideally, every network should be complete, that is, all vertices in the network to be pairwise adjacent. In this situation, connectivity would be maximized and the NDP would be relatively easy to solve since bandwidth would most likely be plentiful.

However, connecting two vertices usually seizes some scarce resource, such as transceivers, of which there is an insufficient supply to connect the entire network. Additionally with wireless networks, weather, ground obstacles, and distance often preclude the possibility of a connection between every pair of vertices. In spite of these restrictions, finding an optimum configuration to satisfy information flow is still desirable.

There are several objectives to consider when designing a network. These are reliability, transparency, economy, convenience, and security (Pooch *et al* 1991). The reliability of a network is its ability to provide service without errors or interruption. The economy of a network is how well the resources available within the network are utilized. Network transparency refers to the ability of a user to access the networks resources without needing to know exactly where the resources are located. For example, a university student being able to utilize MATLAB simply by being logged onto the network without needing to understand where the application is physically stored. Convenience and security are self-explanatory and are not a concern in this context. This research deals with wireless technology assumed to be secure because of the two transceiver types being used. Additionally, the users of the network are aircraft, ground units and SOF teams, so convenience and transparency are assumed to be sufficient to

serve the purposes of each user. In fact, reliability and economy are the only suitable network quality design characteristics within the context of this research and therefore the metrics for quality will measure these two objectives.

One possible solution method is the NDP mixed-integer formulation. Although a solution to the NDP mixed-integer formulation would be optimum, the problem is NP-complete, thus no polynomial-time algorithm exists for generating an optimum solution. The mixed-integer formulation minimizing the total cost of the network is as follows (Ahuja *et al* 1993):

Let x_{ij}^k denote the fraction of the required flow of commodity k to be routed from the source s^k to the destination d^k that flows on arc (i, j).

Let c^k denote the cost vector for commodity k (c_{ij}^k is the per unit cost for commodity k on arc (i, j)).

Let f denote the fixed cost vector for the construction of each arc in the network.

Let y_{ij} be a zero-one variable indicating whether arc (i, j) is selected as part of the network design.

$$\text{Minimize} \quad \sum_{1 \le k \le K} c^k x^k + fy \tag{2.1}$$

Subject to

$$\sum_{\{j:(i,j)\in A\}} x_{ij}^k - \sum_{\{j:(j,i)\in A\}} x_{ji}^k = \begin{cases} 1 & \text{if } i = s^k \\ -1 & \text{if } i = d^k \\ 0 & \text{otherwise} \end{cases} \qquad \forall\ i \in N, k = 1, 2, ..., K \tag{2.2}$$

$$x_{ij}^k \le y_{ij} \qquad \forall\ (i, j) \in A, k = 1, 2, ..., K \tag{2.3}$$

$$x_{ij}^k \ge 0 \qquad \forall\ (i, j) \in A,\ k = 1, 2, ..., K \tag{2.4}$$

$$y_{ij} \text{ is binary} \qquad \forall\ (i, j) \in A \tag{2.5}$$

In this basic formulation, the objective function seeks to minimize the total cost of constructing the edges and assigning flow to the edges. Constraints (2.2) are the flow

balance constraints. These constraints specify that total flow into a vertex, less flow out of a vertex must be equal to 1 if the vertex is a source vertex, 0 if the vertex is a transshipment vertex and -1 if the vertex is a destination vertex. Constraints (2.3) are logical constraints that prevent flow on an arc if the arc is not included in the topology. Constraints (2.4) specify that all flows must be non-negative. Finally, constraints (2.5) specify that variables corresponding to arcs in the topology are binary. That is, an arc is either included in the topology or it is not.

Although this may be a useful formulation, there are several modifications that would need to be made to model the specific situation explored in this research. First, this formulation is an example of the uncapacitated network design problem in which the assumption is made that every edge in the network has an unlimited capacity for handling flow assigned to it. Second, the objective function that is used in this formulation may not reflect the objectives that would be most important in a military wireless context. As presented there is a fixed cost associated with making a connection between two vertices and a unit cost associated with assigning flow to an edge. The objective function seeks to minimize the total cost of constructing the edges and assigning flow to the edges.

The formulation is modified below to more closely model the network type used in this research (Erwin 2006:26-27). The formulation still does not fully capture the nature of the problem but provides a foundation for comparing the approach presented in this research with that presented in Erwin. However, these two formulations for the basis of the problem addressed by this research and are presented in their entirety. In Chapter 3 an updated objective function is presented that more closely evaluates the true objectives of the NDP in a military wireless network.

Let N denote the set of nodes, K the number of commodities, and F the number of interface types.

Let (i, j, f) denote the arc connecting node i to node j by interface type f.

Let A denote the node-incidence matrix where $a_{ijf} = 1$ if node i is incident to node j via interface type f, and $a_{ijf} = 0$ otherwise.

Let x_{ijf}^k denote the fraction of the required flow of commodity k to be routed from the source s^k to the destination d^k that flows on arc (i, j, f).

Let y_{ijf} denote the binary variable indicating whether arc (i, j, f) is selected as part of the network topology.

Let v_{ijf}^k denote the per unit cost for commodity k on arc (i, j, f).

Let c_{ijf} denote the fixed cost of including arc (i, j, f) in the network.

Let u_{if} denote the number of interfaces of type f at node i.

Let b^k denote the the required bandwidth for commodity k.

Let cap_{ijf} denote the capacity of arc (i, j, f).

Let r^k be the total bandwidth of commodity k.

$$\text{Minimize} \quad \sum_k \sum_{\{(i,j,f):\, i<j,\, a_{ijf}=1\}} v_{ijf}^k x_{ijf}^k + \sum_{\{(i,j,f):\, i<j,\, a_{ijf}=1\}} c_{ijf} y_{ijf} \tag{2.6}$$

Subject to

$$\sum_{\{j,f:\, a_{ijf}=1\}} x_{ijf}^k - \sum_{\{j,f:\, a_{jif}=1\}} x_{jif}^k = \begin{cases} 1 & \text{if } i = s^k \\ -1 & \text{if } i = d^k \\ 0 & \text{otherwise} \end{cases} \qquad \forall\ i \in N, k = 1,\ldots,K \tag{2.7}$$

$$\sum_k r^k x_{ijf}^k \le cap_{ijf} \qquad \forall (i, j, f) \in A\ \ni a_{ijf} = 1 \tag{2.8}$$

$$\sum_{j \in N} y_{ijf} \le u_{if} \qquad \forall i \in N,\ f = 1,\ldots,F \tag{2.9}$$

$$x_{ijf}^k \le y_{ijf} \qquad \forall (i, j, f) \in A\ \ni a_{ijf} = 1, k = 1,\ldots,K \tag{2.10}$$

$$y_{ijf} = y_{jif} \qquad \forall (i, j, f) \in A\ \ni a_{ijf} = 1 \tag{2.11}$$

$$x_{ijf}^k \ge 0 \qquad \forall (i, j, f) \in A\ \ni a_{ijf} = 1,\ k = 1,\ldots,K \tag{2.12}$$

y_{ijf} is binary $\qquad \forall (i, j, f) \in A\ \ni a_{ijf} = 1$ (2.13)

Because of equations 2.7 and 2.8, this formulation only has a feasible solution if the network contains sufficient capacity to route all of the commodities (Erwin 2006:27).

Degree Constrained Minimum Spanning Tree Problem

In any network it may be desirable that the network is connected - a path exists between every two vertices in the network. One way to ensure that a network is connected is by solving the Minimum Spanning Tree (MST) problem. However an MST may produce an infeasible solution if there are degree constraints associated with the vertices. In this case, it would be necessary to solve the degree constrained Minimum Spanning Tree Problem (dcMST). Each vertex in the network has associated with it a maximum degree which is equivalent to the number of transceivers located at that vertex. Since every transceiver can communicate with at most one other transceiver, the number of edges connected to any vertex in the network is at most k where k is the number of transceivers located at that vertex. The dcMST ensures connectivity by building a backbone tree structure in which a path exists between every vertex and, since the final structure is a tree, no cycles are present in the solution. The integer linear program formulation for the dcMST is originally presented by Guéret and Christelle (Guéret *et al*. 2000):

Let $N = \{1, 2, ..., n_N\}$ be the set of all n_N nodes in the network.

Let $F = \{1, 2, ..., n_F\}$ denote the set of n_F different types of interfaces used in the network.

Let A be the $(n_N \times n_N \times n_F)$ node-incidence matrix with $a_{ijf} = 1$ if node i is incident to node j by interface type f and $a_{ijf} = 0$ otherwise.

Let y_{ijf} denote the binary decision variable indicating whether or not edge (i, j, f) is chosen. $y_{ijf} = 1$ if chosen, $y_{ijf} = 0$ otherwise.

Let c_{ijf} denote the cost of including edge (i, j, f) in the network.

Let u_{if} denote the number of interfaces of type f at node i.

Let $Level_i$ denote the integer value that corresponds to the number of links in the path from the root node to node i.

$$\text{Minimize } z = \sum_{\{(i,j,f):\, a_{ijf}=1\}} c_{ijf}\, y_{ijf} \tag{2.14}$$

Subject to

$$\sum_{\{(i,j,f):\, a_{ijf}=1\}} y_{ijf} = (n_N - 1) \tag{2.15}$$

$$Level_j \geq Level_i + 1 - n_N + n_N\left(\sum_f y_{ijf}\right) \quad \forall i, j \in N \ni \sum_f a_{ijf} \neq 0 \tag{2.16}$$

$$\sum_{\{(j,f):\, a_{ijf}=1\}} y_{ijf} = 1 \qquad \forall i \in \{2, 3, ..., n_N\} \tag{2.17}$$

$$y_{ijf} \text{ is binary} \qquad \forall (i, j, f) \in A \tag{2.18}$$

$$Level_i \geq 0 \text{ is integer} \qquad \forall i \in N \tag{2.19}$$

The objective in this formulation is to minimize the total cost of building the minimum spanning tree. Equation (2.15) ensures that the number of edges included in the solution is exactly one less than the total number of vertices; a necessary condition for discerning whether the network in question is in fact a tree. Constraints (2.16) ensure that no solution will contain either directed or undirected cycles and constraints (2.17) ensure that every vertex is included in the tree.

The formulation for the dcMST adds constraints that dictate the maximum number of edges connected to each vertex. This set of constraints is given by:

$$\sum_{\substack{i \in N \\ f \in F}} y_{ijf} \leq k \qquad \forall j \in \mathrm{N} \qquad (2.20)$$

Where k is the number of transceivers at vertex j for $\forall j \in N$

Although solving the previous dcMST ensures network connectivity, a more accurate measure of quality in a communication network is the extent to which the network topology is able to support the required commodity flow.

Bipartite Maximum Weight Matching

Every vertex in the network represents a group of transceivers. A connection between two vertices may be composed of several edges between the transceivers at these vertices. If a connection between vertex i and vertex j is included in the network, it should be the best subset of edges from i to j. The bipartite maximum weight matching (BMWM) problem can be employed to guarantee that the best possible connection from i to j has been obtained. The BMWM can be formulated as an assignment problem simply by adding "dummy" transceivers to the partite set (vertex) that is of smaller cardinality and the proceeding to solve the assignment problem given below (Wolsey 1998).

Let G = (V_x, V_y, E) be a graph representation of the node pair (x, y) in question

Let V_x and V_y be the partite sets of transceiver interfaces at vertices x and y respectively.

Let c_e be the weight assigned to arc e for $\forall e \in$ E

Let $\delta(i)$ be the set of potential arcs for $i \in V_x \cup V_y$

Let x_e be the binary variable indicating if arc $e \in E$ is included in the matching

$$Max\ z = \sum_{e \in E} c_e x_e \tag{2.21}$$

Subject to

$$\sum_{e \in \delta(i)} x_e \leq 1 \text{ for } i \in V_x \cup V_y \tag{2.22}$$

$$x_e \geq 0 \text{ for } e \in E \tag{2.23}$$

Network Metrics

Below is a definition for *metric space* used in this research (Marsden & Hoffman 1993):

A *metric space* (M, d) is a set M and a function $d : M \times M \rightarrow R^1$ such that :

i. $d(x, y) \geq 0$ *for all* $x, y \in M$.
ii. $d(x, y) = 0$ *iff* $x = y$.
iii. $d(x, y) = d(y, x)$ *for every* $x, y \in M$.
iv. $d(x, y) \leq d(x, z) + d(z, y)$ *for all* $x, y, z \in M$.

A *metric* is the distance function defined over a metric space. The metric used for rating the quality of a particular network design is based on QoS, packet priority, and packet bandwidth. While two of these quantities are intuitive, QoS has many definitions depending upon the application. We shall define QoS_{ij} in this research as "the probability that flow assigned to arc (*i,j*) successfully completes the traversal from vertex *i* to vertex *j* without being lost." In this research, the *traffic contract* is the commodity flow requests generated by the users located at various nodes throughout the network.

The objective functions presented in the NDP formulations above do not accurately reflect the goals of a military wireless network. For instance, there is generally no fixed cost for establishing a link in a wireless network. The same is true of the variable cost associated with transmitting a message. Instead, QoS is a concern since it acts as a proxy for network reliability. The priority of one message relative to other messages is also not addressed in the previous objective functions. A more suitable objective function for the application of military networks is:

$$\sum_{k \in \text{message routing}} (QoS_k) * (Priority_k) * (Bandwidth_k) \tag{2.24}$$

where *message routing* is the set of requests successfully routed over the current network topology, QoS_i is the quality of service obtained by message i, and $Bandwidth_i$ is the bandwidth of message i. Since a message in the routing may be split into two or more packets, QoS_i is taken to be the minimum QoS of any packet of message i in the routing. Also, $Bandwidth_i$ is the sum of the bandwidths of all packets for message i. A solution to this problem is therefore a topology and routing together that maximizes equation (2.24).

The A* Heuristic

A* is a shortest-path heuristic that is similar to Dijkstra's algorithm but includes a heuristic pricing function to estimate the distance to the end node. A* is preferable to Dijkstra's algorithm in some applications. Dijkstra's algorithm has a run time bound of order $O(n^2)$ while A* is $O(n)$. This savings in computational time is important for dynamic military applications. The tradeoff for this reduction in computational complexity is that the guarantee of optimality is lost. A* becomes Dijkstra's algorithm if the heuristic pricing function value is omitted at each iteration forcing the actual distance to be calculated. It is important to note that the optimality guarantee of Dijkstra's

algorithm only applies to the path that is found. Within the context of the NDP formulation that is addressed by this research, Dijkstra's algorithm may or may not produce better results than A*. Comparing the two path finding approaches will be a main focus of Chapter 4. The basic steps of the A* heuristic are outlined below:

1. Create a search graph, *G* consisting only of the start node.
2. Create a list called "*OPEN*" which contains only the start node.
3. Create a list called "*CLOSED*" that is empty.
4. If *OPEN* is empty then exit with failure.
5. Select the first node on *OPEN*, remove it from *OPEN* and place it on *CLOSED*. Call this node *n*.
6. If *n* is the destination node then exit successfully with a solution obtained by tracing a path along the pointers from *n* to the start node in *G*.
7. Expand node *n*, generating the set, *M*, of its successors that are not already ancestors of *n* in *G*. Install these members of *M* as successors of *n* in *G*.
8. Establish a pointer to *n* from each of those members of *M* that were not already in *G* (not already on either OPEN or CLOSED). Add these members of *M* to OPEN. For each member, *m*, that was already on OPEN or CLOSED, redirect its pointer to *n* if the best path to *m* found so far is through *n*.
9. Reorder the list OPEN in non-decreasing order by pricing function value.
10. Go to step 4.

In the A* heuristic the pricing function is given by $f_i = g_i + h_i$ where g_i is a function that returns a value denoting the shortest distance from the start node to node *i* found by A* so far and h_i is the estimate of the distance remaining from node *i* to the destination node. In order to compare the performance of the A* heuristic and Dijkstra's

algorithm, the set of test cases discussed in Chapter 4 will be solved with a metaheuristic using A* and then solved using Dijkstra's algorithm by turning off the heuristic pricing function (i.e. setting $h_i = 0 \ \forall i \in OPEN$).

Message Routing

The product of the A* shortest path heuristic is a path that connects the source and destination for the communication request being routed. In this case, the "shortest path" is a path that maximizes the minimum QoS connecting source to destination. Once a path is located in the search then message bandwidth is assigned to the path until either the entire message is routed or the bandwidth on the path is exhausted. If the bandwidth on the path is exhausted before the entire message is routed then the A* heuristic is repeated to find another path connecting source and destination for the current message.

Greedy Matching Algorithm

This research adopts a simple greedy heuristic approach to choose the arcs in the initial solution with the weights assigned to each edge calculated in the following manner:

Let QoS_{ij} be the quality of service for arc (i, j).
Let x_{ijk} be the units of bandwidth of message k that is routed over arc (i, j).
Let $Priority_k$ be the priority of message k.

$$arc\_weight_{ij} = \sum_{k \in \text{message routing}} \sum_{(i,j) \in A} QoS_{ij} * x_{ijk} * 10^{Priority_k} \tag{2.25}$$

These edge weights attempt to capture the contribution each edge is making to the overall objective function value. Once the edge weights have been assigned to the

feasible edges in the network, the greedy heuristic operates to develop a maximum weight matching between transceivers as follows (Wolsey 1998):

1. Initialize the set *M*, the set of arcs included in the matching which is empty initially.
2. Select the maximum weight arc (*i*,*j*) from the transceiver adjacency matrix and assign the corresponding arc to the set *M*.
3. Set all non-zero elements in row *i*, row *j*, column *i*, and column *j* to zero since these transceivers have been matched and may not be included in any other matching.
4. Return to step 2 and iterate until all entries in the transceiver adjacency matrix are found to be zero.

Metaheuristics

A metaheuristic is a heuristic that guides one or more other heuristics. The A* shortest path finding heuristic uses a heuristic pricing function as previously discussed. The concept of a heuristic is similar to that of a "rule of thumb". A rule of thumb is simple rule used to guide decisions when an optimal policy is unknown or cannot be determined. In the case of A* the distance remaining to the destination is unknown so it is estimated by simply averaging the QoS on all arcs emanating from the current vertex. Since the heuristic developed in Chapter 3 guides both a greedy matching heuristic and an A* path finding heuristic, it is an example of a metaheuristic.

Summary

This section has presented several formulations that are important to the formulation of the military wireless NDP. Not all of the formulations here are used

directly in the final implementation but all formulations help provide conceptual insights useful in creating the appropriate problem representation.

A military network is dynamic and has several additional constraints not usually present in a standard NDP. These constraints combined with the size of a typical military network employed in network-centric warfare, require the development of efficient solution methods to provide nearly uninterrupted service throughout the network. One advantage of using heuristics is that they generally provide a near-optimal solution in a very reasonable amount of time. Although optimality is not guaranteed they perform exceedingly well in many practical applications.

In Chapter 3 a metaheuristic called NetDesign is developed to solve the NDP discussed earlier. NetDesign begins by assigning each potential arc in the network a weight. Using these weights, a greedy matching heuristic forms a matching which represents a feasible network topology. Once the topology is developed, the shortest path is found for each the prioritized communications requests using either the A* heuristic or Dijkstra's algorithm. The message is routed along this path until the capacity of the path is gone. The routing along with the topology represents a feasible solution. A local improvement scheme is employed to examine the current solution and reweigh the potential arcs.

In addition to presenting the development of NetDesign, several graph theoretic lemmas and corollaries used in the implementation of NetDesign to check for existence of a path and address the topology generated by the greedy matching are provided. Finally, to illustrate the operation of NetDesign a brief example is also provided.

III. Methodology

The NetDesign metaheuristic developed in this research combines the A* shortest-path heuristic with a greedy matching heuristic. An arc weighting scheme is used to produce the greedy matching that represents the network topology portion of the solution. This chapter presents the detailed methodology and some important aspects of the software implementation. For reference, the reader is directed to Appendix A which contains the MATLAB functions necessary to implement the NetDesign metaheuristic.

User Inputs & Data Requirements

All of the data, including network characteristics are randomly generated. The network characteristics are taken from a uniform distribution between the user-specified upper and lower bounds.

Simulation & Mission Planning

Several functions were written to provide the necessary data inputs or to format those inputs for use in the program. The NetDesign metaheuristic can be used in two ways. First, based on user-specified inputs, the software is capable of generating networks for the purpose of simulation or testing. This function may be important if general characteristics of the network are known but no specific instance is given. This functionality can be used to determine the nature and quality of solutions based on a particular set of network characteristics. The list of user-specified inputs is described in Table 1.

Table 1: User Defined Inputs

User Defined Inputs	Description
MAX_TRANSCEIVERS	The maximum number of transceivers to be located at any given vertex.
MAX_COMM_RAD	The upper bound on communication radius for any transceiver.
MAX_COMM_REQ	The maximum number of communications requests to be generated for transmission across the network.
PRIORITY_SCALE	Assigns the maximum priority (this number determines the scale). 1 is the lowest priority and 5 is the highest priority.
AVAILABLE_BANDWIDTH	The maximum bandwidth capacity of an edge in the network. Capacity is a randomly generated number between 1 and AVAILABLE_BANDWIDTH for each edge.
COMM_BANDWIDTH	The maximum size of a communication request to be sent over the network. Communication request size is a randomly generated number between 1 and COMM_BANDWIDTH for each request.

The second and primary application is to find high-quality feasible solutions to the NDP for a predefined network instance. This type of application will be used for mission planning. The data required by NetDesign for this application are described below. All of the data are assumed to be available and in the correct formatted.

Table 2: Data Requirements

NetDesign data requirements	Description
M	The matrix containing information on the total number of transceivers of each type located at each vertex. Each row represents a different vertex.
Trans_Char	The structure containing information about each individual transceiver. There is one element of *Trans_Char* for each transceiver in the network. *Trans_Char* has the following fields.
Trans_Char.Vertex_ID	The unique ID of the vertex at which the transceiver is located.
Trans_Char.Transceiver_ID	The unique ID associated with the transceiver.
Trans_Char.Transceiver_type	The integer (either 1 of 2) indicating the type of transceiver.
Trans_Char.CommRadius	The effective communication radius of the transceiver.
Trans_Char.Xloc	In Euclidean space this is the abscissa of the vertex associated with the transceiver.
Trans_Char.Yloc	In Euclidean space this is the ordinate of the vertex associated with this transceiver.
Trans_Char.XPlotLocation/YPlotLocation	Used for graphical representation and visual clarification. Each transceiver is given a slightly different plot location based on the perimeter of an imaginary circle centered at the vertex coordinates.
A	The matrix of ordered pairs denoting the location of the vertices.
Num_Vertices	The total number of vertices in the network.
Num_Transceivers	The total number of transceivers in the network.
I	The transceiver adjacency matrix filtered for communication radius and transceiver type.
P	The structure storing information about the communication requests to be routed. This structure has fields for ID, Bandwidth, Origin, Destination, and Priority.
QoS	The matrix whose entries represent the QoS for each potential edge in the network.
Bandwidth	The matrix whose entries represent the bandwidth of each potential arc in the network.

Once the data are established, by either of the preceding two methods, the remainder of the metaheuristic operates in the same manner. There are several preprocessing steps that the data must undergo to produce the essential meta-data used to make critical decisions in the metaheuristic.

Prior to producing a topology, limitations on the network connectivity must be determined. Specifically due to communication radius of transceivers or incompatibility of transceiver types it is possible that no path exist between a pair of vertices. Therefore NetDesign determines if a path exists between successive source/destination pairings based on the prioritized communication request list. Any such path will contain at most *Num_Vertices*-1 edges. Connectivity between vertices is verified by summing successive powers of the adjacency matrix in a matrix geometric series.

Lemma 1: Given a vertex adjacency matrix *A* of a graph *G*, the total number of paths of length *n*, connecting vertex *i* and *j* is given by:

$$[A^n]_{ij} = \sum_{k_1,k_2,\ldots,k_{n-1}=1} a_{ik_1}a_{k_1k_2}\ldots a_{k_{n-2}k_{n-1}}a_{k_{n-1}j} \text{ (Meyer 2000).}$$

Corollary 1: In a graph *G* the total number of paths connecting vertices *i* and *j* of length less than or equal to *n* is given by

$$[A]_{ij} + [A^2]_{ij} + [A^3]_{ij} + \cdots + [A^n]_{ij} = [A + A^2 + A^3 + \cdots + A^n]_{ij} \text{ (Meyer 2000)}$$

Proofs for Lemma 1 and Corollary 1 are provided in Meyer (2000).

Proposition 1: Let *n* be the number of vertices in a graph *G*. Given a symmetric vertex adjacency matrix *A*, then for $i, j \in V(G)$ with $i \neq j$, if an *i, j-path* exists then the length of that path is at most *n-1*.

Proof

Given an *i,j-path* in G of length *n* such that all vertices are distinct. The length of a path is defined to be the number of edges in the path. Since a path begins and ends at a vertex there must always be one more vertex than the number of arcs in the path. Thus, the *i,j-path* must contain *n+1* distinct vertices. However G contains only *n* vertices and

therefore the maximum length of a path in a graph G such that all vertices are distinct is *n-1*.

Now by Corollary 1 and Proposition 1 it is possible to determine if a path exists between any two vertices by examining the matrix $V = A + A^2 + A^3 + \ldots + A^{n-1} = \sum_{i=1}^{n-1} A^i$ for a zero entry. If $V_{ij} = 0$ then no *i,j-path* exists in *G*. If no path exist between the source and sink for a message request, the request is removed from the list. This step prevents unnecessary calculations in later segments of the program.

The last item of pre-processing is to sort the communications requests in non-decreasing order by priority. Each request is assumed to have a priority associated with it. The notional priority scale used in this research is the set of integers 1,2,3,4, and 5 with 5 being the highest priority. The relationship between elements in the priority scale is an important factor in calculating the objective function. For the purposes of this research it is assumed a priority 5 request is infinitely more important than a priority 4 request, a priority 4 request is infinitely more important than a priority 3 request, and so on. Within priority groupings the requests are sorted by bandwidth based on the belief larger messages should be routed early in the search process while paths of sufficient bandwidth still exist.

After all data items have been established and the required pre-processing is completed, it is possible to run NetDesign. The following section provides a description of the NetDesign metaheuristic and the solutions that are generated.

The NetDesign Metaheuristic

The NetDesign metaheuristic contains two principal components: an A* shortest-path heuristic and a simple greedy matching heuristic. The table below provides an

overview of the basic steps of NetDesign. Note that steps 1, 2, and 3 are the pre-processing steps discussed above.

1. Determine the nonexistent paths in the network.
2. Build QoS, Bandwidth, and Prioritized_Requests matrices.
3. Determine the messages that are not transmittable and remove these messages from the Prioritized_Requests matrix.
4. Determine the initial communication request routing, used in calculating initial arc weights, based on a full, but infeasible, network topology.
5. Generate arc weights for each potential arc in the network.
6. Create the network topology using a greedy matching heuristic.
7. Determine the initial feasible message routing for the topology generated by the greedy matching heuristic.
8. Calculate the objective function value for the current solution and save the solution if it is the best found so far.
9. Compare the current solution to previous solution. If the topologies are identical then stop. Otherwise return to step 5.

The purpose of step 4 is to provide initial arc weights for generating an initial feasible network topology. All potential arcs are added to the network topology. In other words, the constraint that any transceiver communicates with at most one other transceiver is relaxed.

A* is used to route the communications requests until either all messages have been routed or there is insufficient bandwidth to route any additional messages. First A* finds the highest quality node path to connect the origin and destination for the current element of the prioritized request list. Message traffic can be split into packets

since it may not be possible to find a single path with sufficient bandwidth to route the entire communication request. The proportion of the message bandwidth that path bandwidth can hold is routed and the available bandwidth along the path is updated. The bandwidth of the path is determined by taking the minimum bandwidth of all arcs on the transceiver path since this value is the maximum capacity of the path. This process is continued until either all of the message has been routed, or it has been determined that no more paths exist to connect the origin and destination pair for the current communication request in which case the current element of the prioritized request list is discarded. If it was not possible to route the entire message then the partial routing for the current element is removed from the solution structure and the bandwidth used by the partial message is restored. A* then moves to the next highest priority request on the list.

After the initial routing has been determined, it is used to calculate the initial arc weights in step 5. Each arc is given a weight based on the bandwidth assigned to it, the priority of the messages, and the Quality of Service of the arc. The goal of the weight is to capture the arc's contribution to the objective function. As a result the more desirable arcs for routing will have the highest weight and will be most likely included in the next iteration's topology. All non-adjacent transceiver pairings are assigned a weight of zero.

After updating the arc weights, the greedy matching heuristic is used to determine the topology for the next iteration. At each iteration, the arc with the largest weight is added to the topology. The weights for all arcs touching the transceivers of the added arc are set to zero. The greedy matching heuristic successively selects arcs in this manner and terminates when all non-zero arcs have been added or no non-zero arcs can be added without violating the maximum number of transceiver connections. The resulting

matching is maximal. That is, no larger matching contains it since it cannot be enlarged by adding additional arcs.

Proposition 2: The matching created by the greedy heuristic is maximal.

Proof

At each iteration the maximum element of the adjacency matrix, element a_{ij}, is chosen and the associated arc is added to the topology. All elements in rows i and j and columns i and j are set to zero. This process is repeated until no non-zero elements remain in the matrix. Since no non-zero elements remain in the adjacency matrix, there are no further feasible pairings among the transceivers in G. Therefore, the current matching cannot be enlarged by adding an arc.

The matching produced at the prior step represents a feasible network topology. Once a topology has been established, a routing is again determined by applying A* for each request, and updating the bandwidth and QoS matrices. As before, the communications requests are routed in order of highest priority. After A* has attempted to route each message in the list, the topology and routing are saved in a solution structure. Each element of the structure represents a packet and contains all information needed to route the communication through the current network topology.

Solution Representation

The solution information is stored in a structure segmented based on message packets. The fields of the solution structure are shown in Table 3.

Table 3: Fields contained in "Post_match_comm_routing" solution structure

Description of the solution structure "Post_match_comm_routing"	
message_ID	The unique ID associated with the message being transmitted
message_bandwidth	The bandwidth of the packet routed by this structure element.
node_path	The vector containing, in order, the nodes on the node path from origin to destination.
trans_path	The vector containing, in order, the transceivers of the transceiver path from origin to destination.
path_QoS	The QoS for the path that this packet is being routed over.
message_Priority	The priority of the message to which this packet belongs.

Each element of the structure represents a packet to be routed. Each structure element contains all information needed to route and track the message and calculate objective function values. The *message_ID* field contains the unique identification number for the message of which the packet is a portion. The *message_bandwidth* field contains the scalar value that is the bandwidth of the packet. Summing the bandwidth for all packets associated with message *i* should yield the bandwidth of message *i*. *message_Priority* is the priority of the message of which the packet is a part.

node_path is a vector containing the ordered list of vertices representing the path connecting the message origin and destination nodes. *trans_path* is a vector containing the ordered list of transceivers connecting the vertices in *node_path* The *path_QoS* is the minimum QoS of any arc on the transceiver path. The assumption is made that the QoS of a path is only as high as the weakest arc contained in the path. *Message_bandwidth*, *message_priority* and *path_QoS* are used in the calculation of the objective function value.

The objective function value of the *best_solution* is initially set to zero. At the end of each iteration, the objective function value is calculated and compared to the current *best_solution* objective function value. If the current solution has achieved a

better objective function value than the best solution found so far, then the best solution is replaced by the current solution and the *best_solution* objective function is updated.

Once the messages have been routed a feasible solution to the NDP has been created. That is, both a feasible network topology and a feasible communication request routing have been determined. With an initial feasible solution, the goal now is to improve both the topology and the routing to accommodate as much message traffic as possible while assigning messages of higher priority with paths of higher QoS. The solution improvement scheme is accomplished by iterating steps five though eight. Solution improvement is one important aspect of any metaheuristic.

Another important feature is that of convergence. Due to the way the network topology is represented in the solution it is possible to measure the convergence of successive solution topologies by assigning each potential arc to an element of a binary vector. The topology can then be represented by setting to 1 the elements of the topology vector corresponding to the chosen arcs. Now it is possible to use the standard hamming distance or the number of edges by which two solutions differ as the "distance" between solutions. By measuring the distance between successive solutions it is possible to determine if the NetDesign metaheuristic is indeed converging. Once two successive solutions have identical topologies, implying that the hamming distance between the two successive solutions is equivalent to zero, the search is terminated.

Summary

This chapter describes a methodology to find a high quality network design and routing based on a set of message requests. The NetDesign metaheuristic combines a greedy matching heuristic to design the network based on the projected use for each possible arc with a routing heuristic, A*, to find high quality paths from source to sink for

each message in the prioritized communication request list. The metaheuristic can be iterated until it converges to a fixed network topology. The best solution discovered during the search is returned. Chapter 4 develops a test plan and several test cases to test the NetDesign methodology to determine the quality of solutions produced and the operating characterizes of NetDesign.

IV. Testing

In order to test the NetDesign Metaheuristic, 32 test cases were developed using Design of Experiments. Although the goal of the research was not to perform a full DOE analysis or optimize a response variable, the concept of a 2^k full factorial experimental design was quite useful in preparing test cases for the testing phase and in determining the factor effects. All testing was done on an IBM ThinkPad with 1.86 GHz processor, 1 GB DDR RAM, and a 40 GB hard drive.

Test Data Sets

The network characteristics that were tested are: number of vertices, number of transceivers located at each vertex, the bandwidth of the arcs in the network, number of messages, and differing levels of dispersion about the mean for each of the four preceding factors. With $k = 5$ factors for testing in a 2^k full factorial experiment equates to 32 test cases. Additionally, each of the 32 test cases are tested in NetDesign using both the A* heuristic and Dijkstra's algorithm to compare the performance. Although Dijkstra's algorithm finds an optimal path such paths may provide sub-optimal results overall in the context of the full network topology and routing solution

Table 4 shows the test case characteristics. Note that each factor is tested at a high and low level as indicated. While the number of vertices and dispersion are deterministic, the number of transceivers, arc bandwidth, and message bandwidth are randomly generated integers obtained from a uniform distribution over the interval $(x-d, x+d)$ where x is the center defined by the parameter for vertices, transceiver, message bandwidth, or arc bandwidth, and d is the dispersion parameter.

Table 4: Table of test cases with factor levels

Case	Number of Vertices	Average Number of Transceivers	Average Message Bandwidth	Average Arc Bandwidth	Dispersion
ndp1	30	4	10	30	2
ndp2	30	4	10	30	3
ndp3	30	4	10	90	2
ndp4	30	4	10	90	3
ndp5	30	4	30	30	2
ndp6	30	4	30	30	3
ndp7	30	4	30	90	2
ndp8	30	4	30	90	3
ndp9	30	5	10	30	2
ndp10	30	5	10	30	3
ndp11	30	5	10	90	2
ndp12	30	5	10	90	3
ndp13	30	5	30	30	2
ndp14	30	5	30	30	3
ndp15	30	5	30	90	2
ndp16	30	5	30	90	3
ndp17	40	4	10	30	2
ndp18	40	4	10	30	3
ndp19	40	4	10	90	2
ndp20	40	4	10	90	3
ndp21	40	4	30	30	2
ndp22	40	4	30	30	3
ndp23	40	4	30	90	2
ndp24	40	4	30	90	3
ndp25	40	5	10	30	2
ndp26	40	5	10	30	3
ndp27	40	5	10	90	2
ndp28	40	5	10	90	3
ndp29	40	5	30	30	2
ndp30	40	5	30	30	3
ndp31	40	5	30	90	2
ndp32	40	5	30	90	3

Network Metrics

There are several metrics that are important to the testing of the Metaheuristic developed under this research. Choosing metrics for heuristic testing is difficult in some respects since the optimal solutions for many of the problems being solved by metaheuristics are unknown. The computational complexity of the existing algorithms to solve the NDP to optimality prevents researchers from obtaining optimal solutions to all

but the smallest instances of this formulation. Erwin was not able to provide a feasible solution to the NDP for problem instances with greater than 35 vertices in eight hours of solver run time.

For this reason, this research relies on quality metrics other than optimality to measure the performance of NetDesign. The metrics presented here can be classified as measuring the quality of the network topology, the message routing, or a combination of both. Prior to discussing the quality of a specific solution it is important to examine the network topology potential. A solution topology with an average QoS of 0.60 may appear poor; however, if the average QoS over the set of all possible edges in the network was 0.50 the QoS of the solution topology appears to be quite good under the circumstances.

A QoS metric for the set of all potential arcs in the network is:

$$QoS\_of\_Network = \frac{\sum_{e \in E} Bandwidth_e * QoS_e}{\sum_{e \in E} Bandwidth_e} \tag{4.1}$$

The numerator in this metric is a sum-product of QoS and bandwidth for all potential arcs in the network. Dividing this quantity by the total bandwidth available on the network provides a weighted average of the network QoS. This metric makes it possible to gauge how well the NetDesign heuristic selected arcs from the network by comparing it with the average QoS of the selected topology. If the topology was selected at random then over time the expected average QoS would be the average network QoS. A metric for the weighted average QoS achieved by the solution topology is:

$$QoS\_for\_topology = \frac{\sum_{e \in topology} Bandwidth_e * QoS_e}{\sum_{e \in topology} Bandwidth_e} \tag{4.2}$$

Of course NetDesign does not select arcs solely based on QoS. The message requirements also affect which arcs are selected. It is desirable to transmit as much message bandwidth through high quality paths as possible. A metric for the average of the QoS achieved by the solution routing is given by:

$$QoS\_for\_messages = \frac{\sum_{k \in message\_routing} Bandwidth_k * QoS_k}{\sum_{k \in message\_routing} Bandwidth_k} \quad (4.3)$$

Again this metric can be compared to the average network QoS to gauge how well NetDesign performed in routing messages on high quality arcs.

An additional network metric that can be used to evaluate the quality of a particular solution is that of *efficiency*. The metric below evaluates how much of the bandwidth in the solution topology is utilized by the solution routing. The numerator is the sum of all bandwidth for messages that were transmitted and the denominator is the sum of all bandwidth in the topology. The denominator is an upper bound for the numerator so, the ratio cannot exceed a value of 1.00. The metric to evaluate the relative efficiency of the solution is:

$$Topology\_efficiency = \frac{\sum_{k \in message\_routing} Bandwidth_k}{\sum_{e \in topology} Bandwidth_e} \quad (4.4)$$

Finally, since it is assumed the bandwidth of the messages to be transmitted exceeds the capacity of any feasible solution topology, it is necessary to provide an upper bound for the fraction of message bandwidth that may be transmitted through the network. The following ratio represents the percentage of total message bandwidth that was successfully routed in the solution:

$$Routing\_bandwidth\_upper\_bound = \frac{\sum_{k \in message\_routing} Bandwidth_k}{\sum_{v \in all\_messages} Bandwidth_v} \quad (4.5)$$

The numerator is the total bandwidth that was successfully routed in the solution. The denominator is the total bandwidth of all messages in the Prioritized_Request list. This number is clearly a function of the total bandwidth requested and not just the performance of NetDesign. For comparison an upper bound on the total message traffic that could possibly be transmitted is:

$$Network\_upper\_bound = \frac{\sum_{e \in E} Bandwidth_e}{\sum_{v \in all\_messages} Bandwidth_v} \quad (4.6)$$

The numerator is the total bandwidth available on all arcs in the network and the denominator is the total bandwidth of all messages to be transmitted. This is obviously not a very tight upper bound many messages will need to traverse more than one arc from source to sink using much more additional network bandwidth than just the message size. However, it does provide a basis for comparison that can not be altered by reducing or increasing the size of the Priority_Request list.

As previously stated, the complete set of 32 test cases was solved using both Dijkstra's algorithm and the A* heuristic to determine which method performed better within the context of the NetDesign metaheuristic. Tables 5 compares the run time and objective function value obtained by NetDesign using both the A* heuristic and Dijkstra's algorithm.

Table 5: Test case results for A* and Dijkstra's in NetDesign

Case	NetDesign with A*		NetDesign with Dijkstra's	
	z	time (sec)	z	time (sec)
ndp1	1853.1	1022.7	1525.3	898.14
ndp2	2005.7	609.36	2050.7	550.68
ndp3	4379.2	752.44	4945.1	941.95
ndp4	5212.2	860.39	5350.2	817.48
ndp5	2176.5	832.56	5350.2	817.48
ndp6	1751.3	521.88	1901.3	463.09
ndp7	4846.7	849.24	4897.1	718.58
ndp8	6904.2	709.28	5145	539.13
ndp9	1903.7	1293.5	1604.9	1368.2
ndp10	1756.6	1198.5	1995.1	1047.7
ndp11	5427.3	871.4	4419.4	886.65
ndp12	6030	907.03	4409.4	885.93
ndp13	2246.8	1620.6	2134	1496.4
ndp14	1821	1956.3	1805.3	1663
ndp15	6365.5	1389.6	4806.4	1465.1
ndp16	6273.9	1366.1	4852.1	1385.2
ndp17	1946.2	2195.5	2134	1919.1
ndp18	2037.7	2101.9	1811.7	1736.8
ndp19	5800	1468.2	4824.6	1606
ndp20	6012.6	1426.26	5512	1505.7
ndp21	2726.9	1087.7	3248	1600.3
ndp22	2228	1391	2358.3	1535.7
ndp23	6418.3	2373.5	5011.7	2103.8
ndp24	7096.8	980.21	5068.5	953.41
ndp25	2810.2	2233.1	1955.5	1948.2
ndp26	2087.1	2139.9	2037.5	1824.4
ndp27	5718.2	1548.8	4918.5	1719.7
ndp28	6058.9	1571.2	5036	1728.7
ndp29	2264.8	4523.5	3059.5	3702.9
ndp30	2006.6	4386.7	3021.6	3204.2
ndp31	7280.7	2424.8	5491.2	3042.5
ndp32	8400.8	1493.7	5309.4	1390.3

The following table presents summary statistics for the test data presented above. Notice that the mean run time for Dijkstra's was lower than that of A* while the mean objective function value for A* was higher than that of Dijkstra's algorithm. In order to

test the statistical significance of the difference in the means for both objective function value and run time, a t-test is used. The results of the t-test are presented along with the table of summary statistics. The critical values for the tests done below assume an $\alpha = 0.1$ level of confidence.

Table 6: Summary statistics for objective function value and run time

	NetDesign with A*		**NetDesign with Dijkstra's**	
	z	**time (sec)**	**z**	**time (sec)**
Mean	4120.23	1565.84	3687.17	1483.33
Variance	4722017.85	864068.67	2258632.82	567602.07
Standard Deviation	2173.02	929.55	1502.88	753.39

Table 7: t-test for difference in mean objective function value

	A*	**Dijkstra's**
Mean	4120.23	3687.17
Variance	4722017.85	2258632.82
Observations	32	32
Hypothesized Mean Difference	0	
df	55	
t Stat	0.92720895	
P(T<=t) one-tail	0.17893459	
t Critical one-tail	1.2971343	
P(T<=t) two-tail	0.357869181	
t Critical two-tail	1.673033966	

Since the t-statistic is in the table above is less than the critical value for the one-tail test, we fail to reject the null hypothesis that the difference in the mean objective function value generated using A* and Dijkstra's algorithm is equal to zero..

Table 8: t-test for difference in mean run time

	A*	**Dijkstra's**
Mean	1565.84	1483.33
Variance	864068.67	567602.07
Observations	32	32
Hypothesized Mean Difference	0	
df	59	
t Stat	0.390101926	
P(T<=t) one-tail	0.348933	
t Critical one-tail	1.296065725	
P(T<=t) two-tail	0.697866	
t Critical two-tail	1.671093033	

Since the t-statistic in the table above is less than the critical value for the one tail test, we fail to reject the null hypothesis that the mean run time for NetDesign between A* and Dijkstra's algorithm is equal to zero.

The goal of the NetDesign metaheuristic is to optimize the objective function value by selecting high quality solutions from the available solution space. The goal of the experimentation done for this research is not to optimize a response variable such as run time, but rather to gain an understanding of which network characteristics are significant in affecting the run time. Since this research uses a 2^5 full factorial design, it is necessary to test for linearity in the response since this is one of the basic assumptions of the 2^k design. Linearity can be tested using a center point with all factors set to the "zero" level. In the context of this research this amounts to generating an additional test case with the following network characteristics: number of vertices 35, average number of transceivers 5, average message bandwidth 20, average arc bandwidth 60, and dispersion 3 Although this is not a true center point due to the "transceiver" and "dispersion" factor center levels being non integral, the high factor level is arbitrarily chosen for this test run and it will be assumed that the point to be tested is in fact close

enough to constitute a center point. The transceiver level must be integral since fractional transceivers do not make sense and cannot exist in practice. This assumption is reasonable since the first order model is still accurate even if slight non-linearity is present (Montgomery, 2005). The results of the center point runs are given in the Table 6

Table 9: NetDesign center point test case

Case	NetDesign with A*		NetDesign with Dijkstra's	
	z	time (sec)	z	time (sec)
ndp_center run#1	4062.6	1965.7	3638.2	2354.7
ndp_center run#2	4062.6	1964.1	3638.2	2354.2
ndp_center run#3	4062.6	1971.9	3638.2	2349.6

To verify the linearity assumption the average response from the 32 test cases is compared with the response of the center point. If the difference is relatively small then it may be concluded that the response is linear over the chosen factor range and therefore the first order model is adequate in the analysis of the test results in this research. This test is completed for both the A^* heuristic and Dijkstra's algorithm.

Testing the linearity assumption is done by comparing the average of the responses for the three center point runs, with the average response for all 32 of the factorial runs. The results of this test are presented in the table below.

Table 10: Comparison of Factorial Averages with Center Point Averages

	NetDesign with A*	NetDesign with Dijkstra's
Center run response avg. (sec.)	1967.2	2352.8
Factorial run response avg. (sec.)	4120.2	1483.3
Difference in Center vs. Factorial	2153	869.5

From the table presented above it is apparent that the response is not linear over the range of the factor levels since the difference between the center run averages and the factorial experiment averages is large in both cases. This result is in keeping with the experimental results obtained in similar experiments conducted by Erwin. Although, a

second-order model would be needed in order to perform accurate regression analysis, it is assumed that the simple factor screening results are still accurate, since the first order model is capable of handling some non-linearity (Montgomery, 2005).

A* Factor Screening

It is advantageous to perform a factor screening to determine the influential main effects and interaction effects. Determining which factors are important provides a basis for qualifying NetDesign and the network characteristics which drive run time. The following analysis is done twice; once for each shortest-path approach. The first set of tables and analysis pertains to NetDesign using the A* path-finding approach.

Before determining which factors are significant in affecting the run tine, it is necessary to check normality assumptions for the residuals of the response. This is done using a Box-Cox plot with a 95% confidence interval shown below. The CI is given by the red lines in the plot. The plot indicates that a logarithm transformation on the response is required in order to satisfy normality assumptions. By performing the transformation, it is possible to achieve a greater level of accuracy in deciding which factors are significant using the normal probability plot.

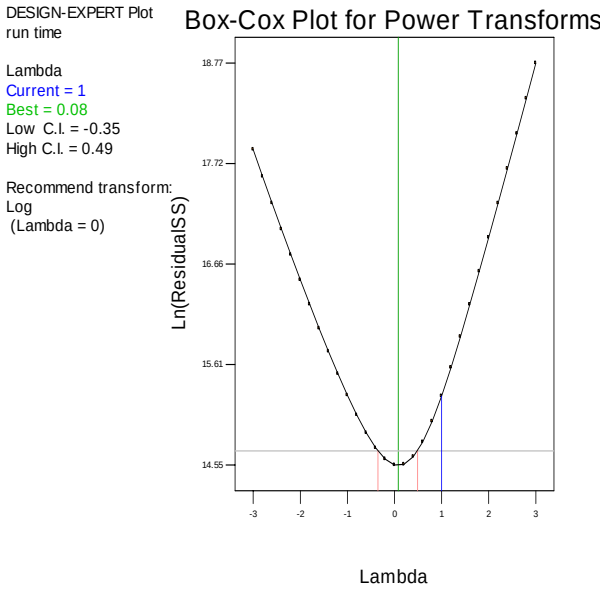


Figure 3: Box-Cox plot to examine Residual Normality in A*

After the transformation is complete the next task is to determine which factors are significant by examining the normal probability plot below. Figure 4 is a normal probability plot for the five factors used in the factorial design. As the reader can see from the chart, factors A, B, and D are significant since they differ from the normal probability plot line significantly. Additionally, the following two factor interactions are significant and are explained in detail below: BC and BD.

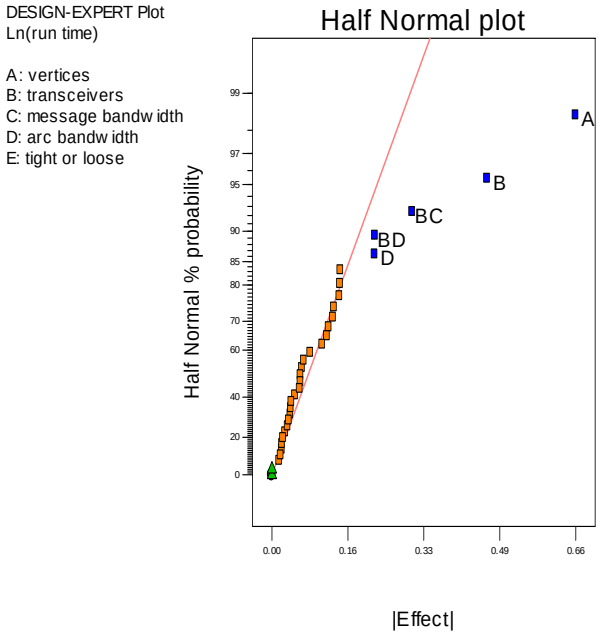


Figure 4: Normal probability factor plot for NetDesign using A*

From the normal probability plot it is clear that all main effects except message bandwidth and dispersion are significant. Dispersion is the variable that controls the range over which the uniform random integers are drawn. Additionally, the two factor interactions of transceivers & message bandwidth, transceivers & arc bandwidth, message bandwidth & dispersion are significant. Although the main effect of message bandwidth is not significant by itself it is included in two separate two factor interactions. Finally, the three factor interaction involving transceivers, message bandwidth & arc bandwidth is also significant. The plots for the main effects and the two factor interaction are given in Figures 5 and 6.

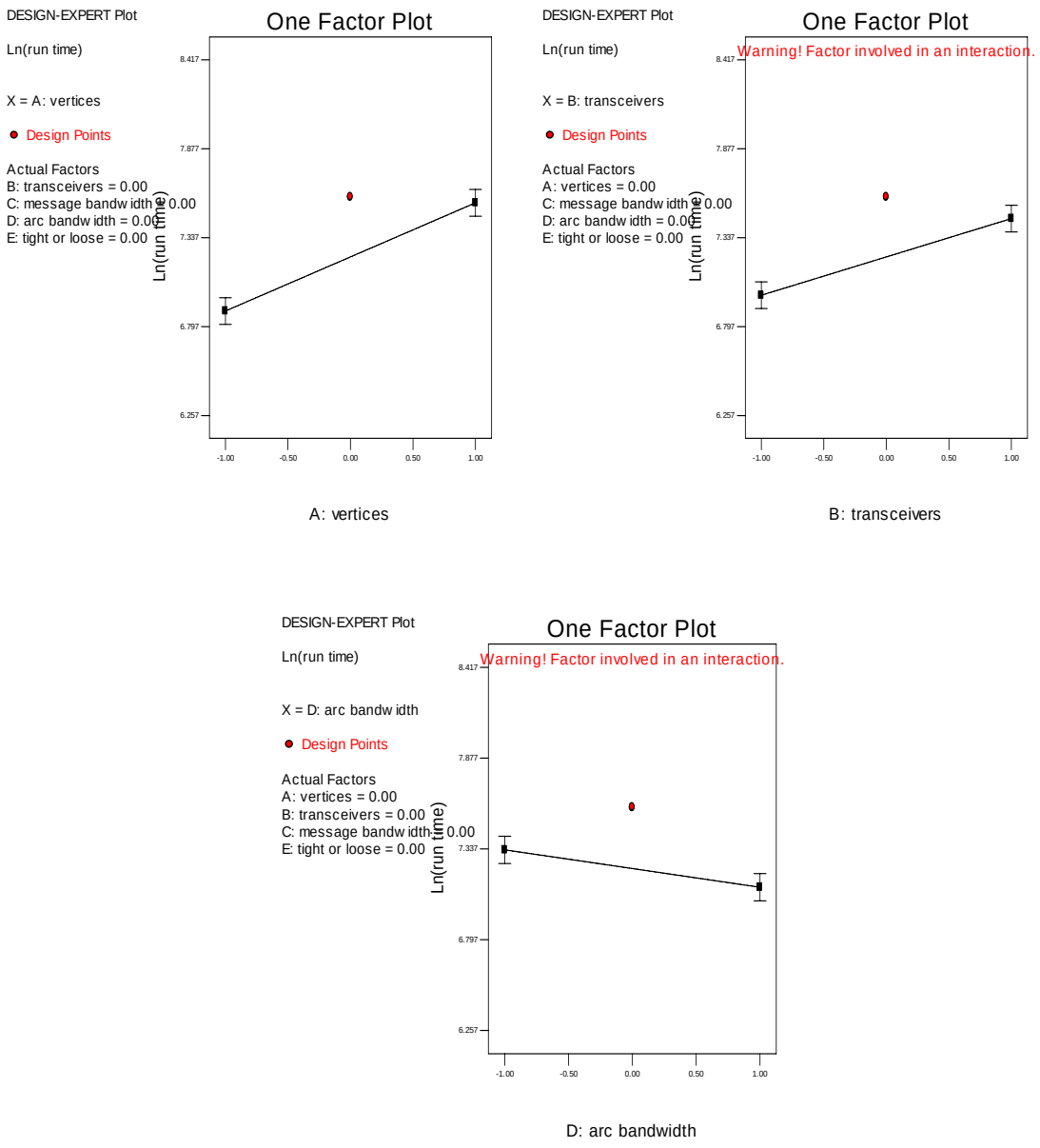


Figure 5: Main Effects vs. Run Time A* plots

The single factor plots presented above are consistent with intuition. Specifically, it is expected that as vertices, transceivers and message bandwidth are increased, the run time should increase since the size of the solution increases. Also, as the arc bandwidth increases, the run time tends to decrease. This makes intuitive sense because arc bandwidth is a measure of capacity. The implication is that as the average arc bandwidth

increases, the average bandwidth of the paths generated for communications requests also increases. Since the bandwidth of the paths is larger, each communication request is split into fewer communication packets, thus the number function calls to the A* heuristic decreases. The test data empirically corroborate this assertion. These data can be found in Appendix B.

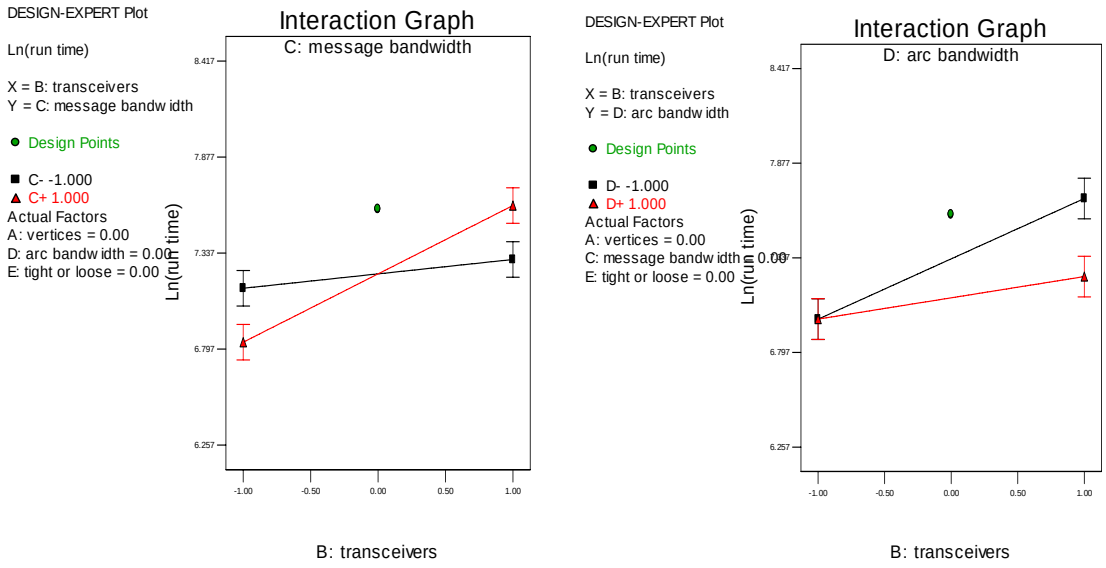


Figure 6: Two factor interaction plots for A*

The two factor interaction plots require slightly more explanation. The interaction plot for transceivers and message bandwidth indicates that the variation in run time is greater as the number of transceivers is varied from low to high when message bandwidth is set at the high level. Similarly, the interaction plot for arc bandwidth and transceivers indicates the variation in the response is greater as the number of transceivers is varied from low to high when arc bandwidth is set to the low level.

The preceding information may be useful in certain instances when time is limited and it is desirable to reduce run time. For instance, the interaction graphs for transceivers & message bandwidth and transceivers and arc bandwidth indicate that as the number of

transceivers are varied from low to high that it is possible to reduce the run time, regardless of the level of transceivers, by setting message bandwidth to the low level and arc bandwidth to the high level. While this idea may be theoretically appealing, it may not be feasible to control such factors in an operational setting.

Dijkstra's Factor Screening

An identical analysis to that presented above for A* is completed in this section for NetDesign using Dijkstra's algorithm. One assumption in regression and DOE is that the residuals for the response variable are normally distributed. This assumption may be tested using the Box-Cox plot below generated in Design Expert. The 95% confidence interval on λ, the variable indicating weather a transformation is necessary or not, is shown in red. Since the current λ value is outside of this range, the indication is that a logarithm transformation of the residual terms is required in order for these terms to satisfy the normality assumption.

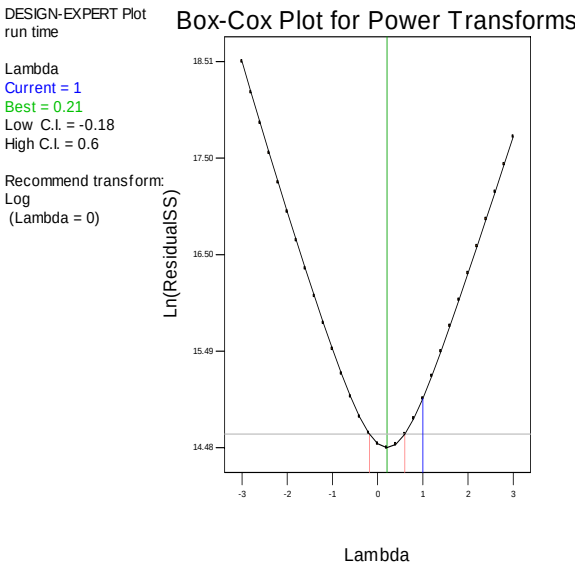


Figure 7: Box-Cox plot to examine Residual Normality in Dijkstra's

After applying this transformation, the significant factors can be determined by examining the normal probability plot given below. The plot indicates that the main effects of vertices, transceivers, and dispersion are significant. Additionally the two factor interaction between transceivers and message bandwidth is also significant.

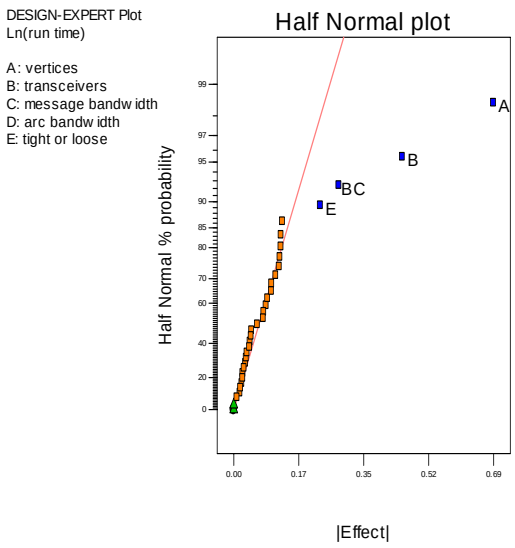
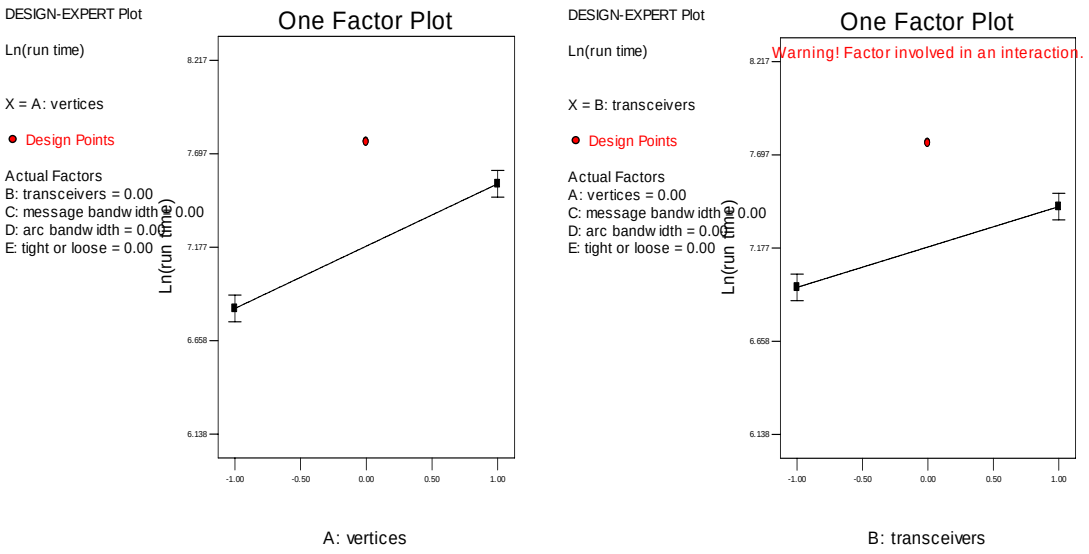


Figure 8: Normal probability factor plot for NetDesign using Dijkstra's



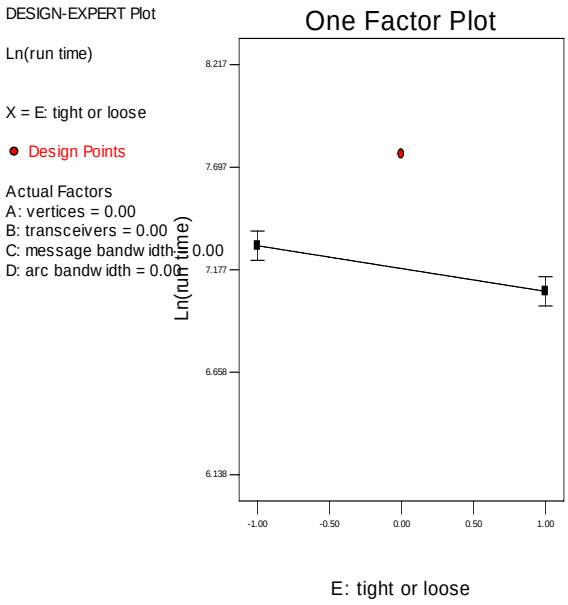


Figure 9: Main Effects vs. Run Time Dijkstra's plots

The first two single factor plots above follow intuition since it is expected that as the number of vertices and transceivers is increased, the run time will also increase. The third plot indicates that as dispersion increases the run time decreases. The two factor interaction plot for transceivers and message bandwidth is given below.

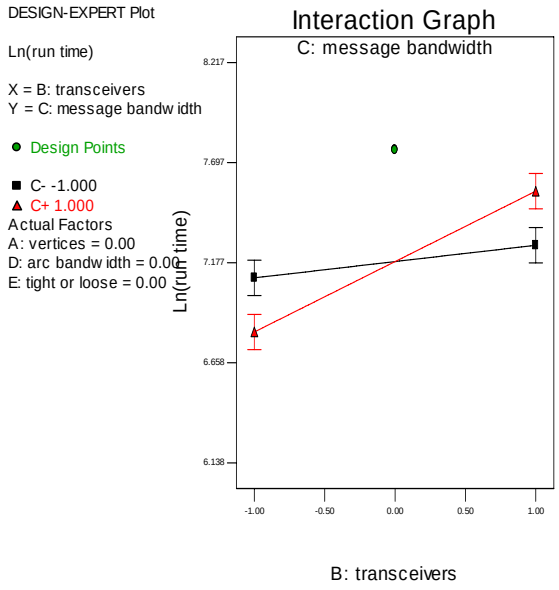


Figure 10: Two factor interaction plot for Dijkstra's

The interaction plot above indicates that the variation in run time is greater as transceivers is varied from low to high when message bandwidth is set at the high level than when message bandwidth is set to the low level.

Metrics for NetDesign Performance

The tabulation of metrics for all test runs is presented in Appendix B. In this section a brief summary is presented to determine how well NetDesign was able to perform relative to the upper bounds and other metrics presented earlier in this chapter. The tabulation incorporates both the A* heuristic and Dijkstra's algorithm in order to provide a side-by-side comparison as they have been implemented in NetDesign, by comparing average performance based on the number of vertices and transceivers in the network. The choice to compare performance based on these two factors is reasonable since they are the two largest main effects and as such the choice of level for each of

these two factors will provide the largest variation in the response of run time. The table below provides the averages for both the A* heuristic and Dijkstra's algorithm.

Table 11: Network metrics breakout by Vertices and Transceivers.

Network Metrics	Factor Level	A*	Dijkstra's
QoS_for_chosen_network	Average_v30_t4.mat	.89624	.90038
	Average_v30_t5.mat	.88325	.9118
	Average_v40_t4.mat	.80601	.84251
	Average_v40_t5.mat	.82074	.85574
QoS_for_messages_routed	Average_v30_t4.mat	.96264	.98243
	Average_v30_t5.mat	.96964	.98851
	Average_v40_t4.mat	.96848	.98512
	Average_v40_t5.mat	.97197	.99021
topology_efficiency	Average_v30_t4.mat	.19248	.18856
	Average_v30_t5.mat	.15388	.14048
	Average_v40_t4.mat	.13585	.13761
	Average_v40_t5.mat	.11805	.12078
time	Average_v30_t4.mat	769.73	719.36
	Average_v30_t5.mat	1325.4	1274.8
	Average_v40_t4.mat	1628.1	1620.1
	Average_v40_t5.mat	2540.2	2320.1
objective_value	Average_v30_t4.mat	3641.1	3498.8
	Average_v30_t5.mat	3978.2	3253.3
	Average_v40_t4.mat	4283.3	3646.1
	Average_v40_t5.mat	4578.4	3853.7

From the table above it seems that on average, using A* in NetDesign produces higher quality solutions but generally requires a longer running time in order to find the solutions, while using Dijkstra's algorithm in NetDesign yields solutions more quickly but a tradeoff is made in terms of the solution quality. Previous statistical tests indicate that one cannot reject the null hypothesis that there is no statistical difference in the mean run time or objective function value for NetDesign using A* or Dijkstra's algorithm. In order to test the statistical difference between the means for the remaining three network metrics in the table presented above it is necessary to perform similar statistical tests. Using a t-test for difference of means, the following results were obtained. Note that the full results are included in Appendix B:

Table 12: t-tests for difference in means in Network Metrics

Network Metrics	Factor Level	p-value of t-test
QoS_for_chosen_network	Average_v30_t4.mat	.42
	Average_v30_t5.mat	.22
	Average_v40_t4.mat	.23
	Average_v40_t5.mat	.28
QoS_for_messages_routed	Average_v30_t4.mat	$1.9*10^{-6}$
	Average_v30_t5.mat	0.0005
	Average_v40_t4.mat	0.0002
	Average_v40_t5.mat	0.0005
topology_efficiency	Average_v30_t4.mat	0.40
	Average_v30_t5.mat	0.18
	Average_v40_t4.mat	0.46
	Average_v40_t5.mat	0.45

By inspection of the p-values in table 12, one would fail to reject that the null hypothesis that there is no difference in the mean values for all cases in QoS_for_chosen_network and topology_efficiency. However at the $\alpha = 0.1$ level of confidence one would reject the null hypothesis that there is no difference in the means and conclude that using Dijkstra's algorithm in NetDesign produces message routings with a higher QoS than those generated by A*.

V. Conclusions & Recommendations

Measurement of Success

The results presented in this chapter highlight several improvements over those presented by Erwin. The problem instances used for testing NetDesign are larger than any instances for which Erwin was able to generate complete solutions using the MILP approach. For instance, Erwin's MILP took a minimum running time of 700 seconds to generate a complete solution for a 15 vertex instance of the problem even when imposing a 12% optimality gap. Depending on the approach used, some 15 vertex instances required running time in excess of 850 seconds. The MILP approach was not able to produce feasible integer solutions to any instance of the problem greater than 15 nodes in the 30 minute time window imposed by Erwin. By contrast, NetDesign is able to provide complete solutions to many 30 node and even 40 node instances in under 30 minutes. All attempts to generate solutions for problem instances with 39 or greater vertices using the MILP approach failed to produce a feasible solution in 8 hours of running time,. In the testing done for this research, NetDesign consistently found solutions to 40 vertex instances of the NDP in less than an hour, although the average running time for all of the 40 vertex instances tested was around 1 hour.

Summary of Test Implications

Through the use of a 2^5 full factorial experiment it was possible to determine that there were several differences in the NetDesign when Dijkstra's algorithm is used as opposed to A*. If less is known about message bandwidth and dispersion of the factors then it is advantageous to use A* since these two factors do not affect run time of NetDesign when A* is used. Similarly, if it is know that the average message size is large relative to average arc capacity then Dijkstra's algorithm should be used since this

factor configuration less significant in driving the run time of NetDesign when Dijkstra's is used.

Recommendations for Future Work

There are several opportunities for future research efforts. In heuristic approximation, one important aspect is the choice of pricing function used in evaluating candidates in a search. In this research, the pricing function determined the breadth and depth of the search tree that was built in trying the find shortest paths. Further research needs to be done to establish optimum policies for the balance of breadth vs. depth in selecting these pricing functions. Even more basic than this issue is the approximation itself. The heuristic portion of the pricing function was established to be the average QoS for all arcs emanating from the node under evaluation. There may be better approximations of the "distance" remaining to the goal node than the one used here.

Another area that could benefit from further research is the arc weighting scheme. One issue that needs to be addressed is that of arc weighting. It may be possible to improve the speed of solution convergence by determining an improved arc weighting scheme.

Finally, an important part of any heuristic search is efficient implementation. In many ways, a heuristic is ultimately judged by how quickly solutions are produced. Therefore, a heuristic with sound theoretical components that are not implemented well is less useful. The software produced in this research was implemented in MATLAB due to the author's limited experience with coding. Surely, a more efficient implementation could be produced using MATLAB or could further benefit from an implementation in C++ or another much faster compiled programming language.

Appendix A: MATLAB implementation of NetDesign

This section contains all functions, implemented in MATLAB, that are needed to run the NetDesign metaheuristic. In addition, the files used for generating the test cases and results are included for completeness and as an additional reference to the reader. Below is a complete listing of the functions and control files that are included in this appendix. A description of each function/file is contained in the header for each.

1. a_star.m

```
function [failure, path_info] =
a_star_test(Prioritized_Requests,Vert_adj,Trans_adj,Trans_Char,QoS,Bandwidth,QoS_Path)

Num_Tranceivers = size(Trans_Char,2);
Num_Vertices = size(Vert_adj,1);
failure = 0;
success = 0;
count = 0;

% To indicate "perfect" QoS for transceivers located at the same vertex,
% set corresponding elements = 2
for i = 1:Num_Tranceivers
    for j = i:Num_Tranceivers
        if Trans_Char(i).Vertex_ID == Trans_Char(j).Vertex_ID && i ~= j
            QoS(i,j) = 2;
            QoS(j,i) = 2;
        end
    end
end

Node = struct([]);
path = struct([]);
best_path = struct([]);
for i = 1:size(Vert_adj,1)
    % This is for returning the best path info for n_goal
    path_info.node_path = [];
    path_info.trans_path = [];
    path_info.path_bandwidth = [];
    path_info.trans_path_QoS = 0;

    % stores and updates best path info
    best_path(i).node_path = [];
    best_path(i).trans_path = [];
    best_path(i).path_bandwidth = [];
    best_path(i).path_QoS = -inf;

    % Stores information on the nodes
    Node(i).successors = [];
    Node(i).ancestors = [];
    Node(i).point_to = [];
end

% Set the start and end node
n_zero = Prioritized_Requests(1,3);
n_goal = Prioritized_Requests(1,4);

% Begin Step 1
OPEN = [n_zero];
% End Step 1
```

```
% Begin Step 2
CLOSED = [];
% End Step 2

while failure == 0 && success == 0

    % Begin Step 3
    if isempty(OPEN) && n ~= n_goal % Then no node path exists
        failure = 1;
    end
    % End Step 3

    if failure == 0

        % Begin Step 4
        n = OPEN(1);
        CLOSED = [n; CLOSED];
        OPEN(1) = [];
        OPEN_CLOSED = union(OPEN,CLOSED);
        % End Step 4

        % Begin Step 5
        if n == n_goal
            success = 1;
            failure = 0;
            % Return the best path to n_goal best path
            path_info.node_path = best_path(n_goal).node_path;
            path_info.trans_path = best_path(n_goal).trans_path;
            path_info.path_bandwidth = best_path(n_goal).path_bandwidth;
            path_info.trans_path_QoS = best_path(n_goal).path_QoS;
        end
        % End Step 5

        if success == 0 && failure == 0
            % Begin Step 6

            % Create initial successor list
            for i = 1:size(Vert_adj,1)
                if Vert_adj(n,i) == 1
                    Node(n).successors = [Node(n).successors,i];
                end
            end

            % Now remove ancestors from the successor list
            Node(n).successors = setdiff(Node(n).successors,Node(n).ancestors);

            % Here find the best arc connecting n to each of the successors and
            % find the QoS of the path by finding the minimum QoS of any arc on
            % the path using trans_path_finder.
            for i = 1:size(path,2)
                path(i).node_path = [];
                path(i).trans_path = [];
                path(i).bandwidth = [];
                path(i).QoS = [];
            end

            % Build "path" for each successor of n
            if count == 1
                for i = 1:length(Node(n).successors)
                    path(Node(n).successors(i)).node_path = [best_path(n).node_path,
Node(n).successors(i)];
                end
            elseif count == 0
                for i = 1:length(Node(n).successors)
                    path(Node(n).successors(i)).node_path = [n, Node(n).successors(i)];
                end
                %path(Node(n).successors(i)).node_path =
horzcat(path(Node(n).successors(i)).node_path,best_path(Node(n).successors(i)).node_path)
;
            end

            % Determine trans path and QoS of trans path if it exists.
```

```
            for i = 1:length(Node(n).successors)
                if failure == 0
                    [n_zero_to_n,failure] =
trans_path_finder(path(Node(n).successors(i)).node_path,Trans_Char,Bandwidth,QoS,QoS_Path
);
                    path(Node(n).successors(i)).trans_path = n_zero_to_n.Trans_path;
                    path(Node(n).successors(i)).bandwidth = n_zero_to_n.Path_Bandwidth;
                    path(Node(n).successors(i)).QoS = n_zero_to_n.Trans_Path_QoS;
                end
            end

            % Compare "path" QoS to "best_path" QoS.  if path QoS is better
            % than best paht QoS then replace best path with path
            for i = 1:length(Node(n).successors)
                if path(Node(n).successors(i)).QoS >
best_path(Node(n).successors(i)).path_QoS
                    best_path(Node(n).successors(i)).node_path =
path(Node(n).successors(i)).node_path;
                    best_path(Node(n).successors(i)).trans_path =
path(Node(n).successors(i)).trans_path;
                    best_path(Node(n).successors(i)).path_bandwidth =
path(Node(n).successors(i)).bandwidth;
                    best_path(Node(n).successors(i)).path_QoS =
path(Node(n).successors(i)).QoS;
                end
            end
            % End Step 6

            % Begin Step 7

            % Detemine successors not already on OPEN or CLOSED
            establish_pointers_1 = setdiff(Node(n).successors, OPEN_CLOSED);

            % Determine successors already on OPEN or CLOSED
            establish_pointers_2 = intersect(Node(n).successors, OPEN_CLOSED);

            % Establish a pointer to n from each successor not already on
            % OPEN_CLOSED and add these elements to OPEN
            for i = 1:length(establish_pointers_1)
                Node(establish_pointers_1(i)).point_to = n;
                OPEN = [OPEN; establish_pointers_1(i)];
                OPEN_CLOSED = union(OPEN,CLOSED);
            end

            % Redirect pointers for elements already on OPEN_CLOSED to n if
            % the best path so far is through n
            for i = 1:length(establish_pointers_2)
                if ~isempty(intersect(n, best_path(establish_pointers_2(i)).node_path))
                    Node(establish_pointers_2(i)).point_to = n;
                end
            end

            %Update ancestors for each node based on the pointers
            for i = 1:length(OPEN_CLOSED)
                Node(OPEN_CLOSED(i)).ancestors = [];
            end

            for i = 1:length(OPEN_CLOSED)
                current = OPEN_CLOSED(i);
                at_node = current;
                while at_node ~= n_zero
                    Node(current).ancestors = [Node(current).ancestors,
Node(at_node).point_to];
                    at_node = Node(at_node).point_to;
                end
            end
            % End Step 7

            %Begin Step 8
            % Calculate h (some elements of OPEN may be removed here)
            % OPEN_h =
calculate_h(QoS,Num_Tranceivers,Num_Vertices,Trans_adj,Trans_Char,OPEN);
```

```
            % Use this one to omit heuristic pricing and run Dijkstra's
             OPEN_h = horzcat(OPEN,zeros(length(OPEN),1));

            % Calculate g (some elements of OPEN may be removed here)
            % Must run calculate_h prior to calculate_g
            % Return the node path info "n_zero_to_n" for use in pointers below
            [OPEN_h_g,OPEN_path_info] =
calculate_g(OPEN_h,Trans_Char,Bandwidth,QoS,Node,n_zero,Num_Vertices,QoS_Path);

            % Here continue only if the size of open_h_g is greater than zero...
            % if not and n is not equal to n_goal then set failure equal to 1
            if ~isempty(OPEN_h_g)

                % Now that elements of OPEN may have been removed recalculate
                % OPEN_CLOSED.  the "union" operator should work even though it
                % sorts the elements in OPEN_CLOSED since the elements is
                % n_zero_to_n are indexed by node ID
                OPEN_CLOSED = union(OPEN_h_g(:,1),CLOSED);

                % Pricing updates for members of OPEN
                % 1.  Calculate f values for each member on OPEN_g_h and call this OPEN_f
                OPEN_f = [];
                for i = 1:size(OPEN_h_g,1)
                    OPEN_f(i,1) = OPEN_h_g(i,1);
                    OPEN_f(i,2) = OPEN_h_g(i,2) + OPEN_h_g(i,3);
                end
                % 2.  Sort elements in OPEN_f in descending order by f value
                OPEN_f = sortrows(OPEN_f, [-2]);
                % 3.  Set OPEN = OPEN_f(:,1) in preperation for Step 3
                OPEN = OPEN_f(:,1);
                % End Step 8

                count = 1;

            end % if ~isempty(OPEN_h_g)
        end % if success == 0
    end % if failure == 0
end % while failure == 0
```

2. assign_message_route.m

```
function [Comm_routing,Bandwidth,Prioritized_Requests,QoS] =
assign_message_route(Prioritized_Requests,current_path_info,Bandwidth,QoS)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%{

Function:

    [Comm_routing,Bandwidth,Prioritized_Requests,QoS] = assign_message_
    route(Prioritized_Requests,current_path_info,Bandwidth,QoS)

Author:

    R. Benjamin Hartlage, Capt, USAF

Description:

    assign_message_route updates the information in Bandwidth, QoS,
    and Prioritized Requests which effectively routes the current
    communications request.

    Return "Comm_routing" along with updated versions
    of the Bandwidth, Prioritized_Requests, and QoS matrices.  "Comm_routing"
    is a structure containing information about the current comm request routing.

Inputs:
```

```
    Prioritized_Requests:  A matrix containing information on the comm
                           requests.  Each row represents a request.  The
                           format for a row is:
                           [priority, bandwidth, origin, destination, request_ID]

    current_path_info:  The structure containing the path information obtained
                        from the a_star.  The fields contained in the structure
                        are:

                            current_path_info.node_path
                            current_path_info.trans_path
                            current_path_info.path_bandwidth
                            current_path_info.trans_path_QoS

    QoS:  As defined in a_star

    Bandwidth:  As defined in a_star

Outputs:

    Comm_routing:  Contains all information about the routing of the message.
                   Since the path may or may not accomodate the entire request
                   it may be necessary to route it in "packets."  There are two
                   possible cases as described below.

    Bandwidth:  This matrix is updated based on the bandwidth used on edges
                contained in the current path.

    QoS:  Updated by setting QoS for edges whose bandwidth is used up to "0"

    Prioritized_Requests:  Updated to reflect the message routing.  If part
                           of the message is routed then the message bandwidth
                           is reduced by the appropriate amount.  If all of
                           the message is routed then the corresponding row
                           (always row 1 since the messages are sorted in
                           non-decreasing order by priority and bandwidth)
                           is deleted.

%}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%{
First case:  The bandwidth of the message is greater than or equal
             to the bandwidth available on the path.

    1: subtract path bandwidth from message bandwidth in
       "Prioritized_Requests"

    2: set current_message.bandwidth = current_path_info.path_bandwidth
       to reflect actual amount being transmitted over that path.

    3: subtract path bandwidth from every arc on the path.  Do this by
       updating the coresponding element of "Bandwidth"

    4: Check each arc on the transceiver path.  If its bandwidth is zero then change
       the corresponding element in QoS to a "0" so that it is not
       selected during future iterations of A*

    5: If Prioritized_Requests(1,2) == 0 then it has been completely routed
       so remove it from the request list

%}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

current_message.bandwidth = Prioritized_Requests(1,2);
current_message.ID = Prioritized_Requests(1,5);

current_message.Priority = Prioritized_Requests(1,1);

if current_message.bandwidth >= current_path_info.path_bandwidth
```

```
    %1
    Prioritized_Requests(1,2) = Prioritized_Requests(1,2) -
current_path_info.path_bandwidth;
    %2
    current_message.bandwidth = current_path_info.path_bandwidth;
    %3
    for i = 1:length(current_path_info.trans_path)-1
        Bandwidth(current_path_info.trans_path(i), current_path_info.trans_path(i+1)) =
Bandwidth(current_path_info.trans_path(i), current_path_info.trans_path(i+1)) -
current_path_info.path_bandwidth;
        Bandwidth(current_path_info.trans_path(i+1), current_path_info.trans_path(i)) =
Bandwidth(current_path_info.trans_path(i+1), current_path_info.trans_path(i)) -
current_path_info.path_bandwidth;
        %4
        if Bandwidth(current_path_info.trans_path(i), current_path_info.trans_path(i+1))
== 0
            QoS(current_path_info.trans_path(i), current_path_info.trans_path(i+1)) = 0;
            QoS(current_path_info.trans_path(i+1), current_path_info.trans_path(i)) = 0;
        end
    end
    %5
    if Prioritized_Requests(1,2) == 0
        Prioritized_Requests(1,:) = [];
    end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%{

Second case: The bandwidth of the message is less than the bandwidth
             available on the path.

    1: subtract message bandwidth from every arc on the path.  Do this by
       updating the coresponding element of "Bandwidth."

    2: Remove the first row of the "Prioritized_Requests" matrix since all
       of the message has been routed.

%}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

elseif current_message.bandwidth < current_path_info.path_bandwidth
    %1
    for i = 1:length(current_path_info.trans_path)-1
        Bandwidth(current_path_info.trans_path(i), current_path_info.trans_path(i+1)) =
Bandwidth(current_path_info.trans_path(i), current_path_info.trans_path(i+1)) -
current_message.bandwidth;
        Bandwidth(current_path_info.trans_path(i+1), current_path_info.trans_path(i)) =
Bandwidth(current_path_info.trans_path(i+1), current_path_info.trans_path(i)) -
current_message.bandwidth;
    end
    %2
    Prioritized_Requests(1,:) = [];
end

Comm_routing.message_ID = current_message.ID;
Comm_routing.message_bandwidth = current_message.bandwidth;
Comm_routing.node_path = current_path_info.node_path;
Comm_routing.trans_path = current_path_info.trans_path;
Comm_routing.path_QoS = current_path_info.trans_path_QoS;
Comm_routing.message_Priority = current_message.Priority;
% didn't include path bandwidth since it is unimportant after the message
% has been routed.
```

3. Bandwidth_QoS_matching_update.m

```
function [Bandwidth,QoS] = Bandwidth_QoS_matching_update(Bandwidth,QoS,I,Trans_Char)

% This function will update the Bandwidth and QoS matrices after the
% matching is formed each time.  After a matching is formed, the result is
% that several arcs are removed.  To represent this set the corresponding
% elements of Bandwidth and QoS to "0" for the next A* search.

a = size(Bandwidth,1);
b = a;

for i = 1:a
    for j = i:b
        if Trans_Char(i).Vertex_ID ~= Trans_Char(j).Vertex_ID
            Bandwidth(i,j) = Bandwidth(i,j)*I(i,j);
            Bandwidth(j,i) = Bandwidth(j,i)*I(i,j); % the I(i,j) here is correct since
some of the adjacency
                                                    % matrices are upper
                                                    % triangular
            QoS(i,j) = QoS(i,j)*I(i,j);
            QoS(j,i) = QoS(j,i)*I(i,j);
        end
    end
end
```

4. calculate_g.m

```
function [OPEN_h_g,n_zero_to_n] =
calculate_g(OPEN_h,Trans_Char,Bandwidth,QoS,Node,n_zero,Num_Vertices,QoS_Path)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%{

Function:

    [OPEN_h_g,n_zero_to_n] = calculate_g(OPEN_h,Trans_Char,Bandwidth,QoS,
    Node,n_zero,Num_Vertices)

Author:

    R. Benjamin Hartlage, Capt, USAF

Description:

    calculate_g calculates the "shortest path" (max QoS) from n_zero to n
    found by A* so far for each element of OPEN. In order to find the
    shortest path, from n_zero to n the shortest transceiver path from
    n_zero to n must first be determined.  If the transceicer path for
    n_zero to n doesn't exist then remove it from OPEN since it will not be
    on the shortest path from n_zero to n_goal.

Inputs:

    OPEN_h:  As defined in calculate_h

    Node:  The structure containing pointers for the elements in the search
           graph.  These pointers are used to determine a path once the
           goal node has been reached.  Node has only one field:

              Node.point_to

    Num_Vertices:  A scalar value denoting the number of vertices in the
                   network.

    n_zero:  The node that the search began from.  In this case it is the
             origin node for the communication request.
```

```
    Bandwidth:  As defined in a_star.

    Trans_Char:  As defined in a_star.

    QoS:  As defined in a_star.

Outputs:

    OPEN_h_g:  The OPEN_h matrix with an added column containing the price
               obtained from the g pricing function.  the g value is the
               length of the shortest path (QoS) for the shortest path
               from n_zero to n (where n is an element of OPEN) found by
               a_star so far.

    n_zero_to_n:  The structure containg path info for the elements of
                  OPEN.  This path info is used to determine the g value
                  of vertices.  The fields contained in n_zero_to_n are:

                    n_zero_to_n.Node_path
                    n_zero_to_n.Trans_path
                    n_zero_to_n.Trans_Path_QoS
                    n_zero_to_n.no_trans_path_indicator

%}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Need this structure to have "Num_Vertices" elements for operations that
% take place in "redirect_pointers"
for i = 1:Num_Vertices
    n_zero_to_n(i).Node_path = [];
end

% Step 1:  Determine the node path from n_zero to n for each element on OPEN
%(guaranteed to exist since each n was arrived at by pointers)
for i = 1:size(OPEN_h,1)
    current = OPEN_h(i,1);
    temp_path = [current];
    while current ~= n_zero
        current = Node(current).point_to;
        temp_path = [current; temp_path];
    end
    %index using the actual node...for instance, n_zero_to_n(5).Node_path
    %is the node path from n_zero to n corresponding to node 5
    n_zero_to_n(OPEN_h(i,1)).Node_path = temp_path;
end

% Step 2:  Determine the transceiver path for each element of n_zero_to_n
% or determine that one does not exist.  If one exists determine
% Trans_Path_QoS...use "trans_path_structure" since "n_zero_to_n" is an
% input...then reassign later.
for i = 1:size(OPEN_h,1)
    [path_info,no_trans_path_indicator] =
trans_path_finder(n_zero_to_n(OPEN_h(i,1)).Node_path,Trans_Char,Bandwidth,QoS,QoS_Path);
    trans_path_structure(OPEN_h(i,1)).Trans_path = path_info.Trans_path;
    trans_path_structure(OPEN_h(i,1)).Trans_Path_QoS = path_info.Trans_Path_QoS;
    trans_path_structure(OPEN_h(i,1)).no_trans_path_indicator = no_trans_path_indicator;
end

% Here's the reassignment from "trans_path_structure" to "n_zero_to_n"
for i = 1:size(OPEN_h,1)
    n_zero_to_n(OPEN_h(i,1)).Trans_path = trans_path_structure(OPEN_h(i,1)).Trans_path;
    n_zero_to_n(OPEN_h(i,1)).Trans_Path_QoS =
trans_path_structure(OPEN_h(i,1)).Trans_Path_QoS;
    n_zero_to_n(OPEN_h(i,1)).no_trans_path_indicator =
trans_path_structure(OPEN_h(i,1)).no_trans_path_indicator;
end

g = [];
for i = 1:size(OPEN_h,1)
    % If no path was found then indicate with a price of 100
    if n_zero_to_n(OPEN_h(i,1)).no_trans_path_indicator == 1
        g(i) = 100;
```

```
    elseif n_zero_to_n(OPEN_h(i,1)).no_trans_path_indicator == 0
        g(i) = trans_path_structure(OPEN_h(i,1)).Trans_Path_QoS;
    end
end
g = g';

OPEN_h_g = horzcat(OPEN_h,g);

% Now remove elements of OPEN_h_g with column 3 entries of 1
i = 1;
v = size(OPEN_h_g,1);
while i <= v;
    if OPEN_h_g(i,3) == 100
        % if the node is removed, also remove the node path info
        n_zero_to_n(OPEN_h_g(i,1)).Node_path = [];
        n_zero_to_n(OPEN_h_g(i,1)).Trans_path = [];
        n_zero_to_n(OPEN_h_g(i,1)).Trans_Path_QoS = 0;
        OPEN_h_g(i,:) = [];
        v = size(OPEN_h_g,1);
    else
        i = i+1;
    end
end
```

5. calculate_h.m

```
function [OPEN_h] = calculate_h(QoS,Num_Tranceivers,Num_Vertices,I,Trans_Char,OPEN)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%{

Function:

    [OPEN_h] = calculate_h(QoS,Num_Tranceivers,Num_Vertices,I,Trans_Char,
    OPEN)

Author:

    R. Benjamin Hartlage, Capt, USAF

Description:

    calculate_h is the heuristic portion of the pricing function that
    provides an estimate of the QoS on the path from n to n_goal.  The
    estimate is calculated for a node by averaging the QoS on all arcs
    eminating from the node.

    Notes:  if the estimated QoS on the path is "0" then the average QoS on
            all arcs eminating from node n is "0" and and no path exists to
            connect n to n_goal.  In this case, remove n from OPEN since no
            path connecting n_zero to n_goal will contain n. OPEN_h is a
            matrix associating the heuristic estimates for the elements on
            OPEN (which have not been removed in this function) with the
            elements of open.  Row i contains node OPEN(i) and the h value
            for OPEN(i).  Elements on OPEN whose h value is zero are
            removed from "OPEN_h."

Inputs:

    QoS: As defined in a_star.

    Num_Vertices: As defined in NetDesign.

    Trans_Char:  As defined in a_star.

    Num_Transceivers:  A scalar value denoting the total number of
                       transceivers in the network.

    I:  The transceiver adjacency matrix
```

```
    OPEN:  The vector of vertices that are currently being explored for by
           a_star.

Outputs:

    OPEN_h:  The matrix containing the nodes listed in OPEN along with
             their heuristic function price.  There is a row in OPEN_h for
             each element of OPEN.  The format of the row is:
             [open_element, h_value]

%}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Trans_QoS_Sum = [];
Denominator = [];
one_calc = ones(Num_Tranceivers,1);
n_to_goal_estimate = [zeros(Num_Vertices,3)];
I_calc = I + I';

for i = 1:Num_Tranceivers
    % Calculate the total QoS eminating from each transceiver i
    Trans_QoS_Sum(i) = QoS(i,:)*I_calc(i,:)';
    % Determine how many QoS measures were summed for each transceiver
    Denominator(i) = I_calc(i,:)*one_calc;
end

% Now determine total QoS and number of measures added up for each node
for i = 1:Num_Tranceivers
    n_to_goal_estimate(Trans_Char(i).Vertex_ID,1) =
n_to_goal_estimate(Trans_Char(i).Vertex_ID,1) + Trans_QoS_Sum(i); % Tracks total QoS
    n_to_goal_estimate(Trans_Char(i).Vertex_ID,2) =
n_to_goal_estimate(Trans_Char(i).Vertex_ID,2) + Denominator(i);
    n_to_goal_estimate(Trans_Char(i).Vertex_ID,3) = Trans_Char(i).Vertex_ID;
end

OPEN_h = [];
for i = 1:length(OPEN)
    OPEN_h(i,1) = OPEN(i);
    if n_to_goal_estimate(OPEN(i),1) > 0
        OPEN_h(i,2) = n_to_goal_estimate(OPEN(i),1)/n_to_goal_estimate(OPEN(i),2);
    elseif n_to_goal_estimate(OPEN(i),1) == 0
        OPEN_h(i,2) = 0;
    end
end

% Remove nodes on OPEN that have estimated QoS of "0"
i = 1;
v = size(OPEN_h,1);
while i <= v;
    if OPEN_h(i,2) == 0
        OPEN_h(i,:) = [];
        v = size(OPEN_h,1);
    else
        i = i+1;
    end
end
```

6. calculate_network_metrics.m

```
function [metrics] =
calculate_network_metrics(best_solution,Bandwidth,QoS,Match,Prioritized_Requests)

% best_solution:  is the structure containing informatoin on the routing
% Bandwidth:  is the matrix of bandwidth values for each arc in the network
% QoS:  is the matrix of QoS values for each arc in the network
% Match:  is the transceiver adjacency matrix after the matching has been made.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 1. QoS metric for the entire network

% First change inf entries to 0 and make entries below main diagonal == 0
% so arcs are not added twice
for i = 1:size(Bandwidth,1)
    for j = i:size(Bandwidth,1)
        if Bandwidth(i,j) == inf
            Bandwidth(i,j) = 0;
        end
        Bandwidth(j,i) = 0;
    end
end

% Numerator
BQ_1 = Bandwidth.*QoS;
BQ_1 = sum(BQ_1,1);
BQ_1 = sum(BQ_1,2);

% Denominator
B_1 = sum(Bandwidth,1);
B_1 = sum(B_1,2);

metrics.QoS_for_all_arcs_network = BQ_1/B_1;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 2. QoS metric for my network

% Numerator
BQ_2 = Bandwidth.*Match; % Bandwidth on chosen arcs
BQ_2 = BQ_2.*QoS; % Band_QoS on chosen arcs
BQ_2 = sum(BQ_2,1);
BQ_2 = sum(BQ_2,2);

% Denominator
B_2 = Bandwidth.*Match;
B_2 = sum(B_2,1);
B_2 = sum(B_2,2);

metrics.QoS_for_my_network = BQ_2/B_2;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 3. QoS metric for messages sent

% Determine the highest message ID which will determine the size of the
% message_collect structure
high_ID = 0;
for i = 1:size(best_solution,2)
    if best_solution(i).message_ID > high_ID
        high_ID = best_solution(i).message_ID;
    end
end


% initialize bandwidth to zero, Band_QoS to 0 and QoS to infinity
for i = 1:high_ID;
    message_collect(i).QoS = 2;
    message_collect(i).Bandwidth = 0;
    message_collect(i).Band_QoS = 0;
end

% Determine the QoS and Bandwidth of each message by examining the routing
for i = 1:size(best_solution,2);
    if best_solution(i).path_QoS < message_collect(best_solution(i).message_ID).QoS
        message_collect(best_solution(i).message_ID).QoS = best_solution(i).path_QoS;
    end
    message_collect(best_solution(i).message_ID).Bandwidth =
message_collect(best_solution(i).message_ID).Bandwidth +
best_solution(i).message_bandwidth;
end

% if QoS of any message is > 1 at this stage then set it to 0
```

```
for i = 1:high_ID
    if message_collect(i).QoS > 1
        message_collect(i).QoS = 0;
    end
end

% calculate the Band_QoS for each message and the total bandwidth of all
% messages transmitted.
total_band = 0;
for i = 1:size(message_collect,2)
    message_collect(i).Band_QoS = message_collect(i).QoS * message_collect(i).Bandwidth;
    total_band = total_band + message_collect(i).Bandwidth;
end

% sum the Band_QoS for all messages sent
total_Band_QoS = 0;
for i = 1:size(message_collect,2)
    total_Band_QoS = total_Band_QoS + message_collect(i).Band_QoS;
end

% Calculate the ratio of total_Band_QoS to total_band
metrics.QoS_for_messages_routed = total_Band_QoS/total_band;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 4. Efficiency of a given topology
metrics.topology_efficiency = total_band/B_2;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 5. Determine the upper bound on bandwidth routed and compare to the ratio
% for actual bandwidth routed to see how well the heuristic performed.

all_messages_bandwidth = 0;
for i = 1:size(Prioritized_Requests,1)
    all_messages_bandwidth = all_messages_bandwidth + Prioritized_Requests(i,2);
end

metrics.total_band_transmitted_ratio = total_band/all_messages_bandwidth;


% Determine the total bandwidth of all arcs in the matching.
B_3 = Bandwidth.*Match;
B_3 = sum(B_3,1);
B_3 = sum(B_3,2);


metrics.total_band_upper_bound = B_3/all_messages_bandwidth;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 6. Determine the number of messages in the routing;
total_requests = size(Prioritized_Requests,1);
message_count_vector = zeros(total_requests,1);
message_count_vector1 = ones(total_requests,1);
for i = 1:size(best_solution,2)
    message_count_vector(best_solution(i).message_ID) = 1;
end
metric.messages_in_routing = message_count_vector' * message_count_vector1;
```

7. calculate_objective_function.m

```
function [objective_value] = calculate_obj_fcn(best_solution,P)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%{

Function:
```

```
    [objective_value] = calculate_obj_fcn(message_routing,P)

Author:

    R. Benjamin Hartlage, Capt, USAF

Description:

    calculate_obj_fcn determines the quality of the current solution in
    terms of the following objective function: sum across all messages in
    the routing (QoS)*(Bandwidth)*(Priority)
    The QoS and Bandwidth are taken directly from the message_routing
    structure, but priority is calculated as follows:
    Priority = 2^(message_Priority - 1)  In this case we are assuming that
    a priority 5 message is twice as important as a priority 4, a priority
    4 message is two times as important as a priority 3, etc.

Inputs:

    message_routing: A structure with fields identical to those contained
                     in Pre_match_comm_routing.  See assign_message_route
                     for a description of the fields.

    P: A structure containing information on each of the requests

Outputs:

    objective value:  A scalar value denoting the solution quality.
                      the higher the objective_value the better a
                      particular solution is.

%}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Determine the highest message ID which will determine the size of the
% message_collect structure
high_ID = 0;
for i = 1:size(best_solution,2)
    if best_solution(i).message_ID > high_ID
        high_ID = best_solution(i).message_ID;
    end
end

% initialize bandwidth to zero, Band_QoS to 0 and QoS to 2
for i = 1:high_ID;
    message_collect(i).QoS = 2;
    message_collect(i).Bandwidth = 0;
end

% Determine the QoS and Bandwidth of each message by examining the routing
for i = 1:size(best_solution,2);
    if best_solution(i).path_QoS < message_collect(best_solution(i).message_ID).QoS
        message_collect(best_solution(i).message_ID).QoS = best_solution(i).path_QoS;
    end
    message_collect(best_solution(i).message_ID).Bandwidth =
message_collect(best_solution(i).message_ID).Bandwidth +
best_solution(i).message_bandwidth;
end

% if QoS of any message is > 1 at this stage then set it to 0
for i = 1:high_ID
    if message_collect(i).QoS > 1
        message_collect(i).QoS = 0;
    end
end

objective_value = 0;
for i = 1:size(message_collect,2)
    objective_value = objective_value +
(message_collect(i).QoS*message_collect(i).Bandwidth*P(i).Priority);
end
```

8. comm_not_transmittable.m

```
function [P,Num_Comm_Requests,Vect_Comm_Requests] =
comm_not_transmittable(P,Isolated_vertices_I,Unreachable_I)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%{

Function:

    comm_not_transmittable(P,Isolated_vertices_I,Unreachable_I)

Author:

    R. Benjamin Hartlage, Capt, USAF

Description:

    comm_not_transmittable removes comm requests which have
    origin/destination pairs that reside in seperate components.  This is
    to be done prior to prioritization or routing so that computation time
    is not wasted for mesages that are easily determined to be unroutable.

Inputs:

    P:  As defined in create_requests.

    Isolated_vertices_I:  As defined in Isolated_vert

    Unreachable_I:  Is the matrix whose "0" entries denote transceiver pairs for
                    which no paths of less than or equal to "CONNECTIVITY."
                    connect the transceivers.  In other words, if entry (i,j) is
                    "0" then j is unreachable from i in "CONNECTIVITY" hops.

Outputs:

    P:  As defined in create_requests...but modified by removing the comm
        requests that cannot be transmitted.

    Num_Comm_Requests:  Simply a scalar constant that denotes the total
                        number of communications requests remaining after
                        non-transmittable requests are removed.

    Vect_Comm_Requests: Can be used in vectorizing code...just a vector
                        of numbers from 1 to Num_Comm_Requests.



Notes:

    Messages_not_transmittable:  Is the vector whose entries represent the
                                 communication ID #'s for comm. requests
                                 that cannot be sent because they are
                                 coming from or going to an isolated
                                 vertex.

%}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


% Messages_not_transmittable:  Is the vector whose entries represent the
%                              communication ID #'s for comm. requests that
%                              cannot be sent because they are coming from or
%                              going to an isolated vertex.
%

Messages_not_transmittable_I = [];
```

```
for i = 1:size(Isolated_vertices_I, 1)
    for j = 1:size(P, 2)
        if P(j).Origin == Isolated_vertices_I(i) || P(j).Destination ==
Isolated_vertices_I(i)
            Messages_not_transmittable_I = [Messages_not_transmittable_I; P(j).ID,
P(j).Priority, P(j).Bandwidth];
        end
    end
end

% Get rid of comm requests at origins and destinations not connected
while i <= size(P,2)
    if Unreachable_I(P(i).Origin, P(i).Destination) == 0
        P(i) = [];
        i = 1;
    else
        i = i + 1;
    end
end
Num_Comm_Requests = size(P,2);
Vect_Comm_Requests = [1:Num_Comm_Requests];
```

9. create_I

```
function [I,Num_Transceivers,Vect_Transceivers] = create_I(Trans_Char,D,M)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%{

Function:

    [I,Num_Transceivers,Vect_Transceivers] = create_I(Trans_Char,D,M)

Author:

    R. Benjamin Hartlage, Capt, USAF

Description:

    create_I is the function that generates the initial transceiver
    incidence matrix.  The matrix is created based on two filtering
    criteria: transceiver type and communications radius.  If two
    transceivers are not within each others comm radius then they
    are not adjacent.  Also, if two transceivers are not of the same type
    then they are not adjacent.

Inputs:

    Trans_Char:  As defined in create_Trans_Char.

    D:  As defined in NetDesign

    M:  As defined in create_Trans_Char

Outputs:

    I:  Is the initial transceiver incidence matrix with main diagonal
        elements removed since vertex i is not adjacent to itself.  I is
        filtered based on the 2 criteria in Operation 2.

    Num_Transceivers:  The number of Transceivers contained in the initial
                       transceiver list.

    Vect_Transceivers:  A vector with all transceiver ID #'s.

%}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

I = ones(size(Trans_Char, 2));
```

```
for i = 1:size(I,1)
    I(i,i) = 0;
    % Entries on the main diagonal are set to zero since
    % transceivers are not adjacent to themselves
    for j = i:size(I,1)
        I(j,i) = 0;
        % This sets the entries below the main diagonal to zero
        % since all arcs are undirected
        if I(i,j) > 0
            if (D((Trans_Char(i).Vertex_ID),(Trans_Char(j).Vertex_ID)) >
Trans_Char(i).CommRadius) || (D((Trans_Char(i).Vertex_ID),(Trans_Char(j).Vertex_ID)) >
Trans_Char(j).CommRadius)
                I(i,j) = 0;
                % filters based on comm radius if either transceiver is
                % outside of the comm radius of the other
            end
            if (D((Trans_Char(i).Vertex_ID),(Trans_Char(j).Vertex_ID)) == 0)
                I(i,j) = 0;
                % If the transceivers are at the same vertex they are not
                % adjacent
            end
            if (Trans_Char(i).Transceiver_type ~= Trans_Char(j).Transceiver_type)
                I(i,j) = 0;
            end
            if (M((Trans_Char(i).Vertex_ID),1) == 0 || M((Trans_Char(j).Vertex_ID),1) ==
0) && (M((Trans_Char(i).Vertex_ID),2) == 0 || M((Trans_Char(j).Vertex_ID),2) == 0)
                I(i,j) = 0;
                % Filters based on transceiver types at each vertex
                % If neither i or j have either transciever type than they
                % are not adjacent
            end
        end % I(i,j) > 0
    end % for j = i:size(I,1)
end % for i = 1:size(I,1)
Num_Transceivers = size(I,1);
Vect_Transceivers = [1:Num_Transceivers];
```

10. create_matching.m

```
function [Matching_final,Matching, Vert_adj_post_match, Match] =
create_matching(Arc_weight,I,Trans_Char,Num_Vertices)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%{

Operation 8:  Determine the Matching between transceivers.  A simple greedy
              heuristic is used to obtain a good initial starting solution.
              The greedy heuristic chooses the pairs of transceivers
              in non-decreasing order of elements in Match.  Also
              determine the Vertex adjacency post match.


Function:

    [Matching_final,Matching, Vert_adj_post_match, Match] = create_matching
    (Arc_weight,I,Trans_Char,Num_Vertices)

Author:

    R. Benjamin Hartlage, Capt, USAF

Description:

    Determine the Matching between transceivers.  A simple greedy
    heuristic is used to obtain a good initial starting solution.
    The greedy heuristic chooses the pairs of transceivers
    in non-decreasing order of elements in Match.  Also
    determine the Vertex adjacency post match.
```

```
Input:

    Arc_weight:  As defined in reweight_arcs

    I:  As defined in create_I

    Trans_Char:  As defined in create_Trans_Char

    Num_Vertices:  As defined in NetDesign

Output:

    Matching:  Is the matrix containing information on the transceiver
               matching created by the greedy heuristic.

    Match:  Is a copy of "I" that can be modified to reflect the actual
            matching that has been chosen.  "I" reflects only the potential
            connections.  "Match" reflects the transceiver adjacencies
            after the match and can be used in the A* in place of  the "I"
            matrix used initially.

    Vert_adj_post_match:  Is the matriz that gives information on the
                          adjacency of the vertices after the greedy
                          heuristic has been run. Also an input for A*.

    Matching_final:  Is the matrix that integrates transceiver and vertex
                     information for use in graphing and for presentation
                     to the user.

%}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Arc_weight1 = Arc_weight;
Match = I;
Matching = [];
a = 1;

while a > 0
    [C,Y] = max(max(Arc_weight1)); % Returns the column index of the max as Y
    [V,F] = max(Arc_weight1(:,Y)); % Returns the row index of the max as F
    % V and C are identical values; both are value of the the max Arc_weight1
    Matching = [Matching; F, Y, C];
    Match(:,F) = 0;
    Match(Y,:) = 0;
    Match(F,:) = 0;
    Match(:,Y) = 0;
    Arc_weight1 = Arc_weight1.*Match;
    a = max(max(Arc_weight1));
end

Matching_final = [];
for i = 1:size(Matching,1)
    Matching_final = [Matching_final; Trans_Char(Matching(i,1)).Vertex_ID,
Trans_Char(Matching(i,2)).Vertex_ID, Matching(i,1), Matching(i,2), Matching(i,3)];
end

% Rebuild "Match" which contains information about the transceiver
% adjacencies after the matching has taken place
for i = 1:size(Matching_final,1)
    Match(Matching(i,1),Matching(i,2)) = 1;
    Match(Matching(i,2),Matching(i,1)) = 1;
end

%Put vert adj post match loop here
Vert_adj_post_match = zeros(Num_Vertices);

for i = 1:size(Matching_final,1)
    Vert_adj_post_match(Matching_final(i,1),Matching_final(i,2)) = 1;
    Vert_adj_post_match(Matching_final(i,2),Matching_final(i,1)) = 1;
end
```

11. create_requests.m

```
function [P] = create_requests(User_inputs,Num_Vertices)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%{

Function:

    [P] = create_requests(User_inputs,Num_Vertices)

Author:

    R. Benjamin Hartlage, Capt, USAF

Description:

    Generate the list of communications requests. Each request
    has associated with it a bandwidth, priority, time window,
    origin, destination, and a unique ID.  It must arrive at
    the destination in during the specified time window in
    order to be considered successful.  Additionally, if the
    request is split into packets then all packets must arrive
    (within the specified time window) in order to be
    considered complete.

Inputs:

    User_inputs:  As defined in NetDesign

    Num_Vertices:  As defined in NetDesign

Outputs:

    P:  is the MATLAB structure containining communication requests along
        with the information needed to prioritize and route each one.
        The fields contained in P are:

            P.ID
            P.Bandwidth
            P.Priority
            P.Origin
            P.Destination

%}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

num_requests = random_integers(User_inputs.MAX_COMM_REQ);
request_number = 1;
for i = 1:num_requests
    P(request_number).ID = i;
    P(request_number).Bandwidth = random_integers(User_inputs.MAX_BAND);
    P(request_number).Priority = random_integers(User_inputs.PRIORITY_SCALE);
    P(request_number).Origin = random_integers(Num_Vertices);
    P(request_number).Destination = random_integers(Num_Vertices);

    %The following loop ensures that the origin and destination for each
    %communication request are distinct
    vvv = 1;
    while P(request_number).Origin == P(request_number).Destination
        P(request_number).Destination = random_integers(Num_Vertices);
    end
    request_number = request_number + 1;
end
```

12. create_Trans_Char.m

```
function [Trans_Char, M] = create_Trans_Char(User_inputs,Num_Vertices,A)
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%{

Function:

    [Trans_Char, M] = create_Trans_Char(User_inputs,Num_Vertices,A)

Author:

    R. Benjamin Hartlage, Capt, USAF

Description:

    create_Trans_Char is a function that generates a structure containing
    the transceiver characteristics for each transceiver in the network.

Inputs:

    User_inputs:  A structure with fields defined in NetDesign

    Num_Vertices:  A scalar value denoting the totla number of vertices
                   in the network.

    A:  As defined in NetDesign.

Outputs:

    M:  Is the matrix containing information on the total number of
        transceivers located at each vertex.  Each row represents a
        different vertex.


        total_transceivers:  Is the total number of transceivers possible
                             at vertex i.

        total_type1_transceivers:  Is the total number of transceivers of
                                   type 1 at vertex i.

        total_type2_transceivers:  Is the total number of transceivers of
                                   type 2 at vertex i.

    Trans_Char:  Is the MATLAB structure that contains the important
                 characteristics about each transceiver.  These
                 characteristics are stored in the fields listed below.

        Vertex_ID:  In "Trans_Char" denotes the vertex at which the current
                    transceiver is located.

        Transceiver_ID:  In "Trans_Char" denotes the unique transceiver
                         identifier

        Transceiver_type:  In "Trans_Char" denotes the type of transceiver.
                           Every transceiver is either of type 1 or type 2

        CommRadius:  In "Trans_Char" denotes The radius of communication of
                     the transceiver.

        XLoc:  In "Trans_Char" denotes the abscissa of the vertex
               containing the transceiver.

        YLoc:  In "Trans_Char" denotes the ordinate of the vertex
               containing the transceiver.

        XPlotLocation:  In "Trans_Char" denotes the abscissa plot location
                        of the transceiver.  Transceivers are plotted
                        around an imaginary circle for purposes of
                        graphing.

        YPlotLocation:  In "Trans_Char" denotes the ordinate plot location
                        of the transceiver.
```

```
%}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

M = [];

for i = 1:Num_Vertices
    total_transceivers = random_integers(User_inputs.MAX_TRANSCEIVERS);
    total_type1_transceivers = random_integers(total_transceivers);
    total_type2_transceivers = total_transceivers - total_type1_transceivers;
    M = [M; total_type1_transceivers, total_type2_transceivers];
end

transceiver_number = 1;

for i = 1:Num_Vertices% Moves through the vertices
    ComRad = random_integers(User_inputs.MAX_COMM_RAD,2);
    ttt = M(i,1) + M(i,2);
    tn = 1;
    for k = 1:2 % Move through tranceiver types
        for j = 1:M(i,k)% Move through the individual transceivers
            Trans_Char(transceiver_number).Vertex_ID = i;
            Trans_Char(transceiver_number).Transceiver_ID = transceiver_number;
            Trans_Char(transceiver_number).Transceiver_type = k;
            Trans_Char(transceiver_number).CommRadius = ComRad(k);
            Trans_Char(transceiver_number).XLoc = A(i,1);
            Trans_Char(transceiver_number).YLoc = A(i,2);
            Trans_Char(transceiver_number).XPlotLocation = A(i,1) +
1.*cos(2*3.14159265/ttt * tn);
            Trans_Char(transceiver_number).YPlotLocation = A(i,2) +
1.*sin(2*3.14159265/ttt * tn);
            transceiver_number = transceiver_number + 1;
            tn = tn + 1;
        end
    end
end
```

13. distance_matrix.m

```
function [D] = distance_matrix(A)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%{

Function:

    [D] = distance_matrix(A)

Author:

    R. Benjamin Hartlage, Capt, USAF

Description:

    "distance_matrix" calculates the distance between all pairs and returns
    this information in a symetric distance matrix.

Inputs:

    A: The n-by-2 matrix of ordered pairs.

Outputs:

    D: the symetric matrix whose (i,j) entries represent the distance
       between vertex i and vertex j.

%}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

D = [];
```

```
for i = 1:size(A,1)
    for j = i:size(A,1)
        if i == j
            D(i,j) = 0;
        else
            x = [A(i,:)-A(j,:)];
            D(i,j) = norm(x,2);
        end
    end
end
D = D + D';
```

14. generate_test_cases

```
function [file] = generate_test_cases(Network)
%{
This file is used to generate test cases.

Run this file and then save the workspace as a *.mat file to save
specific cases.

Notes:

1.  The priority for all test cases is a number between 1 and 5 with 5
    being the highest priority.

2.  The communication radius of every transceiver is set to a constant
    value of 100.

3.  Available bandwidth on arcs is randomly generated between 1 and 100.

%}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

for i = 1:2
    vertices = Network.vertices(i);
    for j = 1:2
        transceivers = Network.transceivers(j);
        for k = 1:2
            number_of_messages = Network.number_of_messages(k);
            for l = 1:2
                message_bandwidth = Network.message_bandwidth(l);
                for m = 1:2
                    arc_bandwidth = Network.arc_bandwidth(m);

                    tic
                    % 3.  Load the appropriate distance matrix.  Either the
                    % 50 or 100 depending on the number of vertices being
                    % tested.  This is generated outside of here so that it
                    % is helad constant throughout testing.
                    dist = sprintf('distance_matrix_%d.mat',vertices);
                    load(dist);
                    Num_Vertices = size(D,1);
                    Vect_Vertices = [1:Num_Vertices];

                    % 4.  Generate Transceiver characteristics
                    [Trans_Char,M] =
create_Trans_Char_test_case(Network,Num_Vertices,vertex_positions);

                    % 5.  Generate a square matrix of ones called "I" which has a row and
a column for each transceiver.
                    [I,Num_Transceivers,Vect_Transceivers] =
create_I_test_case(Trans_Char,D,M);

                    % 6.  Generate the list of communications requests.
                    P = create_requests_test_case(Network,Num_Vertices);

                    total_requests = size(P,2);
```

```
                    % 7.  Determine if there are any disconnected nodes in the network.
                    [Isolated_vertices_I, Vert_adj_pre_match, Unreachable_I] =
Isolated_vert_test_case(I,Num_Vertices,Trans_Char);

                    % 8.  Determine the messages not transmittable and remove them.
                    [P,Num_Comm_Requests,Vect_Comm_Requests] =
comm_not_transmittable_test_case(P,Isolated_vertices_I,Unreachable_I);

                    % 9.  Build matrices containing info on QoS, Bandwidth, and Requests
                    [QoS,Bandwidth,Prioritized_Requests] =
QoS_Bandwidth_Request_test_case(Trans_Char,I,P);

                    % Generate the file name and save the variables to thefile
                    file =
sprintf('test_case_v%d_t%d_n%d_m%d_a%d',vertices,transceivers,number_of_messages,message_
bandwidth,arc_bandwidth);
                    save(file);
                    toc
                end
            end
        end
    end
end
```

15. hamming_distance.m

```
function [ham_dist] = hamming_distance(ham_1,ham_2)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%{

Function:

    [ham_dist] = hamming_distance(ham_1,ham_2)

Author:

    R. Benjamin Hartlage, Capt, USAF

Description:

    hamming_distance calculates the hamming distance between two binary
    column vectors.

Inputs:

    ham_1/ham_2:  Binary column vectors based on network topology.  Has an
                  entry for each possible arc in the network.  If an arc is
                  included in the topology, the corresponding element of the
                  hamming vector is set to "1."  If the arc is not included
                  in the topology, then the corresponding element of the
                  hamming vector is set to "0."

Outputs:

    ham_dist:  A scalar value denoting the hamming distance between ham_1
               and ham_2.  Hamming distanc eis the number of elements by
               which the two vectors differ.

%}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

ham_dist_ones = ones(1,length(ham_1));

x = abs(ham_1-ham_2);

ham_dist = ham_dist_ones*x;
```

16. hamming_vector.m

```
function [ham_vect] = hamming_vector(comm_routing_structure,Num_Transceivers)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%{

Function:

    [ham_vect] = hamming_vector(comm_routing_structure,Num_Transceivers)

Author:

    R. Benjamin Hartlage, Capt, USAF

Description:

    Use the vertex paths in the comm_routing_structure to determine which
    arcs are in the solution.  Arcs in the solution have a value of "1" in
    the hamming vector and arcs not in the solution have a value of "0" in
    the hamming vector.

Inputs:

    comm_routing_structure:  as defined in route_message_traffic

    Num_Transceivers:  As defined in create_I

Outputs:

    ham_vect:  The binary vector representing the current network topology.

%}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

ham_mat = zeros(Num_Transceivers,Num_Transceivers);
for i = 1:size(comm_routing_structure,2)
    for j = 1:length(comm_routing_structure(i).trans_path)-1

ham_mat(comm_routing_structure(i).trans_path(j),comm_routing_structure(i).trans_path(j+1)
) = 1;

ham_mat(comm_routing_structure(i).trans_path(j+1),comm_routing_structure(i).trans_path(j)
) = 1;
    end
end

ham_vect = [];
for i = 1:Num_Transceivers
    ham_vect = [ham_vect;ham_mat(:,i)];
end
```

17. Isolated_vert.m

```
function [Isolated_vertices_I, Vert_adj_pre_match, Unreachable_I] =
Isolated_vert(I,Num_Vertices,Trans_Char)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%{

Function:

    Isolated_vert(I,Num_Vertices,Trans_Char):

Author:

    R. Benjamin Hartlage, Capt, USAF
```

```
Description:

    Isolated_vert

Inputs:

    I:  The transceiver adjacency matrix

    Num_Vertices:  The scalar denoting the number vertices in the network.

    Trans_Char:  As defined in a_star.

Outputs:

    Isolated_vertices_I:  A vector of vertex ID's for vertices in the
                          network that are disconnected.  If a vertex is
                          disconnected then no message traffic can be
                          routed to/from it.  Any messages with an origin
                          or destination that is isolated may be removed
                          from the list of communications to be sent.

     Note:    Use the Adjacency matrix to determine if a path
              of length n-1 connects the two nodes.  Note that
              the (i,j) entries denote the number of paths of length
              "power" that connect node i and j. "Power" is the power to
              which the matrix is raised Generate a matrix to track
              connectivity.  If a path is found then put an entry of 1
              into the appropriate position of the matrix.

%}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Vert_adj_pre_match = zeros(Num_Vertices);
Unreachable_I = zeros(Num_Vertices);
Isolated_vertices_I = [];

for i = 1:size(I,1)
    for j = 1:size(I,1)
        if I(i,j) == 1
            Vert_adj_pre_match(Trans_Char(i).Vertex_ID, Trans_Char(j).Vertex_ID) = 1;
            Vert_adj_pre_match(Trans_Char(j).Vertex_ID, Trans_Char(i).Vertex_ID) = 1;
        end
    end
end

for i = 1:(Num_Vertices-1)
    Unreachable_I = Unreachable_I + (Vert_adj_pre_match^(i));
end

for i = 1:Num_Vertices
    indicator = 0;
    for j = 1:Num_Vertices
        if Unreachable_I(i,j) > 0
            indicator = 1;
        end
    end

    if indicator == 0
            Isolated_vertices_I = [Isolated_vertices_I; i];
    end
end
```

18. master_test_generator.m

```
% Master test case generator (control file)

Network.vertices = [30 40]; % values are deterministic...if value is "x"
                            % then there will be exactly "x"
                            % vertices in the network at every vertex.
```

```
Network.transceivers = [4 5]; % values act as the mean number of
                              % transceivers at vertices in the network.

Network.message_bandwidth = [10 30]; % values act as the "mean" message
                                     % bandwidth over all mesages.

Network.arc_bandwidth = [30 90]; % values act as the "mean" arc bandwidth
                                 % over all arc capacities in the network.

Network.tight_or_loose = [2 3]; % Specify the range around the mean...i.e.
                                % 2 translates to "generate numbers within
                                % a range of 2 units on either side of the
                                % specified mean."

Network.number_of_messages =  2000; % make sure there are more messages
                                    % (look at total bandwidth) can be
                                    % routed through the network.

% First generate the two distance matrices for the two network sizes
for i = 1:2
    [D,Num_Vertices,vertex_positions] = distance_matrix_generator(Network.vertices(i));
    file = sprintf('distance_matrix_%d',Network.vertices(i));
    save(file,'D','Num_Vertices','vertex_positions');
end
clear file

% Generate lists of communications requests.
for i = 1:2
    for j = 1:2
        for k = 1:2
            P =
create_requests_test_case(Network,Network.message_bandwidth(j),Network.vertices(i),Networ
k.tight_or_loose(k));
            total_requests = size(P,2);
            file =
sprintf('requests_v%d_m%d_tl%d',Network.vertices(i),Network.message_bandwidth(j),Network.
tight_or_loose(k));
            save(file,'P','total_requests');
        end
    end
end

clear D Num_Vertices P file total_requests

% Generate the test cases
for i = 1:2
    vertices = Network.vertices(i);
    for j = 1:2
        transceivers = Network.transceivers(j);
        for l = 1:2
            message_bandwidth = Network.message_bandwidth(l);
            for m = 1:2
                arc_bandwidth = Network.arc_bandwidth(m);
                for k = 1:2
                    tight_loose = Network.tight_or_loose(k);

                    tic
                    % 3.  Load the appropriate distance matrix.  Either the
                    % 50 or 100 depending on the number of vertices being
                    % tested.  This is generated outside of here so that it
                    % is helad constant throughout testing.
                    dist = sprintf('distance_matrix_%d.mat',vertices);
                    load(dist);
                    Num_Vertices = size(D,1);
                    Vect_Vertices = [1:Num_Vertices];

                    % 4.  Generate Transceiver characteristics
                    [Trans_Char,M] =
create_Trans_Char_test_case(transceivers,Num_Vertices,vertex_positions,tight_loose);
```

```
                    % 5.  Generate a square matrix of ones called "I" which has a row and
a column for each transceiver.
                    [I,Num_Transceivers,Vect_Transceivers] =
create_I_test_case(Trans_Char,D,M);

                    % 6.  Load the list of communications requests based on the
                    % number of vertices, message bandwidth, and tight_loose
                    req =
sprintf('requests_v%d_m%d_tl%d.mat',vertices,message_bandwidth,tight_loose);
                    load(req);

                    % 7.  Determine if there are any disconnected nodes in the network.
                    [Isolated_vertices_I, Vert_adj_pre_match, Unreachable_I] =
Isolated_vert_test_case(I,Num_Vertices,Trans_Char);

                    % 8.  Determine the messages not transmittable and remove them.
                    [P,Num_Comm_Requests,Vect_Comm_Requests] =
comm_not_transmittable_test_case(P,Isolated_vertices_I,Unreachable_I);

                    % 9.  Build matrices containing info on QoS, Bandwidth, and Requests
                    [QoS,Bandwidth,Prioritized_Requests] =
QoS_Bandwidth_Request_test_case(Trans_Char,I,P,arc_bandwidth,tight_loose);

                    % Generate the file name and save the variables to thefile
                    file =
sprintf('test_case_v%d_t%d_m%d_a%d_tl%d',vertices,transceivers,message_bandwidth,arc_band
width,tight_loose);

save(file,'Bandwidth','D','I','Isolated_vertices_I','M','Network','Num_Comm_Requests',...

'Num_Transceivers','Num_Vertices','P','Prioritized_Requests','QoS','Trans_Char','Unreacha
ble_I',...

'Vect_Comm_Requests','Vect_Transceivers','Vect_Vertices','Vert_adj_pre_match','total_requ
ests',...
                    'vertex_positions');
                    toc
                end
            end
        end
    end
end
```

19. NetDesign.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%{

Title:

    NetDesign

Author:

    R. Benjamin Hartlage, Capt, USAF

Description:

    NetDesign is the control file for the metaheuristic developed in this
    research using the A* graph search Algorithm.  NetDesign uses a greedy
    heuristic to develop a weighted matching between tranceivers at the
    vertices given in the input file specified.  The matching is used as a
    network topology.  Message traffic is routed through this topology by use
    of the A* shortest path heuristic.  The topology along with the routing,
    form an initial solution which is then improved upon through an iterative
    arc reweighting scheme to converge on a topology with arc weights that
    reflect the true importance of each arc in the topology.

Inputs:
```

```
    random_graphxxx.txt:  This is the text file containing ordered pairs
                          representing coordinates of the vertices.

    MAX_TRANSCEIVERS:  the user defined quantity denoting the maximum number
                       of transceivers to be located at any given vertex.

    MAX_COMM_RAD:  the upper bound on communication radius for any transceiver.

    MAX_COMM_REQ:  the maximum number of communications requests to be
                   generated for transmission across the network.

    PRIORITY_SCALE:  assigns the maximum priority (this number determines
                     the scale) I arbitrarily choose 1 as the lowest priority
                     and 5 as the highest priority.

    MAX_BAND:  the maximum bandwidth of a communication to be sent across the
               network.

    AVAILABLE_BANDWIDTH:  the maximum bandwidth capacity of an edge in the
                          network.  Capacity is a randomly generated number
                          between 1 and AVAILABLE_BANDWIDTH for each edge.

    COMM_BANDWIDTH:  the maximum size of a communication request to be sent
                     over the network. Communication request size is a randomly
                     generated number between 1 and AVAILABLE_BANDWIDTH for
                     each request.

Output:

    Post_match_comm_routing:  This structure contains all information
                              needed to route the communications requests
                              for which paths of sufficient bandwidth were
                              discovered and established (topology created)

    Graphs:  While not essential to the solution, graphs are generated to
             aid in visualizing the solution.  Two graphs are generated for
             compariason.  The first is a graph showing all possible arcs.
             The second is a graph showing the final topology created by
             the metaheuristic once convergence is determined.

%}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

QoS_Path = [Trans_Char(:).Vertex_ID]';

t = cputime; % Start the timer here

% First update the Bandwidth and QoS matrices to reflect adjacencies in the
% I matrix
[Bandwidth,QoS] = Bandwidth_QoS_matching_update(Bandwidth,QoS,I,Trans_Char);

% Operation 1:  Determine initial comm routing using A*.  This routing assumes all
possible connections exist and is used in determining the initial arc weights.
Pre_match_comm_routing =
route_message_traffic(Prioritized_Requests,Bandwidth,QoS,Trans_Char,Vert_adj_pre_match,I,
P,QoS_Path);

% Operation 2:  Begin calculations for iterative reweighting loop.
Post_match_comm_routing = Pre_match_comm_routing;

% Initialize the first hamming vector to all zeros.
Ham_previous = zeros(Num_Transceivers^2,1);

% Initialize the second hamming vector from "Pre_match_comm_routing"
Ham_current = hamming_vector(Pre_match_comm_routing,Num_Transceivers);

% calculate the hamming distance between vectors
ham_dist = hamming_distance(Ham_previous,Ham_current);

% While hamming distance of binary topology vectors is > ??
count = 0;
```

```
best_objective = 0;

%while ham_dist > 0 && count < 5

    % Operation 3:  Determine the final edge weights for use in developing a matching for
the initial solution.
    [Arc_weight] = reweight_arcs(Post_match_comm_routing,Num_Transceivers,QoS);

    clear Post_match_comm_routing % after the reweighting is done then clear it for the
next time around

    % Operation 4:  Determine the Matching between transceivers and rerun "a_star" to get
a new comm routing after the matching has been made.
    [Matching_final,Matching,Vert_adj_post_match,Match] =
create_matching(Arc_weight,I,Trans_Char,Num_Vertices);

    % Operation 4a: Update Bandwidth matrix and QoS matrix for use in
    % determining the routing by using a copy of each.  Since these matrices
    % will contain different "0" entries each time, assign them to
    % Bandwidth_matching and QoS_matching to avoid changing the originals.
    [Bandwidth_matching,QoS_matching] =
Bandwidth_QoS_matching_update(Bandwidth,QoS,Match,Trans_Char);

    % Note: use the QoS_matching and Bandwidth_matching matrices here
    % Operation 5:  Perform A* using "Vert_adj_post_match" and "Match" instead of
"Vert_adj_pre_match" and "I"
    Post_match_comm_routing =
route_message_traffic(Prioritized_Requests,Bandwidth_matching,QoS_matching,Trans_Char,Ver
t_adj_post_match,Match,P,QoS_Path);

    objective_value = calculate_obj_fcn(Post_match_comm_routing,P);

    %if count > 0
    %    if objective_value > best_objective
            best_solution = Post_match_comm_routing;
            best_objective = objective_value;
    %    end
    %end

    Ham_previous = Ham_current;

    Ham_current = hamming_vector(Post_match_comm_routing,Num_Transceivers);

    ham_dist = hamming_distance(Ham_previous,Ham_current);

    count = count + 1;
%end

v = cputime - t;
```

20. QoS_Bandwidth_Request.m

```
function [QoS,Bandwidth,Prioritized_Requests] =
QoS_Bandwidth_Request(User_inputs,Trans_Char,I,P)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%{

Function:

    [QoS,Bandwidth,Prioritized_Requests] = QoS_Bandwidth_Request
    (User_inputs,Trans_Char,I,P)

Author:

    R. Benjamin Hartlage, Capt, USAF

Description:
```

```
    Generates the Bandwidth, QoS, and Prioritized_Requests matrices used in
    the A* search.

Inputs:

    P:  As defined in create_requests.

    I:  As defined in create_I.

    Trans_Char:  As defined in create_Trans_Char.

    User_inputs:  As defined in NetDesign.

Outputs:

    QoS:  Is the matrix of randomly generated Quality of Service values.
          The i,j entries denote the quality obtained if a connection is
          formed between transceiver i and transceiver j.  These values are
          randomly generated since they will be obtained through expert
          opinion of communications personnel.

    Bandwidth:  Is the matrix denoting the bandwidth available on each edge
                in the network.

    Prioritized_Requests: Is the matrix containing the communications
                          requests generated in "P" that have been
                          prioritized in non-increasing order by
                          "Priority."  Within priority levels, ties are
                          broken by bandwidth (also in non-increasing
                          order.)

%}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

QoS = rand(size(Trans_Char,2));
QoS = QoS.*I;
QoS = QoS + QoS';

% Build Bandwidth and make it symetric
% Bandwidth = randint(size(Trans_Char,2),size(Trans_Char,2),[1 AVAILABLE_BANDWIDTH]);
Bandwidth =
random_integers(User_inputs.AVAILABLE_BANDWIDTH,size(Trans_Char,2),size(Trans_Char,2));
for i = 1:size(Bandwidth,1)
    for j = i:size(Bandwidth,2)
        Bandwidth(j,i) = 0;
    end
end
Bandwidth = Bandwidth + Bandwidth';

% Modify Bandwidth so that Bandwidth(i,j) = inf if i and j are at a common
% vertex and "0" if they are the same vertex.
for i = 1:size(Bandwidth,1)
    for j = i:size(Bandwidth,2)
        if Trans_Char(i).Vertex_ID == Trans_Char(j).Vertex_ID &&
Trans_Char(i).Transceiver_ID ~= Trans_Char(j).Transceiver_ID
            Bandwidth(i,j) = inf;
            Bandwidth(j,i) = inf;
        elseif i == j
            Bandwidth(i,j) = 0;
        end
    end
end

% Do the request prioritization here
PR1 = [P(:).Priority]';
PR2 = [P(:).Bandwidth]';
PR3 = [P(:).Origin]';
PR4 = [P(:).Destination]';
PR5 = [P(:).ID]';
Prioritized_Requests = horzcat(PR1, PR2, PR3, PR4, PR5);
Prioritized_Requests = sortrows(Prioritized_Requests, [-1 -2]);
```

21. random_integers.m

```
function [f] = random_integers(low,high,number_of_rows,number_of_columns,state)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%{

Function:

    [f] = random_integers(range,number_of_rows,number_of_columns,state)

Author:

    R. Benjamin Hartlage, Capt, USAF

Description:

    random_integers is used to generate a user specified number of random
    integers over a user specified range of values.  The numbers are
    uniformly distrbuted.

Inputs:

    low/high:  Defines the range of values over which the uniformly
               distributed random numbers will fall.

    number_of_rows:  The number of rows in the output.

    number_of_colums: The number of columns in the output.

    state: Seeds the random number generator so that the state can be reset
           prior to each use for comparison purposes, if desired.

Outputs:

    f:  the (number_of_rows)-by-(number_of_columns) matrix/vector/scalar
        of uniformly distributed randomly distributed integers in the
        specified range.

%}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% If all arguments are given set the state
if nargin == 5
    rand('state',state);
end

% If less than 4 are given set the columns
if nargin < 4
    number_of_columns = 1;
end

% If less than 3 then set the rows
if nargin < 3
    number_of_rows = 1;
end

% If only 1 is given then make it "high" and set low to 1
if nargin < 2
    high = low;
    low = 1;
end

f = low + ceil((high-low).*rand(number_of_rows,number_of_columns));
```

22. redirect_pointers.m

```
function [Node,redirect_path_structure] =
redirect_pointers(n_zero,Node,OPEN_g_h,OPEN_CLOSED,redirect_path_structure,OPEN_path_info
,n)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%{

Function:

    [Node,redirect_path_structure] = redirect_pointers(n_zero,Node,OPEN_g_h
    ,OPEN_CLOSED,redirect_path_structure,OPEN_path_info,n)

Author:

    R. Benjamin Hartlage, Capt, USAF

Description:

    For each member of OPEN_CLOSED redirect its pointer to n if the
    shortest path to the node found by A* so far is through n.

Inputs:

    n_zero:  The source node.  This node is the origin for the current
             communication request.

    OPEN_g_h:  As defined in calculate_g.

    Node:  The structiure that contains pointers for each node in the
           search graph.

    OPEN_CLOSED:  The list of nodes that are in the union of the OPEN and
                  CLOSED lists.

    redirect_path_structure:  Structure containing node_path, trans_path
                              and path_QoS for the list of nodes on
                              OPEN_CLOSED.

    OPEN_path_info: As defined in calculate_g.

    n:  The current node being considered.  n is an element of OPEN_CLOSED.

Outputs:

    Node:  Updated structure containing pointers for each node in the
           search graph.  Some of the nodes on OPEN_CLOSED may have had
           their pointers in Node redirected.

    redirect_path_structure:  Updated structure that contains new QoS if a
                              pointer is redirected.

Notes:

Now redirect pointers on OPEN_CLOSED to n if the best path to m found
so far is through n (this is part of step 7).  This must be done
here since we may remove elements of OPEN (hence elements of OPEN_CLOSED
also removed) in the pricing operations "calculate_h" and "calculate_g"

    1.  determine node path for n_zero to each element of
        OPEN (done in calculate_g).  This information is available from
        calculate_g in the structure "OPEN_path_info" This info is later
        stored in a structure called "redirect_path_structure" which is
        only updated in this function.  "redirect_path_structure" will
        contain path info for each element of OPEN_CLOSED since each
        element of CLOSED was on OPEN at one time.  Fields in
        "redirect_path_structure" are:

            redirect_path_structure.node_path
            redirect_path_structure.trans_path
```

```
            redirect_path_structure.path_QoS

    2.  determine if current transceiver path (the one obtained in
        "OPEN_path_info") is "shorter" (higher QoS) than the one contained
        in "redirect_path_structure", if it is then redirect the
        corresponding pointer to n

    2a. Update a structure called "redirect_path_structure" which is
        only updated in this function.

    Note that n_zero doesn't point to anything so need to account for
    that in this function.

    Conditions for redirecting a pointer are:
      1.  current element of OPEN_CLOSED is not n_zero
      2.  QoS of current path to m is greater than QoS stored in
"redirect_path_structure"
      3.  the path is through node n then redirect pointer for m in structure "Node" to n

%}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

for i = 1:size(OPEN_CLOSED,1)
    if (OPEN_CLOSED(i) ~= n_zero) &&
(~isempty(intersect(OPEN_path_info(OPEN_CLOSED(i)).Node_path,n)))
        if OPEN_path_info(OPEN_CLOSED(i)).Trans_Path_QoS >
redirect_path_structure(OPEN_CLOSED(i)).path_QoS
            Node(OPEN_CLOSED(i)).point_to = n;
            % Update "redirect_path_structure.path_QoS if the pointer is redirected"
            redirect_path_structure(OPEN_CLOSED(i)).path_QoS =
OPEN_path_info(OPEN_CLOSED(i)).Trans_Path_QoS;
        end
    end
end
```

23. reweight_arcs.m

```
function [Arc_weight] = reweight_arcs(Pre_match_comm_routing,Num_Transceivers,QoS)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%{

Function:

    [Arc_weight] = reweight_arcs_alternate(Pre_match_comm_routing,Num_Transceivers,QoS)

Author:

    R. Benjamin Hartlage, Capt, USAF

Description:

    Determine the final edge weights for use in developing a
    matching for the network topology created later by the greedy matching.
    Edge weights are based on the bandwidth that is carried by an arc in
    the initial routing.

Inputs:

    Pre_match_comm_routing:  The structure containing matching information
                             from the initial solution.

    Num_Transceivers:  As defined in create_I

Outputs:

    Arc_weight:  Is the matrix containing the arc weights for all
                 possible arcs that could be included in the matching.
```

```
%}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Arc_weight1 = [zeros(Num_Transceivers)];

for i = 1:size(Pre_match_comm_routing,2) %count through packets
    for j = 1:size(Pre_match_comm_routing(i).trans_path,2)-1 % count through all arcs
that carry a given packet
        Arc_weight1(Pre_match_comm_routing(i).trans_path(j),
Pre_match_comm_routing(i).trans_path(j+1)) =
Arc_weight1(Pre_match_comm_routing(i).trans_path(j),
Pre_match_comm_routing(i).trans_path(j+1)) + (Pre_match_comm_routing(i).message_bandwidth
* 10^(Pre_match_comm_routing(i).message_Priority));
        Arc_weight1(Pre_match_comm_routing(i).trans_path(j+1),
Pre_match_comm_routing(i).trans_path(j)) =
Arc_weight1(Pre_match_comm_routing(i).trans_path(j+1),
Pre_match_comm_routing(i).trans_path(j)) + (Pre_match_comm_routing(i).message_bandwidth *
10^(Pre_match_comm_routing(i).message_Priority));
    end
end

Arc_weight = Arc_weight1 .* QoS; % final arc weight is the product of arc QoS and the sum
of priority weighted bandwidth carried by the arc.
```

24. route_message_traffic.m

```
function [message_routing_structure] =
route_message_traffic(Prioritized_Requests,Bandwidth,QoS,Trans_Char,Vert_adj_pre_match,I,
P,QoS_Path)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%{

Function:

    [message_routing_structure] = route_message_traffic
    (Prioritized_Requests,Bandwidth,QoS,Trans_Char,Vert_adj_pre_match,I)

Author:

    R. Benjamin Hartlage, Capt, USAF

Description:

    route_message_traffic uses information about the existing network
    topology to find high QoS paths and route as much message traffic as
    possible through the topology.

Inputs:

    Prioritized_Requests:  As defined in QoS_Bandwidth_Request

    Bandwidth:  As defined in QoS_Bandwidth_Request

    QoS: As defined in QoS_Bandwidth_Request

    Trans_Char: As defined in create_Trans_Char

    Vert_adj_pre_match: As defined in Isolated_vert

    I:  As defined in create_I

    P:  As defined in create_requests

Outputs:

    message_routing_structure:  Contains the information on the message
                                routing.  This structure contains only
                                complete message routings.  Messages that
```

```
                                    are partial (sufficient bandwidth between
                                    origin and destination does not exist) are
                                    removed since partial messages are assumed
                                    to be of no value.
                                    message_routing_structure contains the
                                    following fields:

                                    message_ID
                                    message_bandwidth
                                    node_path
                                    trans_path
                                    path_QoS
                                    message_Priority

Notes:

    Vert_adj_pre_match and I need to be changed out depending on where this
    function is being run.  If it's run to obtain an initial solution then
    use the pre match matrices and if it's being run in the iterative
    reweighting loop then use the post match matrices instead.

%}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

i = 1;
Prioritized_Requests_copy = Prioritized_Requests;
Bandwidth_copy = Bandwidth;
QoS_copy = QoS;
prioritized_req_size = size(Prioritized_Requests_copy,1);
Pre_match_comm_routing.message_ID = [];
Pre_match_comm_routing.message_bandwidth = [];
Pre_match_comm_routing.node_path = [];
Pre_match_comm_routing.trans_path = [];
Pre_match_comm_routing.path_QoS = [];
Pre_match_comm_routing.message_Priority = [];
last_ID = -1;

while prioritized_req_size > 0
    if last_ID ~= Prioritized_Requests_copy(1,5)
        % take a snapshot
        %Prioritized_Requests_copy_snapshot = Prioritized_Requests_copy;
        QoS_copy_snapshot = QoS_copy;
        Bandwidth_copy_snapshot = Bandwidth_copy;
        i_snapshot = i;
        if i > 1
            Pre_match_comm_routing_snapshot = Pre_match_comm_routing;
        elseif i == 1
            Pre_match_comm_routing_snapshot.message_ID = [];
            Pre_match_comm_routing_snapshot.message_bandwidth = [];
            Pre_match_comm_routing_snapshot.node_path = [];
            Pre_match_comm_routing_snapshot.trans_path = [];
            Pre_match_comm_routing_snapshot.trans_path = [];
            Pre_match_comm_routing_snapshot.path_QoS = [];
            Pre_match_comm_routing.message_Priority = [];
        end
        last_ID = Prioritized_Requests_copy(1,5);
    end
    clear failure current_path_info;
    [failure, current_path_info] =
a_star(Prioritized_Requests_copy,Vert_adj_pre_match,I,Trans_Char,QoS_copy,Bandwidth_copy,
QoS_Path);
    if failure == 0
        [Pre_match_comm_routing(i),Bandwidth_copy,Prioritized_Requests_copy,QoS_copy] =
assign_message_route(Prioritized_Requests_copy,current_path_info,Bandwidth_copy,QoS_copy)
;
        i = i + 1;
    elseif failure == 1
        % If failure == 1 then restore from the snapshot and remove the
        % current element of Prioritized_Requests_copy since it could not
        % be routed in its entirety.
        Prioritized_Requests_copy(1,:) = [];
        Pre_match_comm_routing = Pre_match_comm_routing_snapshot;
```

```
        QoS_copy = QoS_copy_snapshot;
        Bandwidth_copy = Bandwidth_copy_snapshot;
        i = i_snapshot;
    end

    prioritized_req_size = size(Prioritized_Requests_copy,1);
end % while prioritized_req_size > 0

message_routing_structure = Pre_match_comm_routing;
```

25. run_test_cases.m

```
Network.vertices = [35 40];

Network.transceivers = [5 5];

Network.message_bandwidth = [20 30];

Network.arc_bandwidth = [60 90];

Network.tight_or_loose = [3 3];

for i = 1:2
    vertices = Network.vertices(i);
    for j = 1:2
        transceivers = Network.transceivers(j);
        for l = 1:2
            message_bandwidth = Network.message_bandwidth(l);
            for m = 1:2
                arc_bandwidth = Network.arc_bandwidth(m);
                for k = 1:2
                    tight_loose = Network.tight_or_loose(k);

                    file =
sprintf('test_case_v%d_t%d_m%d_a%d_tl%d',vertices,transceivers,message_bandwidth,arc_band
width,tight_loose);
                    load(file);

                    % here run NetDesign
                    NetDesign;

                    % calculate and collect metrics and objective function
                    % and label the output based on the testfile used.
                    results =
sprintf('results_v%d_t%d_m%d_a%d_tl%d',vertices,transceivers,message_bandwidth,arc_bandwi
dth,tight_loose);
                    metrics =
calculate_network_metrics(best_solution,Bandwidth,QoS,Match,Prioritized_Requests);
                    metrics.time = v;
                    metrics.final_solution = Post_match_comm_routing;
                    save(results,'metrics');
                end
            end
        end
    end
end
```

26. trans_path_finder.m

```
function [n_zero_to_n,no_trans_path_indicator] =
trans_path_finder(node_path,Trans_Char,Bandwidth,QoS,QoS_Path)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%{

Function:
```

```
    [n_zero_to_n,no_trans_path_indicator] = trans_path_finder(node_path,
    Trans_Char,Bandwidth,QoS)

Author:

    R. Benjamin Hartlage, Capt, USAF

Description:

    "trans_path_finder" finds the highest quality QoS path for the input
    "node_path" input or determines that a transceiver path does not exist.
    The output is "n_zero_to_n" which has the following elements:

Inputs:

    node_path:  The node path that was found in the A* search for the
                current message.

    Trans_Char:  As defined in create_Trans_Char

    Bandwidth:  As defined in QoS_Bandwidth_Request

    QoS:  As defined in QoS_Bandwidth_Request

Outputs:

    n_zero_to_n:  The structure containing information on the transceiver
                  path that is found by trans_path_finder.  See the notes
                  below for a listing of the fields contained in
                  n_zero_to_n.

    no_trans_path_indicator:  Assumes a value of "0" if a transceiver path
                              is found and a value of "1" if a transceiver
                              is determined not to exist.

Notes:

    If a transceiver path is found then output looks like:
        n_zero_to_n.Trans_path = (appropriate vector of transceivers)
        n_zero_to_n.Path_Bandwidth = (appropriate bandwidth)
        n_zero_to_n.Trans_Path_QoS = (appropriate QoS measure)
        no_trans_path_indicator = 0

    If no transceiver path exists then the output looks like:
        n_zero_to_n.Trans_path = []
        n_zero_to_n.Path_Bandwidth = 0
        n_zero_to_n.Trans_Path_QoS = 0
        no_trans_path_indicator = 1

%}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Transceiver_Path_g = [];
%QoS_Path = [Trans_Char(:).Vertex_ID]';
current1 = [find(QoS_Path == node_path(1))];
QoS_of_the_path = inf;
no_trans_path_indicator = 0;

for j = 1:length(node_path)-1
    current2 = [find(QoS_Path == node_path(j+1))];
    QoS_connect = QoS(current1,current2);
    Bandwidth_connect = Bandwidth(current1,current2);

    % Here be sure that bandwidth is positive.  If not then
    % change the corresponding element of QoS_connect to
    % "0" so the loop below won't select the arc
    % corresponding to it
    for zy = 1:size(QoS_connect,1)
        for zz = 1:size(QoS_connect,2)
            if Bandwidth_connect(zy,zz) == 0
                QoS_connect(zy,zz) = 0;
```

```
            end
        end
    end

    % Now connect node j and j+1 with the max element in QoS_connect.
    %[C,Y] = max(max(QoS_connect)); % Returns the column index of the max as Y
    %[V,F] = max(QoS_connect(:,Y)); % Returns the row index of the max as F

    [V,F] = max(QoS_connect,[],1);
    [V,Y] = max(V); % Returns the column index of the max as Y
    F = F(Y); % Returns the row index of the max as F

    % if the QoS is positive but not "perfect" (perfect QoS indicated by
    % "2" entries assigned to transceivers that are at the same vertices)
    % and it has not been determined that a transceiver path does not
    % exist.

    if V > 0 && V ~= 2 && no_trans_path_indicator == 0
        Transceiver_Path_g = [Transceiver_Path_g, current1(F), current2(Y)];
        if (V < QoS_of_the_path)
            QoS_of_the_path = V;
        end

    % If V == 2 then the connection is perfect since j and j+1 are
    % at a single vertex.  The implications of this type of
    % connection are that it has infinite bandwidth and
    % perfect QoS.  Need an indicator so use "2"
    elseif V == 2 && no_trans_path_indicator == 0
        Transceiver_Path_g = [Transceiver_Path_g, current1(F) current2(Y)];

    % if V == 0 then the QoS (or Bandwidth) on that link is
    % 0 and no transceiver path exists to connect the
    % n_zero to n for the current element of
    % OPEN
    elseif V == 0
        no_trans_path_indicator = 1;
    end

    % Note that this path may contain up to 2 nodes from the
    % same vertex since the matching is based on I
    current1 = current2;
end

if no_trans_path_indicator == 1
    % outside this function to check to see if a transceiver path exists
    % just check to see if n_zero_to_n.Trans_path is empty or not
    % if it's empty then no trans path with positive bandwidth could be
    % found
    Transceiver_Path_g = [];
    n_zero_to_n.Path_Bandwidth = 0;
    n_zero_to_n.Trans_Path_QoS = 0;
end

n_zero_to_n.Trans_path = Transceiver_Path_g;

if length(n_zero_to_n.Trans_path) > 0
    % First remove redundant elements of the transceiver path
    zzpath = [n_zero_to_n.Trans_path(1)];
    for zz = 2:size(n_zero_to_n.Trans_path,2)
        if n_zero_to_n.Trans_path(zz) ~= n_zero_to_n.Trans_path(zz-1)
            zzpath = [zzpath, n_zero_to_n.Trans_path(zz)];
        end
    end
    n_zero_to_n.Trans_path = zzpath;

    % Calculate the bandwidth on the transceiver path.
    path_bandwidth = inf;
    for j = 1:size(n_zero_to_n.Trans_path,2)-1
        if Bandwidth(n_zero_to_n.Trans_path(j),n_zero_to_n.Trans_path(j+1)) <
path_bandwidth
            path_bandwidth =
Bandwidth(n_zero_to_n.Trans_path(j),n_zero_to_n.Trans_path(j+1));
```

```
            arc_limiting_path_Bandwidth =
[n_zero_to_n.Trans_path(j),n_zero_to_n.Trans_path(j+1)];
        end
    end
    n_zero_to_n.Trans_Path_QoS = QoS_of_the_path;
    n_zero_to_n.Path_Bandwidth = path_bandwidth;

end

% this can be used to return an indicator of 1 if no transceiver path
% with positive bandwidth could be found...should be redundant but leave it
% here anyway for right now.
if n_zero_to_n.Path_Bandwidth == 0
    no_trans_path_indicator = 1;
end
```

Appendix B: Tabulated Metrics for Test Cases

Test Case	Metric	A*	Dijkstra's
Test_case_v30_t4_m10_a30_tl2.mat	QoS_for_all_arcs_network	.49962	.49962
	QoS_for_chosen_network	.94718	.92971
	QoS_for_messages_routed	.96469	.98001
	topology_efficiency	.19805	.15908
	total_band_transmitted_ratio	.02024	.01673
	total_band_upper_bound	.10218	.10517
	time	1022.7	898.14
	objective_value	1853.1	1525.3
Test_case_v30_t4_m10_a30_tl3.mat	QoS_for_all_arcs_network	.49609	.49609
	QoS_for_chosen_network	.91816	.93033
	QoS_for_messages_routed	.95429	.97536
	topology_efficiency	.20204	.20483
	total_band_transmitted_ratio	.02176	.02186
	total_band_upper_bound	.10771	.1067
	time	609.36	550.68
	objective_value	2005.7	2050.7
Test_case_v30_t4_m10_a90_tl2.mat	QoS_for_all_arcs_network	.50142	.50142
	QoS_for_chosen_network	.84954	.93888
	QoS_for_messages_routed	.95861	.97592
	topology_efficiency	.15462	.20648
	total_band_transmitted_ratio	.04929	.05616
	total_band_upper_bound	.31878	.272
	time	752.44	941.95
	objective_value	4379.2	4945.1
Test_case_v30_t4_m10_a90_tl3.mat	QoS_for_all_arcs_network	.50038	.50038
	QoS_for_chosen_network	.87271	.82588
	QoS_for_messages_routed	.97008	.98943
	topology_efficiency	.14978	.14697
	total_band_transmitted_ratio	.05784	.05817
	total_band_upper_bound	.38616	.3958
	time	860.39	817.48
	objective_value	5212.2	5350.2
Test_case_v30_t4_m30_a30_tl2.mat	QoS_for_all_arcs_network	.50916	.50916
	QoS_for_chosen_network	.91569	.91569
	QoS_for_messages_routed	.96076	.98513
	topology_efficiency	.23297	.23297
	total_band_transmitted_ratio	.00813	.00813
	total_band_upper_bound	.03491	.03491
	time	832.56	825.83
	objective_value	2176.5	2176.1
Test_case_v30_t4_m30_a30_tl3.mat	QoS_for_all_arcs_network	.49775	.49775
	QoS_for_chosen_network	.91959	.89644
	QoS_for_messages_routed	.96742	.98475
	topology_efficiency	.1936	.20961

	total_band_transmitted_ratio	.00653	.00749
	total_band_upper_bound	.03372	.03574
	time	521.88	463.09
	objective_value	1751.3	1901.3
Test_case_v30_t4_m30_a90_tl2.mat	QoS_for_all_arcs_network	.50422	.50422
	QoS_for_chosen_network	.92166	.92166
	QoS_for_messages_routed	.95676	.98499
	topology_efficiency	.17388	.18164
	total_band_transmitted_ratio	.01764	.01843
	total_band_upper_bound	.10146	.10146
	time	849.24	718.58
	objective_value	4846.7	4897.1
Test_case_v30_t4_m30_a90_tl3.mat	QoS_for_all_arcs_network	.499	.499
	QoS_for_chosen_network	.82542	.84474
	QoS_for_messages_routed	.96852	.98386
	topology_efficiency	.23486	.1669
	total_band_transmitted_ratio	.02734	.01919
	total_band_upper_bound	.11639	.11497
	time	709.28	539.13
	objective_value	6904.2	5145
Test_case_v30_t5_m10_a30_tl2.mat	QoS_for_all_arcs_network	.50542	.50542
	QoS_for_chosen_network	.88368	.95394
	QoS_for_messages_routed	.95911	.98762
	topology_efficiency	.16056	.13671
	total_band_transmitted_ratio	.02119	.01611
	total_band_upper_bound	.13195	.11787
	time	1293.5	1368.2
	objective_value	1903.7	1604.9
Test_case_v30_t5_m10_a30_tl3.mat	QoS_for_all_arcs_network	.50073	.50073
	QoS_for_chosen_network	.94275	.92051
	QoS_for_messages_routed	.95964	.98505
	topology_efficiency	.15294	.16953
	total_band_transmitted_ratio	.01861	.02119
	total_band_upper_bound	.12169	.12498
	time	1198.5	1047.7
	objective_value	1757.6	1995.1
Test_case_v30_t5_m10_a90_tl2.mat	QoS_for_all_arcs_network	.49754	.49754
	QoS_for_chosen_network	.75169	.81279
	QoS_for_messages_routed	.9816	.99038
	topology_efficiency	.11399	.10981
	total_band_transmitted_ratio	.05981	.04915
	total_band_upper_bound	.52472	.4476
	time	871.4	886.65
	objective_value	5427.3	4419.4
Test_case_v30_t5_m10_a90_tl3.mat	QoS_for_all_arcs_network	.49956	.49956
	QoS_for_chosen_network	.74802	.83551
	QoS_for_messages_routed	.97552	.99203
	topology_efficiency	.10545	.10009
	total_band_transmitted_ratio	.06767	.04834

	total_band_upper_bound	.64171	.48299
	time	907.03	885.93
	objective_value	6030	4409.4
Test_case_v30_t5_m30_a30_tl2.mat	QoS_for_all_arcs_network	.50084	.50084
	QoS_for_chosen_network	.95223	.95223
	QoS_for_messages_routed	.96479	.98698
	topology_efficiency	.18582	.1752
	total_band_transmitted_ratio	.00803	.007575
	total_band_upper_bound	.04324	.04324
	time	1620.6	1496.4
	objective_value	2246.8	2134
Test_case_v30_t5_m30_a30_tl3.mat	QoS_for_all_arcs_network	.50451	.50451
	QoS_for_chosen_network	.94807	.94807
	QoS_for_messages_routed	.95818	.98719
	topology_efficiency	.14685	.14503
	total_band_transmitted_ratio	.00663	.00654
	total_band_upper_bound	.04512	.04512
	time	1956.3	1663
	objective_value	1821	1805.3
Test_case_v30_t5_m30_a90_tl2.mat	QoS_for_all_arcs_network	.49255	.49255
	QoS_for_chosen_network	.90041	.93562
	QoS_for_messages_routed	.98172	.99312
	topology_efficiency	.175	.1355
	total_band_transmitted_ratio	.02348	.01677
	total_band_upper_bound	.13418	.1238
	time	1389.6	1465.1
	objective_value	6365.5	4806.4
Test_case_v30_t5_m30_a90_tl3.mat	QoS_for_all_arcs_network	.49996	.49996
	QoS_for_chosen_network	.93917	.93575
	QoS_for_messages_routed	.97654	.98568
	topology_efficiency	.19045	.15201
	total_band_transmitted_ratio	.02362	.01886
	total_band_upper_bound	.12403	.12408
	time	1366.1	1385.2
	objective_value	6273.9	4852.1
Test_case_v40_t4_m10_a30_tl2.mat	QoS_for_all_arcs_network	.50389	.50389
	QoS_for_chosen_network	.90283	.87689
	QoS_for_messages_routed	.96177	.98636
	topology_efficiency	.16118	.1604
	total_band_transmitted_ratio	.02183	.02269
	total_band_upper_bound	.13546	.14144
	time	2195.5	1919.1
	objective_value	1946.2	2134
Test_case_v40_t4_m10_a30_tl3.mat	QoS_for_all_arcs_network	.49466	.49466
	QoS_for_chosen_network	.85306	.88221
	QoS_for_messages_routed	.96114	.97724
	topology_efficiency	.13628	.11946
	total_band_transmitted_ratio	.02362	.01896
	total_band_upper_bound	.17329	.1587

	time	2101.9	1736.8
	objective_value	2037.7	1811.7
Test_case_v40_t4_m10_a90_tl2.mat	QoS_for_all_arcs_network	.5027	.5027
	QoS_for_chosen_network	.70384	.89688
	QoS_for_messages_routed	.97796	.99022
	topology_efficiency	.09265	.12058
	total_band_transmitted_ratio	.06616	.05207
	total_band_upper_bound	.71413	.43182
	time	1468.2	1606
	objective_value	5800	4824.6
Test_case_v40_t4_m10_a90_tl3.mat	QoS_for_all_arcs_network	.50302	.50302
	QoS_for_chosen_network	.85684	.7634
	QoS_for_messages_routed	.97888	.9893
	topology_efficiency	.13863	.10578
	total_band_transmitted_ratio	.06950	.06044
	total_band_upper_bound	.47539	.57137
	time	1426.8	1505.7
	objective_value	6012.6	5512
Test_case_v40_t4_m30_a30_tl2.mat	QoS_for_all_arcs_network	.49993	.49993
	QoS_for_chosen_network	.63608	.79745
	QoS_for_messages_routed	.97374	.98482
	topology_efficiency	.09982	.20011
	total_band_transmitted_ratio	.01069	.01221
	total_band_upper_bound	.10717	.06103
	time	1087.7	1600.3
	objective_value	2726.9	3248
Test_case_v40_t4_m30_a30_tl3.mat	QoS_for_all_arcs_network	.50059	.50059
	QoS_for_chosen_network	.89344	.91099
	QoS_for_messages_routed	.96653	.97997
	topology_efficiency	.17398	.18286
	total_band_transmitted_ratio	.00787	.00808
	total_band_upper_bound	.04522	.04419
	time	1391	1535.7
	objective_value	2228	2358.3
Test_case_v40_t4_m30_a90_tl2.mat	QoS_for_all_arcs_network	.49433	.49433
	QoS_for_chosen_network	.91986	.89967
	QoS_for_messages_routed	.95808	.98889
	topology_efficiency	.1741	.12693
	total_band_transmitted_ratio	.02444	.01819
	total_band_upper_bound	.14038	.14335
	time	2373.5	2103.8
	objective_value	6418.3	5011.7
Test_case_v40_t4_m30_a90_tl3.mat	QoS_for_all_arcs_network	.49965	.49965
	QoS_for_chosen_network	.68215	.71263
	QoS_for_messages_routed	.96972	.98413
	topology_efficiency	.1102	.08475
	total_band_transmitted_ratio	.02722	.01868
	total_band_upper_bound	.24703	.22046
	time	980.21	953.41

	objective_value	7096.8	5068.5
Test_case_v40_t5_m10_a30_tl2.mat	QoS_for_all_arcs_network	.50075	.50075
	QoS_for_chosen_network	.94172	.9254
	QoS_for_messages_routed	.97337	.98911
	topology_efficiency	.18588	.1406
	total_band_transmitted_ratio	.02924	.02236
	total_band_upper_bound	.15729	.159
	time	2233.1	1948.2
	objective_value	2810.2	1955.5
Test_case_v40_t5_m10_a30_tl3.mat	QoS_for_all_arcs_network	.50295	.50295
	QoS_for_chosen_network	.82142	.87863
	QoS_for_messages_routed	.96667	.9867
	topology_efficiency	.11184	.11357
	total_band_transmitted_ratio	.02343	.02095
	total_band_upper_bound	.20945	.1845
	time	2139.9	1824.4
	objective_value	2087.1	2037.5
Test_case_v40_t5_m10_a90_tl2.mat	QoS_for_all_arcs_network	.50323	.50323
	QoS_for_chosen_network	.67799	.77936
	QoS_for_messages_routed	.97743	.9899
	topology_efficiency	.06584	.07436
	total_band_transmitted_ratio	.97181	.05197
	total_band_upper_bound	.97181	.69894
	time	1548.8	1719.7
	objective_value	5718.2	4918.5
Test_case_v40_t5_m10_a90_tl3.mat	QoS_for_all_arcs_network	.49883	.4988.3
	QoS_for_chosen_network	.70798	.70807
	QoS_for_messages_routed	.98212	.9932
	topology_efficiency	.07385	.06260
	total_band_transmitted_ratio	.06690	.05702
	total_band_upper_bound	.90592	.91081
	time	1571.2	1728.7
	objective_value	6058.9	5036
Test_case_v40_t5_m30_a30_tl2.mat	QoS_for_all_arcs_network	.50154	.50154
	QoS_for_chosen_network	.92598	.92598
	QoS_for_messages_routed	.97402	.99296
	topology_efficiency	.13577	.18568
	total_band_transmitted_ratio	.00811	.01110
	total_band_upper_bound	.05977	.05977
	time	4523.5	3702.9
	objective_value	2264.8	3059.5
Test_case_v40_t5_m30_a30_tl3.mat	QoS_for_all_arcs_network	.50128	.50128
	QoS_for_chosen_network	.95418	.95401
	QoS_for_messages_routed	.95044	.98552
	topology_efficiency	.13206	.20131
	total_band_transmitted_ratio	.00759	.01159
	total_band_upper_bound	.05746	.05756
	time	4386.7	3204.2
	objective_value	2006.6	3021.6

Test_case_v40_t5_m30_a90_tl2.mat	QoS_for_all_arcs_network	.50099	.50099
	QoS_for_chosen_network	.85885	.9733
	QoS_for_messages_routed	.97777	.99164
	topology_efficiency	.14053	.12427
	total_band_transmitted_ratio	.02824	.01893
	total_band_upper_bound	.20098	.15961
	time	2424.8	3042.5
	objective_value	7280.7	5491.2
Test_case_v40_t5_m30_a90_tl3.mat	QoS_for_all_arcs_network	.50126	.50126
	QoS_for_chosen_network	.67776	.70116
	QoS_for_messages_routed	.97389	.99269
	topology_efficiency	.09861	.06386
	total_band_transmitted_ratio	.03294	.01926
	total_band_upper_bound	.33407	.30159
	time	1493.7	1390.3
	objective_value	8400.8	5309.4

Test Cases vs. Number of Packets		
Test Case	A*	Dijkstra's
ndp1	44	37
ndp2	45	46
ndp3	98	112
ndp4	114	109
ndp5	19	17
ndp6	18	15
ndp7	43	37
ndp8	60	41
ndp9	47	31
ndp10	41	40
ndp11	118	96
ndp12	127	92
ndp13	22	18
ndp14	18	13
ndp15	54	33
ndp16	53	43
ndp17	50	45
ndp18	48	37
ndp19	132	99
ndp20	124	112
ndp21	21	24
ndp22	19	17
ndp23	55	36
ndp24	58	39
ndp25	65	49
ndp26	52	37
ndp27	125	99

ndp28	130	107
ndp29	21	22
ndp30	21	24
ndp31	64	44
ndp32	71	37

QoS_of_chosen_network:

Average_v30_t4.mat	*Variable 1*	*Variable 2*
Mean	0.896244302	0.900380172
Variance	0.001770724	0.001804801
Observations	8	8
Hypothesized Mean Difference	0	
df	14	
t Stat	-0.195632941	
P(T<=t) one-tail	**0.423854236**	
t Critical one-tail	1.761310115	
P(T<=t) two-tail	0.847708472	
t Critical two-tail	2.144786681	

Average_v30_t5.mat	*Variable 1*	*Variable 2*
Mean	0.883252551	0.911802305
Variance	0.007359442	0.003080707
Observations	8	8
Hypothesized Mean Difference	0	
df	12	
t Stat	-0.790303686	
P(T<=t) one-tail	**0.222344109**	
t Critical one-tail	1.782287548	
P(T<=t) two-tail	0.444688217	
t Critical two-tail	2.178812827	

Average_v40_t4.mat	*Variable 1*	*Variable 2*
Mean	0.806012588	0.842513025
Variance	0.012776031	0.005546905
Observations	8	8
Hypothesized Mean Difference	0	
df	12	
t Stat	-0.762685575	
P(T<=t) one-tail	**0.230190549**	
t Critical one-tail	1.782287548	
P(T<=t) two-tail	0.460381098	
t Critical two-tail	2.178812827	

Average_v40_t5.mat	*Variable 1*	*Variable 2*
Mean	0.820735325	0.855739057
Variance	0.01407181	0.012189212
Observations	8	8
Hypothesized Mean Difference	0	
df	14	
t Stat	-0.610946998	
P(T<=t) one-tail	**0.275512839**	
t Critical one-tail	1.761310115	
P(T<=t) two-tail	0.551025679	
t Critical two-tail	2.144786681	

QoS_for_messages_routed:

Average_v30_t4.mat	*Variable 1*	*Variable 2*
Mean	0.962641067	0.982430122
Variance	3.44226E-05	2.40692E-05
Observations	8	8
Hypothesized Mean Difference	0	
df	14	
t Stat	-7.318508715	
P(T<=t) one-tail	**1.90E-06**	
t Critical one-tail	1.761310115	
P(T<=t) two-tail	3.79958E-06	
t Critical two-tail	2.144786681	

Average_v30_t5.mat	*Variable 1*	*Variable 2*
Mean	0.969637251	0.988506544
Variance	0.00010528	8.86209E-06
Observations	8	8
Hypothesized Mean Difference	0	
df	8	
t Stat	-4.995483634	
P(T<=t) one-tail	**0.000529392**	
t Critical one-tail	1.859548033	
P(T<=t) two-tail	0.001058784	
t Critical two-tail	2.306004133	

Average_v40_t4.mat	*Variable 1*	*Variable 2*
Mean	0.968477406	0.985115439
Variance	6.25854E-05	2.12555E-05
Observations	8	8
Hypothesized Mean Difference	0	
df	11	
t Stat	-5.139478029	
P(T<=t) one-tail	**0.000161749**	
t Critical one-tail	1.795884814	
P(T<=t) two-tail	0.000323498	
t Critical two-tail	2.200985159	

Average_v40_t5.mat	*Variable 1*	*Variable 2*
Mean	0.971966768	0.990214603
Variance	9.53576E-05	8.62333E-06
Observations	8	8
Hypothesized Mean Difference	0	
df	8	
t Stat	-5.061502373	
P(T<=t) one-tail	**0.000487615**	
t Critical one-tail	1.859548033	
P(T<=t) two-tail	0.00097523	
t Critical two-tail	2.306004133	

topology_efficiency:

Average_v30_t4.mat	*Variable 1*	*Variable 2*
Mean	0.192475113	0.188562039
Variance	0.001023062	0.000873842
Observations	8	8
Hypothesized Mean Difference	0	
df	14	
t Stat	0.25412092	
P(T<=t) one-tail	**0.401548234**	
t Critical one-tail	1.761310115	
P(T<=t) two-tail	0.803096469	
t Critical two-tail	2.144786681	

Average_v30_t5.mat	*Variable 1*	*Variable 2*
Mean	0.153881854	0.140484288
Variance	0.0009764	0.000687777
Observations	8	8
Hypothesized Mean Difference	0	
df	14	
t Stat	0.928904428	
P(T<=t) one-tail	**0.184339226**	
t Critical one-tail	1.761310115	
P(T<=t) two-tail	0.368678453	
t Critical two-tail	2.144786681	

Average_v40_t4.mat	*Variable 1*	*Variable 2*
Mean	0.135853898	0.137609002
Variance	0.001055507	0.001573437
Observations	8	8
Hypothesized Mean Difference	0	
df	13	
t Stat	-0.096818218	
P(T<=t) one-tail	**0.462173734**	
t Critical one-tail	1.770933383	
P(T<=t) two-tail	0.924347467	
t Critical two-tail	2.160368652	

Average_v40_t5.mat	*Variable 1*	*Variable 2*
Mean	0.118046041	0.12078057
Variance	0.001530336	0.002847782
Observations	8	8
Hypothesized Mean Difference	0	
df	13	
t Stat	-0.116891739	
P(T<=t) one-tail	**0.45436621**	
t Critical one-tail	1.770933383	
P(T<=t) two-tail	0.90873242	
t Critical two-tail	2.160368652	

Bibliography

Ahuja, R. K., Magnanti, T. L., & Orlin, J. B. (1993). *Network flows* (First ed.). Upper Saddle River, NJ: Prentice Hall.

Wikipedia (2006) for definition of Quality of Service, http://en.wikipedia.org/wiki/Quality_of_Service, retrieved June 27, 2006

Department of Defense. (2000). *"Joint vision 2020. america's military: Preparing for tomorrow"*. Washington DC: US Government Printing Office.

Erwin, M. C. (2006). Combining quality of service and topology control in directional hybrid wireless networks. (MS Thesis, AFIT/GOR/ENS/06-07, Graduate school of Engineering and Management, Air Force Institute of Technology (AU)).

Guéret, C., Prins, C., & Sevaux, M. (2000). Applications of optimization with xpress-MP [Electronic version]. *Editions Eyrolles,*

Gurumohan, P.,C., Taylor, T.,J., & Syrotiuk, V.,R. (2004). Topology control for MATNETs. , *1* 599-600-603.

Marsden, J. E., & Hoffman, M. J. (1993). *Elementary classical analysis* (2nd ed. ed.). Monticello, IL: W.H. Freeman and Company.

Meyer, C. D. (2000). *Matrix analysis and applied linear algebra* (1st edition ed.). Philadelphia, PA: Society for Industrial and Applied Mathematics.

Montgomery, D. C. (2005). *Design and analysis of experiments* (1st ed.). Danvers, MA: John Wiley & Sons, Inc.

Pooch, U. W., Machuel, D., & McCahn, J. (1991). *Telecommunications networking* (First ed.). Boca Raton, FL: CRC Press, Inc.

West, D., B. (2001). *Introduction to graph theory* (Second Edition ed.). Upper Saddle River, NJ: Prentice Hall.

Wolsey, L. A. (1998). *Integer programming* (First ed.). New York: John Wiley & Sons.

Wong, R. T. (1978). *Worst-case analysis of network design problem heuristics* No. OR-085-78). Massachusetts Institute of Technology, Operations Research Center: MIT

Vita

Captain Hartlage received his B.S. in Mathematics from the University of Louisville in 2002. His first assignment in the Air Force was with AFRL/HECS as a Program Manager. While working in AFRL he earned his M.S. Eng. in Human Factors from Wright State University in 2004. His follow-on assignment is to the Air Force Information Operations Center in AIA at Lackland AFB, TX. He resides in Huber Heights, Ohio with his lovely wife and his wonderful daughter.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From – To)*
23-02-2007	**Master's Thesis**	Sep 2005 - Mar 2007

4. TITLE AND SUBTITLE
AN EFFICIENT METAHEURISTIC FOR DYNAMIC NETWORK DESIGN AND MESSAGE ROUTING

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Hartlage, Robert, B., Captain, USAF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Street, Building 642
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GOR/ENS/07-10

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
N/A

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The implementation of Net-centric warfare presents major challenges in terms of effectively and efficiently delivering critical information across the Global Information Grid. In many cases, the amount of information requested will exceed the capabilities of the network. One challenge is to dynamically design the network (assign transceivers) to maximize the amount of required information that can be transmitted and the quality of service for those transmissions – to best implement the communications tasking order. The problem is as follows: given a list of required message traffic, to include source, destination, size, and priority, design the network to maximize the delivery of the message traffic based on message priority and quality of service. Once the network is designed, the routing for the messages must be determined. Due to the dynamic nature of the problem and the combinatorial explosion in size as new network nodes are added, a quick-running heuristic approach is needed. In this research, metaheuristic to dynamically design the network based on the projected message traffic requirements and to efficiently route the required messages on the network to maximize priority of messages successfully delivered and the quality of service of the delivery. The meta-heuristic is tested against previous efforts and is shown to generate high quality solutions in a very short amount of time.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Stephen P. Chambal, Capt, USAF (ENS)
U	U	U	UU	118	19b. TELEPHONE NUMBER *(Include area code)* (937) 255-3636, ext 4601; e-mail: Maj. Gary W. Kinney

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39-18