1

PM-1 3 1/2"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Notice: Acquisition

Notice: Notice d'acquisition

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

**Operations Research Tools
Useful in
Network Fault Management**

by

Pavel Kalab

**A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of
Master of Science**

**Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada
December 25, 1992**

© 1992 by Pavel Kalab

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file - Votre référence

Our file - Notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-84150-8

Canada

The undersigned recommend to the Faculty of Graduate Studies and Research acceptance of the thesis

**Operations Research Tools
Useful in Network Fault Management**

submitted by Pavel Kalab in partial fulfillment of the requirements for the degree of Master of Science.

.....
Thesis Supervisor

.....
Thesis Supervisor

.....
**Chair,
Department of Systems and
Computer Engineering**

**Carleton University
March 15, 1993**

Abstract

As the complexity of modern communication networks increases, new tools must be developed for efficient network fault management. The Pegasus multi-paradigm approach is one such tool. Individual knowledge sources rely on information obtained from data collectors, whose optimal location in the network is highly desirable. Dynamic routing of packets also hinders the knowledge sources in rapidly identifying a faulty device or line.

This thesis addresses the optimal location of multiple data collectors in a general network and calculation of n shortest paths between an arbitrary pair of nodes. Output generated by software modules developed as part of this thesis can be used by other knowledge sources to make the Pegasus system more efficient.

key words:

median location, shortest path, network optimization, graph theory

Acknowledgments

I wish to express sincere gratitude to my supervisors, Dr. N. W. Dawes and Dr. B. Pagurek, for their support and guidance throughout the course of this work, and for sharing with me their knowledge about network fault management.

My sincere thanks are due to Lyle Zary and every other member of the Pegasus design team for their numerous suggestions and comments, and to Don Bailey for suggestions about programming style.

I also wish to thank my other friends, especially Robert Brenning, for the numerous useful pointers on programming in C.

Table of Contents

1. Introduction and Organization	1
1.1 Introduction	1
1.2 Objectives	2
1.3 Organization	3
2. The BNR/NT Network	4
2.1 Network Architecture	4
2.2 Diagnostic Support	7
2.3 Network Faults	10
3. The Pegasus Project.....	11
3.1 Objectives	11
3.2 General System Architecture	11
3.3 Dragon Simulator	12
3.4 <i>BANES</i> (Basic Analysis in Nested Evidence Spaces)	14
3.5 <i>END</i> (Expert Network Diagnostician).....	14
3.6 <i>LEAPR</i> (Line Element Anomaly Pattern Recognition).....	16
3.7 Other Knowledge Sources	17
4. Optimization Techniques.....	19
4.1 Basic Definitions.....	19
4.2 Shortest Path Problem	21
4.3 Shortest Paths Between All Pairs of Nodes	24
4.4 Calculation of the n^{th} Shortest Path	25
4.5 Location of Medians in a Network.....	26
4.6 Optimization of Computer Networks.....	32
5. Data Representation and Support Functions	33
5.1 Programming Environment	33
5.2 Data Representation.....	34
5.3 Internal Representation of Network Topology	38
5.4 Assigning Arc Lengths	41
5.5 Support Functions	45
5.6 Shortest Path Functions.....	46
6. Optimal Location of Data Collectors.....	49
6.1 Selection of the Algorithm.....	49
6.2 Distance Matrix Calculated by <i>MGEN</i>	50
6.3 General Logic of the <i>MEDIAN</i> Algorithm	53
6.4 Function <i>localMedians</i>	58

7. Calculation of n Shortest Paths (<i>NPATH</i>)	61
7.1 General Description of the Algorithm	61
7.2 Implementation of the Algorithm	63
7.3 <i>NPATH</i> Knowledge Source	69
7.4 Node Compression	71
7.5 Client Process Testing <i>NPATH</i>	76
7.6 Running the Programs	80
8. Complexity of the Network Topology	82
8.1 Network Topology	82
8.2 Hierarchical Methodology	89
9. Finding Multiple Optimal Medians	91
9.1 <i>MEDIAN</i> Input and Output	91
9.2 Network Size and Running Time	95
9.3 Topologies Using Various Arc Lengths	97
9.4 Calculation of an Optimal Solution	103
9.5 Convergence to an Optimal Solution and Cycling	106
9.6 Random Sequences	108
9.7 Dragon Simulator Results	109
10. Searching for Shortest Paths	116
10.1 Fully Connected Networks	116
10.2 Complete Paths in the BNR/NT Network	119
10.3 Partial Paths and Node Compression	124
10.4 Correlation with the Dragon Simulator	127
10.5 Use of <i>NPATH</i> by other Knowledge Sources	128
11. Conclusions and Recommendations	129
11.1 Conclusions	129
11.2 Recommendations	132
Glossary of Selected Terms	134
References	136
Appendices	139
A - Data Structure Definitions - <i>bb_pktype.h</i>	139
B - General Functions - <i>bb_pkfun.c</i>	141
C - Path-related Functions - <i>bb_pathfun.c</i>	149
D - Distance Matrix Generator - <i>MGEN</i>	161
E - <i>MEDIAN</i> Program	165
F - <i>NPATH</i> Program	180
G - Node Compression Functions - <i>bb_compfun.c</i>	185
H - <i>NP</i> Program	195

List of Figures

2.1	Typical connectivity of devices in the BNR/NT network.....	6
5.1	Sample network showing the correct format of topology and arc length data files.	37
5.2	Internal representation of the network topology.....	40
5.3	Flow chart showing the logic of assigning arc lengths.....	44
5.4	Algorithm used to calculate the shortest path in a weighted graph.	48
6.1	Flow chart of the logic used in the <i>MGEN</i> program.....	52
6.2	Flow chart of the logic used in the <i>MEDIAN</i> program.....	54
6.3	Flow chart of the logic used in the function <i>localMedians</i>	60
7.1	Calculation of candidate paths in the sample network.....	62
7.2	Flow chart of the logic used in the function <i>findShortestPaths</i>	65
7.3	Flow chart of the logic used in the <i>NPATH</i> program.	68
7.4	Flow chart of the logic used in the function <i>compressNodes</i>	75
7.5	Flow chart of the logic used in the <i>NP</i> program.....	79
8.1	Topology of the compressed nodes in the BNR/NT network.	84
8.2	BNR/NT network topology.....	86
9.1	Typical data entry for the <i>MEDIAN</i> program.	93
9.2	Typical output produced by the <i>MEDIAN</i> program.	94
10.1	Search times for shortest paths in fully-connected networks.....	118
10.2	Search times for the shortest paths in the BNR/NT network.....	120
10.3	Detailed topology of the Carling site.....	123

List of Tables

7.1	Data fields posted to the Blackboard by NPATH.	71
7.2	Delimiters posted to the Blackboard at the end of a specific block of data.	71
8.1	Summary of network devices compressed according to geographical location.	85
8.2	Arc lengths for the various BNR/NT network topologies studied.	88
9.1	Summary of the selection of 8 optimal medians in networks having 76 and 251 nodes.	100
9.2	Summary of the selection of 8 optimal medians in networks having 497 and 1262 nodes.	101
9.3	The effect of random sequences and network topologies on the selection of optimal medians in the BNR/NT network.	102
9.4	Packet transfers with standard Type 1A topology.	113
9.5	Packet transfers with optimal Type 1A topology.	113
9.6	Packet transfers with optimal Type 1B topology.	113
9.7	Packet transfers with Type 1B topology and 7 data collectors.	114
9.8	Packet transfers with Type 1B topology and 6 data collectors.	114
9.9	Packet transfers with Type 1B topology and 2 data collectors.	114
9.10	Packet transfers with Type 1B topology and 9 data collectors.	115
9.11	Packet transfers with Type 1B topology and 10 data collectors.	115
9.12	Packet transfers with Type 1B topology and 15 data collectors.	115

Chapter 1

Introduction and Organization

1.1 Introduction

During the past few decades, computers have become essential tools used by professionals in many occupations. The reliance of our society on computer technology has reached the point where it becomes difficult if not impossible to function without it. Most of the developed countries throughout the world are today linked by vast interconnected networks which permit the relatively easy transfer of information around the globe, regardless of whether that information is voice, data, or some other medium.

In conjunction with the growing need to transfer larger and larger quantities of data as quickly as possible between distant locations came the need to efficiently monitor and manage the networks on which this information is being transferred. Network management and network fault diagnosis developed as an important field of systems and computer engineering. As the size and complexity of the networks grew, the methods and tools which engineers used to monitor networks also grew in complexity. Ideally, any fault which might occur in a network should be detected and corrected automatically by special computer hardware and software with minimal intervention from the network manager, a human. Such automated fault diagnosis is the ultimate objective of the Pegasus project.

1.2 Objectives

The objective of this thesis is to develop tools which can be used by various modules of the Pegasus multi-paradigm system in the detection and diagnosis of faults in wide area networks. Specifically, the goals of this thesis are:

1. Using a hierarchical approach, investigate the feasibility of simplifying the network topology by selectively compressing several nodes of the network into supernodes, processing them, and again decompressing them into their constituent parts.
2. Review the present location of data collectors which monitor the network and propose an optimal location in order to minimize the network traffic generated by the data collectors.
3. Develop a system which would provide a ranked list of probable paths by which packets might travel from the origin to the destination under conditions of dynamic routing, and incorporate this system into the Pegasus network diagnostic software.

Several computer programs, run as stand-alone modules or connected to the other modules of the Pegasus diagnostic system, have been developed to accomplish these goals. Simulations were subsequently carried out on a small test subset of the BNR/NT network to test the hypotheses proposed in this thesis.

1.3 Organization

The remainder of this thesis is divided into 11 chapters. The first 4 chapters provide a description of the Pegasus project and the BNR/NT network used as a test platform by the Pegasus group. A review of relevant literature is also contained in these chapters. Chapters 5 to 7 describe the algorithms of the software modules developed as part of this thesis. Chapters 8 to 10 discuss the results of simulations carried out as part of the independent testing of the algorithms and also provide comparisons with results obtained from simulations run on the Dragon network simulator. The conclusions of this research are presented in Chapter 11, and listings of the source code of the modules have been included in the appendices.

Chapter 2

The BNR/NT Network

The Bell-Northern Research (BNR) and Northern Telecom (NT) computer network is being used as the test bed for new ideas developed under the Pegasus network diagnostic project sponsored by the Telecommunications Research Institute of Ontario (TRIO). This chapter provides a brief summary of the architecture and operation of the BNR/NT network.

2.1 Network Architecture

The BNR/NT network contains approximately 40,000 devices interconnected by a complex mesh network covering all continents except Antarctica. Organized as a hierarchy, Local Area Networks (LANs) interconnect at the city level to form Metropolitan Area Networks (MANs), and these in turn link together to form a Wide Area Network (WAN).

At the lowest level of the hierarchy is the so-called "workgroup" [3], which was defined to refer to the individual clients of a specific LAN. Most of the traffic in the network occurs between sites within the same workgroup. All members of a single workgroup are located in close proximity to each other, generally within the same building. All communication between workgroup nodes takes place using unshielded twisted pair Ethernet lines, which may span a maximum distance of approximately 100 metres.

Each workstation of a workgroup is connected to a primary hub. A hub has 12 ports, one of which is reserved for diagnostic or emergency use. If

there are more than 11 clients in a workgroup, the hubs are cascaded and primary hubs are connected to a secondary hub, which may also provide connectivity to file servers used by the workgroup. The Ethernet workgroup consisting of workstations, file servers, and primary and secondary hubs comprise a single LAN. The secondary hubs also provide communication links with the network beyond the workgroup LAN.

Secondary hubs are connected to bridges, which are two-port devices linking secondary hubs with fanout devices, which form the LAN backbone. Fanout devices contain 8 or 16 ports and may be connected hierarchically to permit up to 256 backbone connections. The LAN backbone environment connects major devices such as file servers, directory servers, mainframes, and routers. Macintosh workstations are also connected to the BNR/NT network via fastpath devices where appletalk protocols piggyback on the Ethernet packets (appletalk protocol is carried in the data portion of the packets). The basic architecture of the BNR/NT network is illustrated in Figure 2.1.

Routers are interconnected via serial lines having transmission speeds ranging from 56 Kb to 1.544 Mb (T1 links). Typically there are two routers per backbone, and some of the larger sites, such as Carling, Meriline Court, Corkstown, etc. have several routers interconnected for redundancy in case one of the routers fails. Each router is typically connected to two LANs and three serial lines. If the quality of a line falls below a tolerable limit, the router will consider that line to be inoperable. If a connection between two routers fails, the affected router will attempt to find another path to the other router via one or more other routers. Asymmetry in the network occurs when a line becomes inoperable in one direction only, but is still operating in the opposite direction [8].

Figure 2.1.
Typical connectivity of devices in the BNR/NT network.

Routers also perform load balancing on the lines. Internal router tables, specific to each router, determine the next hop to be taken by packets traveling to a particular destination. These tables, maintained by each router, are constructed by proprietary algorithms which consider the capacity, cost, and current performance of their lines when determining the optimal route. This results in constant dynamic rerouting. Since packets can exist in the network for a limited number of hops and for a finite period of time before they are destroyed, dynamic routing may lead to data loss. Fault diagnosis is further complicated by the fact that it is very difficult to determine which route was used by the packets sent from one device to another at any given time [8]. The routing algorithms permit packets to be sent by different routes between a common origin and destination, so that the path followed may not be the "shortest" path, either in terms of the number of hops or the actual physical distance. The ability to be able to trace the most likely path taken by the packets would therefore greatly benefit the diagnostic methods being used at the present time.

2.2 Diagnostic Support

The entire BNR/NT network is monitored from a single site located in the Carling lab in Ottawa. The monitoring system, called Magnet, consists of a static topological representation of the network components. The information is organized as a hierarchy according to geographical regions. At the top level, only the most important routers and the intercontinental connections are shown. Progressively lower levels of the hierarchy show the connections between cities, sites within a city, and devices within a site.

Individual devices and lines between them are represented by icons, the colour of which indicates the status of the device or line. The device status data is sent to the network operations centre by remote data collectors (DCDs), special devices which monitor the status of other important devices in the network. Monitoring is accomplished by periodically sending special packets (an SNMP or PING request) to the device currently being checked. If the device receiving the request is operating properly, it will send a reply back to the data collector, which will then cycle through the list of all devices which it must monitor. If the data collector fails to receive a reply from a specific device, it will send an SNMP or PING request to that device in rapid succession, and if it fails to receive a reply after a specific number of retries, the data collector will send a status message to the network operations center.

This monitoring activity of the data collectors thus adds to the traffic already present in the network. The amount of additional traffic generated depends on the frequency at which the various devices are polled, the number of data collectors which are monitoring the network, and the number of hops that SNMP or PING requests take between data collectors and the devices they monitor. One extreme of network monitoring occurs when a large number of data collectors each frequently poll a small number of devices so that the network is flooded with a large volume of monitoring traffic and failed devices are detected immediately. The other extreme occurs when a low volume of monitoring traffic is generated so that a long period of time may elapse between the failure of a device and the reporting of this fact to the network operations center. A balance must therefore be struck between these two extremes. This involves

- a) optimizing the number of DCDs needed to monitor the network, and
- b) finding the optimal location of the DCDs to provide adequate monitoring of the network while minimizing the traffic that they generate.

For historical reasons, the BNR/NT network has 8 data collectors, all located in the Ottawa area. In some cases, this means that a large number of hops must be made by the monitoring packets. If a line between the data collector and the devices it monitors is broken, SNMP or PING requests or replies may be lost and the data collector will report that it has had no contact with a particular device or a group of devices. In this way, faults in parts of the network between the data collector and its monitored devices interfere with the correct assessment of the status of the network, and diagnostic tools such as BANES and RUNES, which rely on the status messages generated by the data collectors, become confused [8]. Additional confusion is caused by the dynamic rerouting around faults in order to restore network connectivity. Since the probability of incorrect diagnoses increases with the length of the path between data collectors and its monitored devices as well as with the growing uncertainty of paths actually being used, it is desirable to keep paths between data collectors and the devices being monitored as short as possible. In other words, the optimal location of data collectors within the network must be determined, and a means for determining the probable paths being used must exist.

The topology of other networks may change more frequently, and the optimal location of data collectors must be calculated more often. The difficulties associated with fault management in future communication networks consisting of large numbers of highly mobile nodes will increase by several orders of magnitude, and the ability to rapidly calculate the locations of optimal data collector sites will become extremely important.

2.3 Network Faults

There are a large number of faults which may occur in the network. Some faults can be easily diagnosed, while others may require the joint effort of several experts. As discussed by Iqneibi [22], the factors which make network fault diagnosis challenging are the large network size and complex topology, large volume of status messages generated by the data collectors, constant changes in the network topology, transmission of status messages over the same network which is being monitored, interaction between multiple faults, and difficulties in reproducing faults due to dynamic routing and error-correcting protocols [22]. The various faults which may occur have been summarized by Iqneibi [22] and Dussault [12]. The three main categories of faults are caused by hardware failure, errors in software, and human operator error.

Generally, something is almost always broken in the network. Routine faults occur several times each day, and are usually correctly diagnosed and repaired within a short period of time. Severe faults occur less frequently, usually every few days, and may require several days to be resolved. Interaction between faults further complicates matters because symptoms of some faults may be masked by symptoms of other faults. Furthermore, a single fault may generate thousands of status messages, so that the network manager is overwhelmed by the sheer volume of messages.

Many tools are available to assist with the detection and diagnosis of faults, and new tools are under development. Ideally, the diagnostic system should be fully automated so that minimal intervention would be required by the human operators. Such an automated diagnostic system is the ultimate objective of the Pegasus project.

Chapter 3

The Pegasus Project

3.1 Objectives

The methods used at the present time for network monitoring and diagnosis generally use a single approach. Each approach has certain strengths and weakness, and the next logical step is to use several methods cooperating with each other to solve a common goal. The objective of the Pegasus research project is to develop a multi-paradigm system which would automatically monitor, diagnose, and possibly correct faults occurring in wide area computer networks, and thereby improve the performance of existing networks and the potential performance of future networks [28].

3.2 General System Architecture

The Pegasus diagnostic system is being developed for a UNIX environment and is intended to run on a Sparc 2 workstation. Most of the computer programs have been written in standard C, but some modules have been written in Pascal and C++. The Pegasus system uses a blackboard architecture through which the various processes communicate. The present system consists of the blackboard, network simulator, and various knowledge sources.

The blackboard is a means by which the various processes communicate with each other. It performs the functions of a central database and

provides access control functions for the individual processes. It uses shared memory and semaphores to control access to the various areas, which include status messages, requests, replies, problems, and hypotheses [22]. Integral to the blackboard is a special dynamic database, whose schema can be modified during execution and whose record structures can simultaneously have different formats as required by the knowledge sources accessing it [37].

Attached to the blackboard are the various modules of the Pegasus system. These include Dragon, the network simulator, and various knowledge sources. Using this modular architecture, each component can be developed and tested independently, and once operational, it can be activated only as needed. The architecture used is flexible since any number of processes can be connected to the blackboard. Little or no modification of existing processes is required as new modules are developed, because all communication takes place via the blackboard rather than via direct message passing.

3.3 Dragon Simulator

The Dragon simulator enables very large, busy, datagram WANs to be simulated [7]. It is based on the actual topology of the BNR/NT computer network and uses operational traffic data obtained from the real network. Many features of the real network, such as PING and SNMP network protocols, regular polling of network devices by data collectors, random failure and repair of devices, and dynamic rerouting, are supported by the Dragon simulator, but performed much faster than in real time and without any actual transfer of data. A user interface supports additional features which permit users to deliberately introduce faults into the network in order to observe the

behaviour of the network and the various diagnostic agents. The user interface also provides direct control over the generation of traffic by workstations, the error rates of serial lines, and specific interrogation of any device by PING or SNMP protocols. The simulator runs approximately 20 times faster than real time, which permits comprehensive network reliability analysis. Dragon has been designed to run the equivalent of 25,000 years in simulation time, and can simulate up to 32,000 devices [7].

The simulator has three ways in which the operator may describe the network topology. First, the simulator can read text files, *nm2.no* and *nm2.ls*, which list the network topology and the location of data collectors, respectively. These files can be modified to examine different network topologies and different locations of data collectors. They are in the same format as the BNR/NT network topology files, which means that the Pegasus system can be ported into the real-world environment simply by replacing the Dragon simulator by the actual BNR/NT network.

The second method of specifying the network topology is to set up a list in the simulator code, which can be merged with the file or take the place of the file. The third method is to add or subtract devices of the network while the simulator is running [7].

The Dragon simulator is a vital component of the Pegasus diagnostic system, because it facilitates the thorough testing of individual modules before they are ported into a production environment.

3.4 *BANES* (Basic Analysis in Nested Evidence Spaces)

Knowledge sources are software modules of the Pegasus system which gather knowledge about the network being monitored in order to diagnose network faults. The knowledge gathered may be used by the module which obtained it, or it may be passed to other modules for further processing.

BANES was the first diagnostic method to be used in the Pegasus system [8]. *BANES* examines the status messages generated by the data collectors and uses a probabilistic paradigm to propagate beliefs in an internal structure called the nested evidence space [8]. The assumption that a device is broken is propagated from devices which are furthest from the data collectors, i.e. workstations, toward devices which are closest to the data collectors, i.e. bridges and routers.

The main advantage of *BANES* is that the time complexity of computations is linear with respect to the number of devices being monitored. This makes it very fast and the method can be used on very large networks. Unlike human operators, the performance of *BANES* improves as the size of the network increases. Tests carried out on the actual BNR/NT network showed that the accuracy rate of fault diagnosis using *BANES* was greater than 99% and on simulated WANs the accuracy rate was 99.93% [9].

3.5 *END* (Expert Network Diagnostician)

The main disadvantage of *BANES* is that the knowledge source is not proactive; it cannot perform live queries to obtain additional information or

confirm hypotheses. The only information available to *BANES* is the network topology and the status messages generated by the data collectors.

The *END* knowledge source uses expert systems technology for diagnosing network faults [12]. Rule-based systems were among the first of the artificial intelligence (AI) techniques to be implemented, and are generally well understood. Knowledge obtained from human experts is represented by rules, and the computer follows these rules in formulating a hypothesis. Rule-based systems may request additional information if the rules alone are not sufficient to reach a satisfactory conclusion.

In the case of the *END* knowledge source, several network operators were consulted to establish how they diagnosed network faults, and the procedures were translated into rules recorded using a COSMIC CLIPS shell [12]. Subsequent testing and fine-tuning of the expert system was required to ensure that the network diagnosis was being performed correctly.

The main task of *END* is to verify hypotheses formulated by other diagnostic methods such as *BANES*. Unlike *BANES*, which is completely passive, *END* can send SNMP or PING requests to network devices if it needs more information. This means that additional monitoring traffic is being generated in order to diagnose a fault, which is not desirable, since the capacity of the network has already been reduced due to a failure of one of its components.

Although *END* is very good at solving a specific class of problems, this strength is also, paradoxically, its weakness. It is very difficult to develop a general rule-based system to diagnose all faults, because the interactions between the various rules needed become extremely complex. For that reason, Dussault concluded that it would be better to have several small expert

systems, each diagnosing a specific class of problems, rather than using a single general expert system [12].

In the prototype that was implemented, *END* used the probable path between two nodes to verify the proper operation of all devices and lines along that path. With dynamic routing, determination of this probable path becomes difficult, as already discussed. If another knowledge source was to provide a list of candidate paths which should be examined to locate a fault, the diagnostic power of *END* would increase considerably. The knowledge source *NPATH*, developed as part of this thesis, provides such a list of candidate paths and thus extends the capabilities of *END*.

3.6 LEAPR (Line Element Anomaly Pattern Recognition)

Modern computer networks have high reliability due to built-in redundancy and automatic recovery capabilities. Unfortunately, this feature may also hide gradual degradation in the performance of the network until it reaches an unacceptably low level at which point a large number of users experience communication difficulties over the network. Faults which cause these conditions may be difficult to detect by knowledge sources such as *BANES* and *END*.

The *LEAPR* knowledge source was developed to detect gradual degradation in the performance of the network before it becomes critical [35]. It is based on a back propagation neural network which uses traffic statistics of each line of a specific router as input and produces a hypothesis about the condition of a router. The assumptions are that differences between "normal" and "abnormal" traffic patterns can be detected and that abnormal traffic

patterns around a router are a symptom of a faulty router. A hypothesis formulated by *LEAPR* activates another knowledge source, such as *END*, to carry out further investigations and either confirm or refute the hypothesis.

The neural network is trained to classify traffic into normal and abnormal by analyzing examples of each type of traffic previously classified by communication network experts. The main problem in such training is that the traffic is highly variable since it depends on the time of day and day of the week.

3.7 Other Knowledge Sources

The previous sections briefly describe the knowledge sources which have already been implemented. Research is also in progress to develop other knowledge sources to further extend the capabilities of the Pegasus system.

Abnormal router behaviour as a symptom of network faults is being studied by Farrell [13]. Traffic anomalies around routers are examined and hypotheses regarding probable malfunctions of devices are formulated. The routers on the most probable paths are examined in decreasing order of importance, and the algorithm therefore needs to know the probable paths along which packets are being sent when problems are reported, and also the order of importance of the various routers along those paths. This information is provided by *NPATH*, a knowledge source which provides a list of likely paths, ranked in increasing path length, connecting two devices, and a list of critical nodes and arcs, ranked in decreasing order of importance.

Zary is investigating the use of case-based reasoning in the diagnosis of network faults [39]. Important parameters describing the network when a

specific fault occurred are classified and stored in a database. When new problems in the network are detected, the library of previous cases is examined, and hypotheses are formulated on the basis of similarities which may exist between the current status of the network and symptoms associated with faults recorded at an earlier time.

As the number of knowledge sources used to diagnose network faults increases, coordination of the individual modules to cooperate in solving a common goal will become increasingly important. Such coordination will be undertaken by a high-level master knowledge source, whose function will be to activate various diagnostic methods as required, resolve contradictory hypotheses formulated by different knowledge sources, and prioritize problems so that the most important network devices are investigated first. The importance of such a master knowledge source in improving the efficiency of the Pegasus system is obvious, but to date, it has not yet been implemented. Its development will become imperative as the complexity of the Pegasus system increases.

Chapter 4

Optimization Techniques

During World War II, British military planners asked scientists and engineers to develop methods by which mathematics and the scientific method could be used to solve various problems associated with military operations. The field of applied mathematics which resulted from this effort came to be known as operations research. Today, this term is applied to a much wider field of engineering and mathematics which seeks to determine how best to design and operate a system or a process. In general, this involves some type of optimization, and operations research is therefore synonymous with optimization techniques.

This chapter presents a brief review of relevant operations research techniques, namely the calculation of the shortest path and n shortest paths, and the location of medians in a general network.

4.1 Basic Definitions

Many important optimization problems can best be analyzed by using a graphical or network representation. Shortest path, maximum flow, project scheduling, minimum spanning tree, and resource allocation problems are just a few examples for which efficient solution procedures exist. This chapter begins with a brief review of some of the basic terms used to describe graphs and networks.

A *graph* consists of two sets of symbols: nodes and arcs [36]. A *node*, also called a *vertex*, is a point in the graph which generally represents a physical entity such as a city, facility, or device, or a task when graphs are used to solve project scheduling and task allocation problems. An *arc*, also called an *edge* or *link*, consists of an ordered pair of vertices, and represents a possible direction of motion which may occur between vertices.

Arcs usually have an attribute of length or capacity associated with them. The *length* of an arc is a measure of time required to travel from the vertex at one end to the vertex at the other end. It may be as simple as the physical distance, or a much more complex function of several parameters such as the physical distance, importance, installation cost, operating cost, and capacity of the arc. The *capacity* of the arc is generally a measure of the number of items which may traverse the arc at the same time, and is used mainly in network flow problems.

A *chain* is a sequence of arcs such that every arc has exactly one vertex in common with the previous arc. A *cycle* or a *loop* is a chain connecting a node back to itself without retracing any arcs. A *path* is a chain in which the terminal node of each arc is identical to the initial node of the next arc [36]. The *shortest path* is a minimum-length path between two nodes.

An arc which can be traversed in both directions is said to be *undirected*, but if it can be traversed in only one direction, it is said to be *directed*. Graphs may contain both directed and undirected arcs. A *weighted graph* is a graph with lengths associated with the arcs. A graph where every node is connected to every other node is a *fully-connected graph*. A *planar graph* is a graph that can be embedded in the plane such that arcs do not cross, and represents a special case of general graphs. A *tree* is another special type of graph which does not contain any cycles.

In facilities location problems, certain nodes of a graph have special meaning. A *center* is a node selected so that the distance in the graph from it to the most distant vertex is as small as possible [17, 18, 19, 33]. A *median* is a vertex selected so that the total distance from it to each vertex in the graph is as small as possible [18, 19, 33]. *Multiple median* location problems are concerned with finding multiple optimal medians in a graph, i.e., a set of nodes, the medians, selected so that the sum of all distances between each non-median node and its nearest median is minimized [33].

4.2 Shortest Path Problem

The shortest path problem is one of the most well-studied topics in operations research. It is an important problem by itself, since it has many practical applications in the real world, and it is also important because it is a subproblem of many other optimization techniques.

Dreyfus [11] presents an excellent comparison of shortest path algorithms on the basis of theoretical time bounds. Hung and Divoky [21] report on the computational efficiency of five shortest path algorithms implemented in Fortran on a Burroughs 6800 computer. Mondou et al. [27] compared several shortest path algorithms and measured the benefits of using the C programming language. The running times for 8 algorithms were compared, and various implementation issues were discussed [27]. Several papers have been written on improvements to existing algorithms, and specific application of shortest path algorithms in various fields of engineering. The papers by Frederickson [15], Frederickson and Janardan [16], Sherali [32], Chen and Chin [4], and Rosen et al. [31] are typical examples.

The computationally most efficient procedure was described first by Dijkstra [10] in 1959. The algorithm requires that all arcs of the network have non-negative arc lengths and proceeds in the following manner. Let node 1 be the node where the path is to begin. Find the node that is closest to node 1. Then find the second closest node to node 1, and continue in this manner until the destination has been reached. The essence of the algorithm is that a list is maintained of candidate nodes which are to be examined in order to determine whether the destination of the path has been reached. The manner in which nodes are inserted, selected, and removed from such a list affects the manner in which the graph is searched to find the shortest route.

Generally speaking, the shortest path algorithms can be divided into two classes: (i) methods applied to graphs with non-negative arcs that always remove from the candidate list the node which has the minimum label (shortest distance from the origin); (ii) algorithms which apply to networks of arbitrary arc lengths but without negative cycles and that only consider the position of a node in the candidate list. The first group of algorithms is known as *label-setting*, and the second group as *label-correcting* [27].

A priority queue is an abstract data structure used to maintain a collection of labeled items. The items are sorted in order. The most common implementation of priority queues is as arrays or linked lists. The operations required to insert or delete nodes from priority queues implemented in this manner are a linear function of the number of nodes in the priority queue. In the worst case, items will be added at a much faster rate than they will be removed, so that a maximum of $n(n-1)/2$ additions and $n(n-1)$ comparisons will be required [11]. This implies an $O(n^2)$ time complexity of algorithms using them, where n is the number of nodes in the graph. The idea of using a

priority queue to compute the shortest path between two nodes is credited to Dijkstra, and hence his original algorithm had a time complexity of $O(n^2)$.

A significantly more efficient implementation of priority queues is obtained by using bucket structures or heaps [27]. In bucket structures, nodes which have the same label value are grouped together in what are called buckets. Instead of sequentially searching all items of the candidate list in order to find the correct position where a new candidate node should be inserted, the search algorithm merely looks for the appropriate bucket.

A further improvement is to use heaps to store the lists of candidate nodes. A heap-ordered tree is a binary tree where each node corresponds to one item of a priority queue and the value label of each node is greater than or equal to the value label of its parent node. Heaps are slightly more complex to implement and manipulate than linked lists, because each time that an item is added to or removed from a heap, the data structure must be reorganized to maintain the heap property. In contrast, linked lists require a longer time to find the correct insertion point, but no reorganization of the structure is needed once the insertion point is found. Variations of heaps include d -heaps, where each node has at most d children and the time complexity for inserting k items is $O(\log_d k)$, and Fibonacci heaps, where the number of children varies according to the Fibonacci sequence [27]. Such implementations are sophisticated, and most algorithms implementing priority queues as heaps use only binary trees. In such implementations, the time complexity of Dijkstra's algorithm is improved to $O(n \log n)$, which appears to be the fastest implementation of the shortest path for general graphs [27]. In planar graphs, the most efficient algorithms for finding the shortest path have a time complexity of $O(n\sqrt{\log n})$ [15], but planar graphs are very special cases of general graphs and not applicable to communication networks.

4.3 Shortest Paths Between All Pairs of Nodes

In some applications, the shortest paths between all pairs of nodes must be found. As with the single-pair shortest path problem, considerable research has been undertaken to find the best all-pairs shortest path algorithm.

The naive approach is to repeatedly use Dijkstra's algorithm, in which case the time complexity is $O(n^3)$ or $O(n^2 \log n)$, depending on how the priority queue is implemented. A survey of various all-pairs shortest path algorithms is presented by Dreyfus [11] and Moffat and Takaoka [26]. Many of the specialized algorithms are complex and generally involve sophisticated record-keeping. In the algorithm proposed by Floyd [14], optimal paths are produced by inserting nodes into more direct paths. Starting with an $n \times n$ matrix of direct distances, n matrices are constructed sequentially. The k^{th} such matrix can be interpreted as giving the lengths of the shortest allowable paths between all node pairs where only paths with intermediate nodes belonging to the set of nodes 1 through k are allowed. Similarly, Dantzig's [6] scheme involves the construction of successive matrices of increasing size based on the number of arcs in the shortest paths. Both methods are much more difficult to program than the repeated application of a single-pair shortest path algorithm. Actual computational experiments indicate that computing optimal paths between all pairs of nodes by n applications of Dijkstra's algorithm requires approximately one and one-half times longer than the application of one of the algorithms specifically designed to solve the all-pairs problem [11].

4.4 Calculation of the n^{th} Shortest Path

It is sometimes desirable to calculate the n^{th} shortest path between two pairs of nodes. Dreyfus [11] and Pollack [30] present a survey of literature on this subject.

If n is small, i.e. if only the second-shortest or third-shortest path is desired, the simplest algorithm is presented by Pollack [29]. Given the shortest route consisting of x arcs from the origin to the destination, the length of each arc is cut, one arc at a time, and the shortest path problem is recalculated for each case. The shortest of the paths calculated in this manner is the second-shortest path. To find the third-shortest path, arcs on the shortest and second-shortest paths must be cut. If the second-shortest path contains y arcs, the maximum number of shortest-path problems which must then be solved is xy . As this method depends on the network topology, no general estimate can be made of the time complexity of the algorithm. Although this method is very efficient for finding the n^{th} shortest path if n is small, it becomes computationally overburdening as n increases [29].

The algorithm of Hoffman and Pavley [20] is a general method which can be used to find the n^{th} shortest path, regardless of the size of n . It starts by calculating the shortest path through a graph and then examines all paths which can be generated by following the shortest path from the origin to an arbitrary node, deviating to a neighbouring node, and following the shortest path from the deviation to the destination. The shortest of these perturbations will be the second-shortest path. To find the third-shortest path, all deviations from the shortest and second-shortest paths are examined, and the shortest of these is the third-shortest path. To find n shortest paths, all possible perturbations from $(n-1)$ paths must be examined. In this way,

successively longer paths are generated. The disadvantage is that all deviations from the current path must be examined before the next shortest path can be determined. If the average node has x outgoing arcs and the average shortest path contains y arcs, the average problem is solved in approximately xy additions and comparisons beyond those required for the solution of the shortest path problem. The algorithm is therefore very sensitive to the complexity of the interconnections between nodes.

A third algorithm was presented by Bellman and Kalaba [1]. The algorithm consists of finding the shortest and second-shortest paths from node i to a specified destination node N . If the route from i to N is to be the second shortest, then regardless of the path segment used from the origin to i , the continuation must be either the shortest route or the second-shortest route. The paths through all the nodes of the network are therefore examined. The algorithm proposed by Bellman and Kalaba appears to be more difficult to implement than the Hoffman and Pavley algorithm. Furthermore, the n^{th} shortest path must be computed for all of the nodes in the network by the Bellman and Kalaba method, which makes it considerably slower than Hoffman and Pavley's algorithm if the shortest paths between a specific pair of nodes is desired. If the n shortest paths between all pairs of nodes are desired, the two methods are equivalent [11].

4.5 Location of Medians in a Network

Network location problems occur when special facilities must be located on a network. The network in question may be a road network, power grid, communication network, or any other network where facilities may be

represented by nodes and the spatial or temporal relationships between them may be represented by arcs. The benefit which may be derived from the optimal placement of the facilities is described by an equation referred to as the objective function, and the task of the network designer is to minimize or maximize the objective function subject to certain constraints which may be imposed on the network. In general, the objective function in location problems is the cost of transporting goods through the network, and the ideal location of the required facilities is such that the cost function is minimized.

Since the facilities location problems are so important to our society, considerable research has been done to find the best way of solving these problems. Tansel et al. [33] and [34] and Handler and Mirchandani [19] provide excellent surveys of literature. They discuss several location problems in great detail and provide a large number of references in their papers. Zanakis et al. [38] examine different types of heuristic methods and applications to solving various problems. The authors also present extensive references.

Facilities location problems may be divided into two broad categories locating centers and locating medians. As discussed in a previous section, a center is a node chosen so that the distance from it to the most distant node is as small as possible. In contrast, a median is a node selected so that the total distance from it to each node in the graph is as small as possible. Location problems in communication networks generally involve the location of one or more special devices such as switches, file servers, mainframes, data collectors, etc. Such devices can be represented by medians in the network, since they must be accessible to a large number of nodes subject to the constraint that the sum of distances to all nodes serviced by these special devices is

minimized. The following discussion will therefore deal only with this aspect of the facilities location problem.

Hakimi [18] was the first to examine this problem with regard to locating a single switching center in a communication network. He showed that there must exist an optimal location which is a node, and introduced a simple algorithm for determining it [18]. Hakimi introduced the concept of the distance matrix, which is a matrix containing the shortest distances between every pair of nodes in the graph. His algorithm for finding the single median of a graph consisted of calculating and comparing the sums of entries in each row of the distance matrix (row sums). The node which has the minimum row sum is the median of the graph. Most of the work is required in finding the shortest paths between every pair of nodes, which has a time complexity of $O(n^3)$ or $O(n^2 \log n)$, depending on which shortest path algorithm is used, and this is followed by $O(n^2)$ operations to find the single median [34].

Numerous authors have taken advantage of the underlying nature of the network in order to develop more efficient algorithms. For tree networks, Goldman developed an $O(n)$ algorithm for finding a single median [17]. The algorithm reduces the search to successively smaller subtrees until a median is found. Unfortunately, communication networks are more complex than a simple tree, and such a method for finding a median is not applicable.

Although solving the single median problem is relatively easy and can be achieved in polynomial time, finding multiple medians in a graph is more complex. Kariv and Hakimi showed that the multiple median problem is NP-hard [23], i.e. no polynomial-time algorithm is known to exist to solve such a problem. Bienstock and Marcotte studied a related problem called path bundling, in which the total number of cables in a network must be

minimized [2], and this problem is similar to the multiple median problem. The authors present an extensive analysis and proofs showing why this and related problems are NP-hard [2].

The methods for solving the multiple median (m -median) problem can be classified into the following categories: enumeration, graph theoretic, primal-based mathematical programming, dual-based mathematical programming, and heuristic methods [19]. Enumerating all possible solutions to determine the optimal one is a naive approach and for large networks it is not practical due to the computations involved. However, if the network is small or if only a single median must be found, it is a feasible approach. Graph theoretic methods generally take advantage of the network structure, and may be efficient if the network is a tree. Mathematical programming approaches, either primal-based or dual-based, use an integer programming formulation for the m -median problem. The network topology and constraints are described by a set of equations which are solved by commercially-available integer programming computer routines. Such methods have proved successful for general networks. The limiting factors are the size of the network and the equations describing it, since one equation is required for each of the constraints in the problem. Since the formulation of such problems becomes complex very quickly as the network size increases, integer programming techniques are not particularly suitable for solving very large networks [19].

This leaves heuristic methods. Frequently, a "good" solution rather than an "optimal" solution is required, and heuristic methods have therefore proved to be relatively successful in solving the m -median problem. In some cases, heuristic methods are used to generate an initial solution for more precise methods in order to obtain faster convergence. The most important of

these methods have been described in literature as node partitioning, myopic or greedy algorithms, node substitution, and genetic algorithms [19].

Node partitioning was probably one of the first heuristics to be studied for the m -median problem. The algorithm developed by Maranzana [25] starts by arbitrarily selecting m nodes in the network as candidate medians. Each non-median node is then assigned to its closest candidate median so that the network is partitioned into m domains. The median of each domain is then calculated, and the original candidate medians are replaced by the new medians. The procedure is repeated until the median set does not change. Maranzana showed that this algorithm produces a monotonically nonincreasing sequence of objective function values and that the solution may converge to a nonoptimal value. Furthermore, the heuristic may cycle. Very limited computational work has been reported on the performance of this algorithm, but Maranzana claimed that the algorithm performed well on sample problems [25]. Maranzana's algorithm was used in the *MEDIAN* knowledge source to find the optimal location of data collectors in the BNR/NT network.

The myopic or greedy algorithm recursively adds medians to the solution starting with one median and then sequentially adds medians until m have been added. The first median to be selected is the absolute median of the network. The next median to be entered into the solution is any nonselected node that maximizes the improvement in the objective function value. The greedy algorithm was first suggested by Kuehn and Hamburger [24], and a more recent study was carried out by Cornuejols, Fisher, and Nemhauser [5], who also presented extensive computational results on the performance of this heuristic.

Node substitution starts by selecting any m nodes as candidate medians. Nonmedian nodes are then examined in turn to see whether substituting a nonmedian node for a current candidate median improves the objective function. The algorithm stops when no further improvement can be obtained. This algorithm was first suggested by Kuehn and Hamburger [24] and subsequently studied by Cornuejols et al. [5] and other authors [19].

Genetic algorithms use a paradigm of evolution and natural selection in arriving at an optimal solution. The candidate solutions are "chromosome strings" usually represented as a sequence of characters or numbers. The algorithm must start with a large number of candidate solutions (the genetic pool), and by using operations such as reproduction, crossover, and mutation, the algorithm spawns new solutions from the old generation of candidate solutions. The "fitter" solutions will pass their genetic material to subsequent generations, while those solutions which are not fit will become extinct and the information contained in their chromosome strings will be removed from the genetic pool. The probability of survival is proportional to fitness. The advantages of these types of algorithms are that they are simple to implement, they work with complete solutions, and they can be applied to any problem whose solution can be represented as a string. Their strength lies in solving problems which are too large for the efficient application of other algorithms. However, their disadvantage is that a very large number of candidate solutions (generations) must be examined and there is no guarantee that the solution will converge to the optimal value.

Due to the complexity of the m -median problem, heuristic methods have been successfully used for finding a suboptimal solution for problems where finding the true optimum is impractical. Their strength becomes particularly apparent as networks become larger and more complex.

4.6 Optimization of Computer Networks

The preceding sections presented a general summary of optimization techniques used for finding the shortest paths through a network and for locating single or multiple medians in general networks. The algorithms discussed are general and may be applied to any network, regardless of whether the network is used to transmit packets between computers or trucks between cities.

Designers of computer networks use operations research techniques in all aspects of the design process. For example, the length of cable and number of devices needed to provide a specific level of service must be minimized, location of devices such as file servers, data collectors, and mainframes must be optimized, time required to transfer information between devices must be minimized, etc. Operations research is a large field and it is not possible to provide a review of the various typical applications in the design and diagnosis of communication networks. This chapter was intended to provide the reader with the background necessary to understand the concepts and implementation of the operations research techniques in solving two problems vital for the Pegasus diagnostic systems – the n shortest paths and the optimal location of data collectors in the BNR/NT network.

Chapter 5

Data Representation and Support Functions

5.1 Programming Environment

Two main software modules have been written to illustrate the concepts described in this thesis. These modules are *MEDIAN*, which calculates the optimal assignment of n data collectors in an arbitrary network, and *NPATH*, which finds the n shortest paths. A third module, *NC*, used for the compression of several nodes into a smaller number of supernodes was initially developed as a stand-alone program, but was subsequently incorporated into *NPATH* as a support function. The algorithms and a description of how the software modules are used will be described in detail in subsequent chapters. This chapter describes the basic methods of data representation and manipulation and some functions which are common to the various software modules.

The programs were written in the C programming language and were compiled and linked using SUN's C compiler or the Objectworks C++ compiler. Sun 3 and Sparc 2 workstations running the SUN OS Version 4.1.1 UNIX operating system were used for testing.

5.2 Data Representation

The modules which have been developed as part of this thesis were intended to serve as tools which can be used by other knowledge sources comprising the Pegasus network diagnostic program. Classical graph theory concepts have been used to examine the network topology and provide information regarding the optimal paths between two devices of the network and to divide the network into several smaller domains.

The computer network which was used to test the concepts consists of a small subset of a much larger corporate wide area network (WAN) which spans all continents except Antarctica. The actual data network used by Bell Northern Research (BNR) and Northern Telecom (NT) contains approximately 40,000 nodes of which approximately 3,000 are monitored by data collectors. The major components which make up this network include routers, bridges, LANs, and workstations. The model network which was used to test the algorithms consisted of 777 devices all of which were monitored. The size of the model network can be increased by adding random workstations, which, for the purposes of the simulations, can also be monitored by data collectors. The maximum size of network with additional workstations which can be modeled by Dragon, the network simulator, can contain 32,000 nodes. The maximum size of the network which was examined with *MEDIAN* and *NPATH* contained 1262 nodes, the basic 777 nodes plus an additional 485 fictitious workstations. Where arrays were used, the sizes of the arrays were set at 1400 in order to conserve computer memory, but the array sizes can easily be modified as needed.

For the purposes of the graph theory approaches, all devices are treated as general "nodes" and the communications lines between them are

treated as "arcs". A generic "length", which is a measure of the arc's efficiency, is associated with each arc. This efficiency is a complex function of the number of packets that the physical connection can carry, the speed at which the packets may be transmitted, the cost associated with operating that link, the required reliability, and several other factors. The manner in which the nodes and arcs are connected is referred to as the network topology, and is generally stored as an ASCII text file which is accessed by the various diagnostic modules. A simplification has been made which assumes that the network topology is static and can therefore be stored in a file. This is not precisely true because the topology changes constantly as devices are deliberately removed from or added to the network, or as devices randomly fail and get repaired. However, the major components such as routers, LANs, bridges, and the connections between them are not likely to be deliberately frequently changed. The workstations, which are the most volatile nodes, are not monitored in the real network, and the model which represents the real network by a stable topology is valid.

Accessing a text file each time any computation involving the network topology must be made would be too time-consuming and hence impractical for diagnostic purposes, and an internal representation of the network topology is therefore required. If the network is small, the topology of a network consisting of n nodes may be stored as a two-dimensional n by n array where the rows and columns represent the nodes and the cells of the array represent the arcs connecting them. Although this method is acceptable for small networks because the access time for arc lengths is very fast, it is impractical for intermediate and large networks because it is very wasteful in terms of computer memory. The fact that the array is a static data structure further

restricts the application of this approach to networks whose size does not vary appreciably over a period of time.

A second approach is to represent the network topology internally by means of only dynamic data structures such as binary search trees and linked lists. This representation has the advantage that it uses only as much memory as is necessary to describe the network topology, and hence the waste of storing non-existent connections, as is the case with a two-dimensional array, is eliminated. The disadvantage of using dynamic data structures is that programs using them are generally more complex and the risk of having "bugs", especially due to "dangling pointers", is much greater than when using static data structures. Binary search trees can be very efficient as long as they are balanced. The code necessary to keep a tree balanced introduces additional programming complexity and will increase the execution time of the algorithm, which may or may not be important, depending on the application. However, if a binary tree is constructed from data which is already sorted without any attempt at balancing the tree, all advantages of binary search are lost and the problem is reduced to a linear search.

A third approach is to use a hybrid of both static and dynamic data structures in order to utilize the advantages while minimizing the disadvantages of both approaches.

A) Schematic diagram of a sample network.

B) Topology data file for the network shown above:

```

1= e0.bCAR LN: bCARc1 e1.bCAR
2= e1.bCAR LN: e0.bCAR e2.bCAR bCARc2
3= e2.bCAR LN: e1.bCAR bCARc2
4= bCARc1 RT: e0.bCAR bCARc2 bMDHc1 bMERC1
5= bCARc2 RT: e1.bCAR e2.bCAR bCARc1 bMERC1 nTORc1 bRTPc1
6= bMDHc1 RT: bCARc1 bMERC1 nMTVc1
7= bMERC1 RT: bCARc1 bCARc2 bMDHc1 nTORc1
8= nTORc1 RT: bCARc2 bMERC1 nANNc1
9= nMTVc1 RT: bMDHc1 nANNc1 nRCHc1 nMELc1
10= nANNc1 RT: nTORc1 nRCHc1 nMTVc1
11= bRTPc1 RT: bCARc2 nRCHc1
12= nRCHc1 RT: nMTVc1 nANNc1 bRTPc1
13= nMELc1 RT: nMTVc1

```

C) Data file defining the arc lengths for the network shown above:

```

1 2
MDH MTV 20
MTV MEL 10
MTV RCH 3

```

Figure 5.1.

Sample network showing the correct format of topology and arc length data files.

5.3 Internal Representation of Network Topology

The network topology file which is read by the Dragon simulator is very large and contains more detail than is required by the modules using a graph theory approach. A simplified version of this topology file was created by the simulator and was used in all subsequent testing. The format of the topology file was as follows:

```
id_number= device_name device_type: device_1 device_2 ... device_x
```

where *id_number* is an integer identifier, *device_name* and *device_type* are the name and type of the device, respectively, and *device_1* to *device_x* are the names of all devices connected to *device_name*. A sample network consisting of 13 nodes and the corresponding topology and arc length data files is illustrated in Figure 5.1.

Initial prototypes of *NPATH*, *MEDIAN*, and the associated functions stored the network topology only in dynamic data structures. A preprocessor function parsed the topology data file and constructed the internal representation of the topology in two passes. On the first pass, it extracted the device name and the device type and constructed a binary search tree. Each node of the tree contained a field for the device name, which was the key, the device type, a pointer to a linked list which contained the devices connected to this specific device, and pointers to the left and right children. On the second pass, the parser built the linked lists attached to each node of the binary tree, and carried out some rudimentary checks to ensure that each node which was added to the linked lists was present in the binary tree. The nodes of the linked lists contained the name of the device, a field for the generic length of the arc, and a pointer to the next node in the linked list. The name

of the device was stored explicitly in the linked list rather than as a pointer to the node of the tree in order to simplify subsequent operations.

No attempt was made to balance the binary search tree. This resulted in poor performance of the algorithms if the original topology data file was already sorted. Furthermore, algorithms which subsequently processed the topology, for example to find the shortest distance, were cumbersome because string manipulations had to be carried out. The internal representation of the topology was therefore modified, and a hybrid of dynamic and static data structures was used in all subsequent work.

The hybrid representation consisted of three global arrays: *nameArray* which stored the device names, *typeArray* which stored the device types, and *listArray* which stored pointers to linked lists containing the devices connected to the particular device. The organization of these arrays and linked lists is illustrated in Figure 5.2.

During the first pass, the parser constructed a binary search tree of the device names and types as previously, and this information was sorted as it was transferred to the arrays by an in-order tree traversal. On the second pass, the linked lists were constructed as in the previous version, but the nodes of the linked lists now contained only integer fields, one being the array index of the node, and the other being the generic length of the arc.

Although this approach introduced the restriction of fixed array sizes, the performance of the algorithm was faster than with the previous approach. Binary search could be used for finding the array index of a specific node without the increased overhead of constructing a balanced binary search tree. All subsequent functions processing the network topology could then use integer arithmetic rather than character string manipulation, and the danger of inadvertently leaving dangling pointers was greatly reduced.

Internal representation of network topology using arrays and linked lists.

<i>nameArray</i>	<i>typeArray</i>	<i>listArray</i>
0 bCARc1	0 RT	0 ▶ 1 1
1 bCARc2	1 RT	1 ▶ 0 1
2 bMDHc1	2 RT	2 ▶ 0 2
3 bMERC1	3 RT	3 ▶ 0 2
4 bRTPc1	4 RT	4 ▶ 1 2
5 e0.bCAR	5 LN	5 ▶ 0 1
6 e1.bCAR	6 LN	6 ▶ 1 1
7 e2.bCAR	7 LN	7 ▶ 1 1
8 nANNc1	8 RT	8 ▶ 10 2
9 nMELc1	9 RT	9 ▶ 10 10
10 nMTVc1	10 RT	10 ▶ 2 20
11 nRCHc1	11 RT	11 ▶ 4 2
12 nTORc1	12 RT	12 ▶ 1 2
13 ~~~	13	13

Index field of list item contains device connectivity, e.g. e1.bCAR (6) to e2.bCAR (7).

Null pointers ending each linked list not shown for clarity.

Dist field of list item contains arc length, e.g. 1

End of data in arrays is marked by variable *endArray*, set to the next free space, e.g. 13. *NameArray* has been initialized to "~~~", *listArray* to NULL.

Figure 5.2.
Internal representation of the network topology.

5.4 Assigning Arc Lengths

The initial prototypes of the modules based all path length calculations on the number of hops taken between the origin and destination. No specific "lengths" were provided in the network topology, and it was therefore assumed that the length of each arc could be set to 1. This design philosophy resulted in the development of a series of functions which treated the network as an unweighted graph, i.e., a graph in which all arcs and all nodes have the same importance.

Initial testing showed that such an assumption is not valid and it was necessary to incorporate actual arc lengths into the model. In the absence of more accurate determination of the cost associated with each arc, which would translate to the arc "length", the arcs were assigned arbitrary lengths of 3 for any router-router link, and 1 for all other links. This resulted in the generation of paths via LANs rather than by direct router-router connections and corresponded to the paths generated by explorer packets transmitted in the Dragon simulator [7].

Since the lengths of arcs could be easily divided into two sets either of length 1 or 3, a function called *getArcLengths* was written to assign arc lengths after the internal representation of the network topology was created. This function also counted the number of arcs present in the network and the number of nodes which had only a single incident arc in order to provide a rough measure of the interconnectivity of the network.

Although this approximation of arc lengths was at first satisfactory for *NPATH*, simulations carried out with *MEDIAN* required a different definition of arc lengths. *MEDIAN* read a matrix of distances between all pairs of nodes in the network which was generated by the *MGEN* (matrix generator)

preprocessor. Initially *MGEN* assumed that all arc lengths were set to 1 and repeatedly used Dijkstra's shortest path algorithms [10] for unweighted networks to find the shortest distances between every pair of nodes. When it became obvious that a more accurate representation of different arc lengths was required, a different approach was taken.

The information about arc lengths was stored in a separate data file which was read by function *assignArcLengths*. When called by *NPATH*, the arc definition file must be named *topology.len* and must reside in the same directory as *NPATH*. If the file does not exist, *NPATH* bypasses the relevant functions and assigns the default arc lengths as described above. When called by *MGEN*, the user must specify the name of the file where the arc lengths are stored, and the program checks whether the file is present or not.

The first line of the arc definition file contains two integers, the first one being the default length of arcs at the same site and the second being the default length of arcs connecting different sites. For example, the arc definition data file for an unweighted network would only contain two 1's. For weighted networks, these numbers would differ depending on the relative importance of the arcs. A sample data file is shown in Figure 5.1.

The default arc lengths could be modified according to additional entries in the arc definition data file. These entries must be in the format

```
string_1 string_2 length
```

where *string_1* and *string_2* are substrings of node names at either end of the arc, and *length* is an integer length of the arc. The default arc lengths as well as the specific arc length information is read from the data file by function *assignArcLengths*, which calls *defaultArcLengths* and *specialArcLengths* to

perform the actual insertion of arc lengths into the internal topology. A flow chart showing the logic of assigning arc lengths is presented in Figure 5.3.

The names of all devices in the network are examined to determine whether they contain *string_1* or *string_2*, and if a match is made for both nodes at either end of an arc, that arc is assigned the new length. The format thus permits the user to be general, so that, for example, the definition

CAR MER 10

would assign a length of 10 to arcs *bCARc2-bMERc1* and *bCARc1-bMERc1*, while

bCARc2 bMERc1 10

would only assign the length of 10 to the *bCARc2-bMERc1* arc while the *bCARc1-bMERc1* arc would retain its default value.

This approach provided a relatively easy "patch" and gives users relative flexibility in assigning arc lengths. However, it would be highly desirable to read the true arc lengths from the data file which contains the network topology and assign the correct lengths at the time that the internal topology is being created rather than going about it in this round-about fashion. It is assumed that once the true lengths of all arcs are known and stored in the topology data file, these functions will be discarded.

Figure 5.3.
Flow chart showing logic of assigning arc lengths.

5.5 Support Functions

The support functions which are used repeatedly by the various modules can be grouped according to the data structure which they construct and manipulate. These data structures, presented in Appendix A, are binary search trees, linked lists, arrays, and queues.

The functions associated with trees, linked lists, and arrays are used to create and initialize these data structures, enter, find, and print data, transfer information from one structure to another, and delete the data structures. Standard algorithms have been used and the source code listing is provided in Appendix B. The source code for additional general path-related functions is provided in Appendix C.

Queues, especially priority queues, have been used in algorithms which calculate the shortest path and in algorithms which divide the network into several domains. The queue has been implemented as a record with an integer field for the length of the queue and pointers to the head and tail of a linked list which contains the items of the queue. The nodes of the queue are records with three integer fields for the data and a pointer to the next node in the linked list. Standard functions which enqueue and dequeue items, search for specific items in the queue, and delete the queue have been supplemented by functions *priorityNQ* and *sortQ*. As the names suggest, *priorityNQ* inserts an item into the queue so that the linked list remains sorted in increasing order of one of the integer fields (partial path length), and *sortQ* sorts an existing queue in descending order of one of the integer fields (number of nodes assigned to a specific domain).

5.6 Shortest Path Functions

Dijkstra's algorithm [10] was used to calculate the shortest path between two nodes in a network. This algorithm for a breadth-first search was implemented by using arrays *nameArray* and *listArray*, which contained the internal representation of the network topology, *seenArray* which was an array of integers representing the nearest predecessor of the node just examined, and a priority queue which contained the arcs and nodes which should be examined and the order that they should be examined in.

The basic algorithm was implemented as follows:

1. Initialize *seenArray* to indicate that all nodes are marked as "unvisited" (by using -1), initialize the current path length to 0, and initialize all other variables as required.
2. In turn, examine all the neighbours (candidates) of the origin node. If these candidates have not been visited, mark them as "visited" by recording the array index of the origin in the appropriate location of *seenArray*. The origin thus becomes the "nearest predecessor" of the candidate node.
3. If the candidate has not been visited, record the array index of the candidate, the array index of the nearest predecessor, and the cumulative path length from the global origin to the candidate in a priority queue sorted in increasing order of partial path length.
4. Dequeue the first item from the priority queue. If the candidate node is the destination proceed to Step 5, otherwise make the candidate node the local origin and go back to Step 2.
5. The path has been found. The length associated with the destination is the path length. The path proper has been built up in *seenArray* by array

indices of the predecessor node of every visited node, so follow the trail from the destination to the origin and print the path.

The algorithm, shown as a flowchart in Figure 5.4, assumes that the network is not partitioned so that a path exists from the origin to the destination. If the path cannot be found, function *findWeightedPath* returns -1 instead of the path length. Furthermore, any entry in *seenArray* which is not -1 is the index of a node which belongs to one of the partitions, and any entry which is -1 belongs to the other partition. If it is determined that the network is partitioned, this is one way of finding all of the nodes which belong to one of the partitions. The sample network topology contained several smaller embedded networks, and this property was used to identify the smaller partitions which were subsequently physically removed from the topology data file.

Since the algorithm stores predecessor nodes, the path is recorded in reverse order. The user expects to receive the path in a specific order, and for that reason the origin and destination specified by the user are interchanged when they are read by the program. Additional functions were developed to transfer the path from *seenArray* to a queue of arcs, which was then used by *NPATH* when constructing ranked lists of paths, and to calculate the length of any path in such a queue.

The data representation and support functions were the building blocks upon which the more complex algorithms were developed. They have been used by the modules *MEDIAN* and *NPATH* to perform the useful tasks of finding the optimal allocation of domains and the n shortest paths in a network, respectively.

Figure 5.4.
Algorithm used to calculate the shortest path in a weighted graph.

Chapter 6

Optimal Location of Data Collectors

6.1 Selection of the Algorithm

The optimal allocation of data collectors in the network under study consists of solving the minisum (medians) location problem. After reviewing the various algorithms which could be used to solve the multiple median problem, Maranzana's node partitioning algorithm [25] was selected due to its programming simplicity and efficiency. This algorithm for finding n medians in a network consists of the following steps:

1. Select n random nodes in the network as the candidate medians.
2. Assign every node to its closest median. In this way, the entire network is divided into n domains.
3. Within each domain, find the optimal median of that domain and replace the candidate median selected earlier by this optimal median.
4. Repeat the previous steps until the solution converges to a stable value or cycling of the solution is observed.
5. Repeat the previous steps to find several solutions. The best solution will be the optimal or near-optimal solution.

This algorithm can be considered to be the "shotgun approach" because if a large number of iterations is run and random medians are used to initialize the search process for each iteration, there is a high probability that at least one of the iterations will converge to the optimal solution.

6.2 Distance Matrix Calculated by *MGEN*

The median of a domain is the node from which the sum of distances to all other nodes within the domain is minimized. Distances between all pairs of nodes within the domain are recorded in the so-called distance matrix. The row sum of the distance matrix represents the sum of distances from a specific node to all other nodes in the domain, and the node which has the lowest row sum is the median of that domain.

In order to have rapid access to the distances between any two nodes of the network containing n nodes, the distance matrix was implemented as a two-dimensional $n \times n$ array where the index of the rows and columns corresponds to the name of that node contained in *nameArray*. If the network is nonoriented, the length of each arc is the same regardless of the direction in which it is traversed, and the distance matrix is symmetrical about the diagonal. For nonoriented networks it would be possible to store the distance information in a more compact array of size $\frac{1}{2}(n^2 - n)$. The latter approach was not implemented in order to keep the algorithm general so that it could also be applied to oriented networks, i.e. networks where the length of each arc is different depending on the direction in which it is traversed.

As it is unrealistic to expect that the distance matrix for large networks would be calculated on-line, the *MGEN* preprocessor was used to generate the distance matrix and store it as a text file, which can subsequently be read by *MEDIAN*.

MGEN first reads the topology data file and constructs the internal representation of the topology. Initially, the length of each arc was assumed to be 1 since for the purposes of the *MEDIAN* algorithm, the distance between each pair of nodes represented the number of hops that packets must

make when traveling along the most direct route. Subsequent modifications of the *MEDIAN* program required arc lengths other than 1, and *MGEN* had to be modified accordingly. The algorithm used to assign arc lengths has been described in detail in Section 5.4, and a flow chart of the logic used was presented in Figure 5.3.

The distance matrix needs to be calculated only once as it does not change unless the topology of the network changes. Although elegant algorithms exist for finding the shortest paths between all pairs of nodes in a graph, these algorithms are not trivial to implement [11]. In contrast, the repeated execution of Dijkstra's algorithm [10] is relatively trivial to implement, and the overall performance of the algorithm which calculates the shortest distance between every pair of nodes is not significantly slower than the more elegant algorithms. Since the shortest path algorithm had already been developed for another diagnostic module, it was considered more expedient to utilize the functions already implemented rather than implement the more elegant algorithms.

A flow chart of the logic used by the *MGEN* program is shown in Figure 6.1, and the source code listing is provided in Appendix D. The first part of the program deals with opening of files, initialization of the required arrays and variables, parsing of data, and construction of the internal topology. The data files may be entered as command line arguments or the program prompts the user to input the required file names during execution. Limited error checking is carried out and the program aborts if any problem is encountered. This design logic was also used in the other programs developed.

Figure 6.1.
Flow chart of the logic used in the *MGEN* program.

Dijkstra's algorithm was used by function *fillDistArray* to calculate the shortest distance between each pair of nodes. Due to the symmetry of the distance matrix for nonoriented graphs, *MGEN* calculated less than $\frac{1}{2}(n^2 - n)$ shortest paths. In order to further speed up the preprocessor, the distances between every pair of neighbouring nodes were entered directly into the matrix by function *fillAdjDist* without invoking the shortest path function.

Although the complete distance matrix was stored in memory, only the upper half of the matrix above the diagonal was stored as a text file in order to save disk storage space. Since the time required by *MGEN* to generate the distance matrix for a large network (1262 nodes) was in the order of 2 days, the distance matrix file was written to the file in append mode each time a new complete line of the matrix was calculated. This approach would facilitate the restart of the program in the event of a computer crash.

6.3 General Logic of the *MEDIAN* Algorithm

The program for calculating the optimal location of data collectors (*MEDIAN*) was implemented as a stand-alone program which would not be an integral part of the Pegasus system. The program logic is shown as a flow chart in Figure 6.2, and the source code listing is contained in Appendix E.

MEDIAN uses three input data files which are the network topology, distance matrix generated by *MGEN*, and file *nm2.no*, which is the topology file used by the Dragon network simulator [7]. *MEDIAN* generates two output files, one containing a log of the search for the optimal medians, and the other containing a new topology for the Dragon simulator with the revised location of data collectors and domains that they monitor.

Figure 6.2.

Flow chart of the logic used in the *MEDIAN* program.

The program starts by constructing the internal network topology. The variable *endArray* is the index of the next available entry in the node name array, *nameArray*, and represents the number of nodes in the network. It is used to limit manipulation of the arrays to only that portion of the array which is actually being used.

MEDIAN then reads the distance matrix data file. The first line of that file is the *endArray* value recorded when the matrix was generated and must correspond to the *endArray* value of the current network topology. This ensures at least that the matrix being read is of the expected size, but the algorithm cannot guarantee the correctness of the data being read. It does not detect deliberate tampering with the data or accidental reading of the wrong file. The number of integers read per line is compared with the expected number and any deviations trigger an alarm, which may indicate that one line of the data file was deleted or added, or an incomplete line was read. Under normal conditions, the data files will be read without any problems.

Once the data files have been read, the user is asked to input any fixed medians, or a blank line if none exist. As the name implies, fixed medians do not change as the program works toward an optimal solution. The name of the median is followed by an integer code which can be 1 or 0. If the code number is missing, it is assumed to be 0. If the code number is 1, the median is treated as a fixed median throughout all iterations. If the code number is 0, the median which was entered is used as the candidate median for the first iteration only, but may subsequently be replaced. This feature was used when specific medians were known *a priori* or when the domain and approximate location were known but it was necessary to determine the precise location within the domain. In both cases the known or suspected medians were entered and *MEDIAN* was run through a single iteration.

After the fixed medians, if any, have been entered, the user is requested to enter the total number of medians and the number of iterations which should be performed in arriving at an optimal solution. If the total number of medians is the same as the number of fixed medians entered, the program bypasses all optimization functions which iteratively improve the solution and merely changes the topology data file which will be used as input for the Dragon simulator. Similarly, if only one median is to be located, the program merely finds the absolute median of the entire network, updates the Dragon topology file, and exits.

Function *getMedians*, which records the number of fixed medians requested by the user, also requests the user to enter the tolerance on the size differences between any two domains and between the largest and smallest domains. During initial testing of this program on a network containing 1262 nodes, it was observed that one domain might contain 600 nodes while another domain only contained 1 node. The restriction in the differences between the sizes of domains was one way that this problem could be overcome. Furthermore, the candidate medians for the start of each iteration are selected at random. The sum of the domain tolerances is used to initialize (seed) the random number generator to ensure that simulations are repeatable and a mechanism for generating various random sequences exists.

After all data is read, the necessary variables are initialized. The initialization process includes the generation of random medians, one for each variable median requested by the user. The medians are stored in a queue, *medianQueue*, and the value of the *flag* field indicates the type of median – 0 for variable medians and 1 for fixed medians. This queue is passed among the various functions and is updated as required. A second queue, *optMedQ*, contains the global optimal medians. The variable *globalMin* stores the

global minimum value of the sum of distance between the medians and their closest nodes. Variables *globalIter* and *globalSubIter* store the iteration and subiteration numbers where the minimum occurs, respectively.

After all appropriate variables are initialized, the program enters a FOR loop which gets executed as many times as was specified by the user. These are the iterations over which the user has direct control. Inside the FOR loop, a call is made to function *localMedians*, which receives the appropriate control variables and counters, and the initial candidate medians, either randomly generated or specified as being fixed by the user. The function then searches for an optimal solution for this particular iteration, and if the local optimum is better than the global optimum, it updates the optimal median queue and associated variables. Each solution calculated by *localMedians*, called a subiteration, is recorded in the log file. The user has no control over the number of these subiterations, but safeguards have been incorporated into *localMedians* to prevent cycling.

Upon exiting the FOR loop, the best allocation of medians is stored on the optimal median queue. The optimality of the solution depends on the number of iterations which were executed; the higher the number of iterations, the greater the probability that the solution will be the optimal one. The program then assigns each node of the network to the closest median. By default, the size restrictions imposed on the domains by the user are ignored during the final allocation of nodes to their closest median, but such restrictions may be imposed even for the final result.

As a last step, function *modSimFile* modifies the Dragon simulator topology data file. The modification consists of reading the original topology, replacing the old data collector definitions and domains monitored by the data collectors by the new medians and associated domains calculated by this

program, and writing the modified topology to a new data file. The optimal values are then printed, all files are closed and the program stops.

6.4 Function *localMedians*

Function *localMedians*, shown as a flow chart in Figure 6.3, receives an initial guess of the location of the medians and iteratively revises the network domains and the absolute median of each domain. The stopping conditions for this function are:

1. The median queue passed to *localMedians* is empty. No calculations are carried out and control immediately returns to the calling function.
2. Only one variable median should be calculated. No domain balancing, optimization, or subiterations are carried out as the calculations only involve finding the absolute median of the entire network, and there is only one possible answer.
3. The current subiteration is the same as the previous subiteration. The local optimum has been found.
4. A total of five subiterations have been observed which are worse than the previous subiteration. This indicates cycling of the solution and it is possible that the solution will never converge.
5. A total of 50 subiterations have been carried out. This indicates that the solution is converging but very slowly. Although this condition has never been encountered during testing of the program, it was included as a fail-safe mechanism to exit out of an apparently endless loop.

If size restrictions have been placed on the domains into which the network is divided, function *equalRegions* is called to carry out the balancing

of the domains. The number of nodes which belong to each domain is recorded in the median queue, and the queue is sorted in decreasing size of the domains. Function *equalRegions* examines the sizes of the domains and transfers nodes from the source domain to the destination domain so that the size differences between a particular domain and the next largest domain and between the largest and the smallest domains are within the tolerances specified by the user. The nodes which are transferred from one domain to another are those nodes which are farthest from the origin median and closest to the destination median. All nodes are examined and moved in order of their location with respect to the origin and destination medians, i.e., the nodes farthest from the origin median are moved first. Redistribution of nodes continues until the sizes of the domains are within the tolerances specified by the user.

It should be noted that this function is executed only while the program searches for the optimal medians. Once the optimal medians have been found, the size restrictions are ignored and every node is assigned to its closest median. This heuristic was necessary to initially prevent large discrepancies between the sizes of the domains, yet keep nodes together in the final solution if they would otherwise be arbitrarily placed into unrelated domains only to satisfy the domain size restrictions.

The performance of this algorithm and the results obtained are discussed in Chapter 9.

Figure 6.3.
Flow chart of the logic used in the function *localMedians*.

Chapter 7

Calculation of n Shortest Paths (*NPATH*)

7.1 General Description of the Algorithm

The algorithm uses Hoffman and Pavley's perturbation method [20] to find the n shortest paths through a graph. The basic idea of this method consists of finding the shortest path and successively examining all possible loopless paths which can be generated by following the shortest path to some arbitrary node on the shortest path, deviating from the shortest path to a neighbouring node (the perturbation), and finding the shortest path from the perturbation to the destination. All paths generated in this manner, called candidate paths, are recorded and ranked in increasing order of their length. Since the shortest path through the graph is already known, any paths which pass through a perturbation from this shortest path are guaranteed to be as long as or longer than this shortest path. However the rank of a candidate path is not known until all possible perturbations from the candidate path itself have also been examined, because no new path can be shorter than the path currently under investigation. If a shorter path does in fact exist, it will have been recorded during a previous iteration of the algorithm.

A specific example showing all of the candidate paths generated from the shortest and second-shortest paths of the sample graph of Figure 5.1 is illustrated in Figure 7.1. Detailed descriptions of the algorithm and program logic follow.

ORDER IN WHICH ALTERNATE PATHS ARE ADDED TO LIST
(Perturbations from shortest and second-shortest paths)

Rank	Length	Shortest path
1	[17]	e1 bCAR - bCARc2 - nTORc1 - nANNc1 - nMTVc1 - nMELc1
4	[19]	Perturbation paths from <i>e1 bCAR</i> added to list in the following order
2	[18]	e1 bCAR - e0 bCAR - bCARc1 - bCARc2 - nTORc1 - nANNc1 - nMTVc1 - nMELc1
		e1 bCAR - e2 bCAR - bCARc2 - nTORc1 - nANNc1 - nMTVc1 - nMELc1
8	[20]	Perturbation paths from <i>bCARc2</i> added to list in the following order:
5	[19]	e1 bCAR - bCARc2 - bCARc1 - bMERc1 - nTORc1 - nANNc1 - nMTVc1 - nMELc1
3	[18]	e1 bCAR - bCARc2 - bMERc1 - nTORc1 - nANNc1 - nMTVc1 - nMELc1
		e1 bCAR - bCARc2 - bRTPc1 - nRCHc1 - nMTVc1 - nMELc1
47	[37]	Perturbation path from <i>nTORc1</i> added to list
		e1 bCAR - bCARc2 - nTORc1 - bMERc1 - bMDHc1 - nMTVc1 - nMELc1
9	[20]	Perturbation path from <i>nANNc1</i> added to list
		e1 bCAR - bCARc2 - nTORc1 - nANNc1 - nRCHc1 - nMTVc1 - nMELc1
		No perturbation paths exist from <i>nMTVc1</i>
		Perturbations from the second-shortest path give the following new paths
14	[21]	e1 bCAR - e2 bCAR - bCARc2 - bCARc1 - bMERc1 - nTORc1 - nANNc1 - nMTVc1 - nMELc1
10	[20]	e1 bCAR - e2 bCAR - bCARc2 - bMERc1 - nTORc1 - nANNc1 - nMTVc1 - nMELc1
6	[19]	e1 bCAR - e2 bCAR - bCARc2 - bRTPc1 - nRCHc1 - nMTVc1 - nMELc1
54	[38]	e1 bCAR - e2 bCAR - bCARc2 - nTORc1 - bMERc1 - bMDHc1 - nMTVc1 - nMELc1
15	[21]	e1 bCAR - e2 bCAR - bCARc2 - nTORc1 - nANNc1 - nRCHc1 - nMTVc1 - nMELc1
63	[46]	Longest path
		e1 bCAR - e2 bCAR - bCARc2 - bRTPc1 - nRCHc1 - nANNc1 - nTORc1 - bMERc1 - bCARc1 - bMDHc1 - nMTVc1 - nMELc1

Figure 7.1.
Calculation of candidate paths in the sample network.

The algorithm consists of the following steps:

1. Find the shortest path from the origin to the destination.
2. Record this candidate path in the ranked list of paths.
3. Examine all nodes of the candidate path and record each new loopless path passing through a single perturbation from the candidate path.
4. After all perturbations have been examined, select the next path in the ranked list as the candidate path. Stop if the required number of paths have been found, otherwise repeat Step 3.

7.2 Implementation of the Algorithm

The algorithm was implemented using the external and internal representation of the network topology and the shortest path algorithm already described. The bulk of the calculations was performed by function *findShortestPaths* and its auxiliary functions which constructed the ranked list of paths. Functions *countNodesInPaths* and *countArcsInPaths* were used to construct lists of nodes and arcs, respectively, ranked in decreasing order of frequency of occurrence in the n shortest paths. All of the relevant functions were encapsulated in a shell which was a knowledge source attached to the Blackboard so that the *NPATH* module became an integral part of the Pegasus diagnostic package. The source code listing for the path-related functions is provided in Appendix C and the source code for *NPATH* is provided in Appendix F. Functions developed by Iqneibi [22] and Yuan [37] have been used in order to run *NPATH* in the Pegasus environment. These computer programs have been described in detail by the respective authors.

Function *findShortestPaths*, the logic of which is illustrated in Figure 7.2, receives the names of the origin and destination nodes and the number of paths that it must find in its argument list, and returns the pointer to the data structure which contains the ranked list of paths. The argument list of *findShortestPaths* also includes the variable *endArray*, which is required by various auxiliary functions to limit searches to only those portions of the arrays which are actually being used. *FindShortestPaths* is also passed pointers to integer variables *candPaths* and *pathSearched*, which are counters recording the total number of paths contained in the ranked list and the total number of paths examined, respectively. These counters are returned to the calling function and are used to monitor the operation of the algorithm, but are not essential to the correct operation of the algorithm.

FindShortestPaths first initializes the appropriate variables and finds the shortest path from the origin to the destination using function *findWeightedPath*, which returns the length of the path. Since this function records the path by constructing a trail of immediate predecessor nodes in *seenArray*, function *recordShortestPath* is used to transfer this information into a more useful format. *RecordShortestPath* receives the indices of the origin and destination nodes, follows the trail of predecessor nodes recorded in *seenArray*, and constructs a queue of arcs. Each item of the queue contains one arc of the path by recording the indices of the nodes at either end of the arc. Although this approach is slightly wasteful in terms of memory storage space, it was used because all of the queue functions used for other portions of the program may be reused without any modification, and subsequent data manipulation is simplified when the path is recorded as a series of arcs rather than as a series of nodes.

Figure 7.2.

Flow chart of the logic used in the function *findShortestPaths*.

After all variables are initialized and the shortest path is recorded as the first path in the ranked list of paths, *findShortestPaths* enters a triple nested loop which calculates the n shortest paths. The outermost WHILE loop is executed until all possible perturbations from the first n paths contained in the ranked list have been examined, at which point control returns to the calling function. The purpose of the outermost loop is to keep track of which candidate path in the ranked list is being examined and to set up all temporary variables required to examine all perturbations from the candidate path.

The middle WHILE loop keeps track of the node along the candidate path which is currently being examined. The innermost WHILE loop examines all neighbours (perturbations) of the node currently being examined. It first initializes *seenArray* and records the partial path from the origin to the perturbation node. The function then finds the shortest path from the perturbation to the destination by calling function *findWeightedPath*. Since *seenArray* contains a partial path up to this point, it is impossible to construct a path which contains a loop. If a loopless path does not exist, *findWeightedPath* returns a negative number as the path length, the new partial path contained in *nextPathQ* is deleted, and the next perturbation is examined. If a loopless path does exist, the ranked list is examined to determine whether that path had been recorded during previous iterations, and if the path is new, it is inserted into the appropriate location in the ranked list. If several paths having the same length are found, each new path is recorded at the end of that group of paths to ensure that all possible perturbations can be examined. Using this logic, an exhaustive search is made of all perturbations until the required number of paths is found.

The variable *counter* is incremented each time all perturbations from a candidate path have been examined, since the rank of the next candidate path is known with certainty only at that point. This counter controls the outermost loop. The variable *candPaths* is a counter recording the number of paths in the ranked list whose rank is not yet determined. It is incremented each time a new path is added to the ranked list and decremented each time *counter* is incremented. The sum of *counter* plus *candPaths* is therefore the size of the ranked list at any given time. A third counter, *pathSearched*, is incremented each time that the innermost loop is executed and thus records the total number of perturbations which have been examined, regardless of whether the perturbation resulted in a new path being recorded or not. A clock timer is used to print these counters at 15-second intervals to indicate the progress of the search and let the user know that the program has not crashed, but at the same time not slow the program down as would be the case with continuous output. During testing of the program, this was found to be the best compromise between continuous printout, which unnecessarily slowed down the program, and a long interval between outputs, which raised doubts about the status of the program.

Once the ranked list of paths has been created, functions *countArcsInPaths* and *countNodesInPaths* record the occurrences of critical arcs and nodes in the specified paths, respectively. The functions are passed the pointer to the ranked list of paths and the number of paths that were to be recorded. The functions then construct linked lists of the arcs or nodes in the n shortest paths sorted in decreasing order of frequency of occurrence, and return the pointer to these lists to the calling function.

Figure 7.3.
Flow chart of the logic used in the *NPATH* program.

7.3 *NPATH* Knowledge Source

The functions which find the n shortest paths and the critical arcs and nodes in those paths have been encapsulated in a knowledge source running as a server process accessed through the Blackboard of the Pegasus diagnostic package. The flow chart of the logic used in *NPATH* is presented in Figure 7.3, and the source code is provided in Appendix F.

NPATH first gains access to the Blackboard and registers as a new knowledge source. The internal topology is constructed from the *topology.dat* file and arc lengths are assigned from the optional *topology.len* file. Both files must be in the same directory as *NPATH*. As already discussed, this section of the code will need to be modified once the knowledge source is fully integrated into the Pegasus diagnostic system so that it accesses the same topology data file used by other modules such as the Dragon simulator.

NPATH then waits until a request for paths, nodes, or arcs is posted to the Blackboard. The following function numbers are reserved for *NPATH*:

- 10 – post list of paths only
- 11 – post list of arcs only
- 12 – post list of nodes only
- 13 – post list of paths, arcs, and nodes
- 14 – not used
- 15 – load new topology file *topology.cmp* (compressed nodes)

When function request number 15 is passed to *NPATH*, the program deletes the previously read topology, loads the network topology contained in the file *topology.cmp*, created by the node compression program, and waits for the next request.

After receiving any other valid function number, *NPATH* decodes the request and constructs the ranked list of shortest paths. The input parameters expected by *NPATH* in the request data fields are as follows:

<code>request_str0</code>	– name of origin node
<code>request_str1</code>	– name of destination node
<code>request_para0</code>	– number of shortest paths required
<code>request_para1</code>	– number of most frequent arcs required
<code>request_para2</code>	– number of most frequent nodes required

If the number of shortest paths required is negative, *NPATH* is deregistered from the Blackboard and stops. Otherwise, it checks for errors. An error condition occurs if the node names do not exist in the network, if the origin is the same as the destination, or if no path exists between the origin and destination. If an error condition is detected, the string *ERROR* is posted to the Blackboard in `reply_str0` and -1 is posted in `reply_para0`.

Otherwise there is no problem and *NPATH* proceeds with the calculations. Function *findShortestPaths* is called to construct the ranked list of paths and functions *countArcsInPaths* and *countNodesInPaths* are called as needed to count the arcs and/or nodes. The results are then posted to the Blackboard, one data item per reply. The paths are posted to the Blackboard as sequences of arcs. Special delimiters are used to indicate the end of a specific path and also the end of transmission (i.e. posting replies on the Blackboard) of a specific group of data. The data transmitted consists of the shortest paths, critical arcs, and critical nodes. Depending on the option selected, one or more of the blocks of data may be omitted from the transmission stream, but the sequence of data blocks transmitted will always follow the same order. The reply data fields and delimiters are summarized in Tables 7.1 and 7.2.

Table 7.1.
Data fields posted to the Blackboard by NPATH.

REPLY FIELD	ERROR	PATH DATA	ARC DATA	NODE DATA
reply_str0	ERROR	node name at start of arc	node name at start of arc	node name
reply_str1		node name at end of arc	node name at end of arc	node type
reply_para0	-1	path number	frequency of occurrence	frequency of occurrence
reply_para1		arc number	arc number	node number
reply_para2		path length		

Table 7.2.
Delimiters posted to the Blackboard at the end of a specific block of data.

REPLY FIELD	NEXT PATH DELIMITER	END OF PATH DATA	END OF ARC DATA	END OF NODE DATA
reply_str0	NextPath	EndPaths	EndArcs	EndNodes
reply_str1				
reply_para0		total number of paths searched	total number of paths searched	total number of paths searched
reply_para1		number of candidate paths	number of candidate paths	number of candidate paths
reply_para2	0	-1	-2	-3

7.4 Node Compression

A utility function was developed to permit users to examine and process the network topology in a hierarchical manner. The function *compressNodes* merges several network nodes into larger supernodes and thus simplifies the network topology. This compression feature is especially useful in merging nodes which are far from the anticipated paths being sought by *NPATH*, because the network topology is simplified and the number of candidate paths which may have to be examined is greatly reduced.

Nodes may be merged together in one of two ways – automatically according to a substring which is common to the node names, or as explicitly defined by the user. The merge information may either be recorded in a text file, or it may be entered from the terminal during execution of the program. If the merge information is stored in a file, the two correct formats which may be used together in the same file are as follows:

```
new_name: string_1 string_2 ... string_n
```

or

```
string_1 string_2 string_3 ... string_n
```

In the first example, the first character string *new_name* is followed by a colon and one or more character strings on the same line. This will result in all nodes which have one of the character strings following the colon as part of their name to be combined into a single node having the name *new_name*. In the second example, there is no colon so that each character string on that line will be treated individually and all nodes having this character string as part of that name will be merged and combined into a single node having that substring as its new name. For example, the compression instructions

```
OTTAWA:  MER CAR KAN
RTP ANN MEL
```

would result in all nodes having MER, CAR, or KAN as part of their names being merged into a single node called OTTAWA, and all nodes having RTP, ANN, and MEL as part of their names would be combined into three separate nodes called RTP, ANN, and MEL, respectively. All redundant arcs are also removed so that nodes remain connected by at most one arc. If the merge information is entered from the terminal, the user must specify the new node

name followed by a substring present in the old node name, one such pair for each group of nodes to be merged. This is similar to the first example. If only a single entry is made, it is assumed that the user entered a name substring which is to be the new name of all nodes sharing this substring, and is similar to the second example.

The compression algorithm constructs a compression list, implemented as a linked list, which contains a character substring of the old name and the corresponding new name of the node for each group of nodes to be merged together. As the computer program is constructing the internal network topology, it refers to this compression list, examines each node name which is read to determine whether it contains a substring present in the compression list, and if such a substring is present, it replaces that node name by the new node name. The new network topology is then written to file *topology.cmp* which may be accessed by *NPATH*.

Function *compressNodes* also has other options which permit the user to modify the compression list. They consist of:

- 0 – abort without compressing nodes
- 1 – add nodes to compression list
- 2 – remove nodes from compression list
- 3 – display nodes in compression list
- 4 – list nodes in the original data file
- 5 – exit and compress nodes

Options 1 and 2 permit the user to add nodes to or remove nodes from the compression list. Option 3 displays the nodes currently in the compression list by displaying the new and old node names. Options 0 and 5 allow the user to exit and either abort or complete the node compression

Option 4 is used to display the device names contained in the original topology data file. The user is prompted to enter the device name and device type search parameters. The user may specify either a substring of the device name, or blank line if all device names are to be considered, and the device type, or blank line if all device types are to be considered. The function then prints all devices, sorted by alphabetical order of their names, which match the search parameters.

If the user requests a hard copy of the compression statistics, a file will be created which will contain a listing of the number of nodes, listed by device type, which were merged into each new supernode. This file provides a useful record of the numbers of nodes which were merged together and can also be used to improve the heuristic of assigning a specific "length" to arcs connecting compressed nodes.

At the present time, *NPATH* arbitrarily increases the length of any arc connected to a compressed node by 7. This arbitrary value is not a very accurate representation since it treats large supernodes, i.e. nodes which may have resulted from the merging of a large number of nodes, in the same manner as small supernodes, i.e. nodes which may have resulted from the merging of only a small number of nodes. By examining the compression log file, a better heuristic could be implemented which would assign arc lengths which would be proportional to the "sizes" of the supernodes, i.e. the higher the number of nodes merged into a single supernode, the higher the arc length which connects this supernode to the rest of the network. However, these heuristics have not been implemented in the algorithms presented in this thesis but should be considered in future research projects.

The flow chart of the logic used in function *compressNodes* is provided in Figure 7.4, and the source code listing is presented in Appendix G.

Figure 7.4.
Flow chart of the logic used in the function *compressNodes*

7.5 Client Process Testing *NPATH*

A simple client process, called *NP*, was written to test all features of *NPATH*, particularly the communication via the Pegasus Blackboard. The main purpose of *NP* was to provide a user interface which passes information supplied by the user to *NPATH* via the Blackboard, requests *NPATH* to carry out the appropriate calculations, and returns the results back to *NP* which then prints them. All testing was carried out with the client-server process pair communicating via the Blackboard where *NP* was the client and *NPATH* was the server. In future, *NP* will be replaced by knowledge sources developed by other researchers which will require the information provided by *NPATH* to solve specific goals in the network diagnostic environment.

The *NP* program first gains access to the Blackboard and registers the process. The user is asked whether the information provided by *NPATH* is to be only displayed to the video terminal, or if a permanent record should also be written to a log file. The program then enters a DO loop which gets executed until the user issues the command to exit the program, at which point a message is sent to *NP'ATH*, and both the client and the server programs de-register from the Blackboard and exit. Exiting either program by typing Control C (CC) is not recommended because such an abnormal exit may sometimes fail to release the semaphores and shared memory used by the Blackboard.

Inside the DO loop, the user is prompted to enter the type of information that *NPATH* should provide, i.e. shortest paths, critical arcs, critical nodes, or all information, or whether the nodes in the network topology should be compressed into supernodes before *NPATH* is executed. *NP* then prompts the user to enter the names of the origin and destination nodes, and

the number of paths, arcs, and nodes required, and checks to ensure that non-negative values have been entered. The client process cannot check whether the node names are legitimate, but if the names are not valid, an error condition will be signaled by the server process and the user will be able to make corrections on a subsequent repetition of the DO loop.

If any negative parameters have been entered, they are automatically changed to the minimum default values. The minimum number of arcs and nodes may be 0, but the minimum number of paths may be 1. The program then passes the information to function *pathInfoToBB*, which posts the request to the Blackboard, and to function *pathInfoFromBB*, which processes the replies posted to the Blackboard by *NPATH*. A timer was placed around these two functions to record the time elapsed from the moment that a request is posted to the Blackboard to the moment that the last reply is read, and is a crude measure of the performance of the *NPATH* algorithm.

Function *pathInfoFromBB* is used to retrieve the replies posted by *NPATH* to the Blackboard and print them to the screen and log file if the user chose this option. The information displayed by the function has the following format:

Shortest Paths:

```
id_number. [path length]: origin node_1 node_2 ... destination
```

Critical Arcs:

```
id_number. [frequency of occurrence]: node_1-node_2
```

Critical Nodes:

```
id_number. [frequency of occurrence]: node name
```

The *id number* is the rank of the path, arc, or node, and ranges from 1 to either the number requested by the user or the maximum number of paths, arcs, or nodes available, whichever is lower. In most cases, the limit specified by the user will be reached long before the number of paths, arcs, or nodes actually present in the network is exhausted. The number listed in the square brackets following the *id number* is either the length of the path or the frequency of occurrence of the arc or node within the number of paths specified. The names following the path length or frequency of occurrence are the names of the nodes which make up the path, which are at either end of the critical arc, or the name of the critical node proper. Depending on the request option selected, one or more blocks of data may be missing, but it should be emphasized that the *n* shortest path calculations *must* be carried out by *NPATH* regardless of which of the request options is chosen.

The merging of several nodes into supernodes is performed by a separate function, *compressNodes*, and depending on the options selected, *NPATH* may be instructed to load a new data file which contains the compressed network topology. The operation of the compression function has been described in detail in the previous section.

Finally, the program prints some statistics indicating how many paths were contained in the ranked list, how many paths were searched, and how long the algorithm took to execute (in real time). The user is then given the option to repeat the DO loop or exit the program. If the loop is repeated, the user may enter new requests for shortest paths, and critical arcs and nodes. Otherwise, *NPATH* is sent a request to look for -1 paths, which is a signal for the server process to stop, and both programs finish executing.

The flow chart of the logic used in *NP* is presented in Figure 7.5, and the source code is provided in Appendix H.

Figure 7.5.

Flow chart of the logic used in the *NP* program.

7.6 Running the Programs

Due to the client-server nature of the knowledge source modules, the programs must be run in a windowing environment such as Sun's Sunview. Three windows are required to run the Blackboard, *NPATH* server process, and *NP* client process. A fourth window may, depending on the options specified by the user, be opened by the Blackboard.

The Blackboard program must be run first so that the shared memory and semaphores are set before any of the knowledge sources may be run. To start the Blackboard, enter the following command after the system prompt in one of the UNIX windows:

```
<pegasus-1>bb
```

The system will respond by printing the identification numbers of the shared memory and semaphores reserved by the system, and will wait for further input which will either alert the Dragon simulator or exit the program. No further input is required in this window until the execution of the Blackboard program is to be stopped. This is achieved by entering the number 9 following the program prompt, at which point the shared memory and the semaphores are released. It is important to stop the Blackboard only after all knowledge sources using it have been stopped, otherwise the shared memory and semaphores may not be released correctly.

The *NPATH* server and *NP* client programs may then be run in the other two UNIX windows. Since *NPATH* will wait for requests posted by *NP*, it must be started before any requests can be posted. To run *NPATH*, enter the name of the program following the system prompt:

<pegasus-2>npath

No further input is required in this window. Messages will be printed to advise the user of the program status, and once the shortest path calculations start, a message will be printed at 15-second intervals to advise the user of the progress of the search. *NPATH* is stopped from the client knowledge source, *NP*, and should not be terminated by typing ^C (Control C).

The *NP* client process is started last in a similar manner in the third window. It is activated by the command:

<pegasus-3>np

Since this program is the main user interface, the user will be prompted for additional information as already described. The shortest paths, critical arcs, and critical nodes as well as the other relevant statistics will be displayed in this window in the format described earlier. The execution of both *NPATH* and *NP* is halted from this window.

Until the knowledge sources are fully integrated into the Pegasus diagnostic package, these modules will continue to rely on a text file which stores the network topology. As this information has been hard-coded into the program to reduce the number of prompts which the user must answer, the topology data must be stored in the format specified in previous chapters in a file called *topology.dat* in the same directory as the knowledge sources.

Although the number of arcs within the network is virtually unlimited, being constrained only by the physical memory of the computer, the number of nodes is limited to 1400 at the present time. Should there be a need to change this limit, the constant *A_SIZE* which controls the sizes of the arrays must be changed in the main header file *bb_pkttype.h*.

Chapter 8

Complexity of the Network Topology

8.1 Network Topology

Maintenance and diagnosis of large-scale computer networks becomes more and more difficult as the complexity of the networks increases. The human operator, who is ultimately responsible for ensuring the proper operation of such a network, can process only limited information, and it is therefore imperative that this information be presented in a logical and concise manner. The sample network being studied by the Pegasus group contains 1262 nodes and is only a subset of a much larger network which contains several thousand devices.

The network under study connects devices on four continents. It consists of several types of components which include workstations, data collectors, Ethernet local area networks, optical fiber networks, bridges, multi-bridges, and routers. For the purposes of the topology model used in this thesis, all of the above devices were considered to be nodes. Specific arc "lengths", which are cost functions incorporating the capacity of a link, its importance, operating cost, etc., were not provided in the original topology file, and the arc length values were arbitrarily assigned during the development and testing of the tools developed as part of this thesis. Originally, these arc lengths were assigned a value of 1, since the number of hops was of primary interest. Subsequently, these lengths were changed and assigned a value of 1 for all connections between devices sharing the same community

string, i.e. devices at the same site, a value of 2 for connections between devices belonging to different sites, and values of 10, 20, or 100 for intercontinental connections. As these values are totally arbitrary, the computer programs using them have been made flexible enough so that the true values, once they are known, can be incorporated into the network model.

The convention used for selecting names in the BNR/NT network encodes information about the location and type of each device. The name contains a so-called characteristic community string, which is a string of 3 upper case alphanumeric characters uniquely identifying a specific physical site. Also encoded in the name is the device type, and whether the device belongs to BNR or NT. For example, the name bCARc1 means that the device is BNR's router C1 located at the Carling site in Ottawa. Similarly, e0 nMEL identifies an Ethernet local area network in Melbourne, Australia.

In order to obtain a clear understanding of the network topology, the nodes were compressed according to the characteristic community string common to all devices at a specific location. The topology of these compressed nodes is shown in Fig. 8.1. It has been organized so that devices are grouped according to geographical regions, which are shown in dashed lines. The number given with each node is the number of devices which were combined into each "supernode", and a summary of these devices is presented in Table 8.1. The full complexity of the BNR/NT network is presented in Fig. 8.2. This diagram is a scaled-down version of a larger drawing and has been included in this thesis to illustrate the complexity of the network being studied. Although this diagram is more complex than Fig. 8.1, it is still a somewhat simplified representation of the network topology, since multiple devices such as workstations, bridges, multi-bridges, and fast paths, have been combined together into single "boxes" representing each group of devices.

Figure 8.1. Topology of compressed nodes in the BNR/NT network.

Table 8.1.

Summary of network devices compressed according to geographic location

NAME	BR	DC	FD	FP	LN	MB	RT	WS	TOT
1 AIR	0	0	0	0	1	0	1	2	4
2 ANN	0	0	0	0	2	0	2	3	7
3 ATG	0	0	0	0	1	0	1	2	4
4 ATL	0	0	0	0	2	0	2	3	7
5 ATP	0	0	0	0	4	0	1	18	23
6 BAS	0	0	0	0	3	0	1	5	9
7 BCS	0	0	0	0	1	0	1	2	4
8 BDC	0	0	0	0	6	0	2	14	22
9 BFL	0	0	0	0	1	0	1	2	4
10 BPD	0	0	0	0	4	0	2	17	23
11 BRD	0	0	0	0	1	0	1	6	8
12 BRK	0	0	0	0	1	0	1	3	5
13 BVW	0	0	0	0	4	0	1	17	22
14 CAR	12	2	1	38	12	6	6	131	208
15 CLD	0	0	0	0	2	0	1	8	11
16 COL	0	0	0	0	0	0	1	0	1
17 CRK	19	2	0	10	6	0	2	42	81
18 CSW	0	0	0	0	4	0	2	10	16
19 DEN	0	0	0	0	2	0	2	3	7
20 DND	0	0	0	0	3	0	1	9	13
21 EDM	0	0	0	0	1	0	1	4	6
22 EMR	0	0	0	0	1	0	1	1	3
23 FFM	0	0	0	0	1	0	1	3	5
24 FTZ	0	0	0	5	2	0	2	5	14
25 GAL	0	0	0	0	2	0	1	6	9
26 GCH	0	0	0	0	1	0	1	1	3
27 GRG	0	0	0	0	2	0	1	2	5
28 GTY	0	0	0	0	1	0	1	2	4
29 HAR	0	0	0	0	6	0	1	27	34
30 HLF	0	0	0	0	0	0	1	0	1
31 HOF	0	0	0	0	1	0	1	1	3
32 KAI	0	0	0	0	1	0	1	6	8
33 KAN	3	1	0	8	2	0	2	8	24
34 KCH	0	0	0	0	1	0	1	1	3
35 LAK	0	0	0	0	2	0	2	3	7
36 LAU	0	0	0	0	3	0	2	12	17
37 LDN	0	0	0	0	2	0	1	5	8
38 MDH	0	0	0	0	1	0	2	1	4
39 MEL	0	0	0	0	1	0	1	6	8
40 MER	49	2	0	31	8	0	10	109	209
41 MGR	0	0	0	0	1	0	1	2	4
42 MHS	0	0	0	0	2	0	1	7	10
43 MIS	0	0	0	0	1	0	1	4	6
44 MKT	0	0	0	0	2	0	1	7	10
45 MLV	0	0	0	0	4	0	1	14	19
46 MSA	0	0	0	0	1	0	1	0	2
47 MTH	0	0	0	0	4	0	2	11	17
48 MTL	0	0	0	0	1	0	2	2	5
49 MTV	0	0	0	0	5	0	3	10	18
50 MUN	0	0	0	0	2	0	1	7	10
51 NCX	0	0	0	0	3	0	1	5	9
52 NSH	0	0	0	0	3	0	2	6	11
53 OWL	0	0	0	0	1	0	1	2	4
54 PAR	0	0	0	0	2	0	2	0	4
55 PEN	0	0	0	0	1	0	1	2	4
56 PIN	0	0	0	0	2	0	2	26	30
57 PLD	0	0	0	0	1	0	1	3	5
58 PNG	0	0	0	0	1	0	1	2	4
59 QNE	0	0	0	5	2	0	2	11	20
60 QNW	0	0	0	0	2	0	2	12	16
61 RAB	0	0	0	0	1	0	1	0	2
62 RCH	0	0	0	0	8	0	3	18	29
63 RTP	0	0	0	0	6	0	8	17	31
64 SCL	0	0	0	0	1	0	1	1	3
65 SDH	0	0	0	0	1	0	1	5	7
66 SGN	0	0	0	0	1	0	1	1	3
67 SHK	0	0	0	0	1	0	1	5	7
68 STJ	0	0	0	0	2	0	1	1	4
69 STL	0	0	0	0	2	0	1	4	7
70 STM	0	0	0	0	4	0		12	17
71 SYD	0	0	0	0	1	0	1	3	5
72 TOK	0	0	0	0	2	0	1	7	10
73 TOR	0	0	0	0	1	0	1	3	5
74 VDH	0	0	0	0	2	0	1	9	12
75 WAH	0	0	0	0	1	0	1	3	5
76 WDL	12	1	0	5	3	0	2	30	53
TOTAL	95	8	1	102	177	6	121	752	1262

LEGEND

BR = bridge

DC = data collector

FD = fiber network

FP = fastpath

LN = local area network (LAN)

MB = multi bridge

RT = router

WS = workstation

Figure 8.2. BNR/NT network topology.

The network illustrated in these figures consists of 1262 nodes. As described in previous sections, it differs from the smaller network consisting of 777 nodes only by the number of workstations. The Dragon simulator provides the option of adding more workstations than are actually contained in the *nm2.no* topology file, which describes the topology of only the basic 777 nodes. The additional approximately 500 "fictitious" workstations were therefore added so that the algorithms presented in this thesis could be tested on networks having more than 1,000 nodes. Simulations were therefore carried out with variations of the two basic topologies, one containing 1262 nodes and the other containing 777 nodes. The topology variations are summarized in Table 8.2.

From Figures 8.1 and 8.2 and Table 8.1, it is evident that most of the devices are located in the Ottawa area. The actual network contains 8 data collectors, which monitor the behaviour of the entire network, and these are also located exclusively in the Ottawa area, namely at the Carling, Merivue Court, Corkstown, Woodline, and Kanata sites. The operators of the network have determined that only 8 data collectors are necessary by considering the number of devices which are present in the network and the number of devices that a single data collector can monitor and the rate at which it can do so. For historical reasons, since Ottawa is the corporate headquarters, all data collectors have been located in this geographic region. However, the network extends over 4 continents, and it is possible that a better location of data collectors can be found so that the monitoring efficiency can be increased. This hypothesis was tested by developing the *MEDIAN* computer program and running a series of simulations on the network under study.

In examining the complex interconnections between individual devices in the network, which are particularly apparent in the Carling and Merivue

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

Court nodes, it is evident that a large number of paths may be taken by data packets traveling between two devices. Since dynamic routing takes place, the packets may not necessarily follow the shortest route from the origin to the destination, because the paths assigned by the routers depend on a large number of factors such as the traffic, capacity, transmission speed, operating cost, etc. of the various arcs which the packets must traverse, and the actual path which is followed is determined by proprietary routing algorithms. To further complicate matters, the return path for replies from the destination back to the origin may be different than the path from the origin to the destination.

The diagnostic methods which have been developed or are currently under development by the Pegasus group need to know the actual path taken or the group of most probable paths which could have been taken when a fault is detected. The *NPATH* program was developed to provide this information for the other knowledge sources.

Table 8.2.

Arc lengths for the various BNR/NT network topologies studied.

Topology type	Number of nodes	Arc lengths at same site	Arc lengths between sites	Intercontinental arc lengths
1A	777	1	1	1
1B	777	1	2	10
1C	777	1	2	20
1D	777	1	2	100
2	1262	1	1	1

8.2 Hierarchical Methodology

The human approach to problem solving is generally hierarchical in nature in that a complex problem is visualized from a global perspective and subsequently broken down into more manageable components which can then be solved. This approach, known as the divide and conquer methodology, is used in many engineering disciplines, and is well suited for the analysis of computer networks.

The network topology can be represented as a geographical hierarchy, whereby the most general level of complexity depicts devices on the four continents and the intercontinental connections, and the next levels in turn represent countries, cities, sites within each city, rooms within each site, and finally devices within a room. If the information about the network needs to be even more detailed, this concept can be extended so that ultimately each chip in the network could be represented.

The hierarchical approach is important because it can be used to greatly simplify the problem being studied. For example, as will be discussed in subsequent sections, the relative sizes of domains serviced by a single data collector can be seen from the top level of the hierarchy, as in Fig. 8.1. The actual interconnections between individual devices contained in each super-node are irrelevant at this point. Similarly, if the shortest path between a workstation attached to e0.bCAR and a workstation attached to e0.nMEL is sought, the probable route will pass through Toronto, Ontario (MTH), Ann Arbor, Michigan (ANN), and Mountainview, California (MTV). All other nodes, such as those in Europe and Asia, can be eliminated from the search space. Similarly, the packets are unlikely to follow a path through Western or Eastern Canada, as this is not the most direct route. However, this

conjecture, which can be easily reached upon the examination of Fig. 8.1, would not necessarily be reached if only a detailed topological map, as in Fig. 8.2, was considered, because what appears quite obvious in the simplified topology is hidden among the unnecessary clutter of the detailed topology.

This concept will become magnified as the network complexity increases. Human operators are unable to simultaneously process large quantities of input and must simplify the input as much as possible. The network management centre therefore displays the network topology at various views in the geographical hierarchy, and the operators have the opportunity to zoom into specific supernodes and examine the constituent devices in greater detail. Diagnostic methods should be able to perform the same functions automatically in order to diagnose faults more quickly and with greater accuracy, and the compression algorithms presented in this thesis demonstrate a method by which this may be achieved.

The node compression algorithms have therefore been developed and used to simplify the problems being studied. However, this concept was not taken to its ultimate limit whereby parts of the network would be automatically excluded from (i.e. by node compression or virtual partitioning of the network), or added to (i.e. by decompression of supernodes or connection to network domains initially discarded) the search space as required when solving a specific problem. This approach will be imperative as the network complexity increases further, and may be a useful area for future research.

Chapter 9

Finding Multiple Optimal Medians

The programming logic and implementation of the computer program which finds the locations of optimal medians in a network was described in Chapter 6. This chapter describes the output generated by the *MEDIAN* program, discusses the simulations carried out, and interprets the results.

9.1 *MEDIAN* Input and Output

The main input for *MEDIAN* consists of the network topology, the distance matrix, and the network topology used by the Dragon simulator (*nm2.no*), stored as three separate text files, the names of which can be entered as command line arguments or when prompted by the program during its execution. Secondary input, consisting of the control parameters which determine the number of iterations which *MEDIAN* is to perform, the domain size tolerances which should be imposed, and the initial selection of fixed or variable medians, is entered during the execution of the program either directly, or by means of UNIX input redirection. The prompts and responses for a typical run are provided in Figure 9.1.

The output is written to the terminal and to a log file, the name of which is specified by the user. The output written to the log file consists of the medians randomly selected at the start of each iteration, the medians and corresponding domain sizes as well as the sum of distances during each

iteration and subiteration, a global summary which lists the control parameters and the optimal solution, and a list of nodes distributed among the optimal domains of the network. A sample listing of the global summary is shown in Figure 9.2.

The seed which is used to initialize the random number generator is the sum of the tolerances between the domains of the network (i.e., the tolerance between any two domains plus the tolerance between the maximum and minimum domains). This ensures that random sequences of medians may be repeated for the same set of initial input parameters, yet different sequences may be selected by altering the domain size restrictions. In most cases, the domain size restrictions are not enforced in the final assignment of nodes to their closest median, and the size restrictions are therefore irrelevant for the final answer. However, the user does have the option to enforce the size limits even for the final solution.

The user also has the option to enter fixed medians. This feature is especially useful when it is necessary to determine the sum of distances for a particular set of devices which may not be the optimal medians, and was used to obtain the sum of distances for the data collectors used in the actual BNR/NT network. A slight relaxation of this idea is that the user may specify a set of medians, but the program then has freedom to select other nodes which would be better suited as medians for that domain. This feature is useful when an optimal solution is suspected but not known for certain, or when it is desirable to investigate the effect of selecting a median which might not have otherwise been chosen. For example, if other simulations indicate that all medians should be located in the Ottawa area, this feature can be used to place a median in Europe (e.g. at *e0.nMUN*) and another in Eastern USA (e.g. at *e3.bATP*), and let the solution eventually converge to *bMDHc1* and

```

<pegasus-1> median bnrn.dat bnrn0.mtx

PROGRAM TO LOCATE DATA COLLECTORS IN BNR/NT NETWORK

USAGE: median file1 file2 file3 file4
file1 - INPUT: original network topology
file2 - INPUT: distance matrix
file3 - OUTPUT: log of median selection
file4 - OUTPUT: simulator topology definition

Enter log file name:
opt08.log
Enter new simulator topology definition file:
n108.no

Reading data file ... 777 lines read from file.
Building topology ...
Reading distance matrix ...

Enter initial medians and codes in format: MEDIAN_NAME CODE
where CODE = 1 for fixed median
CODE = 0 or blank for variable median
Terminate input by entering blank line.
e0.bMER 1
e0.bCAR 1
bRTPc1 1
e0.bCRK
e0.bWDL
bMERc2
nANNc1
bKANc2

Enter total number of medians and number of iterations:
8 10
Enter size differences between domains (as percent):
Any two domains, largest and smallest:
10 20
Enforce size limits for final domains (y/n)?

Domain size restrictions not enforced in final run.

*****> ITERATION 1 <*****

Original medians:
e0.bMER e0.bCAR bRTPc1 e0.bCRK e0.bWDL bMERc2 nANNc1 bKANc2

1. 1765 = sum of distances
   e0.bMER-175 e0.bCAR-175 nMTHc1-160 e0.bCRK-67 bMERc2-61 bRTPc1-59 e0.bWDL-48 e0.bKAN-32
2. 1786 = sum of distances
   e0.bMER-175 e0.bCAR-175 nMTHc1-171 e0.bCRK-67 bMERc2-61 bRTPc1-55 e0.bWDL-48 e0.bKAN-25
Solutions converged.

*****> ITERATION 2 <*****

Original medians:
e0.bMER e0.bCAR bQNEh26 bCARh280 bCARh514 bRTPc1 nCALc1 bCARh264

1. 2451 = sum of distances
   bCARc5 175 bRTPc1-156 e0.bCAR-156 e0.bMER-156 bCAR008-74 bMERh898-20 bMER646-20 bCRKc1-20
2. 1990 = sum of distances
   bRTPc1-158 e0.bCAR-157 bWDLc2-156 e0.bMER-156 e0.bCRK-90 bMERc1-20 bMERc5-20 e1.bMER-20
...

9. 1897 = sum of distances
   e0.bMER-175 bRTPc1-175 e0.bCAR-156 e0.bCRK-114 bMERc2-64 e0.bWDL-48 e0.bPIN-25 bCARh5-20
Solutions diverged 5 times.

```

Figure 9.1.
Typical data entry for the *MEDIAN* program.

```

=====> ITERATION 9 <=====

Original medians:
e0.bMER e0.bCAR bQNWs7 e5.bCRK e2.nMTH bRTPc1 bWDLh316 e1.bCRK

1: 2023 = sum of distances
   e0.bCAR-175 nMTHc1-175 e0.bMER-163 bRTPc1-130 e0.bCRY 48 e0.bWDL 44 e1.bCRK 22 e0.bQNW 20
2: 2110 = sum of distances
   nMTHc1-175 e0.bCAR-166 e0.bMER-156 bCRKc1-133 bRTPc1-59 e0.bWDL-48 bMERcA 20 bQNWc2 20
3: 2017 = sum of distances
   nMTHc1-175 e0.bCAR-166 e0.bMER-156 bCRKc1-133 bRTPc1-59 e0.bWDL 48 bMERcS 20 bQNWc2 20
4: 1863 = sum of distances
   e0.bCAR-175 nMTHc1-173 e0.bMER-166 e0.bCRK-90 bRTPc1-59 e0.bWDL-48 bPINc2 46 bQNWc2 20
5: 1947 = sum of distances
   e0.bCAR-175 nMTHc1-175 e0.bMER-166 e0.bCRK-109 bRTPc1 59 e0.bWDL-48 e0.bPIN 25 bQNWc2 20
Solutions converged.

=====> ITERATION 10 <=====

Original medians:
e0.bCAR bMER626 e0.bMER bQNEc1 bRTPc1 bCRKa377 e1.nATL bWDL857

1: 1961 = sum of distances
   bRTPc1-175 e0.bMER-167 e0.bCAR-157 bCARcS-136 nMTVc2-57 e0.bWDL-45 bCARh547-20 bQNEc2 20
2: 1875 = sum of distances
   bRTPc1-175 e0.bCAR-167 e0.bMER-156 bCRKc1-120 nMTVc2-71 e0.bWDL-48 bMERh145-20 bQNEc2 20
3: 1803 = sum of distances
   bRTPc1-175 e0.bCAR-167 e0.bMER-156 e0.bCRK-120 nMTVc2-71 e0.bWDL 48 bMERc1-20 bQNEc2 20
4: 1752 = sum of distances
   bRTPc1-175 e0.bMER-167 e0.bCAR-162 nANnc1-115 e0.bCRK-70 e0.bWDL-48 bMERc1-20 bQNEc2 20
5: 2031 = sum of distances
   nMTHc1-175 e0.bCAR-167 e0.bMER-162 bRTPc1-115 e0.bCRK-70 e0.bWDL 48 bMERc1 20 bQNEc2 20
6: 2021 = sum of distances
   nMTHc1-175 e0.bCAR-167 e0.bMER-162 bCRKc1-126 bRTPc1-59 e0.bWDL-48 bMERc1 20 e0.bQNE 20
7: 2013 = sum of distances
   nMTHc1-175 e0.bCAR-167 e0.bMER-162 bCRKc1-126 bRTPc1-59 e0.bWDL-48 bMERc1-20 e0.bQNE 20
Solutions converged.

***** GLOBAL OPTIMUM *****

Topology data file: 'bnrn.dat'. Distance matrix file: 'bnrn0.mtx'.
Medians: 8; Iterations: 10; Nodes: 777.
Tolerance: Any 2 domains = 77 (10.00%); Maximum = 155 (20.00%)
Domain size restrictions not enforced in final node allocation.
Fixed medians:
e0.bMER e0.bCAR bRTPc1
Optimum in Iteration:      5-03
Optimal sum of distances:  1487
Combinations examined:    53
Elapsed time:              00:06:41

Final sum of distances:    1361

Optimal medians and corresponding domain sizes:
e0.bMER-206 e0.bCAR-165 nANnc1-99 bCARc2-80 e0.bCRK-67 bRTPc1-54 bMERc2-61 e0.bWDL-45

Printing median sets to file ...
Modifying simulator topology file ...

Simulator topology saved as 'n108.no'.
Log file saved as 'opt08.log'.

=====> NORMAL TERMINATION <=====

<pegasus-2>

```

Figure 9.2.
Typical output produced by the *MEDIAN* program.

bRTPc1, respectively. During data entry, the distinction between a fixed and an initial variable median is made according to the integer code following the name of the trial median – 1 for a fixed median and 0 or blank for a variable median. If specific medians are entered, the sequence of generating random medians to start each iteration is postponed depending on the number of medians entered. During most simulations, no initial medians were entered and the program was permitted to select all random medians to start each iteration.

9.2 Network Size and Running Time

The distance matrix which contains the distances between every pair of nodes in the network is calculated by a separate program, *MGEN*, and is read by *MEDIAN* during the program initialization phase. After the distance matrix has been read, all distances are accessed directly from the matrix, now stored in memory, and no new shortest path calculations are performed. The only calculations which are carried out for each iteration are the allocation of random medians at the start of each iteration, assignment of all nodes to their closest median, reassignment of nodes to satisfy the domain size tolerances, and calculation of the absolute medians of the new domains. Integer arithmetic is used for all calculations, and all nodes of the network are referenced by their index according to their location in *nameArray*.

Simulations were carried out using 4 different sizes of networks which had 76, 251, 497, and 1262 nodes. The smallest network containing 76 nodes was obtained by fully compressing all nodes according to the characteristic community string and was illustrated in Figure 8.1. The largest network

containing 1262 nodes was the full-size BNR/NT network under study (777 actual nodes plus 485 fictitious workstations). The intermediate sizes of the networks, those containing 251 nodes and 497 nodes, were obtained by compressing selected nodes of the actual BNR/NT network in order to obtain networks of the required sizes. This ensured that the shortest path between every pair of nodes was sufficiently random.

Since the actual BNR/NT network contains 8 data collectors, all simulations consisted of finding 8 optimal medians. The imposed domain size tolerances ranged from 0% to 100% for the tolerances between any two domains as well as between the largest and smallest domain. Five domain size restrictions were used in order to obtain 5 different random sequences. The running times of the *MEDIAN* program were recorded and are summarized in Tables 9.1 and 9.2. The relevant parameters showing the optimality of the solution are also listed. *MEDIAN* was run for 40, 100, and 500 iterations for each of the 5 starting conditions and each of the 4 network sizes.

As can be seen from the tables, the running time varied from a few seconds to almost a full day. Although the actual running time of the program is not an accurate measure of the speed of the algorithm, since it depends on a large number of factors such as the computer hardware (processor speed, memory access, server access, etc.) as well as the time slice which could be allocated by the CPU to the *MEDIAN* process run in a multi-tasking environment, it was recorded in order to obtain at least a general idea of how long it takes to solve a single problem. In an attempt to keep the simulations as consistent as possible, for those simulations which required half an hour or less to run, the *MEDIAN* program was the only process (except for operating system processes) run on a single workstation (Artie) in a Sunview environment with output being directed to the screen and to the log file (i.e.,

screen lock not active). For the other longer simulations, it was impractical to deny access to the workstation to other users and to run all simulations on only one workstation, and the running times are inaccurate for this reason.

Although CPU time has been used for a long time as a measure of algorithm performance, it has been recognized to have difficulties. According to Hung and Divoky, the clock in a multi-tasking system can vary by as much as 20% on exactly the same job [21]. The actual running time instead of the CPU time allocated to the *MEDIAN* process was measured because it was desirable to estimate the completion time of a particular simulation in order that a new simulation could be started. Should it be necessary to determine the actual CPU time used during the simulation, the *MEDIAN* program would need to be modified and the simulations rerun. This could be part of future research investigating further optimization of the multiple median algorithm.

9.3 Topologies Using Various Arc Lengths

The first network model assumed that all arcs between nodes had a length of 1. This model, topology Type A, represented the case where only the numbers of hops between devices were of interest, regardless of how expensive each hop might be, and the results presented in the previous sections are based on this topology.

As this model is not entirely realistic, additional simulations were run in which the arc lengths were assigned values other than 1 in order to differentiate the cost of individual hops. Since the real costs were unknown, three additional models, in which a value of 1 was assigned to all arcs belonging to

the same physical location, a value of 2 was assigned to all arcs between different sites on the same continent, and all intercontinental connections were assigned a value of 10, 20, or 100 for topology types B, C, and D, respectively. The network topologies are summarized in Table 8.2.

In contrast to the model where all arcs were set to 1, *MGEN*, which generated the distance matrices, had to be modified to use the more complex shortest path functions. Calculation of the distance matrix for an unweighted network consisting of 1262 nodes required approximately 8 hours to complete. Calculation of the distance matrices for the weighted networks of the same size required approximately 2.5 days each. Several algorithms which are more efficient for finding the distances between all pairs of nodes in a network than repeated use of Dijkstra's algorithm exist [11], and in view of the length of time required to calculate the distance matrix for even a network with 1262 nodes, it may be worthwhile to implement the more efficient algorithms. Calculations of the distance matrices for the networks containing 777 nodes required approximately 8 hours to complete.

In reviewing the results of the various simulation, it was observed that as long as the intercontinental arc lengths were greater than 1, there was no significant difference with regard to the optimal location of data collectors between topology types B, C, and D (intercontinental lengths of 10, 20, and 100, respectively). This indicates that as long as the length of intercontinental arcs is greater than the arc lengths between sites on the same continent, the actual value of that arc length is not important. For that reason, all subsequent simulations used topology type 1B (777 nodes and intercontinental arc lengths set to 10).

The main difference between topology types A and B was that one of the medians which was assigned to Ottawa (*bCARc2*) when all arc lengths

were set to 1 was moved to Maidenhead in England (*bMDHc1*) when the lengths of intercontinental connections were increased, but otherwise there were only minor changes. A comparison of the optimal locations for the two topologies is presented in Table 9.3.

Network topologies containing the true data collectors used by BNR/NT at the present time were compared with topologies consisting of optimal data collector locations. In one simulation, the domain size restriction was not imposed for the final assignment of nodes to their closest median, which resulted in large differences in domain sizes. Several of the data collectors were monitoring only 1 or two nodes, while a single data collector was monitoring as many as 374 nodes. If the domain size restrictions were imposed during the final assignment of nodes to their closest median, the size differential between various domains was as specified by the user, but the final sum of distances was more than double the sum of distances in an optimal configuration. A third simulation was run where the actual data collectors were used as initial variable medians to establish the domains, but the *MEDIAN* program was allowed to go through a single iteration to establish the optimal location of medians within the specified domains. The results indicate that the present location of the data collectors in the BNR/NT network results in poor monitoring of the network, and better configurations exist. This hypothesis was also verified by other simulations described in subsequent sections.

Table 9.1.

Summary of the selection of 8 optimal medians in networks having 76 and 251 nodes. All arc lengths are 1.

Iter	Network size																
	Domain size tolerance (%)		76 nodes								251 nodes						
			Any two domains	Max and min. domains	Final sum of distances	Search time	Optimal iteration	Comb. examined	Distances for optimal iteration	Size of largest domain	Size of smallest domain	Final sum of distances	Search time	Optimal iteration	Comb. examined	Distances for optimal iteration	Size of largest domain
40	0	0	96	0 00 12	33-03	166	119	25	3	512	0 02 27	10-02	189	542	47	21	
40	5	10	97	0 00 17	14-03	306	103	15	2	477	0 02 36	02-04	231	479	46	16	
40	10	20	93	0 00 12	23-01	200	97	23	3	477	0 02 20	15-03	182	477	50	16	
40	50	50	95	0 00 06	05-03	79	95	24	3	493	0 01 43	12-03	105	493	88	8	
40	100	100	96	0 00 06	39-03	81	96	27	3	480	0 01 23	22-03	103	480	66	16	
100	0	0	96	0 00 27	33-03	393	119	25	3	512	0 07 25	10-02	486	542	47	21	
100	5	10	97	0 00 42	14-03	806	103	15	2	477	0 07 01	02-04	566	479	46	16	
100	10	20	94	0 00 29	57-02	497	95	18	3	477	0 05 25	15-03	470	477	50	16	
100	50	50	95	0 00 15	05-03	194	95	24	3	492	0 04 00	80-02	264	492	83	26	
100	100	100	94	0 00 15	72-02	199	94	26	3	480	0 03 52	22-03	751	480	66	16	
500	0	0	100	0 02 20	340-01	1994	114	28	4	489	0 32 40	104-05	2343	527	42	16	
500	5	10	95	0 07 06	482-03	4174	97	16	4	477	0 36 33	109-04	2946	477	42	16	
500	10	20	93	0 04 29	216-09	2411	93	18	3	477	0 32 01	15-03	2536	477	50	16	
500	50	50	94	0 02 24	170-02	983	94	24	3	479	0 20 13	296-03	1305	479	66	16	
500	100	100	93	0 02 20	207-02	960	93	28	3	477	0 20 15	407-04	1312	477	50	16	

NOTES

- 1 The "seed" which was used to initialize the random number sequence was the sum of the domain size tolerances
- 2 The search time recorded represents the actual running time of the simulation
- 3 The optimal iteration column shows the iteration (first number) and subiteration (second number) of the optimal selection of medians
- 4 The combinations examined column gives the sum of iterations and subiterations examined during the simulation
- 5 Domain size restrictions were not enforced for the final assignment of nodes to their closest median

Table 9 2

Summary of the selection of 8 optimal medians in networks having 497 and 1262 nodes. All arc lengths are 1

Iter	Domain size tolerance (%)		Network size													
			497 nodes					1262 nodes								
			Any two domains	Max and min domains	Final sum of distances	Search time	Optimal iteration	Comb examined	Distances for optimal iteration	Size of largest domain	Size of smallest domain	Final sum of distances	Search time	Optimal iteration	Comb examined	Distances for optimal iteration
40	0	0	967	0 15 17	38-06	262	1115	185	4	3389	1 06 44	16-03	262	3551	358	75
40	5	10	977	0 17 11	11-03	373	1097	118	1	3245	1 22 29	23-02	409	3402	348	73
40	10	20	917	0 16 38	04-12	336	971	123	17	3174	1 00 27	01-05	284	3192	302	53
40	50	50	918	0 06 55	27-02	111	918	213	16	3288	0 36 03	22-04	126	3288	367	50
40	100	100	929	0 06 35	26-02	102	929	205	5	3361	0 29 06	31-05	112	3361	416	32
100	0	0	975	0 42 20	95-13	697	1108	185	4	3389	2 44 42	16-03	661	3551	358	75
100	5	10	977	0 46 58	11-03	945	1097	118	1	3137	4 15 26	83-07	992	3177	242	80
100	10	20	917	0 38 34	04-12	766	971	123	17	3174	2 18 59	01-05	667	3192	302	53
100	50	50	895	0 18 04	93-06	299	895	234	10	3241	1 28 20	56-03	312	3241	363	47
100	100	100	906	0 16 22	94-12	258	906	242	13	3293	1 23 04	59-02	272	3293	362	30
500	0	0	969	2 36 57	257-08	3521	1096	185	4	3389	14 29 31	16-03	3450	3551	358	75
500	5	10	957	3 03 05	434-04	4594	1058	119	23	3132	16 58 22	213-07	5043	3150	256	81
500	10	20	933	2 13 44	240-03	3740	965	121	20	3145	18 02 11	142-04	3390	3145	257	75
500	50	50	874	1 07 14	118-05	1554	874	183	25	3241	11 03 43	56-03	1503	3241	363	47
500	100	100	876	0 58 18	134-03	1263	876	242	16	3216	6 05 53	425-03	1401	3216	405	53

NOTES:

1. The "seed" which was used to initialize the random number sequence was the sum of the domain size tolerances.
2. The search time recorded represents the actual running time of the simulation.
3. The optimal iteration column shows the iteration (first number) and subiteration (second number) of the optimal selection of medians.
4. The combinations examined column gives the sum of iterations and subiterations examined during the simulation.
5. Domain size restrictions were not enforced for the final assignment of nodes to their closest median.

Table 9.3.

The effect of random sequences and network topologies on the selection of optimal medians in the BNR/NT network.

Domain size tolerance (%)		Optimal median locations and domain sizes in networks consisting of 777 nodes																				Note
Any two	Max & min	Iter	Top Type	Sum of dist	Median 1	Size of Domain 1	Median 2	Size of Domain 2	Median 3	Size of Domain 3	Median 4	Size of Domain 4	Median 5	Size of Domain 5	Median 6	Size of Domain 6	Median 7	Size of Domain 7	Median 8	Size of Domain 8		
n/a	n/a	n/a	1	2642	BCARh1	374	BCARh2	2	BCRKh1	66	BCRKh2	1	DMERh1	260	DMERh2	1	BCARh1	25	BWDLh1	48	1	
10	20	n/a	1	3549	BWDLh1	175	BCRKh1	157	DMERh1	156	BCARh1	156	BCARh1	73	DMERh2	20	BCRKh2	20	BCARh2	20	2	
10	20	1	1	1893	e0 DMER	185	nMThc1	107	BCARh2	287	BCRKh1	70	DMERh2	61	e0 BWDL	45	DMERh5	21	BCARh150	1	3	
0	0	40	1	1408	e0 BCAR	198	e0 DMER	185	nLAKc1	101	e0 BCAR	70	DMERh2	65	BCARh4	56	DMERh5	54	e0 BWDL	48	4	
5	10	40	1	1417	BCRKh1	110	DMERh2	105	e0 BCAR	173	nANNC1	97	e0 DMER	167	DMERh2	25	e0 BWDL	48	BRTPc1	52	4	
8	16	40	1	1366	e0 BCAR	178	e0 DMER	243	DMERh5	56	nANNC1	99	e0 BCAR	67	BRTPc1	54	e0 BWDL	48	BCARh2	32	4	
10	20	40	1	1366	e0 BCAR	224	e0 DMER	196	DMERh2	65	nANNC1	97	e0 BCAR	67	BRTPc1	52	e0 BWDL	48	e0 BPIN	25	4	
15	30	40	1	1426	e0 DMER	248	BCARh2	142	e0 BCAR	153	nMTVc2	80	e0 BCAR	67	e0 BWDL	45	nBDCc1	28	nBPCc1	14	4	
50	50	40	1	1371	e0 DMER	247	e0 BCAR	165	nANNC2	101	BCARh2	80	e0 BCAR	67	BRTPc1	52	e0 BWDL	45	nCLDc1	20	4	
100	100	40	1	1403	e0 DMER	290	e0 BCAR	183	nLAKc1	97	e0 BCAR	67	BRTPc1	56	e0 BWDL	48	e0 BCAR	25	e0 BCAR	11	4	
10	20	500	1	1366	e0 BCAR	224	e0 DMER	196	DMERh2	65	nANNC1	97	e0 BCAR	67	BRTPc1	52	e0 BWDL	48	e0 BPIN	25	5	
best estimate	n/a	n/a	1	1361	e0 DMER	206	e0 BCAR	165	nANNC1	99	BCARh2	80	e0 BCAR	67	BRTPc1	54	DMERh2	61	e0 BWDL	45	4	
n/a	n/a	n/a	2	3634	BCARh1	374	BCARh2	2	BCRKh1	66	BCRKh2	1	DMERh1	260	DMERh2	1	BCARh1	25	BWDLh1	48	1	
10	20	n/a	2	5072	BWDLh1	175	BCRKh1	157	DMERh1	156	BCARh1	156	BCARh1	73	DMERh2	20	BCRKh2	20	BCARh2	20	2	
10	20	1	2	2563	e0 BCAR	224	BCARh2	102	e0 DMER	254	nANNC1	103	e0 BCAR	67	e0 BPIN	25	DMERh231	1	e0 DMER	1	3	
0	0	40	2	2384	e0 BCAR	198	e0 DMER	185	nANNC2	104	e0 BCAR	70	DMERh2	62	BCARh4	56	DMERh5	54	e0 BWDL	48	4	
5	10	40	2	2513	DMERh2	102	e0 BCAR	164	e0 DMER	167	nANNC1	94	e0 BCAR	70	BCARh2	74	e0 BWDL	48	BRTPc1	58	4	
8	16	40	2	2064	DMERh5	116	e0 DMER	167	e0 BCAR	217	nANNC2	104	e0 BCAR	67	e0 BWDL	48	e0 BCAR	25	BWDLh1	33	4	
10	20	40	2	2216	e0 DMER	239	e0 BCAR	241	DMERh2	54	nANNC2	104	e0 BCAR	70	BWDLh1	33	e0 BPIN	25	nANNC1	11	4	
15	30	40	2	2394	e0 DMER	303	BCARh4	128	e0 BCAR	152	e0 BCAR	70	nMTVc3	66	e0 BWDL	48	nTKC1	9	BCARh401	1	4	
50	50	40	2	2112	e0 BCAR	281	e0 DMER	236	nANNC2	124	e0 BCAR	70	e0 BWDL	48	BWDLh1	33	e0 BPIN	24	BWDLh175	1	4	
100	100	40	2	2093	e0 DMER	277	e0 BCAR	237	nMTVc2	79	e0 BCAR	67	e0 BWDL	48	BWDLh1	33	e0 BCAR	25	e0 BCAR	11	4	
10	20	500	2	2364	e0 BCAR	224	e0 DMER	223	DMERh5	56	nANNC2	104	e0 BCAR	67	e0 BWDL	48	BWDLh1	33	e0 BCAR	25	4	
best estimate	n/a	n/a	2	2031	e0 DMER	224	e0 BCAR	206	BCARh2	70	e0 BWDL	45	BRTPc1	47	DMERh5	56	nANNC2	93	BWDLh1	33	4	

TOPOLOGY TYPE		COMMENTS				
1	2	No	1	2	3	
1	1	1	1	1	1	
1	2	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	10	1	1	1	1	
1	1					

COMMENTS

No

- 1 No iterations actual data collectors used as fixed medians domain size restrictions not enforced
- 2 No iterations actual data collectors used as fixed medians domain size restrictions enforced
- 3 Single iteration original data collectors used to start iteration process
- 4 Random selection of initial medians, normal rules for terminating iteration applied
- 5 Optimal configuration was established in Subiteration 5 of iteration 30. Run of 500 iterations not required

TOPOLOGY TYPE

1	2
1	1
1	2
1	10

Arc lengths at same site
Arc lengths between sites
Intercontinental connections

9.4 Calculation of an Optimal Solution

The *MEDIAN* algorithm is used to find a near-optimal solution for a problem for which it is not practical to calculate the exact optimal solution. The optimal solution can be determined only by enumerating all of the possible combinations. In a network of n devices from which m medians are to be selected, there are a total of

$$C(n, m) = \frac{n!}{m!(n-m)!}$$

possible combinations. Even in a "small" network of 76 nodes, there are approximately 1.89×10^{10} combinations of selecting 8 medians. In the real network consisting of 777 or 1262 devices, the possible combinations of selecting 8 medians is considerably larger.

Fortunately, it is not necessary to calculate the exact optimal solution, since an approximation to an optimal solution is adequate. The *MEDIAN* algorithm uses a hill-climbing technique by selecting an initial random solution and improving upon it during subsequent iterations. Due to the nature of the problem, there may be several near-optimal solutions, so the solution space can be viewed not as a single hill but as a rolling terrain where each peak represents one of the near-optimal solutions. The true optimal solution is assumed to lie at the apex of the tallest hill. At the start of each iteration, *MEDIAN* randomly selects one of these hills and calculates the local optimal solution. After the specified number of iterations has been executed, the best solution of the entire solution set is reported as the "global optimal" solution. Clearly, the more solutions that are examined, the higher the probability that the true optimal solution will be found.

From Tables 9.1 and 9.2, it is evident that the slopes of the hills are very gentle. This means that as the number of iterations is increased, the optimality of the solution does not markedly improve. For example, in the network consisting of 1262 nodes, the optimal sum of distances from the 8 medians to their closest nodes was 3174 hops determined from 40 iterations. After 100 iterations, the solution improved to 3137, and after 500 iterations, there was a further improvement to 3132. The running times required to generate these solutions were approximately 1 hour, 4 hours, and 17 hours, respectively.

The speed with which the approximation to the global optimum is found depends mainly on the random selection of the initial medians. If the hill at whose apex lies the approximation to the global optimum is selected during the first iteration, any subsequent iterations will not improve the solution already discovered. This is evident from the random sequence initiated by domain size tolerances of 10% between any two domains and 20% between the largest and smallest domain. In the network consisting of 1262 nodes, the approximation to the global optimum was found in subiteration 5 of iteration 1 in both the simulation consisting of 40 iterations as well as in the simulation consisting of 100 iterations. The final sum of distances recorded in this case was 3174. In this random sequence, the next improvement was observed in subiteration 4 of iteration 142, at which time the final sum of distances dropped to 3145. However, the random sequences initialized by domain tolerances of 5% between any two domains and 10% between the largest and smallest contained better solutions of 3137 and 3132 for 100 and 500 iterations, respectively.

The analysis of results for the other 3 networks showed a similar trend. Since the hills are not steep, there is no need to run the algorithm

through a large number of iterations in order to obtain a thorough sampling of the solution space. Instead, it may be more beneficial to examine a large number of initial random sequences by specifying several different domain size tolerances. No conclusions can be drawn regarding the domain size tolerances which should be imposed, because the optimal solution was found in different random sequences for the different networks.

It is interesting to note that in some cases, the final optimal sum of distances recorded became greater (less optimal) as the number of iterations increased. This is evident, for example, in the network containing 497 nodes with the random sequence initialized by domain size tolerances of 0% between any two domains and 0% between the largest and smallest domain. In this random sequence, the optimal solutions had a final sum of distances of 967, 975, and 969, for 40, 100, and 500 iterations, respectively. At first glance, this would indicate a flaw in the algorithm. However, when the sums of distances for the actual iteration, where the size domain restriction are imposed, are examined, it may be seen that the solution did in fact improve as the number of iterations increased. For that same random number sequence, the sums of distances were 1115, 1108, and 1096 for 40, 100, and 500 iterations, respectively. The fact that the sums of distances changed implies that different solutions were obtained when the domain size tolerances were imposed, and the solution improved constantly, subject to the imposed constraints. When these constraints were relaxed for the final assignment of nodes to their closest medians, the final solution which was obtained was in some cases worse than when the constraints were lifted from a suboptimal constrained solution.

This phenomenon was observed in the networks consisting of 76 nodes and 497 nodes, but not in the other two networks. Due to the small number

of networks and random sequences examined, no general comment can be made about the divergence from an optimal solution as the number of iterations increases. The phenomenon does not appear to be restricted to any specific random sequence.

The order in which the medians are selected does not appear to affect the final sum of distances. Simulations were run in which the same 8 medians selected in a suboptimal solution in the network containing 1262 nodes were used as fixed medians, but the order was changed for each simulation. In all cases, the final sum of distances remained the same, but the number of nodes assigned to each domain varied. This indicates that in cases where the distance from a node to several medians is the same, it will get assigned to the first such median. This may lead to large differences in the sizes of the various domains. A better algorithm, which would adjust the final sizes of the domains, could be used in future versions of the *MEDIAN* program.

9.5 Convergence to an Optimal Solution and Cycling

Tables 9.1 and 9.2 also list the total numbers of combinations examined before the approximation to the global optimal solution was found. These numbers provide a crude measure of the convergence to an optimal solution. For example, in searching for 8 medians in the network containing 1262 nodes, the number of combinations examined during 40 iterations was as low as 112 and as high as 409. In some cases the convergence to a stable solution was very fast and the local optimum was reached after as few as 2 subiterations. In other cases, the solution did not converge and one of the other stopping criteria had to be used to terminate an iteration.

One of the stopping criteria is that the solution being sought diverges a specific number of times during a single iteration. The feature had to be included, because as the domain boundaries are adjusted and new medians are calculated, in some cases a solution may temporarily diverge from a previous value and only subsequently converge to a better solution. However, in other cases the algorithm may become locked in a cycle where two or more solutions are examined repeatedly. From initial testing of the program, it was observed that if a solution converged after initially diverging, the number of subiterations where a solution was worse than on a previous subiteration was relatively low, and in all cases less than 10. Similarly, if cycling occurred, it became obvious after only a few subiterations that the program was caught in an infinite cycle. For that reason, an arbitrary limit was set to limit the number of times that intermediate solutions may diverge. There does not appear to be any pattern to the rapid convergence and cycling, and the number of subiterations required for a particular iteration depends exclusively on the initial medians selected at random to start the convergence process.

In examining the complete log files of iterations and subiterations, summarized in Tables 9.1 and 9.2, it is apparent that the final subiteration is not always the solution with the lowest (and hence optimal) sum of distances. The algorithm assumes that the solution has converged if the sum of distances for the current subiteration is the same as for the previous one, and hence the solution converges to a stable rather than an optimal value. The reason for this behaviour is that the boundaries for the domains change from subiteration to subiteration in accordance with the initial random medians selected and the size constraints imposed. It is therefore important to examine all subiterations and not merely the final subiteration of each iteration.

9.6 Random Sequences

Candidate medians are selected at the start of each iteration and the solution is improved during each subsequent subiteration. The random sequence by which these candidate medians are selected is based on a "seed" which initializes the random number generator. In the *MEDIAN* program, this seed is the sum of the domain size tolerances.

Table 9.3 shows the optimal locations of medians based on different random sequences. Initially, 40 iterations were used with 7 different random sequences, and the best random sequence was then allowed to continue through 500 iterations. The medians and the corresponding domain sizes are presented in Table 9.3 in the order that they were selected by the computer program. An additional simulation was carried out for each topology in which all medians were entered as "fixed" and the program established the domain sizes and calculated the sums of distances. The medians which were selected for these simulations, labeled as "best estimate", were selected after reviewing the results of all simulations. For example, *e0.bMER*, *e0.bCAR*, *e0.bCRK*, and *e0.bWDL* were selected as optimal medians in most of the simulations, and it is therefore highly probable that these are four of the true optimal medians. In the topologies where intercontinental arc lengths were longer than arc lengths between sites on the same continent, *bMDHc1* was selected as another median for the domain covering Europe. The locations of the remaining medians were not certain, but additional simulations established that *bRTPc1*, *nANNc?* and *bMERc?* were good candidates.

When comparing the sums of distances of the optimal medians selected for each of the random sequences, it is evident that there is a great variability in the near-optimality of the solutions found. This indicates that the

algorithm is very sensitive to the random sequence selected. The results also confirm an earlier hypothesis that it is better to run several random sequences of a small number of iterations rather than a few sequences of a large number of iterations. The near-optimal locations of medians were found after fewer than 40 iterations and in one case, there was absolutely no additional benefit derived from running 500 iterations.

9.7 Dragon Simulator Results

The final series of simulations consisted of evaluating the performance of the network when the numbers and locations of the data collectors varied. The Dragon network simulator was used for this purpose.

The topology data files read by the Dragon simulator were modified in accordance with the optimal locations of data collectors determined by the *MEDIAN* program. Since all of the optimal data collectors established by *MEDIAN* were either LANs or routers, fictitious data collector nodes (workstations) were created in the *nm2.no* and *nm2.sy* data files read by Dragon. These optimal data collectors were connected to the LAN which itself was supposed to be a data collector or which was connected to a router determined to be an optimal data collector. For example, if *e0.bCAR* was designated as a data collector, a workstation of data collector type was connected to *e0.bCAR* in the *nm2.no* data file. Similarly, if the data collector was supposed to be *bRTPc1*, the actual data collector node was connected to *e0.bRTP*, which was in turn connected to *bRTPc1*. Such a topological representation differs from the topology calculated to be optimal by the *MEDIAN* program since the sum of distances has been altered, and may imply that a different

topology, considered to be suboptimal by *MEDIAN*, might in fact be better. In view of the other modeling approximations made, this slight deviation from the optimal topology was not considered to be critical. If it is necessary to have precise correspondence between the topology calculated by *MEDIAN* and the topology read by the Dragon simulator, the *MEDIAN* algorithm would have to be modified so that only workstations would be considered as valid medians.

The Dragon simulator was run with the various network topologies and the number of data packets transferred after the first 10 minutes of simulated time was recorded. The results provided by Dragon consist of the number of packets transferred, the quantity of data transferred (in bytes), and the total number of hops taken by the packets as the data collectors monitor all of the devices in the network. In all cases, the effect of packet loss rate and the number of additional workstations was investigated. The packet loss rates were 2, 10, and 20, and the number of additional workstations varied from 0 to 1000. The results of the simulations are summarized in Tables 9.4 to 9.12.

Table 9.4 shows the performance of the network when the standard topology was used. The original *nm2.no* topology data file was used and the actual data collectors currently in use by BNR/NT were used to monitor the network devices. This was referred to as the "standard topology". Tables 9.5 and 9.6 show the network performance when the optimal data collector locations for topology types 1A and 1B, respectively, were used. Topology 1B (intercontinental arc lengths longer than arcs connecting different sites on the same continent) was used as the "optimal topology" for comparisons in Tables 9.7 to 9.12. Tables 9.7 to 9.9 show the network performance when fewer than 8 data collectors were used, and Tables 9.10 to 9.12 show the

network performance when more than 8 data collectors were used. In all tables, the hops per packet were calculated, and these numbers were compared with the hops per packet in the standard and optimal topologies.

When comparing the hops per packet in the standard and optimal network topologies, Tables 9.4 and 9.6, respectively, it is evident that the number of hops per packet required to monitor devices in the optimal topology is approximately 20 to 34% lower than in the standard network topology. This means that if the locations of the data collectors being used are changed, the monitoring traffic in the network could be reduced by that amount.

As the number of data collectors was reduced, the monitoring traffic, represented by the hops per packet, increased. A network which contains 7 instead of the usual 8 data collectors had nearly the same number of hops per packet as the optimal topology, and still had a 22 to 32% improvement over the standard topology. When the number of data collectors was reduced to 2, the hops per packet rate was nearly the same as in the standard topology with 8 data collectors, and 16 to 40% worse than in the optimal topology with 8 data collectors.

Conversely, as can be expected, the number of hops per packet was lower as the number of data collectors increased. Tables 9.10, 9.11, and 9.12 show the network traffic when 9, 10, and 15 data collectors, respectively, located at optimal sites throughout the network, were used. There was an 8 to 14% improvement over the network topology using only 8 data collectors at optimum locations and 25 to 42% improvement over the standard topology.

These results can be explained as follows. With the standard topology, all data collectors are in the Ottawa area. In order to monitor, for example, a workstation in Melbourne, a total of 11 hops are required for a one-way route to the nearest data collector located at Carling (*bCARh1* or *bCARh2*).

However, if a data collector is moved to $nANNc2$, only 7 hops are required, a ratio consistent with the results obtained from the Dragon simulator.

The hops per packet ratio is not greatly affected by the packet loss rate. This is probably due to the fact that if a packet is lost, it is retransmitted, the number of hops remains the same and the hops per packet ratio does not change. However, the hops per packet ratio increases considerably as the number of additional workstations is increased. This is due to the fact that the additional workstations are added randomly throughout the network. Since the topology changes, the location of the data collectors which was optimal for 0 additional workstations may no longer be optimal as new workstations are added. In order to maintain optimality, the *MEDIAN* algorithm would have to be rerun with the new topology, but this is not practical because the number and locations of workstations may change frequently. In the real BNR/NT network, the workstations are not monitored, and this is therefore not an issue.

With the results presented in this chapter, it has been shown that even with the limited topological information available, it is possible to select a better location of data collectors (medians) than the locations used at the present time. Such optimization translates into improved monitoring of the network, and, ultimately, increased productivity and higher profits for the organization using the network.

Table 9.4. The effect of data collector locations on packet transfers in the Dragon simulator using topology Type 1A and standard location of data collectors.

777 nodes, all arc lengths equal Location of data collectors not optimized. Data collectors: bCARh1 bCARh2 bCRKh1 bCRKh2 bMERh1 bMERh2 bKANh1 bWDLh1						
Workstations added	Packet loss rate	Packets transferred	Data transferred	No. of hops	Hops per packet	Hops/packet relative to std. topology
0	2	19,288	17,731,683	88,927	4.6105	1.0000
0	10	20,303	18,135,866	93,824	4.6212	1.0000
0	20	20,206	17,334,321	93,357	4.6203	1.0000
200	2	20,586	19,895,248	101,255	4.9186	1.0000
500	2	22,568	23,394,917	120,285	5.3299	1.0000
750	2	24,204	26,473,301	136,021	5.6198	1.0000
1,000	2	25,818	29,319,685	151,629	5.8730	1.0000

Table 9.5. The effect of data collector locations on packet transfers in the Dragon simulator using topology Type 1A and optimal location of data collectors.

777 nodes, all arc lengths equal Location of data collectors optimized. Data collectors: e0.bMER e0.bCAR e0.bCRK e0.bWDL bRTPc1 bMERc2 nANNc1 bCARc2						
Workstations added	Packet loss rate	Packets transferred	Data transferred	No. of hops	Hops per packet	Hops/packet relative to std. topology
0	2	19,224	12,137,934	61,087	3.1776	0.6892
0	10	20,428	12,686,972	65,067	3.1852	0.6893
0	20	20,581	12,327,157	65,583	3.1866	0.6897
200	2	20,522	14,433,108	73,033	3.5588	0.7235
500	2	22,504	17,996,169	91,373	4.0603	0.7618
750	2	24,140	20,956,451	106,587	4.4154	0.7857
1,000	2	25,754	23,845,720	121,457	4.7160	0.8030

Table 9.6. The effect of data collector locations on packet transfers in the Dragon simulator using topology Type 1B and optimal location of data collectors.

777 nodes, arc lengths vary Location of data collectors optimized. Data collectors: e0.bMER e0.bCAR e0.bCRK e0.bWDL bRTPc1 bMERc5 nANNc2 bMDHc1						
Workstations added	Packet loss rate	Packets transferred	Data transferred	No. of hops	Hops per packet	Hops/packet relative to std. topology
0	2	19,232	11,807,754	59,493	3.0934	0.6710
0	10	19,995	11,672,253	61,882	3.0949	0.6897
0	20	20,756	12,737,466	64,353	3.1005	0.6711
200	2	20,530	14,115,082	71,437	3.4798	0.7074
500	2	22,512	17,721,707	89,883	3.9971	0.7489
750	2	24,148	20,733,344	105,455	4.3670	0.7771
1,000	2	25,762	23,653,650	120,467	4.6762	0.7982

Table 9.7. Packet transfer rates in the Dragon simulator using topology Type 1B and 7 optimal data collectors.

777 nodes, arc lengths vary Optimal location of 7 data collectors: e0.bMER e0 bCAR e0 bCRK bRTPc1 bMDHc1 bCARc2 nMTVc2							
Workstations added	Packet loss rate	Packets transferred	Data transferred	No of hops	Hops per packet	Hops/packet relative to standard top	Hops/packet relative to optimal top
0	2	22,734	13,793,693	71,711	3 1544	0 6842	1 0197
0	10	20,551	12,544,412	64,933	3 1562	0 6830	1 0198
0	20	20,873	12,228,937	65,930	3 1586	0 6836	1 0188
200	2	24,094	16,195,391	84,291	3 4984	0 7113	1 0054
500	2	26,184	19,961,760	103,699	3 9604	0 7431	0 9908
750	2	27,940	23,066,711	120,085	4 2980	0 7648	0 9842
1,000	2	29,652	26,164,039	135,941	4 5845	0 7806	0 9804

Table 9.8. Packet transfer rates in the Dragon simulator using topology Type 1B and 6 optimal data collectors.

777 nodes, arc lengths vary Optimal location of 6 data collectors e0.bMER e0 bCAR e0 bCRK e0 bWDL bRTPc1 bMDHc1							
Workstations added	Packet loss rate	Packets transferred	Data transferred	No of hops	Hops per packet	Hops/packet relative to standard top	Hops/packet relative to optimal top
0	2	22,734	16,612,203	86,193	3 7914	0 8223	1 2256
0	10	21,106	15,809,283	79,944	3 7877	0 8196	1 2239
0	20	20,736	14,806,747	78,507	3 7860	0 8194	1 2211
200	2	24,094	18,971,099	98,443	4 0858	0 8307	1 1742
500	2	26,184	22,764,858	118,053	4 5086	0 8459	1 1280
750	2	27,940	25,872,242	134,349	4 8085	0 8556	1 1011
1,000	2	29,652	28,945,111	150,259	5 0674	0 8628	1 0837

Table 9.9. Packet transfer rates in the Dragon simulator using topology Type 1B and 2 optimal data collectors.

777 nodes, arc lengths vary Optimal location of 2 data collectors e0 bMER bCARc4							
Workstations added	Packet loss rate	Packets transferred	Data transferred	No of hops	Hops per packet	Hops/packet relative to standard top	Hops/packet relative to optimal top
0	2	22,734	19,228,328	100,025	4 4000	0 9543	1 4224
0	10	20,437	17,524,658	89,758	4 3919	0 9504	1 4191
0	20	20,740	16,863,425	91,074	4 3912	0 9504	1 4163
200	2	24,094	21,486,172	111,839	4 6418	0 9437	1 3340
500	2	26,184	25,120,169	130,597	4 9877	0 9358	1 2478
750	2	27,940	28,104,075	146,219	5 2333	0 9312	1 1984
1,000	2	29,652	31,152,539	161,673	5 4523	0 9284	1 1660

Table 9.10 Packet transfer rates in the Dragon simulator using topology Type 1B and 9 optimal data collectors.

777 nodes, arc lengths vary Optimal location of 9 data collectors: e0 bMER e0 bCAR e0 bCRK e0 bWDL bMDHc1 bRTPc1 bMERc5 nANNc2 nMTVc1							
Workstations added	Packet loss rate	Packets transferred	Data transferred	No. of hops	Hops per packet	Hops/packet relative to standard top.	Hops/packet relative to optimal top.
0	2	19,224	10,912,492	54,875	2.8545	0.6191	0.9228
0	10	20,705	11,739,912	59,175	2.8580	0.6185	0.9235
0	20	20,858	11,594,830	59,669	2.8607	0.6192	0.9227
200	2	20,522	13,165,519	66,601	3.2453	0.6598	0.9327
500	2	22,504	16,666,100	64,603	3.7595	0.7054	0.9405
750	2	24,140	19,554,083	99,483	4.1211	0.7333	0.9437
1,000	2	25,754	22,390,991	114,041	4.4281	0.7540	0.9470

Table 9.11 Packet transfer rates in the Dragon simulator using topology Type 1B and 10 optimal data collectors.

777 nodes, arc lengths vary Optimal location of 10 data collectors: e0 bCAR e0 bCRK e0 bMER e0 bWDL bMERc5 bMDHc1 bRTPc1 nANNc1 nBPDc1 nMTVc1							
Workstations added	Packet loss rate	Packets transferred	Data transferred	No. of hops	Hops per packet	Hops/packet relative to standard top.	Hops/packet relative to optimal top.
0	2	19,224	10,574,362	53,119	2.7632	0.5993	0.8932
0	10	20,644	11,076,672	57,107	2.7663	0.5986	0.8938
0	20	20,436	10,635,819	56,579	2.7686	0.5992	0.8930
200	2	20,522	12,901,917	65,249	3.1795	0.6464	0.9137
500	2	22,504	16,543,667	83,981	3.7318	0.7002	0.9336
750	2	24,140	19,515,218	99,241	4.1111	0.7315	0.9414
1,000	2	25,574	22,454,300	114,387	4.4728	0.7616	0.9565

Table 9.12 Packet transfer rates in the Dragon simulator using topology Type 1B and 15 optimal data collectors.

777 nodes, arc lengths vary Optimal location of 15 data collectors: e0 bCAR e0 bCRK e0 bKAN e0 bMER e0 bWDL bCARc4 bMERc3 bMERc5 bATPc1 bMDHc1 bRTPc1 nANNc1 nBPDc1 nMTHc1 nMTVc2							
Workstations added	Packet loss rate	Packets transferred	Data transferred	No. of hops	Hops per packet	Hops/packet relative to standard top.	Hops/packet relative to optimal top.
0	2	19,224	10,241,696	51,453	2.6765	0.5805	0.8652
0	10	20,380	10,843,269	54,540	2.6762	0.5791	0.8647
0	20	20,781	10,916,095	55,764	2.6834	0.5808	0.8655
200	2	20,522	12,551,791	63,497	3.0941	0.6291	0.8892
500	2	22,504	16,199,676	82,245	3.6547	0.6857	0.9143
750	2	24,140	19,190,933	97,611	4.0435	0.7195	0.9259
1,000	2	25,754	22,162,678	112,883	4.3831	0.7463	0.9373

Chapter 10

Searching for Shortest Paths

The logic and implementation of the program which finds the n shortest paths between two nodes in a network was described in Chapter 7. This chapter describes how *NPATH* was used and discusses the results obtained.

10.1 Fully Connected Networks

The first series of simulations was carried out on fully connected networks ranging in size from 10 nodes to 100 nodes. Fully connected networks are networks where every node is connected to every other node. This means that a network with n nodes has $\frac{1}{2}(n^2-n)$ arcs, and each node has $(n-1)$ arcs connected to it. The number of paths which exist between any two nodes is

$$\text{number of paths} = \sum_{i=0}^{n-2} \left(\frac{(n-2)!}{(n-2-i)!} \right)$$

where n is the number of nodes in the network and i is the number of intermediate hops in the path. Each term of the summation represents the number of paths between the origin and the destination which pass through i intermediate nodes. Hence, when $i = 0$, the origin and destination is connected directly, and only one possible path exists. There are $(n-2)$ possible paths for 2 hops, $(n-2)(n-3)$ paths for 3 hops, and eventually $(n-2)!$ paths passing through all of the nodes. A fully connected network with only 10 nodes has a total of 109,601 paths between any two nodes. The above

equation shows that the number of paths increases factorially as the number of nodes increases.

The *NPATH* algorithm finds the shortest path between any two nodes and builds a ranked list of new candidate paths obtained by single-node perturbations from a path currently under investigation. The number of candidate paths which are added to the list as each node in the current path is examined depends on the number of nearest neighbours of that particular node. The perturbations from every node on a given path must be examined before the search can proceed to the next path in the list. The search time therefore depends more on the average number of arcs which leave each node than on the number of nodes in the network.

This hypothesis was tested by several simulations. Figure 10.1 shows the search times for finding the shortest paths in fully connected networks. All simulations were run on the same workstation with no other processes (except those associated with the Sunview environment) running in the background. As the diagram illustrates, 200 paths can be calculated in less than 30 seconds in a fully connected network consisting of 10 nodes. However, even the calculation of the second-shortest path required 12 minutes in a fully connected network consisting of 100 nodes. Simulations with larger fully connected networks were not carried out since the trend was established with the results obtained from the 6 networks examined.

In all fully connected networks, the destination could be reached from the origin by a single hop, and this was the shortest path. The next shortest path consisted of 2 hops, and there were $(n-2)$ of these. This means that in the 10-node network, *NPATH* only had to process 9 paths (1 single-hop and 8 two-hop paths) before the second-shortest path was determined, whereas 99 paths had to be processed in the 100-node network.

Search times for shortest paths in fully connected networks

Figure 10.1.

It is also interesting to note that the search times for finding the n shortest paths are linear, with the slope of the line being the number of paths which can be found in a unit of time. In a fully connected network, this relationship is understandable, because the number of perturbations which exist at every node is constant and the amount of work is therefore also constant. As will be seen in the following sections, this relationship was also present in the BNR/NT network where the number of arcs leaving each node was not constant. In real-world mesh networks the interconnections are sufficiently random that the search time is determined by the *average* number of arcs leaving each node.

10.2 Complete Paths in the BNR/NT Network

Fully connected networks are rarely used in practice and therefore results obtained from the topologies of such networks have limited practical applications. However, the search times established with fully connected networks set the upper limit for the search times for paths in real networks of comparable sizes. Fortunately, the actual search times are much lower than the theoretical limit.

The search times for finding the n shortest paths through the BNR/NT network are presented in Figure 10.2. Simulations were carried out with two network topologies. One network, Topology 1, consisted of 777 nodes and 1135 arcs of which 473 nodes (e.g. workstations) had a single incident arc. The other network, Topology 2, consisted of 1262 nodes and 1620 arcs of which 862 nodes had a single incident arc.

Search times for shortest paths in BNR/NT network

Figure 10.2

The paths which were studied connected node *bCARa175*, a workstation at the Carling site in Ottawa, with *e0.nMEL*, a LAN in Melbourne, Australia. This path was selected because it connects devices on opposite sides of the world and several aspects of typical networks could be studied. First, the Carling site is very complex and due to the large number of interconnections, a large number of paths exists at the Carling site proper. Second, Carling is connected to a large number of other sites, so that the numerous paths from Carling to Mountainview, California, the gateway to Australia, have only a small number of cross-connections. Third, only a single path exists from Mountainview to Melbourne, so that this part of the path will be common to all paths found.

The topology for the Carling site is shown in Figure 10.3. It has been simplified to the extent that parallel paths, e.g. through bridges, have been grouped together into a single "box", and connections to individual workstations have not been shown, only the number of workstations connected to each LAN has been provided. The only difference between Topology 1 (777 nodes) and Topology 2 (1262 nodes) is the number of workstations present in the network, otherwise the topologies of the other devices are identical.

The lengths which were assigned to the arcs were 3 for router-router links and 1 for all other connections. This forced the shortest paths through LANs connected to the routers rather than through the direct router connections and corresponded to the manner in which the Dragon simulator routes traffic. The shortest path was

bCARa175 - *e0.bCAR* - *bCARc4* - *nTORc1* - *nMTHc1* - *nANNc1* - *e0.nANN* - *nANNc2* - *nMTVc3* - *e0.nMTV* - *nMTVc1* - *nMELc1* - *e0.nMEL*

and was identical to the path established by explorer packets sent through the Dragon simulator.

Since the paths pass through several LANs, the number of workstations present in the network affects the speed with which a solution is obtained. This is due to the fact that as the algorithm searches for a candidate path, dead-end paths which end at the workstations connected to the intermediate node LANs are considered before they are discarded. For that reason, there is a large difference in the search times for the same numbers of shortest path for the two network topologies. Had the paths passed only through routers, the difference would be less pronounced.

What is more important is the fact that the direction of the path is important. For example, in the 777-node network, it takes 9 minutes to find the 20 shortest paths from *bCARa175* to *e0.nMEL*, but the same number of paths, calculated in the opposite direction, is found in only 2.75 minutes. This difference is caused by the large number of arcs leaving the nodes at one end of the path (at Carling) and the relatively small number at the other end (Melbourne). If the paths are sought from the Carling end, a large number of candidate paths, which will not be part of the final solution set but which must themselves be examined, will be generated by second and higher order perturbations (i.e., perturbations of perturbations). If the paths are sought from the Melbourne end, there are initially smaller numbers perturbations and this number increases only when the search proceeds into the Carling site.

Due to the arc length restrictions imposed, there exist groups of large numbers of paths of equal lengths. For example, there are 4 paths of length 22, 6 paths of length 23, more than 90 paths of length 24, etc. Since the first 20 paths include only some of the paths having length 24, the paths listed will differ depending on the direction in which the search proceeds.

Figure 10.3. Detailed topology of the Carling site.

10.3 Partial Paths and Node Compression

Figure 10.2 shows that several likely paths where a fault might have occurred can be generated in a relatively short period of time in the network topologies being studied. However, the real BNR/NT network has approximately 40,000 devices, and it is quite likely that the generation of candidate paths for diagnostic purposes would be unacceptably slow. The problem must therefore be simplified so that the concepts and algorithms presented in this thesis can still be used within the time constraints imposed by the real world.

The problem can be simplified by incrementally looking at segments of the most likely paths. In the example presented in the previous section, such segments might be *bCARa175* to *bCARc4*, *bCARc4* to *nMTVc3*, and *nMTVc3* to *e0.nMEL*. However, from simulations carried out, it was observed that searching for *n* shortest paths between *bCARa175* and *bCARc4* required almost as much time as searching for the same number of full paths, and hence there was no improvement in the performance. In fact, this approach was less efficient because two segments of the path had to be considered.

Simplifying the problem by reducing the size of the network would be another approach which could be tried. The entire network could be divided into individual domains (e.g. with the *MEDIAN* program), and network monitoring could proceed independently and in parallel for each domain. This approach is analogous to the manner in which air traffic controllers operate. However, the main drawback is that searching for shortest paths within a single domain without examining the trans-domain effects might not produce the desired results. If this approach was used in the above example, the geographic zones as presented in Figure 8.1 might be used as the domains. When viewed from a higher level in the topology hierarchy, the

global shortest path as specified above would be passed to each of the domains through which the path passes, and the tasks would be to find the n shortest paths between the entry and exit points of the domain. Hence, the Central Canada domain would examine all paths terminating at *nMTHc1* which would be connected to *nANNc1* in another domain.

The problem with this segmented approach is that although the first 10 shortest global paths (assuming the paths through Carling remain constant) do in fact follow the same paths through *nMTHc1*, the eleventh path passes through *bRTPc1*, the paths then oscillate between these two main routes until path 102, which passes through *bMDHc2*. It would therefore be difficult to blindly segregate the entire network into domains, perform a local analysis, and hope to obtain the correct results.

A much better approach is to examine the network as a whole and selectively discard those portions which are not required at any given time. This represents a hierarchical approach, because the entire network can be modeled by a small number of supernodes which can be selectively expanded and compressed as required.

This approach was demonstrated by searching for the n shortest paths between *bCARa175* and *e0.nMEL* in two stages. In one stage, all nodes except the Carling node were compressed into single supernodes corresponding to the characteristic community strings. The entire network consisted of 75 supernodes and 168 nodes at the Carling site. The shortest paths between *bCARa175* and *bCARc4* were sought and the search times were recorded.

In the second stage, the Carling site was compressed into a single supernode and the rest of the nodes in the network were restored to their original decompressed topology. The shortest paths between *CAR*, a supernode, and *e0.nMEL* were sought and the search times were recorded. The search

times for each stage independently were approximately the same (e.g. 50 seconds to find 20 paths between *bCARa175* and *bCARc4*, and 51 seconds to find 20 paths between *CAR* and *e0.nMEL*). The search times for both stages were added together and the total search time is shown by the dashed line in Figure 10.2. It should be noted that this method does not provide the n global shortest paths, because the first stage assumes the shortest path outside of Carling to be fixed and the second stage assumes the shortest path within Carling to be fixed when in fact there will be interactions between the two path segments. However, the fact that $2n-1$ paths are generated should offset this deficiency.

Furthermore, the results are obtained much more rapidly than with any other path search in the full network, and this demonstrates the importance of selective compression of network nodes. With the Carling node compressed, 500 shortest paths between *CAR* and *e0.nMEL* were generated in approximately 23 minutes, which is nearly the same time required to generate 50 paths from *bCARa175* to *e0.nMEL* in the full network. In this case, therefore, the search in the compressed network is 10 times faster. When taken to extremes, all 2049 paths from *CAR* to *MEL* in a completely compressed network (76 supernodes as shown in Figure 8.1) were generated in approximately 6 minutes.

Judicious use of selective node compression therefore reduces the search time required to find the n shortest paths through a network. This method can be successfully used even in large networks to find multiple paths within a reasonable time.

10.4 Correlation with the Dragon Simulator

After the shortest paths between *bCARa175* and *e0.nMEL* had been generated using *NPATH*, the Dragon simulator was run with the corresponding *nm2.no* data file describing the topology of a network with 777 nodes. Various devices and links were broken along the paths established by explorer packets sent through the simulator and the new paths obtained by explorer packets were compared with the list of shortest paths generated by *NPATH*. The shortest path with no devices and no links broken corresponded exactly to the shortest path calculated by *NPATH*. As already noted, there are four "shortest" paths of length 22.

It was interesting to note that although the routing algorithm in the Dragon simulator has no problems in choosing paths passing through LANs prefixed by *e0*, no path was selected through a LAN prefixed by *e1*. Several devices and/or links were broken in an attempt to force the path through some *e1* LAN, but instead the simulator selected other paths of equal or longer length. This was due to the nature of the Dragon simulator routing logic, selected so that it was similar to the most common types of commercial router logic. The comparison of the behaviour of *NPATH* with the Dragon simulator was therefore inconclusive. The discrepancies between the paths other than the shortest path generated by the Dragon simulator and *NPATH* are caused by the different algorithms used to compute the best paths, and the paths generated by the two programs cannot be compared.

10.5 Use of *NPATH* by other Knowledge Sources

The need for the development of the multiple paths generator first became apparent in research carried out by Dussault [12]. His prototype Expert Network Diagnostician (*END*) used an expert system to diagnose network faults. One of the faults which can be detected by *END* is a routing problem where a user at station A cannot reach station B. This could be caused by any number of factors of which broken devices, stuck bits in an Ethernet address decoder, and data storms are only a few examples. The present version of *END* considers only the most probable path to verify different elements in the path, but is unable to check other paths. The development of *NPATH* now permits *END* to examine multiple paths and solve a larger set of problems.

Farrell is examining the possibility of using network anomalies to diagnose network faults [13]. Such anomalies generally occur near faulty devices, and it is therefore necessary to construct a ranked list of devices which should be checked. As in Dussault's work, the network fault generally manifests itself when communication difficulties between two nodes occur. Since *NPATH* provides ranked lists of nodes and arcs in order of descending frequency with which they occur in the n shortest paths, *NPATH* can be used to provide the required information for Farrell's programs [13].

Since dynamic routing presents a challenge in establishing the precise path through a network during the occurrence of a fault, some mechanism must exist which would determine a set of the most likely paths. *NPATH* and the utilities associated with it are therefore an important tool which can be used for this purpose.

Chapter 11

Conclusions and Recommendations

11.1 Conclusions

The objectives of this thesis were to apply operations research techniques to the development of tools used by the Pegasus multi-paradigm approach in network fault diagnosis. These objectives were achieved by studying node compression as a means of simplifying network topologies, optimal location of data collectors as a means of reducing traffic required to monitor the network, and generation of n shortest paths as a means of rapidly identifying a faulty device or line. The following conclusions have been reached:

1. Algorithms to compress several network nodes into a smaller number of supernodes in order to simplify the network topology, and restore (decompress) supernodes into their original constituent parts have been developed and tested, and appear to be an efficient way of simplifying network topologies.
2. The compression algorithms were used to obtain a simplified hierarchical view of the network topology and list the devices located at various physical sites. Such a hierarchical view of a network topology is important particularly in large and complex networks as it hides details which are not required at any given point in time.

3. The location of the present data collectors was examined and optimal locations were determined. By comparing alternate locations of the data collectors, it was observed that monitoring traffic within the sample network studied could be reduced by approximately 30%. Such a reduction implies that the cost of the network is reduced, since the excess capacity released in this manner can be utilized by "paying customers" rather than network overhead. In real-world networks, the actual savings which would be realized would depend on the ratio of the monitoring traffic to the total network traffic.
4. The *MEDIAN* program which was developed to find the optimal locations of data collectors in the network can also be used to study the effect of locating data collectors in suboptimal locations. This is important in the design of new networks.
5. With the *MEDIAN* program, the effect of using other numbers of data collectors to monitor the network can also be investigated and the data can be used in a cost-benefit analysis to determine the optimal number of data collectors for a given level of service.
6. The *MEDIAN* program divides the network into several domains, each of which is served by a data collector. If it is not practical to search for faults in the full-size network, current diagnostic techniques can be applied to each domain separately and the diagnosis can proceed in parallel. Such parallel processing and simplified network topologies may result in faster detection of faults in very large and complex networks.
7. The *NPATH* program and associated functions have been developed to find the n shortest paths through the network. This is particularly important to locate the set of most probable paths in networks where dynamic routing is used. *NPATH* provides a tool by which network

managers can examine a number of candidate paths to search for possible faults.

8. The speed with which *NPATH* calculates the n shortest paths depends on the complexity of the network. When the node compression program is used to selectively compress the portions of the network which are not of interest and thereby simplifies the network topology, the desired n shortest paths can be calculated in a fraction of the time which would otherwise be required. Such savings imply that *NPATH* and the node compression program can be efficiently used to rapidly assist network managers in locating faults even in very large and complex networks.
9. *NPATH* has been linked to the other Pegasus diagnostic modules via the Blackboard. This means that Dussault's Expert Network Diagnostician (END) [12], which requires the correct path where the fault occurred as input in order to diagnose the fault, can be made more versatile since it can be applied to problems where the actual faulty path is not known with 100% certainty.
10. In addition to providing the n shortest paths, *NPATH* also provides ranked lists of the most critical nodes and arcs which occur along those paths. This information can be used by other Pegasus modules, for example the module currently being developed by Farrell [13], which searches for faults by examining anomalies around suspected faulty devices. *NPATH* can provide the candidate faulty devices as input for Farrell's computer programs.
11. Although the BNR/NT network was used as a test bed for the algorithms developed in this thesis, the methods and implementations are general and can be applied to any network.

The results obtained in this thesis demonstrate that operations research techniques can be used to develop valuable tools to assist network managers with the detection and correction of faults in networks. This means that fully automated network fault diagnosis, the ultimate goal of the Pegasus project, may be one step closer to realization.

11.2 Recommendations

The programs developed in this thesis demonstrate the effectiveness of operations research techniques as an integral part of a multi-paradigm approach to network fault diagnosis. However, there are still areas where additional research is required. These are as follows:

1. The lengths of arcs used by *MGEN*, *MEDIAN* and *NPATH* were selected arbitrarily and the model of the network may not be accurate. More realistic arc lengths, which would take into consideration the importance, capacity, transfer rate, cost, and other relevant factors associated with a particular arc, should be used to represent the actual network being modeled more accurately.
2. In some cases, the programs require a long time to perform the required computations. Further work may be required to optimize the algorithms so that they are more efficient, i.e., so that they run faster, require less computer memory, and are easier to use.
3. The *MGEN* program which generates the distance matrix for the *MEDIAN* program repeatedly uses Dijkstra's shortest path algorithm [10]. More efficient methods for finding the shortest distances between

all pairs of nodes in a network exist, and *MGEN* should be rewritten to make use of these algorithms.

4. Although arrays have been successfully used in all of the programs developed, the fixed array sizes may present limitations if the methodology developed in this thesis is to be applied to larger networks. In that case, the computer programs may need to be rewritten so that only dynamic data structures are used.
5. Although priority queues implemented as heaps appear to require more computations when adding or removing items than linked list implementations, results in literature show that shortest path algorithms using heaps are in fact faster than those using linked lists. The performance of functions using priority queues implemented as linked lists in this thesis can be improved if a binary heap is used instead.
6. The programs which have been developed as part of this thesis read the network topology from special data files. It would be useful if these programs read the information directly from the data files used by other components of the Pegasus system in order to ensure that the network topology is consistent for all modules at all times.

The tools which have been developed as part of this thesis have been successfully used to calculate the optimal locations of multiple medians, a set of n shortest paths, and to simplify network topologies as needed. The output generated by these tools has been used by other modules of the Pegasus diagnostic system to make network fault detection and diagnosis more efficient. The objectives stated for this research work have therefore been successfully satisfied.

Glossary of Selected Terms

Refer to Section 4.1, *Basic Definitions*, for definitions of terms used in graph theory.

arc	connection between two nodes which represents communication lines in a communication network; also called an edge or a link.
BANES	diagnostic program which generates hypotheses about the status of network elements from status messages provided by data collectors [8].
data collector	special device which monitors other network devices and reports their status to a network operation center.
domain	subset of a network which contains a single median and its closest nodes.
Dragon	network simulator used for testing various components of the Pegasus system [7].
END	rule-based knowledge source used to verify hypotheses formulated by BANES [10].
graph theory	branch of mathematics which deals with solving problems formulated as networks.
heap	dynamic data structure, usually represented as a balanced binary tree, where data is organized with smallest or largest item at the top of the heap.
LEAPR	neural network knowledge source used to detect performance degradation or abnormal traffic in a network [35].
linked list	dynamic data structure where data is stored as a chain of items which may or may not be sorted.
Magnet	program used by BNR/NT to monitor their communication network by means of a hierarchical representation of devices and connections between them.
MEDIAN	program which calculates the optimal location of data collectors in a network.

median	network node located so that the sum of distances to all other nodes in its domain is minimized.
MGEN	preprocessor for MEDIAN which generates the matrix of shortest distances between every pair of nodes in a network.
network	also called a graph – collection of nodes and arcs used to model certain types of problems; <i>communication network</i> is a set of devices and lines used to transfer data.
node	junction of two or more arcs in a network which represents devices in communication networks; also called a vertex.
NPATH	program which finds the n shortest paths through a network.
Pegasus	multiple paradigm system developed at Carleton University for network fault management [9].
PING	(Packet Internet Groper) program used to analyze network problems.
queue	dynamic data structure organized so that items are removed from the front (head) and inserted at the end (tail); <i>priority queue</i> contains sorted items.
SNMP	(Simple Network Management Protocol) a widely-used management protocol for the Internet network.
supernode	single node into which a larger number of network nodes have been compressed to simplify the network topology.
topology	physical layout of a network showing how nodes are connected.
tree	special type of graph which does not contain any loops; dynamic data structure where data is organized so that <i>left child < root < right child</i> (binary search tree).

References

1. BELLMAN, R. and R. KALABA: *On k^{th} best policies*. Journal of the Society for Industrial and Applied Mathematics, **8**, 582-588 (1960).
2. BIENSTOCK, Dan and Odile MARCOTTE: *On a network design problem that is intractable on trees*. Mathematics of Operations Research, **15** (3), 530-544 (1990).
3. BNR Information Technology: *BNR Local Area Network Environment: An Introduction for the Network User*, 1990, 35 p.
4. CHEN, Y. L. and Y. H. CHIN: *The quickest path problem*. Computers and Operations Research, **17** (2), 153-161 (1990).
5. CORNUEJOIS, G., M. L. FISHER, and G. L. NEMHAUSER: *Location of bank accounts to optimize float: An analytic study of exact and approximate algorithms*. Management Science, **23**, 789-810 (1977).
6. DANTZIG, G. B.: *All shortest routes in a graph*. Technical Report 66-3, Operations Research House, Stanford University (1966). IN DREYFUS, Stuart E. *An appraisal of some shortest-path algorithms*. Operations Research, **17**, 395-412 (1969).
7. DAWES, N. W.: *Dragon - A very high speed, very large scale network simulator*. Technical Report SCE-91-31, Dept. of Systems and Computer Engineering, Carleton University, 1991.
8. DAWES, N. W., J. MUTOFT, and B. PAGUREK: *Network diagnosis by reasoning in uncertain nested evidence spaces*. Report SCE-91-26, Dept. of Systems and Computer Engineering, Carleton University, 1991.
9. DAWES, N. W., R. KAYE, and B. PAGUREK: *PEGASUS Demonstration*. Dept. of Systems and Computer Engineering, Carleton University, 1992.
10. DIJKSTRA, E. W.: *A note on two problems in connection with graphs*. Numerische Mathematik, **1**, 269-271 (1959). IN DREYFUS, Stuart E. *An appraisal of some shortest-path algorithms*. Operations Research, **17**, 395-412 (1969).
11. DREYFUS, Stuart E.: *An appraisal of some shortest-path algorithms*. Operations Research, **17**, 395-412 (1969).
12. DUSSAULT, R.: *A diagnostic expert system for wide area networks*. Master of Engineering Thesis, Dept. of Systems and Computer Engineering, Carleton University 1992.
13. FARRELL, Alan: *Detection of abnormal router behaviour*. Master of Engineering Thesis, Dept. of Systems and Computer Engineering, Carleton University, in preparation.

- 14 FLOYD, R. W. *Algorithm 97, Shortest path*. Communications of the ACM, **5**, 345 (1962).
- 15 FREDERICKSON, Greg N.: *Fast algorithms for shortest paths in planar graphs*. SIAM Journal on Computing, **16** (6), 1004-1022 (1987).
- 16 FREDERICKSON, Greg N. and Ravi JANARDAN: *Efficient message routing in planar networks*. SIAM Journal on Computing, **18** (4), 843-857 (1989).
- 17 GOLDMAN, A. J. *Optimal center location in simple networks*. Transportation Science, **5**, pp. 212-221 (1971).
- 18 HAKIMI, S. L.: *Optimum locations of switching centers and the absolute centers and medians of a graph*. Operations Research, **12**, 450-459 (1964).
- 19 HANDLER, Gabriel Y. and Pitu B. MIRCHANDANI: *Location on Networks: Theory and Algorithms*. The MIT Press, Cambridge, Mass., 1979, 233 p.
- 20 HOFFMAN, Walter and Richard PAVLEY: *A method for the solution of the Nth best path problem*. Journal of the ACM, **6** (4), 506-514 (1959).
- 21 HUNG, Ming S. and James J. DIVOKY: *A computational study of efficient shortest path algorithms*. Computers and Operations Research, **15** (6), 567-576 (1988).
- 22 IQNEIBI, Sami: *A blackboard architecture to support network fault diagnosis*. Master of Engineering Thesis, Dept. of Systems and Computer Engineering, Carleton University, 1992.
- 23 KARIV, O. and S. L. HAKIMI: *An algorithmic approach to network location problems. Part 2: The p-medians*. SIAM Journal of Applied Mathematics, **37**, 539-560 (1979).
- 24 KUEHN, A. A. and M. J. HAMBURGER: *A heuristic program for locating warehouses*. Management Science, **9**, 643-666 (1963).
- 25 MARANZANA, F. E.: *On the location of supply points to minimize transport costs*. Operational Research Quarterly, **15**, 261-270 (1964). IN: HANDLER, Gabriel Y. and Pitu B. MIRCHANDANI: *Location on Networks: Theory and Algorithms*. The MIT Press, Cambridge, Mass., 1979, 233 p.
- 26 MOFFAT, Alistair and Tadao TAKAOKA: *An all pairs shortest path algorithm with expected time $O(n^2 \log n)$* . SIAM Journal on Computing, **16** (6), 1023-1031 (1987).
- 27 MONDOU, Jean-François, Theodor Gabriel CRAINIC and Sang NGUYEN: *Shortest path algorithms: A computational study with the C programming language*. Computers and Operations Research, **18** (8), 767-786 (1991).
- 28 PAGUREK, B., N. DAWES and A. KAYE: *A multiple paradigm diagnostic system for wide area communication networks*. Technical Report SCE-91-37, Dept. of Systems and Computer Engineering, Carleton University, 1991.
- 29 POLLACK, Maurice: *The k^{th} best route through a network*. Operations Research, **9** (4), 578-580 (1961).

30. POLLACK, Maurice: *Solutions of the k^{th} best route through a network - a review* Journal of Mathematical Analysis and Applications, **3**, 547-559 (1961)
31. ROSEN, J. B., S.-Z. SUN and G.-L. XUE: *Algorithms for the quickest path problem and the enumeration of quickest paths* Computers and Operations Research, **18** (6), 579-584 (1991).
32. SHERALI, Hanif D.: *On the equivalence between some shortest path algorithms* Operations Research Letters, **10**, 61-65 (1991).
33. TANSEL, Barbaros C., Richard L. FRANCIS, and Timothy J. LOWE: *Location on networks: A survey. Part I: The p-center and p-median problems* Management Science, **29** (4), 482-497 (1983).
34. TANSEL, Barbaros C., Richard L. FRANCIS, and Timothy J. LOWE: *Location on networks: A survey. Part II: Exploiting tree network structure* Management Science, **29** (4), 498-511 (1983)
35. VIENS, François: *A neural network approach to detect traffic anomalies in a communication network*, Master of Engineering Thesis, Dept. of Systems and Computer Engineering, Carleton University, 1992.
36. WINSTON, Wayne L.: *Operations Research: Applications and Algorithms* PWS-Kent Publishing Company, Boston, 1987, 1025 p
37. YUAN, Bin: *PegasusBase - A run-time schema extendible system* Master of Engineering Thesis, Dept. of Systems and Computer Engineering, Carleton University, 1992
38. ZANAKIS, Stelios H., James R. EVANS and Alkis A. VAZACIOPOULOS: *Heuristic methods and applications: A categorized survey* European Journal of Operations Research, **43**, 88-110 (1989)
39. ZARY, Lyle: *Case-based reasoning for network fault management* Master of Engineering Thesis, Dept. of Systems and Computer Engineering, Carleton University, in preparation.

Appendix A

Data Structure Definitions – bb_pkttype.h

```

#define MALLOC(type) ((type *)malloc(sizeof(type))) /* Macro for allocating memory */

#define BUFSIZE 2048          /* buffer size for topology files      */
#define NAMELEN 12            /* length of device name          */

#define A_SIZE 1400           /* maximum size of array          */
#define LARGENUM 10000000     /* initialization parameter       */
#define LARGEBUFSIZE 8192     /* buffer for reading distance matrix */
#define COMPR_LINK 7          /* arc weight to any compressed node */
#define RT_RT_LINK 3         /* 'router-router arc length      */
#define NORMAL_LINK 1         /* length for all other arcs in network */

typedef char file_spec[30];   /* char string for file names      */
typedef char name_t[NAMELEN]; /* char string for device names    */
typedef char type_t[5];       /* char string for device types    */

/*****
***** DYNAMIC DATA STRUCTURE DEFINITIONS *****/

/* Node definition for linked lists storing device names. */
typedef struct list_def {
    name_t name;
    struct list_def *next;
} list_t;

/* Node definition for binary search tree. */
typedef struct tree_def {
    name_t name;
    type_t type;
    list_t *list;
    struct tree_def *left;
    struct tree_def *right;
} tree_t;

/* Node definitions for linked lists storing array index. */
typedef struct nlist_def {
    int index;
    int dist;
    int length;
    struct nlist_def *next;
} nlist_t;

/* Node definition for queue. */
typedef struct qnode_def {
    int orig;
    int dest;
    int flag;
    struct qnode_def *next;
} qnode_t;

/* Definition of queue. */
typedef struct queue_def {
    int length;
    qnode_t *head;
    qnode_t *tail;
} q_t;

/* Definition of structure containing list of path queues. */
typedef struct qlist_def {
    int length;
    qnode_t *head;
    struct qlist_def *next;
} qlist_t;

```

```
/* Definition of structure for compressing nodes. */
```

```
typedef struct comp_def {
    name_t n_old;
    name_t n_new;
    struct comp_def *next;
    int br;
    int dc;
    int fp;
    int hu;
    int ln;
    int mb;
    int rt;
    int ws;
    int sp;
} comp_t;
```

```
/*.....
***** GLOBAL ARRAYS *****
```

```
name_t  nameArray[A_SIZE];      /* device names          */
type_t  typeArray[A_SIZE];      /* device types          */
nlist_t *listArray[A_SIZE];     /* pointers to linked lists */
int      seenArray[A_SIZE];     /* working array for paths etc. */
int      rsumArray[A_SIZE];     /* row sums of distances  */
int      distArray[A_SIZE][A_SIZE]; /* minimum distance array */
```

Appendix B

General Functions – bb_pkfun.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "bb_pkttype.h"

/*****
***** FUNCTION DEFINITIONS *****/

void initArray()
/* Initializes global arrays nameArray and listArray.
   PRE: None.
   POST: nameArray contains "----" and listArray has pointers set to NULL.*/
{
    int i;

    for (i = 0; i < A_SIZE; i++)
    {
        strcpy(nameArray[i], "----");
        listArray[i] = NULL;
    }
} /* initArray */

/*****

void initSeenArray()
/* Initializes global working array seenArray.
   PRE: None.
   POST: All entries in seenArray are set to 0. */
{
    int i;
    for (i = 0; i < A_SIZE; i++) seenArray[i] = -1;
} /* initSeenArray */

/*****

char *stringstr(char *str1, char *str2)
/* Custom version of strstr(string1, string2). Function scans string1
   for the first occurrence of string2 and returns a pointer to the
   character string where string2 starts or NULL if substring does not
   exist.
   PRE: str1 and str2 must be valid alphanumeric strings.
   POST: Returns pointer to start of str2 within str1 or NULL if substring
        does not exist. */
{
    char *current1, *current2, *stringStart;

    current1 = str1;
    current2 = str2;

    while (*current1 && *current2)
    {
        if (*current1 != *current2) current1++;
        else
        {
            stringStart = current1;
            do {
                current1++;
                current2++;
                if (!(*current2)) return (stringStart);
            } while (*current1 == *current2);
            current1 = stringStart + 1;
            current2 = str2;
        }
    }

    return (NULL);
} /* stringstr */
```

```

.....
..... QUEUE FUNCTIONS .....
.....

q_t *newQ()
/* Creates new queue of length 0.
PRE: None.
POST: New queue node is created, queue length is set to 0, and pointers
to head and tail of queue are set to NULL. Pointer to new queue
is returned to calling function. */
{
    q_t *newNode;

    newNode = MALLOC(q_t);
    newNode->length = 0;
    newNode->head = NULL;
    newNode->tail = NULL;
    return(newNode);
} /* newQ */

.....

void nQ(q_t *queue, int orig, int dest, int flag)
/* Adds new items to the tail of a queue.
PRE: A queue must exist (defined by newQ()).
POST: New queue item is created, inserted at tail of queue, and
data are transferred. */
{
    qnode_t *newNode;

    newNode = MALLOC(qnode_t);
    newNode->orig = orig;
    newNode->dest = dest;
    newNode->flag = flag;
    newNode->next = NULL;
    if (queue->length == 0) queue->head = newNode;
    else queue->tail->next = newNode;
    queue->tail = newNode;
    queue->length++;
} /* nQ */

.....

qnode_t *dQ(q_t *queue)
/* Removes item from the head of a queue.
PRE: A queue must exist (defined by newQ()).
POST: Pointer to item removed is returned and linked list representing
queue is modified */
{
    qnode_t *result;

    if (queue->length == 0)
    {
        printf("\nERROR - Attempted dequeue from empty queue\n");
        printf("=====> ABORTING <=====\n");
        exit(0);
    }
    result = queue->head;
    queue->head = queue->head->next;
    queue->length--;
    if (queue->length == 0) queue->tail = NULL;
    return(result);
} /* dQ */

.....

void killQ(q_t *queue)
/* Removes all items from a queue.
PRE: A queue must exist (defined by newQ())
POST: Every item in queue is deleted, memory is released, and queue
definition reinitialized to zero.*/
{
    qnode_t *current, *next;
    current = queue->head;
    queue->head = NULL;
    queue->tail = NULL;
    queue->length = 0;
    while (current != NULL)
    {
        next = current->next;
        free(current);
        current = next;
    }
} /* killQ */

```

```

/*****
void sortQ(q t *queue)
/* Sorts the elements of a queue in descending order
according to the second field (queue->dest). This is used when
bumping nodes from one median to the next in order to balance the
domains clustered around each local median.
PRE: A queue must exist (defined by newQ()).
POST: Linked list representing queue is sorted in ascending order
of "dest" field. */
{
    qnode t *current, *last;
    int flag;

    /* If queue is empty or has only one element, there is nothing to
    sort so return. */

    if ((queue->head == NULL) || (queue->head->next == NULL))
        return;

    /* Use bubble sort to percolate largest element to the head of the
    queue and stop when no more interchanges have to be made. The
    variable "flag" is used to monitor whether items in queue have
    been swapped. */

    do {
        flag = 0;
        current = queue->head;

        /* Placing largest element at head of queue. */

        if (current->dest < current->next->dest)
        {
            last = current;
            queue->head = current->next;
            last->next = current->next->next;
            queue->head->next = last;
        }
        last = queue->head;
        current = queue->head->next;

        /* Swapping largest elements in remainder of queue. */

        while (current->next != NULL)
        {
            if (current->dest < current->next->dest)
            {
                last->next = current->next;
                current->next = current->next->next;
                last->next->next = current;
                flag = 1;
            }
            last = current;
            if (current->next != NULL) current = current->next;
        }
    } while (flag);
    queue->tail = last->next;
}
**** sortQ ****
/****

qnode_t *nodeInQ(q t *pathQ, int node)
/* Function checks whether the desired node is contained in the path.
The difference between nodeInQ and nodeNotInPath (aside from opposite logic)
is that nodeInQ checks only the ->orig field, whereas nodeNotInPath checks
both ->orig and ->dest fields.
PRE: pathQ contains the path being searched.
POST: If node is contained in the path, function returns a pointer to
its location, otherwise it returns NULL. */
{
    qnode t *current;

    current = pathQ->head;
    while (current != NULL)
    {
        if (current->orig == node) return(current);
        current = current->next;
    }
    return(NULL);
}
**** nodeInQ ****

```

```

/*****
***** PATH FUNCTIONS *****/

void printPath(int origin, int dest)
/* Function to print the shortest path from dest to origin.
   SeenArray contains the indices of the preceding node in the path, so
   the algorithm works backwards from the destination by inspecting the
   node at SeenArray[i]. If this is origin, stop, otherwise repeat loop
   PRE: Path between origin and dest must be recorded as indices of immediate
        predecessor nodes from dest.
   POST: Names of nodes in path printed to screen */
{
    int next, counter;

    counter = 1;
    printf("\nShortest path between '%s' and '%s' is \n",
        nameArray[dest], nameArray[origin]);
    next = dest;
    while ((next >= 0) && (next != origin))
    {
        printf("%s ", nameArray[next]);
        if ((counter % 8) == 0) printf("\n");
        counter++;
        next = seenArray[next];
    }
    if (next == origin) printf("%s", nameArray[next]);
    printf("\n");
} /* printPath */

/*****
***** TREE FUNCTIONS *****/

/**** Function which looks for "name" in the binary search tree, and if
found, it returns the pointer to that node. If it is not found, it
returns NULL. */

tree_t *nameInTree(tree_t *aTree, name_t name)
/* Searches for "name" in binary search tree and returns pointer to that node
   if it exists or NULL if it does not.
   PRE: aTree binary search tree must exist, but can be set to NULL
   POST: Returns pointer to node if found or NULL pointer if not found */
{
    tree_t *current;
    int comp;

    current = aTree;

    while (current != NULL)
    {
        comp = strcmp(name, current->name);
        if (comp == 0) return(current);
        else if (comp < 0) current = current->left;
        else current = current->right;
    }
    return(NULL);
} /* nameInTree */

/*****/

tree_t *instTree(char *name, char *type)
/* Function which instantiates a node of the binary search tree,
   fills in the NAME and TYPE fields, sets all pointers to NULL, and
   returns the pointer to this node
   PRE: None.
   POST: Returns pointer to this node (root node) */
{
    tree_t *newTree;

    newTree = MALLOC(tree_t);
    strcpy(newTree->name, name);
    strcpy(newTree->type, type);
    newTree->left = NULL;
    newTree->right = NULL;
    newTree->list = NULL;
    return(newTree);
} /* instTree */

```



```

/*****
void addleaf(tree_t *root, char *name, char *type)
/* Function which locates the position in the binary search tree
   (ordered by NAME) where this leaf node should be added, and calls
   function instTree() to actually fill in the appropriate values. The
   pointers of the parent node are changed to point to the leaf node.
   This function is iterative.
   PRE: Root node must exist.
   POST: Leaf node added to correct location of binary search tree. */
{
    tree_t *current;
    int comp;

    current = root;

    /* Iteratively traverse the tree and determine where the leaf should
       be added with respect to the current node. */

    while (current != NULL)
    {
        comp = strcmp(name, current->name);
        if (comp < 0)
        {
            /* Leaf belongs on left side. If no left child exists, add leaf
               otherwise keep going until vacant spot is found. */
            if (current->left == NULL)
            {
                current->left = instTree(name, type);
                return;
            }
            else current = current->left;
        }
        else if (comp > 0)
        {
            /* Leaf belongs on right side. If no right child exists, add leaf
               otherwise keep going until vacant spot is found. */
            if (current->right == NULL)
            {
                current->right = instTree(name, type);
                return;
            }
            else current = current->right;
        }
        else return;
        /* Node with given name already exists. Do nothing. */
    }
} /* addleaf */

*****/

tree_t *buildTree(FILE *in_file)
/* Function which reads the topology file and builds the binary
   search tree from the names of the devices. This is the first pass in
   reading the data file and sorting it at the same time. Each node
   represents one device of the network, and the devices that are
   connected to it are listed as linked lists attached to the nodes of
   the tree.
   PRE: Input file in_file must exist and have the proper format.
   POST: Data transferred from input file to binary search tree. Pointer to
         root node returned by function. */
{
    tree_t *root;
    name_t n_name;
    char buffer[BUFSIZE];
    char buf, *name, *type;

    printf("\nReading data file ...\n");

    /* Extract the device name and device type from first line of file.
       Use this information and get the root of the binary search tree. */

    if (fgets(buffer, BUFSIZE, in_file) != NULL)
    {
        buf = strtok(buffer, "\n");
        name = strtok(buf, "=");
        name = strtok(NULL, "");
        strcpy(n_name, name);
        type = strtok(NULL, ":");
        root = instTree(n_name, type);
    }
}

```

```

/* Read the remainder of the file and keep adding nodes to the tree */
while (fgets(buffer,BUFSIZE,in_file) != NULL)
{
    buf = strtok(buffer,"\n");
    name = strtok(buf,"=");
    name = strtok(NULL," ");
    strcpy(n_name,name);
    type = strtok(NULL,"");
    addLeaf(root,n_name,type);
}
rewind(in_file);
return(root);
} /** buildTree **/

/*****

void killTree(tree_t *aTree)
/* Deletes a binary search tree and releases memory back to system
PRE: A tree is assumed to exist.
POST: Tree is deleted and memory released. */
{
    if (aTree != NULL)
    {
        killTree(aTree->left);
        killTree(aTree->right);
        free(aTree);
    }
} /** killTree **/

/*****
/***** ARRAY FUNCTIONS *****/

int getIndex(name_t name, int arrayEnd)
/* Function which returns the index of the nameArray where NAME is
located. If the name is not in the array, the function returns -1.
This function uses an iterative binary search to look for the name.
Variable arrayEnd is used to limit the upper boundary for the search,
since this is the index following the last element which was entered in
the array.
PRE: Entries in nameArray must be sorted alphabetically in ascending order
POST: Returns array index of item or -1 if item not found */
{
    int start,middle,end;

    start = 0;
    end = arrayEnd;
    while (end >= start)
    {
        middle = (start + end) / 2;
        if (strcmp(nameArray[middle],name) == 0) return(middle);
        else if (strcmp(nameArray[middle]) < 0) end = middle - 1;
        else start = middle + 1;
        if (start < 0) start = 0;
        if (end > arrayEnd) end = arrayEnd;
    }
    return(-1);
} /** getIndex **/

/*****

void treeToArray(tree_t *aTree, int *index)
/* Function which transfers the device names and types from the
binary search tree nameArray and typeArray, respectively. Function
performs an in-order recursive tree traversal and fills in the arrays
as it traverses the tree.
PRE: Binary search tree must exist. Index must be initialized to 0
POST: Data transferred from binary search tree to nameArray and
typeArray. Index is set to next free location in arrays (used
as variable endArray). */
{
    if (aTree != NULL)
    {
        treeToArray(aTree->left, index);
        strcpy(nameArray[*index],aTree->name);
        strcpy(typeArray[*index],aTree->type);
        (*index)++;
        treeToArray(aTree->right, index);
    }
} /** treeToArray **/

```

```

/*****
void dellistArray(int endArray)
/* Deletes linked lists pointed to by entries in listArray.
PRE: None.
POST: All pointers in listArray are reset to NULL and lists to which
they were pointing are deleted. */
{
    int i;
    nlist_t *current, *next;

    for (i = 0; i < endArray; i++)
    {
        current = listArray[i];
        listArray[i] = NULL;

        while (current != NULL)
        {
            next = current->next;
            free(current);
            current = next;
        }
    }
}

/*****
LIST FUNCTIONS *****/

nlist_t *newListNode(int index)
/* Function to instantiate a new node of linked list. The data item
INDEX is the index of that node in the nameArray.
PRE: None.
POST: New node of linked list created, data transferred, fields
fields initialized, and pointer to this node returned. */
{
    nlist_t *newList;

    newList = MALLOC(nlist_t);
    newList->index = index;
    newList->dist = 1;
    newList->length = 1;
    newList->next = NULL;
    return(newList);
} /** newListNode */

/*****
void addToList(int srcNode, int destNode)
/* Function which adds new list node to linked list so that it
remains sorted in ascending order according to the index value of the
linked list. SrcNode is the listArray index for the list to which
destNode will be added.
PRE: srcNode and destNode must be indices of valid nodes.
POST: Device connectivity constructed as linked lists pointed to by
listArray. */
{
    nlist_t *current, *newNode, *previous;

    if ((srcNode < 0) || (destNode < 0))
    {
        printf("\nERROR - Negative index passed to function 'addToList'.\n");
        printf("srcNode = %d; destNode = %d\n", srcNode, destNode);
        printf("=====> ABORTING <=====\n");
        exit(0);
    }

    /* Fill in first node of linked list. */
    if (listArray[srcNode] == NULL)
    {
        current = newListNode(destNode);
        listArray[srcNode] = current;
        return;
    }
    current = listArray[srcNode];

```

```

/* Linked list already exists but new node should go before head of
current list, so insert it and adjust list pointers */
if (destNode < current->index)
{
    newNode = newListNode(destNode);
    newNode->next = current;
    listArray[srcNode] = newNode;
    return;
}

/* New node is to be inserted somewhere into the middle of an
existing linked list. First find location where it should be
inserted (CURRENT), determine if it should go before or after
current, and insert it. */

while ((current->next != NULL) && (destNode > current->index))
{
    previous = current;
    current = current->next;
}
if (destNode > current->index)
{
    current->next = newListNode(destNode);
}
else if (destNode < current->index)
{
    newNode = newListNode(destNode);
    previous->next = newNode;
    newNode->next = current;
}
return;
} /** addToList */

/*****

void addList(FILE *in_file)
/* Function which makes second pass through topological data file
and adds the connected devices as linked lists the heads of which are
pointed to by pointers in listArray.
PRE: Input file in_file must be in the proper format
POST: Network connectivity is transferred from input file to listArray */
{
    char buffer[BUFSIZE];
    int srcIndex, index;
    char *buf, *name;

    printf("Building topology ... \n");

    /* Read entire topology file, and extract the "source device name" */

    while (fgets(buffer, BUFSIZE, in_file) != NULL)
    {
        buf = strtok(buffer, "\n");
        name = strtok(buf, "=");
        name = strtok(NULL, " ");
        srcIndex = getIndex(name, A_SIZE-1);
        name = strtok(NULL, ":");

        /* For each line, extract the device names which are attached to
        "source device name", and add these to the linked list for the
        source device. */
        do
        {
            name = strtok(NULL, " ");
            if ((name != NULL) && (stringstr(name, "+") == NULL))
            {
                index = getIndex(name, A_SIZE-1);
                addToList(srcIndex, index);
            }
        } while (name != NULL);
    }
} /** addList */

```

Appendix C

Path-related Functions – bb_pathfun.c

```

#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <malloc.h>
#include <time.h>

#include "bb_pktype.h"      /* data structure definitions      */
#include "bb_pkfun.h"       /* tree, list, and array functions */

#define TIME_INTERVAL 15    /* time for showing program is alive (sec) */

/*****

void priorityNQ(q_t *queue, int orig, int dest, int pathlength)
/* Function to enqueue an item in decreasing value of pathlength.
   PRE:  Orig and dest are indices of origin and destination, respectively,
        queue contains the sorted queue of nearest neighbours already
        examined, and pathlength contains the partial path length between
        origin and destination.
   POST: Item is inserted in the queue so that queue remains sorted according
        to pathlength. */
{
    qnode_t *newNode, *current, *previous;

    newNode = MALLOC(qnode_t);
    newNode->orig = orig;
    newNode->dest = dest;
    newNode->flag = pathlength;
    newNode->next = NULL;

    /* Queue is still empty. Insert first item and return. */
    if (queue->length == 0) {
        queue->head = newNode;
        queue->tail = newNode;
        queue->length++;
        return;
    }

    /* Item must be inserted before first item of existing queue. */
    current = queue->head;
    if (pathlength < current->flag) {
        newNode->next = current;
        queue->head = newNode;
        queue->length++;
        return;
    }

    /* Item belongs after the first item of existing queue. Find the
       correct location and insert the item. */
    while ((current->next != NULL) && (pathlength >= current->flag))
    {
        previous = current;
        current = current->next;
    }

    if (pathlength >= current->flag)
    {
        /* Item is inserted at the very end of the queue. */
        current->next = newNode;
        queue->tail = newNode;
    }
    else
    {
        /* Item inserted into the middle of the queue. */
        previous->next = newNode;
        newNode->next = current;
    }
    queue->length++;
    return;
} /* priorityNQ */

```

```

/*****
char *getGroupID(char *device)
/* Function which extracts the community string of a BNR/NT device.
PRE: None.
POST: The community string is a sequence of 3 alphanumeric (upper case)
characters which are returned to the calling function *.
{
    int i, j, flag;
    name_t answer;

    i = 0;
    j = 0;
    flag = 0;

    while (device[i])
    {
        if ((device[i] > 64) && (device[i] < 91))
        {
            answer[j] = device[i];
            j++;
            flag = 1;
        }
        else if (flag)
        {
            answer[j] = '\0';
            return(&answer[0]);
        }
        i++;
    }

    answer[j] = '\0';
    return(&answer[0]);
} /** getGroupID **/

/*****/

void defaultArcLengths(int sameGrpLength, int difGrpLength, int endArray)
/* Function which reads the default lengths of arcs connecting different
sites and inserts the default lengths into the length field of the data
structure representing the internal topology. Arc lengths are assumed to
be the same in both directions.
PRE: endArray indicates the first free position in the arrays used.
POST: sameGrpLength is assigned to arc lengths between devices sharing
the same community string and difGrpLength is assigned to arcs
between nodes having different community strings. */
{
    int i;
    char *groupID;
    name_t orig;
    nlist_t *current;

    for(i = 0; i < endArray; i++)
    {
        groupID = getGroupID(nameArray[i]);
        strcpy(orig.groupID);
        current = listArray[i];
        while (current != NULL)
        {
            if (strstr(nameArray[current->index], orig) == NULL)
                current->dist = difGrpLength;
            else current->dist = sameGrpLength;
            current = current->next;
        }
    }
} /** defaultArcLengths **/

/*****/

/* Function which examines specific arc lengths overriding the default
arc lengths and inserts them into the length field of the data structure
representing the internal topology. The specific arc lengths follow the
default arc lengths in the text data file. Function assumes that the
arc length is the same in both directions.
PRE: endArray indicates the first free location in arrays used.
POST: Lengths is assigned to arcs having orig and dest as substrings of
the origin and destination nodes, respectively. Any previously
assigned length is overwritten. */

```

```

void specialArcLengths(name_t orig, name_t dest, int length, int endArray)
{
    int i;
    nlist_t *current;

    for (i = 0; i < endArray; i++)
    {
        if (stringstr(nameArray[i], orig) != NULL)
        {
            current = listArray[i];
            while (current != NULL)
            {
                if (stringstr(nameArray[current->index], dest) != NULL)
                {
                    current->dist = length;
                    current = current->next;
                }
            }
        }
        else if (stringstr(nameArray[i], dest) != NULL)
        {
            current = listArray[i];
            while (current != NULL)
            {
                if (stringstr(nameArray[current->index], orig) != NULL)
                {
                    current->dist = length;
                    current = current->next;
                }
            }
        }
    }
} /* specialArcLengths */

/*****

void assignArcLengths(FILE *dist_file, int endArray)
/* Function which assigns default or specific lengths to arcs of the
network. The arc lengths are read from a special text file.
PRE: dist_file must contain the arc length definitions in the correct
format and endArray indicates the end of the arrays used.
POST: Arc lengths are inserted into the ->dist field of the linked
lists of the internal topology. */
{
    char buffer[BUFSIZE];
    char *origStr, *destStr, *lengthStr;
    int sameGrplength, difGrplength, length;

    if (dist_file == NULL) return;

    printf("Assigning arc lengths ...\n");

    /* Read the default arc lengths from the first line of the file and
assign the default arc lengths. */

    if (fgets(buffer, BUFSIZE, dist_file) != NULL)
    {
        sscanf(buffer, "%d %d\n", &sameGrplength, &difGrplength);
        printf("Default arcs at same site = %d\n", sameGrplength);
        printf("Default arcs between sites = %d\n", difGrplength);
        defaultArcLengths(sameGrplength, difGrplength, endArray);
    }

    /* Read any special arc lengths which may override the default lengths
and assign them to the appropriate arcs. */

    printf("\nDefault lengths superceded by following lengths:\n");
    while (fgets(buffer, BUFSIZE, dist_file) != NULL)
    {
        origStr = strtok(buffer, " ");
        destStr = strtok(NULL, " ");
        lengthStr = strtok(NULL, "\n");
        sscanf(lengthStr, "%d\n", &length);
        printf("%s-%s length = %d\n", origStr, destStr, length);
        if ((strlen(origStr) > NAMELEN) || (strlen(destStr) > NAMELEN))
        {
            printf("ERROR - Device names too long: '%s', '%s'\n",
                origStr, destStr);
            printf("====> ABORTING <====\n");
            exit(0);
        }
        specialArcLengths(origStr, destStr, length, endArray);
    }
} /* assignArcLengths */

```

```

/*****
void getArcLengths(int endArray, int *totalArcs, int *singleArcs)
/* Function to assign weights to the arcs of a network and count the
number of arcs present in the network (totalArcs) and the number
of arcs which lead to leaf nodes (singleArcs).
PRE: endArray indicates the end of the arrays used.
POST: If "topology.len" file is missing, arc lengths are assigned
according to device type. If the file is present, arc lengths as
specified in the file are assigned. */
{
    int i, j;
    nlist_t *current;
    FILE *dist_file;

    *totalArcs = 0;
    *singleArcs = 0;

    /* If length file is missing, assign "hard-coded" arc lengths. */
    dist_file = fopen("topology.len", "r");
    if (dist_file == NULL)
    {
        printf("\nRouter-router arc distance      = %d\n", RT_RT_LINK);
        printf("Distance to any compressed node = %d\n", COMPR_LINK);
        printf("Distance for all other arcs      = %d\n\n", NORMAL_LINK);

        for (i = 0; i < endArray; i++)
        {
            current = listArray[i];
            if (stringstr(typeArray[i], "RT") != NULL)
            {
                while (current != NULL) {
                    j = current->index;
                    if (stringstr(typeArray[j], "RT")) current->dist = RT_RT_LINK;
                    else if (stringstr(typeArray[j], "COMPR")) current->dist = COMPR_LINK;
                    else current->dist = NORMAL_LINK;
                    current = current->next;
                }
            }
            else if (stringstr(typeArray[i], "L *R") != NULL) {
                while (current != NULL) {
                    current->dist = COMPR_LINK;
                    current = current->next;
                }
            }
            else {
                while (current != NULL) {
                    if (stringstr(typeArray[j], "COMPR")) current->dist = COMPR_LINK;
                    else current->dist = NORMAL_LINK;
                    current = current->next;
                }
            }
        } /* FOR all devices in array */
    } /* IF distance file not present */
    else
    {
        /* Distance array is present, assign arc lengths and increase length
        to any compressed node by COMPR_LINK. */

        assignArcLengths(dist_file, endArray);
        for (i = 0; i < endArray; i++)
        {
            current = listArray[i];
            if (stringstr(typeArray[i], "COMPR"))
            {
                while (current != NULL)
                {
                    current->dist += COMPR_LINK;
                    current = current->next;
                }
            }
            else {
                while (current != NULL)
                {
                    j = current->index;
                    if (stringstr(typeArray[j], "COMPR")) current->dist += COMPR_LINK;
                    current = current->next;
                }
            }
        } /* FOR increasing arc lengths to compressed nodes */
    } /* ELSE arc distance file exists */
}

```



```

/* Count number of arcs in network. */
for (i = 0; i < endArray; i++)
{
    current = listArray[i];
    if(current->next == NULL) *singleArcs = *singleArcs + 1;
    while (current != NULL)
    {
        *totalArcs = *totalArcs + 1;
        current->length = current->dist;
        current = current->next;
    }
}
*totalArcs = *totalArcs/2;
printf(" %d nodes, %d arcs, %d nodes with single incident arc.\n",
       endArray, *totalArcs, *singleArcs);

} /** getArcLengths **/

/* function which returns the true length of an arc between ORIG and
   DIST. This length is stored in field DIST in the connectivity linked
   list. */

int newArcLength(int orig, int dest)
{
    nlist_t *current;

    current = listArray[orig];
    while ((current != NULL) && (dest > current->index))
        current = current->next;

    return(current->dist);
} /** newArcLength **/

/*****

int truePathLength(q_t *queue)
/* function which returns the true path length.
   PRE: Queue contains the path described by a queue of arcs.
   POST: function returns the length of the path. */
{
    qnode_t *queuePtr;
    nlist_t *listPtr;
    int length;

    length = 0;
    queuePtr = queue->head;
    while (queuePtr != NULL)
    {
        listPtr = listArray[queuePtr->orig];
        while (listPtr != NULL)
        {
            if (listPtr->index == queuePtr->dest)
            {
                length += listPtr->length;
                break;
            }
            listPtr = listPtr->next;
        }
        queuePtr = queuePtr->next;
    }
    return(length);
} /** truePathLength **/

/*****

/* function which adds all arcs between current node and node which
   has not yet been added to the path being constructed
   (seenArray[i] == -1) to the priority queue.
   PRE: Any node which has been examined is assigned the index of its
        immediate predecessor in seenArray. Any unexamined node still has
        its original value of -1. CurLength must be the cumulative path
        length from the origin to the current node. Queue contains the
        list of nodes which are to be examined and must be sorted in
        increasing length of path length (priority queue).
   POST: Any unexamined nearest neighbour node is added to the priority
        queue for future examination. */

```

```

void addWeightedPaths(q_t *queue, int o_index, int curlength)
{
    nlist_t *current;
    int length;

    current = listArray[o_index];
    while (current != NULL)
    {
        if(seenArray[current->index] == -1)
        {
            length = curlength + current->dist;
            priorityNQ(queue, o_index, current->index, length);
        }
        current = current->next;
    }
} /* addWeightedPaths */

/*****

int findWeightedPath(q_t *queue, int origin, int dest)
/* Function which finds the shortest path between the origin and the
destination in a weighted network.
PRE: Internal topology must be stored in the listArray structure.
POST: The predecessor in the path for each node k is recorded as an
index of the predecessor node in seenArray[k]. The path can be
constructed by following the trail of the indices.
The function returns the actual path length if no path exists,
function returns -1 */
{
    qnode_t *nextDest;
    int pathlength;

    pathlength = 0;
    killQ(queue);
    seenArray[origin] = origin;
    addWeightedPaths(queue, origin, 0);
    while (queue->length > 0)
    {
        nextDest = dQ(queue);
        pathlength = nextDest->flag;
        if (seenArray[nextDest->dest] == -1)
        {
            seenArray[nextDest->dest] = nextDest->orig;
            addWeightedPaths(queue, nextDest->dest, pathlength);
        }
        free(nextDest);
        if (nextDest->dest == dest) return(pathlength);
    }
    return(-1);
} /* findWeightedPath */

/*****

q_t *recordShortestPath(int origin, int dest)
/* Function which records the current (shortest) path found by reading
the indices of seenArray[k] and adding the arcs to a queue as they
are found. The queue is passed to the calling function.
PRE: Path between origin and dest must exist and must be recorded in
seenArray as indices of nearest predecessor nodes.
POST: Pointer to a queue which represents a sequence of arcs is returned */
{
    q_t *pathQ;
    int next, previous, length;

    pathQ = newQ();
    if (origin == dest)
    {
        nQ(pathQ, origin, dest, 0);
        return(pathQ);
    }
    next = dest;
    while ((next >= 0) && (next != origin))
    {
        previous = next;
        next = seenArray[next];
        length = newArcLength(previous, next);
        nQ(pathQ, previous, next, length);
    }
    return(pathQ);
} /* recordShortestPath */

```

```

/*****
void prPathQ(q_t *pathQ)
/* function which prints the path, recorded as arcs in pathQ, to the
terminal.
PRE: pathQ must contain the sequence of arcs describing the path.
POST: The path is printed to the screen. */
{
    qnode_t *current;

    printf("[Xd]: ", truePathLength(pathQ));
    current = pathQ->head;
    while (current != NULL)
    {
        printf("%s ", nameArray[current->orig]);
        current = current->next;
    }
    printf("%s n", nameArray[pathQ->tail->dest]);
} /** prPathQ **/

/*****
qnode_t *arcInQ(q_t *queue, qnode_t *item)
/* function which determines whether ITEM is already present in the
QUEUE.
*** IMPORTANT NOTE
In contrast to other functions using the qnode_t data type, in this
function the arc origin is stored in field queue->orig as in other
functions, BUT the destination is stored in queue->flag. The field
queue->dest is used to count the frequency of occurrence of this
particular arc in the paths under consideration.
*** REASON: sortQ(queue) sorts the queue according to the
queue->dest field and can therefore be used directly.
PRE: None.
POST: If given arc is present in the queue, function returns the pointer
to that location, otherwise it returns NULL. */
{
    qnode_t *current;

    current = queue->head;
    while (current != NULL)
    {
        if (((item->orig == current->orig) && (item->dest == current->flag)) ||
            ((item->dest == current->orig) && (item->orig == current->flag)))
            return(current);
        current = current->next;
    }
    return(NULL);
} /** arcInQ **/

/*****
int pathIsNew(qlist_t *qList, q_t *queue)
/* function which determines whether a path listed in QUEUE is already
in the list of paths (QLIST). Function assumes that no path length
will ever be shorter than the shortest path in qList.
PRE: Queue contains the path listed as a sequence of arcs, and qList
is the data structure which contains a list of paths.
POST: Returns 1 if path being checked is not in the list. 0 otherwise. */
{
    qlist_t *listPos;
    qnode_t *oldPathLoc, *newPathLoc;
    int length;

    length = truePathLength(queue);
    listPos = qList;

    if (length < listPos->length)
    {
        printf("\nERROR - path found which is shorter than SHORTEST path \n");
        printf("Shortest length: %d; Current length: %d\n",
            listPos->length, length);
        prPathQ(queue);
        printf("====> ABORTING <=====\n");
        exit(0);
    }
}

```

```

while ((listPos != NULL) && (length >= listPos->length))
{
    if (length == listPos->length)
    {
        /* Perform arc-by-arc comparison only if path length is the same */
        newPathLoc = queue->head;
        oldPathLoc = listPos->head;
        while ((oldPathLoc != NULL) && (newPathLoc != NULL))
        {
            if ((oldPathLoc->orig == newPathLoc->orig) &&
                (oldPathLoc->dest == newPathLoc->dest))
            {
                oldPathLoc = oldPathLoc->next;
                newPathLoc = newPathLoc->next;
            }
            else break;
        }
        if ((oldPathLoc == NULL) && (newPathLoc == NULL)) return(0);
        listPos = listPos->next;
    }
    return(1);
} /* pathIsNew */

/*****

void insertNewQueue(qlist_t *qlist, q_t *queue)
/* Function which adds new path described in QUEUE to the structure
of list of lists so that the paths remain in increasing order
according to their lengths.
PRE: qList must contain the ranked list of paths, and queue must contain
the current path listed as a sequence of arcs.
POST: The path described in queue is transferred to the correct location
in the ranked list. If there are several paths of same length as
the current path, the current path is added at the END of this
group of same-length paths. */
{
    qlist_t *listPos, *prevPos, *newPos;
    int length;

    length = truePathlength(queue);
    listPos = qlist;

    if (length < listPos->length)
    {
        printf("\nERROR - path found which is shorter than SHORTEST path\n");
        printf("=====> ABORTING <=====\n");
        exit(0);
    }

    while ((listPos != NULL) && (length >= listPos->length))
    {
        prevPos = listPos;
        listPos = listPos->next;
    }

    newPos = MALLOC(qlist_t);
    newPos->length = length;
    newPos->head = queue->head;
    prevPos->next = newPos;

    if (listPos == NULL) newPos->next = NULL;
    else newPos->next = listPos;
} /* insertNewQueue */

/*****

int nodeNotInPath(q_t *pathQ, int node)
/* Function checks whether the desired node is contained in the path
PRE: pathQ contains the path being searched.
POST: If node exists in the path, function returns 0, otherwise it
returns 1. */
{
    qnode_t *current;

    current = pathQ->head;
    while (current != NULL)
    {
        if ((current->orig == node) || (current->dest == node)) return(0);
        current = current->next;
    }
    return(1);
} /* nodeNotInPath */

```

```

/*****
q t *countNodesInPaths(qlist_t *pathlist, int numberOfPaths)
/* Function counts the frequency of occurrence of all nodes in numberOfPaths
paths. The frequency of occurrence is the ->dest field of each queue item.
PRE: numberOfPaths specifies the number of paths to be examined and
must be > 0. Pathlist is the pointer to the structure containing
the ranked list of paths.
POST: Returns a pointer to a linked list of nodes sorted in descending
order of the frequency of their occurrence */
{
    q t *nodeQ;
    qnode t *currentArc, *existingNode;
    qlist t *currentQptr;
    int counter;

    nodeQ = newQ();
    counter = 0;
    currentQptr = pathlist;
    while((counter < numberOfPaths) && (currentQptr != NULL))
    {
        currentArc = currentQptr->head;
        while (currentArc != NULL)
        {
            existingNode = nodeInQ(nodeQ, currentArc->orig);
            if (existingNode == NULL)
                nQ(nodeQ, currentArc->orig, 1, 0);
            else existingNode->dest++;
            if (currentArc->next == NULL)
            {
                existingNode = nodeInQ(nodeQ, currentArc->dest);
                if (existingNode == NULL)
                    nQ(nodeQ, currentArc->dest, 1, 0);
                else existingNode->dest++;
            }
            currentArc = currentArc->next;
        }
        counter++;
        currentQptr = currentQptr->next;
    }
    sortQ(nodeQ);
    return(nodeQ);
} ** countNodesInPaths **

/*****
q t *countArcsInPaths(qlist_t *pathlist, int numberOfPaths)
/* Function counts the frequency of occurrence of all arcs in numberOfPaths
paths. The frequency of occurrence is the ->dest field of each queue item.
PRE: numberOfPaths specifies the number of paths to be examined and
must be > 0. Pathlist is the pointer to the structure containing
the ranked list of paths
POST: Returns a pointer to a linked list of arcs sorted in descending
order of the frequency of their occurrence */
{
    q t *arcQ;
    qnode t *currentArc, *existingArc;
    qlist t *currentQptr;
    int counter;

    arcQ = newQ();
    counter = 0;
    currentQptr = pathlist;
    while((counter < numberOfPaths) && (currentQptr != NULL))
    {
        currentArc = currentQptr->head;
        while (currentArc != NULL)
        {
            existingArc = arcInQ(arcQ, currentArc);
            if (existingArc == NULL)
                nQ(arcQ, currentArc->orig, 1, currentArc->dest);
            else existingArc->dest++;
            currentArc = currentArc->next;
        }
        counter++;
        currentQptr = currentQptr->next;
    }
    sortQ(arcQ);
    return(arcQ);
} ** countArcsInPaths **

```

```

.....
/***** MAIN FUNCTION BUILDING RANKED LIST OF PATHS *****/

/* Function which calculates the number of shortest paths specified in
   pathNum and returns the pointer to the complete list of shortest
   paths.

   in_file      - file which contains the network topology
   out_file     - file to which the results are printed
   origin       - name of the origin node
   dest        - name of the destination node
   pathNum      - number of shortest paths to be listed
   endArray     - index of next free space in the arrays
   *candPaths   - pointer to number of candidate paths in data structure
   *pathSearched - pointer to total number of paths examined
*/
.....

qlist_t *findShortestPaths(name_t origin, name_t dest, int pathNum,
                          int endArray, long *candPaths, long *pathSearched)
/* PRE: Internal topology must be stored in listArray and endArray must
   be set to the next available location in the arrays
   POST: Returns a pointer to the data structure containing the ranked list
   of paths. */
{
    long iteration, candPathNum;
    int o_index, d_index;
    int counter, prevNode, nextNode, pathlength;
    q_t *queue, *pathQ, *nextPathQ, *nodeQ, *startPathQ, *restPathQ;
    qnode_t *current, *nextArc;
    nlist_t *endArc;
    qlist_t *pathList, *currentQptr;
    time_t prev_time, current_time;

    /*----- PATH CALCULATIONS -----*/

    /* Since the path algorithm works "backward" by keeping track of the
       preceding node in the path, the origin and destination must be
       reversed so path is printed in correct order
       Also, write the useful information to the output file */

    /* Calculate shortest path, print it, and initialize parameters
       needed for finding alternate paths */

    queue = newQ();
    initSeenArray();
    d_index = getIndex(origin, endArray);
    o_index = getIndex(dest, endArray);
    pathList = MALLOC(qlist_t);

    pathlength = findWeightedPath(queue, o_index, d_index);

    if (pathlength > 0) pathQ = recordShortestPath(o_index, d_index);
    else return(NULL);

    pathlength = truePathLength(pathQ);

    printf("Shortest path:\n");
    prPathQ(pathQ);
    pathList->length = pathlength;
    pathList->head = pathQ->head;
    pathList->next = NULL;

    *candPaths = 0;
    *pathSearched = 1;
    if(pathNum == 1) return(pathList);

    printf("\nSearching for alternate paths. Processing : ", n);
    currentQptr = pathList;
    iteration = 0;
    counter = 1;
    candPathNum = 0;
    nodeQ = newQ();
    startPathQ = newQ();
    nextPathQ = newQ();

    /* Main iteration loop which is repeated until the required paths
       are found.
       Copy the next shortest path in list (pointed to by currentQptr)
       to temporary path queue "pathQ". */

```

```

prevTime = time(NULL);
while (counter < pathNum)
{
    pathQ = newQ();
    current = currentQptr->head;
    while (current != NULL)
    {
        nQ(pathQ, current->orig, current->dest, current->flag);
        current = current->next;
    }
    current = pathQ->head;
    prevNode = current->orig;

    /* Second nested loop which keeps track of the node currently
       being examined and calculates all the possible perturbations
       from this node. Each node of the path currently under investigation
       is examined in turn. The perturbation node is contained in
       endArc->index */

    while (current != NULL)
    {
        nextNode = current->dest;
        endArc = listArray[current->orig];

        /* Third nested loop which tests whether the perturbation
           node is not already in the path and that it is not the previous
           node or the destination node. If this condition is satisfied,
           the perturbation is valid and can be used to calculate alternate
           path. Note that because the path is calculated in "reverse order",
           d_index is in fact the index of the origin node. */

        while (endArc != NULL)
        {
            if ((nodeNotInPath(startPathQ, endArc->index)) &&
                (endArc->index != nextNode) && (endArc->index != d_index))
            {
                initSeenArray();
                nextArc = startPathQ->head;

                /* Transfer path from origin to perturbation node to the
                   next path queue. */

                while (nextArc != NULL) {
                    seenArray[nextArc->orig] = nextArc->dest;
                    nQ(nextPathQ, nextArc->orig, nextArc->dest, nextArc->flag);
                    nextArc = nextArc->next;
                }

                /* Find shortest loopless path from perturbation node to
                   destination. If such a path does not exist, pathlength
                   will be a negative number and the candidate path
                   (nextPathQ) will be deleted. */

                seenArray[current->orig] = endArc->index;
                nQ(nextPathQ, current->orig, endArc->index, endArc->length);
                pathlength = findWeightedPath(queue, o_index, endArc->index);
                if (pathlength < 0)
                {
                    killQ(nextPathQ);
                }
                else
                {
                    restPathQ = recordShortestPath(o_index, endArc->index);
                    while (restPathQ->length > 0)
                    {
                        nextArc = dQ(restPathQ);
                        nQ(nextPathQ, nextArc->orig, nextArc->dest, nextArc->flag);
                        free(nextArc);
                    }
                    killQ(restPathQ);
                    free(restPathQ);

                    if ((pathIsNew(pathList, nextPathQ)) &&
                        (nextPathQ->tail->dest > -1))
                    {
                        candPathNum++;
                        insertNewQueue(pathList, nextPathQ);
                        nextPathQ = newQ();
                    }
                    else killQ(nextPathQ);
                }
            }
            /* While next arc exists */
        }
    }
}

```

```

iteration++;
endArc = endArc->next;
current_time = time(NULL);
if ((current_time - prev_time) > TIME_INTERVAL)
{
    printf("%d certain, %d candidates, %d explored\n",
           counter, candPathNum, iteration);
    prev_time = current_time;
}

} /* third (innermost) WHILE(endArc != NULL) loop */

nQ(startPathQ, current->orig, current->dest, current->flag);
prevNode = current->orig;
current = current->next;

} /* WHILE current != NULL */

killQ(startPathQ);
currentQptr = currentQptr->next;
if (currentQptr == NULL) break;
counter++;
candPathNum--;
killQ(pathQ);
free(pathQ);
} /* outermost WHILE loop looking for pathNum paths */

killQ(queue);
free(queue);
*candPaths = candPathNum;
*pathSearched = iteration;
return(pathList);
} /** findShortestPaths */

.....

void killAllPaths(qlist_t *pathList)
/* Function which deletes the data structure containing all of paths
   PRE: pathList is a pointer to the data structure containing the ranked
       list of paths.
   POST: The data structure is deleted and memory is released */
{
    qnode_t *current, *next;
    qlist_t *currentQptr, *nextQptr;

    currentQptr = pathList;
    while (currentQptr != NULL)
    {
        current = currentQptr->head;
        while (current != NULL)
        {
            next = current->next;
            free(current);
            current = next;
        }
        nextQptr = currentQptr->next;
        free(currentQptr);
        currentQptr = nextQptr;
    }
} /** killAllPaths */

```


Appendix D

Distance Matrix Generator – *MGEN*

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "bb_pctype.h" /* data structure definitions */
#include "bb_pkfun.h" /* general functions */
#include "bb_pathfun.h" /* path-related functions */

/*****

void fillAdjDist(int endArray)
/* Function which inserts the lengths of neighbouring nodes (paths of
a single hop) into the distance array. The shortest path function need
not be called to calculate these path lengths.
PRE: endArray indicates the end of the arrays used and the arc lengths
have been stored in the listArray data structure.
POST: All nearest neighbour distances are entered into distArray by
this function. */
{
    int i, j;
    nlist_t *current;

    for (i = 0; i < endArray; i++)
        for (j = 0; j < endArray; j++)
            distArray[i][j] = 0;

    for (i = 0; i < endArray; i++)
    {
        current = listArray[i];
        while (current != NULL)
        {
            distArray[i][current->index] = current->dist;
            current = current->next;
        }
    }
}
/**** fillAdjDist ****/

/*****

void printPartition(q_t *queue, int i, int j, int endArray)
/* function used to output the smaller of two partitions in a partitioned
network if a path does not exist between nodes i and j. If the network is
divided into several partitions, only one partition will be printed at a
time.
PRE: No path exists between nodes i and j. Queue has been created
(initialization not necessary) and endArray indicates the end of
the arrays used.
POST: The smaller of the two partitions is printed to file "error.mtx".
and program stops. */
{
    FILE *error_file;
    int k, sizePart1, sizePart2, distance, counter;

    printf("ERROR - no path between %s' and %s'.\n",
           nameArray[i], nameArray[j]);
    printf("Halted at connection %d.\n", j);
    error_file = fopen("error.mtx", "w");

    printf("Writing partition information to file 'error.mtx'.\n");
    fprintf(error_file, "ERROR - no path between %s' and %s'.\n",
            nameArray[i], nameArray[j]);
    sizePart1 = 0;
    sizePart2 = 0;
    for (k = 0; k < endArray; k++)
    {
        if (seenArray[k] >= 0) sizePart1++;
        else sizePart2++;
    }
}
/**** printPartition ****/
/*****
```

```

if (sizePart2 < sizePart1)
{
    initSeenArray();
    distance = findWeightedPath(queue, j, 1);
}

counter = 0;
fprintf(error_file, "\nNodes in smaller partition:\n\n");
for(k = 0; k < endArray; k++)
{
    if(seenArray[k] >= 0)
    {
        fprintf(error_file, "%s ", nameArray[k]);
        counter++;
        if((counter % 8) == 0) fprintf(error_file, "\n");
    }
}

printf("====> ABORTING <====\n");
fclose(error_file);
exit(0);

} /** printPartition **/

/*****

void fillDistArray(file_spec matrix_f, int endArray)
/* Function which constructs the array of shortest distances
between every pair of nodes in the network. The distance array is
symmetrical about the diagonal, and the lengths are calculated only
for the upper right triangle of the matrix.
PRE: endArray indicates the end of the arrays used and has been
recorded as the first line of matrix_f file.
POST: The upper part of the distanceArray above the diagonal is written
to matrix_f file. If the network is partitioned, the program
prints the smaller of the two partitions and stops. */
{
    int i, j, distance, counter;
    FILE *matrix_file;
    q_t *queue;

    printf("\nBuilding distance array - processing line:\n"),
    fillAdjDist(endArray);

    queue = newQ();

    counter = 0;
    for (i = 0; i < endArray - 1; i++)
    {
        printf("%d\n", ++counter);
        for (j = i + 1; j < endArray; j++)
        {
            if (distArray[i][j] == 0)
            {
                initSeenArray();
                distance = findWeightedPath(queue, i, j);
                if (distance > 0)
                {
                    distArray[i][j] = distance;
                    distArray[j][i] = distance;
                }
            }
            else
            {
                printPartition(queue, i, j, endArray);
            }
        } /* if distance not calculated yet */
    } /* j FOR loop */

    /* Print complete row of distances to matrix file. */

    matrix_file = fopen(matrix_f, "a");
    fprintf(matrix_file, "%d", i+1);

    for (j = i + 1; j < endArray; j++)
        fprintf(matrix_file, " %d", distArray[i][j]);
    fprintf(matrix_file, " n");
    fclose(matrix_file);

} /* i FOR loop */
} /** fillDistArray **/

```

```

/*****
/*****  MAIN PROGRAM *****/
/*****/

main(int argc, char *argv[])
{
    int noInput, noMatrix, noDist, endArray;
    FILE *in_file, *matrix_file, *dist_file;
    file spec input_f, matrix_f, dist_f;
    tree_t *root;

    printf("\nPROGRAM TO BUILD DISTANCE MATRIX OF BNR/NT NETWORK TOPOLOGY\n");

    /* If the name of the topology data file and the file to which matrix
       should be written are not specified in the command line, prompt
       user for them. */

    noInput = 0;
    noDist = 0;
    noMatrix = 0;

    if (argc > 3) {
        strcpy(input_f, argv[1]);
        strcpy(dist_f, argv[2]);
        strcpy(matrix_f, argv[3]);
    }
    else if (argc > 2) {
        strcpy(input_f, argv[1]);
        strcpy(dist_f, argv[2]);
        noMatrix = 1;
    }
    else if (argc > 1) {
        strcpy(input_f, argv[1]);
        noDist = 1;
        noMatrix = 1;
    }
    else {
        noInput = 1;
        noDist = 1;
        noMatrix = 1;
    }

    if (noInput || noDist || noMatrix) {
        printf("\nUSAGE: %s file1 file2 file3\n", argv[0]);
        printf("        file1 - INPUT:  network topology file\n");
        printf("        file2 - INPUT:  arc length definition file\n");
        printf("        file3 - OUTPUT: distance matrix file\n\n");
    }
    if (noInput) {
        printf("Topology data file name:\n");
        gets(input_f);
    }

    /* Abort if the topology file cannot be opened. */

    in_file = fopen(input_f, "r");
    if (!in_file)
    {
        printf("\nERROR - cannot open file '%s'.\n", input_f);
        printf("====> ABORTING <====\n");
        exit(0);
    }

    if (noDist) {
        printf("Arc length definition file name:\n");
        gets(dist_f);
    }

    /* Abort if the arc length file cannot be opened. */

    dist_file = fopen(dist_f, "r");
    if (!dist_file)
    {
        printf("\nERROR - cannot open file '%s'.\n", dist_f);
        printf("====> ABORTING <====\n");
        exit(0);
    }
}

```

```

/* If file name for matrix file is not specified, use default name */
if (noMatrix) {
    printf("Enter distance matrix file name:\n");
    gets(matrix_f);
}

if (strlen(matrix_f) != 0)
    strcpy(matrix_f, "matrix.out");

/* Construct internal representation of network topology. */
initArray();
root = buildTree(in_file);
endArray = 0;
treeToArray(root, &endArray);
printf("%d nodes read.\n", endArray);
addList(in_file);
fclose(in_file);

/* Create new distance matrix file and close it so that it can
   subsequently be opened in append mode. */
matrix_file = fopen(matrix_f, "w");
fprintf(matrix_file, "%d\n", endArray);
fclose(matrix_file);

/* Assign default and specific arc lengths, and fill in the shortest
   path length in the appropriate cell of DistArray. */
assignArcLengths(dist_file, endArray);
fillDistArray(matrix_f, endArray);

printf("\nDistance matrix written to file '%s'.\n", matrix_f);
printf("\n\n=====> NORMAL TERMINATION =====<=\n\n"),
}

```

Appendix E

MEDIAN Program

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "bb_pctype.h" /* data structure definitions */
#include "bb_pqfun.h" /* general queue, list, tree, etc. functions */

/*****
***** MEDIAN FUNCTIONS *****/

void distArrayFromFile(FILE *dist_file, int matrixSize)
/* function to read the distance matrix from a file.
PRE: Distance matrix must be stored in proper format in dist_file and
file must be opened for reading.
POST: Transfers data from text file to distArray. */
{
    int i, j, endArray, number, length;
    char buffer[LARGEBUFSIZE];
    char *buf, *value;

    printf("Reading distance matrix ... \n");

    /* Extract the array size from the first line of input file. If the
    array size read does not match the expected size, abort the
    program because of data mismatch. */

    if (fgets(buffer, LARGEBUFSIZE, dist_file) != NULL)
    {
        value = strtok(buffer, "\n");
        if (value != NULL)
        {
            endArray = atoi(value);
            if (endArray != matrixSize)
            {
                printf("\nERROR - Matrix size read differs from expected size.\n");
                printf("      You may be accessing wrong matrix file.\n");
                printf("      Array size read: %d. Array size expected: %d\n",
                    value, endArray);
                printf("====> ABORTING <====\n");
                exit(0);
            }
        }
        else
        {
            printf("ERROR - could not read array size.\n");
            printf("====> ABORTING <====\n");
            exit(0);
        }
    }

    i = 0;
    length = endArray - 1;

    /* The data file is assumed to be correct. Read the values from
    the file and fill in the distance array. As each line is read,
    monitor whether this corresponds to expected number of data items,
    and if numbers don't match, abort because file may be corrupted. */

    while (fgets(buffer, LARGEBUFSIZE, dist_file) != NULL)
    {
        buf = strtok(buffer, "\n");
        value = strtok(buf, " ");
        number = atoi(value);
        if (number != i+1)
        {
            printf("ERROR - discrepancy row %d read and row %d expected.\n",
                number, i+1);
            printf("====> ABORTING <====\n");
            exit(0);
        }
    }
}

```

```

value = strtok(NULL, " ");
j = 1 + 1;
do {
    if (value != NULL)
    {
        number = atoi(value);
        if ((i < A_SIZE) && (j < A_SIZE))
        {
            distArray[i][j] = number;
            distArray[j][i] = number;
        }
        else
        {
            printf("ERROR - array size exceeded \n");
            printf("====> ABORTING <====\n");
            exit(0);
        }
    }
    j++;
    value = strtok(NULL, " ");
} while (value != NULL);
j--;
if ((j-1) != length)
{
    printf("\nERROR - file read error at line %d j = %d.\n", i, j);
    printf("====> ABORTING <====\n");
    exit(0);
}
length--;
i++;
}
} /* distArrayFromFile */

/*****

q_t *getMedians(int endArray, int *totMed, int *iterations,
                float *maxDif, float *adjDif)
/* Function which gets the medians from the user. User may specify
some fixed medians, and must enter the total number of medians and the
number of iterations to be used.
PRE:  endArray indicates the end of the arrays used
POST: The medians are placed in a queue, and the pointer to this queue
is returned to the calling program. */
{
    int index, i, flag, fixedMedCounter;
    name_t median;
    q_t *medianQueue;
    char buffer[BUFSIZE];
    char *medianStr, *flagStr, *buf;

    /* Instantiate the median queue. Prompt user to enter any fixed
    medians. A fixed median has the flag field of the queue element
    set to 1, whereas a variable median has it set to 0 */

    medianQueue = newQ();

    fixedMedCounter = 0;
    printf("\nEnter initial medians and codes in format: MEDIAN NAME (CODE n").
    printf("where CODE = 1 for fixed median n");
    printf("        CODE = 0 or blank for variable median n");
    printf("Terminate input by entering blank line. n");
    do {
        fgets(buffer, BUFSIZE, stdin);
        buf = strtok(buffer, "\n");
        medianStr = strtok(buf, " ");
        flagStr = strtok(NULL, " n");
        if (flagStr) sscanf(flagStr, "%d n", &flag);
        else flag = 0;
        if (flag != 0) flag = 1;
        if (medianStr) strcpy(median, medianStr);
        else strcpy(median, "");
        index = getIndex(median, endArray);
    }

```

```

while ((strlen(median) > 0) && (index < 0))
{
    printf("ERROR - Node '%s' is not in network.\n", median);
    printf("Enter valid name or ABORT to quit:\n");
    fgets(buffer, BUFSIZ, stdin);
    buf = strtok(buffer, " ");
    medianStr = strtok(buf, " ");
    flagStr = strtok(NULL, "\n");
    if (flagStr) sscanf(flagStr, "%d", &flag);
    else flag = 0;
    if (flag != 0) flag = 1;
    if (medianStr) strcpy(median, medianStr);
    else strcpy(median, "");
    index = getIndex(median, endArray);
    if (strcmp(median, "ABORT") == 0)
    {
        printf("-----> ABORTING <-----\n\n");
        exit(0);
    }
}

/* Check whether specified median already exists in the queue, and
   if not, add it. If null string was added (no more entries), exit
   loop */

if ((index == 0) && !nodeInQ(medianQueue, index))
{
    nQ(medianQueue, index, -1, flag);
    if (flag == 1) fixedMedCounter++;
}
else if (strlen(median) > 0)
    printf("'Xs' already entered - ignored.\n", nameArray[index]);
} while (strlen(median) > 0);

/* Prompt user for total number of medians and the number of
   iterations program should perform while looking for optimal
   medians. Check for and adjust illegal values. */

printf("Enter total number of medians and number of iterations:\n");
fgets(buffer, BUFSIZ, stdin);
sscanf(buffer, "%d %d", &totMed, &iterations);
*totMed = medianQueue->length;
if (*totMed < 0) *totMed = 0;
if (((medianQueue->length == fixedMedCounter) && (medianQueue->length != 0))
    || (*iterations < 1))
    *iterations = 0;

/* Prompt user to enter size difference in domains (as percent)
   maxDif = difference between largest and smallest domain
   adjDif = difference between consecutive domains in queue */

printf("Enter size differences between domains (as percent):\n");
printf("Any two domains, Largest and smallest:\n");
fgets(buffer, BUFSIZ, stdin);
sscanf(buffer, "%f %f", &adjDif, &maxDif);
if (*adjDif < 0) *adjDif = 0;
if (*maxDif < 0) *maxDif = 0;
if (*adjDif > 100) *adjDif = 100;
if (*maxDif > 100) *maxDif = 100;

/* If any variable medians are to be used, select these at random and
   add them to the queue. */

i = 0;
srand((int)(adjDif + maxDif));
while (i < *totMed)
{
    index = rand() % endArray;
    if ((index < endArray) && !nodeInQ(medianQueue, index))
    {
        nQ(medianQueue, index, -1, 0);
        i++;
    }
}
*totMed = medianQueue->length - fixedMedCounter;
return(medianQueue);
} /* getMedians */

```

```

/*****
void prMedians(FILE *out_file, q_t *medQ, int endArray)
/* Function which prints the devices belonging to each median to
   a file.
   PRE: Medians must be contained in the queue medQ. endArray must
        indicate the end of the arrays used, and out_file must be ready
        for writing.
   POST: Domains of nodes belonging to each median are printed to file */
{
    qnode_t *current;
    int lineCount, i, j;

    current = medQ->head;
    lineCount = 0;
    printf("\n\nPrinting median sets to file ... \n");
    fprintf(out_file, "\n");

    /* The closest median for each device is at this point stored in the
       working array seenArray. Scan the queue and use pointer current to
       keep track of current position. Current->orig is the index of the
       median. SeenArray[i] contains the index of the median to which node
       i belongs. If the two match, the node belongs to the set covered
       by the current median, so print it. */

    while (current != NULL)
    {
        fprintf(out_file, "%s has %d nodes:\n", nameArray[current->orig],
                current->dest);
        j = 0;
        for (i = 0; i < endArray; i++)
        {
            if (seenArray[i] == current->orig)
            {
                fprintf(out_file, "%s ", nameArray[i]);
                j++;
                if ((j % 9) == 0)
                {
                    fprintf(out_file, "\n");
                }
            }
        }
        fprintf(out_file, "\n\n");
        current = current->next;
    }
} /** prMedians **/

/*****/

int getClosestMedian(q_t *medianQueue, int index)
/* Function finds the median which is the closest to node "index"
   and returns it to the calling function.
   PRE: Medians are contained in medianQueue.
   POST: Index of closest median is returned. */
{
    int minDist, nearestMed;
    qnode_t *current;

    /* Initialize the minimum distance to some arbitrarily large number
       and position the current pointer at the head of the median queue.
       Scan the median queue and the appropriate values in the distance
       array to determine which is the closest median to "index" node
       and returns the index of the closest median. */

    minDist = LARGENUM;
    nearestMed = LARGENUM;
    current = medianQueue->head;

    if (current == NULL) {
        printf("\nWARNING - no medians entered\n");
        return(index);
    }
    while (current != NULL)
    {
        if (distArray[index][current->orig] < minDist)
        {
            minDist = distArray[index][current->orig];
            nearestMed = current->orig;
        }
        current = current->next;
    }
    return(nearestMed);
} /** getClosestMedian **/

```



```

/*****
int getfarthestNode(q,t *medQ, int median, int endArray)
/* function returns the index of the farthest node from the
   given median. This function is used only if two or more medians would
   converge on the same value, and it is a means of further randomizing
   the selection of candidate medians.
   PRE: medQ contains the list of medians, median is a candidate median
        which is already present in the queue, and endArray indicates the
        end of the arrays used.
   POST: Index value of farthest node not already in the median queue is
         returned as the new candidate median. */
{
    int i, maxDist, farNode, index;

    maxDist = 0;
    farNode = 0;

    for (i = 0, i < endArray; i++)
    {
        if ((distArray[i][median] > maxDist) &&
            (!nodeInQ(medQ, i)))
        {
            maxDist = distArray[i][median];
            index = i;
        }
    }
    return(index);
} /* getfarthestNode */

/*****
/* function which balances the regions associated with each median.
   When this function is called, seenArray contains the closest median of
   each node. This function calculates how many nodes belong to each median
   and stores this information in medQ->dest. It then sorts the median queue
   by descending order of the number of nodes assigned to each median and
   calculates the size of each region so that all regions
   except the last one are equal (the last region has fewer nodes than the
   rest). It then examines each median in the queue and transfers any surplus
   nodes from the current median to the next median in the queue until all
   nodes are redistributed among the medians. */

void equalRegions(q,t *medQ, int endArray, int maxTolerance, int adjTolerance)
/* PRE: medQ contains the list of medians, endArray indicates the end of
   the arrays used, maxTolerance is the maximum node difference between
   largest and smallest domains, adjTolerance is maximum node
   difference between any two domains.
   POST: Nodes are bumped from one domain to another to satisfy the
         tolerance requirements. Indices of nearest median for each node
         are recorded in seenArray. */
{
    int rsize, i, j, oldmed, newmed, bumpnodes, mindist, maxdist, index;
    int domainDif;
    qnode t *current;

    /* Calculate size of balanced region. Return to calling function if
       there are fewer than two medians in the queue. */

    if (medQ->length < 2) return;
    rsize = endArray/medQ->length;
    domainDif = endArray % medQ->length;
    if (domainDif != 0) rsize++;
    domainDif = medQ->length - domainDif;
    current = medQ->head;

    /* Calculate how many nodes belong to each median and sort the
       median queue in descending order of this number */

    while (current != NULL)
    {
        current->dest = 0;
        for (i = 0, i < endArray; i++)
        {
            if (seenArray[i] == current->orig) current->dest++;
        }
        current = current->next;
    }
    sortQ(medQ);

```

```

/* Examine each node in turn and transfer excess nodes which are
   closest to next median in the queue and farthest from current
   median to the next median. */

current = medQ->head;
while (current->next != NULL) {
    oldmed = current->orig;
    newmed = current->next->orig;
    bumpnodes = current->dest - rsize - adjTolerance;
    if (bumpnodes < 0) bumpnodes = 0;
    current->dest -= bumpnodes;
    current->next->dest += bumpnodes;
    for (i = 0; i < bumpnodes; i++) {
        mindist = LARGENUM;
        maxdist = 0;
        index = 0;
        for (j = 0; j < endArray; j++) {
            if ((seenArray[j] == oldmed) &&
                (distArray[j][newmed] < mindist) &&
                (distArray[j][oldmed] > maxdist)) {
                mindist = distArray[j][newmed];
                maxdist = distArray[j][oldmed];
                index = j;
            }
        }
        seenArray[index] = newmed;
    } /* FOR loop bumping nodes */
    current = current->next;
} /* WHILE balancing adjacent medians */

sortQ(medQ);
if (domainDif > maxTolerance) return;

while (medQ->head->dest - medQ->tail->dest > maxTolerance)
{
    oldmed = medQ->head->orig;
    newmed = medQ->tail->orig;
    bumpnodes = medQ->head->dest - medQ->tail->dest - maxTolerance;
    medQ->head->dest -= bumpnodes;
    medQ->tail->dest += bumpnodes;
    for (i = 0; i < bumpnodes; i++)
    {
        mindist = LARGENUM;
        maxdist = 0;
        index = 0;
        for (j = 0; j < endArray; j++)
        {
            if ((seenArray[j] == oldmed) &&
                (distArray[j][newmed] < mindist) &&
                (distArray[j][oldmed] > maxdist))
            {
                mindist = distArray[j][newmed];
                maxdist = distArray[j][oldmed];
                index = j;
            }
        }
        seenArray[index] = newmed;
    } /* FOR loop bumping nodes */
    sortQ(medQ);
} /* WHILE balancing maximum and minimum medians */
} /* equalRegions */

/*.....*/

/* Function which iteratively calculates the absolute median
   of a region until the solution converges. MedQ is the median queue
   which records the local optimum of one iteration and optMedQ is
   a median queue which records the global optimum from all iterations.
   Variable globalMin records the global minimum sum of distances from
   every node to its respective median. Intermediate results are written
   to out_file
   *out_file      log file to which intermediate results are written
   *medQ          - queue containing current candidate medians
   *optMedQ       - queue containing global optimal medians found so far
   endArray       - index of next free location in the arrays
   iteration      - local counter for the current iteration
   *iterCounter   - counter recording the total combinations examined
   *globalMin     - global minimum sum of distances for optimal medians
   *globalIter    - counter recording occurrence of global optimal iteration
   *globalSubIter  - occurrence of global optimal subiteration
   maxTolerance   - node difference between largest and smallest domain
   adjTolerance   - node difference between any two domains */

```

```

void localMedians(FILE *out_file, q_t *medQ, q_t *optMedQ, int endArray,
    int iteration, int *iterCounter, int *globalMin, int *globalIter,
    int *globalSubIter, int maxTolerance, int adjTolerance)
/* PRE All variables must be initialized before first call to this
function, queues must contain the appropriate sets of medians,
and out file must be ready for writing.
POST Function performs one complete iteration and the required number
of subiterations, and updates counters and global optimum. */
{
    qnode_t *current, *optcur;
    int i, j, medIndex, oldVal, newVal, oldtotal, newtotal,
    int counter, divergeCount,

/* If median queue is empty return, there is nothing to calculate. */
    if (medQ > length == 0)
    {
        printf("WARNING - no initial medians entered.\n");
        return;
    }

/* Record the initial "seed medians" which were entered at start of
iteration. */

    if (medQ > length > 1)
    {
        fprintf(out_file, "Original medians:\n");
        printf("Original medians: n");
        current = medQ->head,
        while (current != NULL)
        {
            fprintf(out_file, "%s ", nameArray[current->orig]);
            printf("%s ", nameArray[current->orig]);
            current = current->next;
        }
        fprintf(out_file, "\n n");
        printf(" n n");
    }

/* Initialize counters and start DO LOOP which will execute until
solution converges, until it diverges 10 times, or until it
is repeated 50 times. Variable newtotal records the sum of
distances to all nodes from their closest median during the
current iteration, oldtotal is the sum of distances from previous
iteration, divergeCount is a counter recording the number
of times the current solution was worse than the previous one,
and counter is a loop counter. */

    newtotal = 0,
    counter = 0,
    divergeCount = 0,
do {
    oldtotal = newtotal,
    newtotal = 0;

/* Calculate the closest median for each node and store this in
seenArray. If there is more than one median, balance regions. */

    initSeenArray(),
    for (i = 0, i < endArray, i++)
        seenArray[i] = getClosestMedian(medQ, i),
    if (medQ > length > 1)
        equalRegions(medQ, endArray, maxTolerance, adjTolerance),

/* Examine each region and calculate the sum of distances between
a node and every other node of the same region. Store the results
in rsumArray. The minimum of these distances is the absolute
median for that region. */

    current = medQ->head,
    while (current != NULL)
    {
        medIndex = current->orig,
        for (i = 0, i < endArray, i++)
            rsumArray[i] = 0,

        for (i = 0, i < endArray, i++)
            for (j = 0, j < endArray, j++)
                if ((seenArray[i] == medIndex) && (seenArray[j] == medIndex))
                    rsumArray[i] += distArray[i][j],

```

```

/* Calculate the absolute median of each domain */
oldVal = rsumArray[medIndex];
newVal = LARGENUM;
for (i = 0; i < endArray; i++)
{
    if ((rsumArray[i] > 0) && (rsumArray[i] < newVal))
    {
        newVal = rsumArray[i];
        medIndex = i;
    }
}
if (newVal == LARGENUM) newVal = 0;

/* If there is only one median, then absolute median of region is
the absolute median of the entire network. The solution is done
so return */

if ((medQ->length == 1) && (medQ->head->flag != 1))
{
    printf("Absolute median: %s', Distance sum: %d \n",
        nameArray[medIndex], rsumArray[medIndex]);
    fprintf(out_file, "Absolute median: %s', Distance sum: %d \n",
        nameArray[medIndex], rsumArray[medIndex]);
    *globalMin = rsumArray[medIndex];
    optMedQ->head->orig = medIndex;
    return;
}

/* If there are multiple medians, improve on current solution. If
the current median is a fixed median, do nothing. If the new
candidate median is not already in the queue and current median
is not fixed, replace current median by candidate median.
Otherwise, if candidate median is already in the queue, select
a candidate median which is as far from current median as possible */

if ((current->flag == 1) || (current->orig == medIndex))
{
}
else if (!nodeInQ(medQ, medIndex))
{
    current->orig = medIndex;
}
else
{
    medIndex = getfarthestNode(medQ, current->orig, endArray);
    current->orig = medIndex;
}

newtotal += newVal;
current = current->next;
} /* WHILE examining medians in queue */

/* If the current solution differs from previous solution, print
local optimum, update counters, and prepare to repeat 100 1000 */

if (newtotal != oldtotal)
{
    current = medQ->head;
    counter++;
    fprintf(out_file, "%2d: %3d - sum of distances n",
        counter, newtotal);
    printf("%2d: %3d - sum of distances n",
        counter, newtotal);

    while (current != NULL)
    {
        fprintf(out_file, "%s %d ", nameArray[current->orig],
            current->dest);
        printf("%s-%d ", nameArray[current->orig], current->dest);
        current = current->next;
    }
    printf(out_file " \n");
    printf(" \n");
} /* If current solution differs from last one */

```

```

/* If current solution is better than global optimum update global
   optimum */
if (newtotal < *globalMin)
{
    current = medQ->head;
    optcur = optMedQ->head;
    while (current != NULL)
    {
        optcur->orig = current->orig;
        optcur->flag = current->flag;
        *globalMin = newtotal;
        *globalSubIter = counter;
        *globalIter = iteration;
        current = current->next;
        optcur = optcur->next;
    }
}

/* If current solution same as last solution, exit loop and assume
   that solution has converged. If current solution is worse than
   last solution, update counter and hope that next solution
   improves. If solution diverges several times, solution is cycling
   so exit after 5 cycles. */

if ((oldtotal - newtotal) == 0)
{
    printf("Solutions converged \n");
    break;
}
else if ((oldtotal - newtotal) < 0) divergeCount++;

if (divergeCount == 5)
    printf("Solutions diverged %d times.\n", divergeCount);
if (counter == 50)
    printf("Maximum permitted iterations reached.\n");

} while ((divergeCount < 5) && (counter < 50)).

*iterCounter += counter;
} /* localMedians */

/***** */

void modSimFile(FILE *in_file, FILE *out_file, int endArray)
/* Function modifies the Dragon simulator data file, nm2.no, by changing
   the domain for each node to correspond to the nearest median. Some
   manual changes in the nm2.no and nm2.sy files are still necessary after
   this function is finished.
PRE   in_file must contain the nm2 no Dragon simulator file and must
       be ready for reading, out_file will store the modified simulator
       file and must be ready for writing, endArray indicates the end
       of arrays used
POST  Dragon topology definition file nm2.no is modified to correspond
       to optimal location of data collectors and written to a new file */
{
    char buffer[BUFSIZ];
    char *startline, *midline, *endline;
    int index;

    printf("Modifying simulator topology file \n");

    /* Parse each line of text file and look for device name which is
       contained in double quotation marks. If this string is a valid name
       present in the topology, variable index will be set to its location
       in nameArray and will be > 1. */

    while(fgets(buffer, BUFSIZ, in_file) != NULL)
    {
        index = 1;
        startline = strtok(buffer, "\"");
        midline = strtok(NULL, "\"");
        endline = strtok(NULL, "\n");
        if ((midline != NULL) && (strlen(midline) > 0) &&
            (strlen(midline) <= NAMELEN))
        {
            index = getIndex(midline, endArray);
        }
    }
}

```

```

if (stringstr(startLine,"object") && (index >= 0))
{
    fprintf(out_file,"%s\\%s\\%s\\n", startLine, midline, endline);
    while (fgets(buffer,BUFSIZE,in file) != NULL)
    {
        startLine = strtok(buffer,"");
        midline = strtok(NULL,"");
        endline = strtok(NULL,"\\n");

        /* Look for key word "serviceName", and if present, replace the
        name of the domain the device belongs to by the name of the
        closest median as recorded in seenArray. Restore all line
        fragments and complete (even blank) lines to their original
        state. */

        if (stringstr(startLine,"serviceName"))
        {
            fprintf(out_file,"%s\\%s\\", startLine,
                nameArray[seenArray[index]]);
            if (endline) fprintf(out_file, "%s", endline);
            else          fprintf(out_file, "\\n");
            break;
        }
        else
        {
            if (endline)
                fprintf(out_file,"%s\\%s\\%s\\n", startLine, midline, endline);
            else if (midline)
                fprintf(out_file,"%s\\%s\\", startLine, midline);
            else if (startLine)
                fprintf(out_file,"%s", startLine);
            else
                fprintf(out_file,"\\n");
        }
    } /* WHILE looking for data collector line */
} /* IF device connected to data collector found */
else
{
    if (endline)
        fprintf(out_file,"%s\\%s\\%s\\n", startLine, midline, endline);
    else if (midline)
        fprintf(out_file,"%s\\%s\\", startLine, midline);
    else if (startLine)
        fprintf(out_file,"%s", startLine);
    else
        fprintf(out_file,"\\n");
} /* WHILE not end of file */
} /* modSimfile */

/*****
/***** MAIN PROGRAM *****/
/*****/

main(int argc, char *argv[])
{
    int lineCount, noOutput, noInput, index, endArray, noMatrix, noTop, medIndex;
    int i, j iterations, runCounter, globalMin;
    int globalIter, globalSubIter, fixedMedFlag, iterCounter;
    int maxTolerance, adjTolerance, totMed, actualDistance;
    int hr, min, sec;
    float maxDif, adjDif;
    FILE *in_file, *out_file, *matrix_file, *sim_old_file, *sim_new_file;
    file_spec input_f, output_f, matrix_f, top_f;
    tree_t *root;
    time_t start_time, finish_time;
    name_t sizelimit;
    q_t *queue, *medianQueue, *optMedQ;
    qnode_t *current;

    /* Print program title, interpret command line arguments and prompt user
    to enter any missing information. Open the required files or abort if
    the specified input files do not exist */

    printf(" nPROGRAM TO LOCATE DATA COLLECTORS IN B&B/ST NETWORK n ");

    noInput = 0;
    noOutput = 0;
    noMatrix = 0;
    noTop = 0;

```

```

if (argc > 4) {
    strcpy(input_f,argv[1]);
    strcpy(matrix_f,argv[2]);
    strcpy(output_f,argv[3]);
    strcpy(top_f,argv[4]);
}
else if (argc > 3) {
    strcpy(input_f,argv[1]);
    strcpy(matrix_f,argv[2]);
    strcpy(output_f,argv[3]);
    noTop = 1;
}
else if (argc > 2) {
    strcpy(input_f,argv[1]);
    strcpy(matrix_f,argv[2]);
    noOutput = 1;
    noTop = 1;
}
else if (argc > 1) {
    strcpy(input_f,argv[1]);
    noOutput = 1;
    noMatrix = 1;
    noTop = 1;
}
else {
    noInput = 1;
    noOutput = 1;
    noMatrix = 1;
    noTop = 1;
}

if (noInput || noOutput || noMatrix || noTop) {
    printf("\nUSAGE: %s file1 file2 file3 file4\n", argv[0]);
    printf(" file1 - INPUT: original network topology\n");
    printf(" file2 - INPUT: distance matrix\n");
    printf(" file3 - OUTPUT: log of median selection\n");
    printf(" file4 - OUTPUT: simulator topology definition\n\n");
}

if (noInput) {
    printf("Enter topology data file name:\n");
    gets(input_f);
}

in_file = fopen(input_f,"r");
if (!in_file)
{
    printf("\nERROR cannot open topology data file '%s'.\n",input_f);
    printf("=====> ABORTING <=====\n");
    exit(0);
}

if (noMatrix) {
    printf("Enter distance matrix file name:\n");
    gets(matrix_f);
}

if (strlen(matrix_f) == 0)
    strcpy(matrix_f,"matrix.out");

matrix_file = fopen(matrix_f,"r");

if (!matrix_file)
{
    printf("\nERROR - cannot open matrix data file '%s'.\n",matrix_f);
    printf("=====> ABORTING <=====\n");
    exit(0);
}

if (noOutput) {
    printf("Enter log file name:\n");
    gets(output_f);
    if (strlen(output_f)==0) strcpy(output_f,"log out");
}

out_file = fopen(output_f,"w");

if (noTop) {
    printf("Enter new simulator topology definition file:\n");
    gets(top_f);
}

```

```

if (strlen(top_f)==0)
{
    sim_new_file = NULL;
}
else
{
    sim_new_file = fopen(top_f,"w");
    sim_old_file = fopen("nm2.no","r");
    if (!sim_old_file)
    {
        printf("\nERROR - cannot open nm2.no' file \n");
        printf("=====> ABORTING <=====\n");
        exit(0);
    }
}

/*-----
/* Read the topology data file, construct the binary search tree,
transfer sorted information to arrays, build linked lists of connected
nodes, and record the extent of array used in variable endArray
Read the distance matrix from file. */

initArray();
initSeenArray();
queue = newQ();

root = buildTree(in_file);
lineCount = 0;
treeToArray(root, &lineCount);
endArray = lineCount;

addList(in_file);
distArrayFromFile(matrix_file, endArray);
fclose(matrix_file);

/* Initialize iteration counter, prompt user to enter any fixed
medians and the total number of medians to be used as well as
the number of iterations to be executed. Transfer this information
to the global optimal median queue (optMedQ) */

runCounter = 0;
medianQueue = getMedians(endArray, &totMed, &iterations,
                        &maxDif, &adjDif);
maxTolerance = (int) (endArray*maxDif/100);
adjTolerance = (int) (endArray*adjDif/100);

printf("Enforce size limits for final domains (y/n)^n"),
gets(sizeLimit);
if (stringstr(sizeLimit,"y") || stringstr(sizeLimit,"Y"))
{
    printf("Sizes of final domains will be limited as specified ^n"),
strcpy(sizeLimit,"YES");
}
else printf("Domain size restrictions not enforced in final run ^n"),

optMedQ = newQ();
current = medianQueue->head;
while (current != NULL)
{
    nQ(optMedQ, current->orig, current->dest, current->flag);
    current = current->next;
}

/* Record starting parameters in log file */

start_time = time(NULL);
globalMin = LARGENUM;
globalIter = 1;
globalSubIter = 1;
fixedMedFlag = 0;
iterCounter = 0;
current = medianQueue->head;
while ((current != NULL) && (current->flag != 1))
    current = current->next;

if (current != NULL) fixedMedFlag = 1;

/* Repeat FOR loop for the number of iterations specified by user
During each iteration calculate local optimal solution and update
global optimal solution if need be. Generate next set of random
medians and repeat iteration */

```



```

for (i = 0; i < iterations; i++)
{
    printf("n = %d =====> ITERATION %d <===== n\n",
        ++runCounter);
    fprintf(out_file, "n =====> ITERATION %d <===== n\n",
        runCounter);
    localMedians(out_file, medianQueue, optMedQ, endArray, runCounter,
        &iterCounter, &globalMin, &globalIter, &globalSubIter,
        maxTolerance, adjTolerance);

    /* Generate next set of random medians for those medians which are
       not fixed (median.queue->flag != 1). */

    if (totMed < 2) break;

    current = medianQueue->head,
    while (current != NULL)
    {
        if (current->flag != 1)
        {
            index = rand() % endArray;
            while (nodeInQ(medianQueue, index))
            {
                index = rand() % endArray;
            }
            current->orig = index;
        }
        current = current->next;
    }
    /* Iterations FOR loop */

    /* After iterations are done, optMedQ contains the global optimal
       medians. Find out which nodes belong to which median, replace old
       data collector tags (DC) in typeArray by "OD", and assign the type
       "DC" to devices determined to be optimal medians. */

    /* Now that optimal medians have been determined, assign each node
       to its closest median regardless of size of domains. */

    for (i = 0; i < endArray; i++)
    {
        seenArray[i] = getClosestMedian(optMedQ, i);
        if (strcmp(typeArray[i], "DC") == 0) strcpy(typeArray[i], "OD");
    }

    if (strcmp(sizeLimit, "YES") == 0)
    {
        equalRegions(optMedQ, endArray, maxTolerance, adjTolerance);
        printf("\nAdjusting sized of final domains\n");
    }

    /* Count the number of nodes which belong to each median. */

    current = optMedQ->head,
    while (current != NULL)
    {
        current->dest = 0;
        for (i = 0; i < endArray; i++)
        {
            if (seenArray[i] == current->orig) current->dest++;
        }
        current = current->next;
    }

    /* Calculate final sum of distances from medians to all nodes */

    actualDistance = 0;
    current = optMedQ->head,
    while (current != NULL)
    {
        medIndex = current->orig;
        for (i = 0; i < endArray; i++)
            rsumArray[i] = 0;

        for (i = 0; i < endArray; i++)
            for (j = 0; j < endArray; j++)
                if ((seenArray[i] == medIndex) && (seenArray[j] == medIndex))
                    rsumArray[i] += distArray[i][j];

        actualDistance += rsumArray[medIndex];
        current = current->next;
    }

```

```

finish_time = time(NULL);
hr = (int) ((finish_time - start_time) / 3600);
min = (int) (((finish_time - start_time) % 3600) / 60);
sec = (int) (((finish_time - start_time) % 3600) % 60);

if (iterations == 0) globalMin = actualDistance;

if (optMedQ->length > 0)
{
    fprintf(out_file, "\n***** GLOBAL OPTIMUM *****\n\n");
    fprintf(out_file, "Topology data file: %s', Distance matrix file: %s' \n",
        input_f, matrix_f);
    fprintf(out_file, "Medians: %d, Iterations: %d; Nodes: %d.\n",
        optMedQ->length, iterations, endArray);
    fprintf(out_file,
        "Tolerance: Any 2 domains = %d (%2.2f%%), Maximum = %d (%2.2f%%)\n",
        adjTolerance, adjDif, maxTolerance, maxDif);

    printf("\n***** GLOBAL OPTIMUM *****\n\n");
    printf("Topology data file: %s', Distance matrix file: %s' \n",
        input_f, matrix_f);
    printf("Medians: %d; Iterations: %d; Nodes: %d.\n",
        optMedQ->length, iterations, endArray);
    printf("Tolerance: Any 2 domains = %d (%2.2f%%), Maximum = %d (%2.2f%%)\n",
        adjTolerance, adjDif, maxTolerance, maxDif);

    if (strcmp(sizeLimit, "YES") == 0)
    {
        fprintf(out_file,
            "Domain size limits enforced in final node allocation.\n");
        printf("Domain size limits enforced in final node allocation.\n");
    }
    else
    {
        fprintf(out_file,
            "Domain size restrictions not enforced in final node allocation.\n");
        printf("Domain size restrictions not enforced in final node allocation.\n");
    }

    if (fixedMedFlag)
    {
        fprintf(out_file, "Fixed medians: n");
        printf("Fixed medians: n");
        current = optMedQ->head;
        while (current != NULL)
        {
            if (current->flag == 1)
            {
                fprintf(out_file, "%s ", nameArray[current->orig]);
                printf("%s ", nameArray[current->orig]);
            }
            current = current->next;
        }
        fprintf(out_file, "\n");
    }
    else
    {
        fprintf(out_file, "No fixed medians entered.\n");
        printf("No fixed medians entered.\n");
    }

    fprintf(out_file, "nOptimum in Iteration %5d %10d n",
        globalIter, globalSubIter);
    fprintf(out_file, "Optimal sum of distances: %8d n", globalMin);
    fprintf(out_file, "Combinations examined %8d n", iterCounter);
    fprintf(out_file, "Elapsed time: %10d.%10d %10d n n",
        hr, min, sec);

    fprintf(out_file, "Final sum of distances %8d n n", actualDistance);
    fprintf(out_file, "Optimal medians and corresponding domain sizes: n");

    printf("nOptimum in Iteration: %5d %10d n",
        globalIter, globalSubIter);
    printf("Optimal sum of distances: %8d n", globalMin);
    printf("Combinations examined: %8d n", iterCounter);
    printf("Elapsed time: %10d.%10d %10d n n", hr, min, sec);
    printf("Final sum of distances %8d n n", actualDistance);
    printf("Optimal medians and corresponding domain sizes: n");
}

```

```

current = optMedQ->head;
while (current != NULL)
{
    fprintf(out_file, "%s-%d ", nameArray[current->orig], current->dest);
    printf("%s-%d ", nameArray[current->orig], current->dest);
    strcpy(typeArray[current->orig], "DC");
    current = current->next;
}
fprintf(out_file, "\n");
prMedians(out_file, optMedQ, endArray);
fprintf(out_file, "\n");
}

if (sim_new_file)
{
    modSimFile(sim_old_file, sim_new_file, endArray);
    printf("\nsimulator topology saved as %s'." , top_f);
    fclose(sim_new_file);
    fclose(sim_old_file);
}
else printf("Dragon simulator topology file not updated.\n");

fclose(in_file);
fclose(out_file);
printf(" nlog file saved as %s'." , output_f);
printf(" n n.      ---> NORMAL TERMINATION <====='\n\n");
}

```

Appendix F

NPATH Program

```

#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <sysent.h> /* getpid(),system(),exit(),open(),read(),close() */

#include "bbtypes.h"      /* blackboard data structure definitions */
#include "bbfunctions.h"  /* blackboard functions */
#include "pbase_types.h"  /* PegasusBase data structure defs */

#include "bb_pktype.h"    /* Paul Kalab's structure definitions */
#include "bb_pkfun.h"     /* Paul's general functions */
#include "bb_pathfun.h"   /* Paul's path related functions */

#define KS_NAME "SERV nPath" /* Pegasus knowledge source name */
#define KS_TYPE 11 /* Pegasus KS type */
#define KS_SUBTYPE 1 /* Pegasus KS subtype */
#define PRINT_PATHS 10 /* option for posting list of paths */
#define PRINT_ARCS 11 /* option for posting list of arcs */
#define PRINT_NODES 12 /* option for posting list of nodes */
#define PRINT_ALL 13 /* option for posting all of the above */
#define COMPR_NODES 15 /* topology with compressed nodes read */

/* global structure definitions */
struct request_record request;
struct reply_record reply;
struct process_record process_env;

/*****
***** MAIN NPATH PROGRAM *****/

main()
{
    long last_request, templong, reply_idx;

    name_t origin, dest;
    int numberOfPaths, lineCount, endArray, numberOfArcs;
    int totalArcs, singleArcs, o_index, d_index;
    int pathID, nodeID, option, numberOfNodes, errorflag;
    long candPaths, pathSearched;
    tree_t *root;
    FILE *in_file;
    qlist_t *pathlist, *listPos;
    gnode_t *current;
    q_t *nodeQ, *arcQ;

    /* Get access to the blackboard and register process */

    if (get_access_to_bb(&process_env) == -1) exit(1);
    strcpy(process_env.KS_name, KS_NAME);
    process_env.KS_type = KS_TYPE;
    process_env.KS_subtype = KS_SUBTYPE;
    if (register_KS(&process_env) == -1) exit(1);

    /* Open topology data file or die if it does not exist */

    in_file = fopen("topology.dat", "r");
    if (!in_file)
    {
        printf("nERROR - cannot open topology data file n");
        printf("***** ABORTING ***** n");
        exit(0);
    }

    /* Initialize parameters, read data file, build data structure */

    printf("n** KS TO PROVIDE n CRITICAL PATHS, ARCS, AND NODES ** n");
    printf("2.0 - 7/15/92 - Upgraded for compatibility with BB's 1.1 n");
    printf("2.1 - 8/05/92 - Minor modifications to replies posted to BB n");
    printf("2.2 - 11/18/92 - Modifications to accept user defined arc lengths n");

```

```

errorflag = 0;
initArray();
root = buildTree(in_file);
linecount = 0;
treeToArray(root, &lineCount);
endArray = lineCount;
addlist(in_file);
fclose(in_file);

getArcLengths(endArray, &totalArcs, &singleArcs);
printf("waiting for requests from blackboard ... \n\n");

/* Get ready for requests posted to blackboard. */

last_request = interested(&process_env, REQUEST_ARRAY);

/* while there is no problem with the blackboard. */
while (wait_change(&process_env) != -1)
{
    /* Get index to new request */

    templong = request_last_id(&process_env);

    /* Check whether a new request was posted to the BB */
    if (templong != 0)
    {
        last_request = templong;
        get_request(&process_env, last_request, &request);

        /* Check if request was for this server. */

        if (request.function_no == COMPR_NODES)
        {
            delistArray(endArray);
            killTree(root);
            printf("***** READING NEW NETWORK TOPOLOGY ***** \n\n");
            in_file = fopen("topology.cmp", "r");
            initArray();
            root = buildTree(in_file);
            lineCount = 0;
            treeToArray(root, &lineCount);
            endArray = lineCount;
            addlist(in_file);
            fclose(in_file);
            getArcLengths(endArray, &totalArcs, &singleArcs);
        }
        else if ((request.function_no > 9) &&
            (request.function_no < 15))
        {
            /* Extract search parameters */

            strcpy(origin, request.request_str0);
            strcpy(dest, request.request_str1);
            numberOfPaths = (int) request.request_para0;
            numberOfArcs = (int) request.request_para1;
            numberOfNodes = request.request_para2;

            /* If number of paths requested < 0, stop process and exit */

            if (numberOfPaths < 0)
            {
                delistArray(endArray);
                killTree(root);
                uninterested(&process_env, REQUEST_ARRAY);
                check_out(&process_env);
                exit(0);
            }

            /* Initialize temporary array, get array indices for origin
            and destination, and check that requested nodes exist.
            NOTE paths are recorded in reverse order and hence
            d_index is the "origin" index and o_index is "destination. */

            initSeenArray();
            d_index = getIndex(origin, endArray);
            o_index = getIndex(dest, endArray);

```

```

if (nameInTree(root, origin) == NULL)
{
    printf("ERROR - Node '%s' not in network.\n", origin);
    errorFlag = 1;
}
if (nameInTree(root, dest) == NULL)
{
    printf("ERROR - Node '%s' not in network.\n", dest);
    errorFlag = 1;
}
if (strcmp(origin, dest) == 0)
{
    printf("ERROR - Origin and destination are same node.\n");
    errorFlag = 1;
}

if (errorFlag)
{
    /* Notify client process of error condition */

    reply_idx = reply_last_idx(&process.env);
    strcpy(reply_reply_str0, "ERROR");
    reply_reply_para0 = -100;
    reply_reply_no = last_request;
    reply_idx = post_reply(&process.env, &reply);
    errorFlag = 0;
}
else
{
    /* There is no problem so proceed with calculations
       "pathlist" is pointer to linked list of linked lists
       giving the paths in ascending order of their length */

    option = request_function_no;
    pathlist = findShortestPaths(origin, dest, numberOfPaths,
                                endArray, &endPaths, &pathSearched);
    if (pathlist == NULL)
    {
        /* Print message that no path exists, notify client process
           that error condition occurred and change option request
           to bypass calculation of critical nodes and arcs */

        printf("WARNING: %s not connected to %s\n", origin, dest);
        reply_idx = reply_last_idx(&process.env);
        strcpy(reply_reply_str0, "ERROR");
        reply_reply_para0 = 100;
        reply_reply_no = 1;
        reply_idx = post_reply(&process.env, &reply);
        option = PRINT_PATHS;
    }

    listPos = pathlist;

    if ((option == PRINT_PATHS) || (option == PRINT_ALL))
    {
        /* Paths are posted to blackboard. End of each path
           is signalled by "NextPath" in reply str0 and end of path
           data by "EndPaths" */

        pathID = 0;
        while ((listPos != NULL) && (pathID < numberOfPaths))
        {
            pathID++;
            nodeID = 0;
            current = listPos->head;
            while (current != NULL)
            {
                reply_idx = reply_last_idx(&process.env);
                strcpy(reply_reply_str0, nameArray[current->orig]);
                reply_reply_para0 = pathID;
                reply_reply_para1 = ++nodeID;
                reply_reply_para2 = listPos->length;
                reply_reply_no = last_request;
                reply_idx = post_reply(&process.env, &reply);
            }
        }
    }
}

```

```

    if (current->next == NULL)
    {
        reply_idx = reply_last_id(&process_env);
        strcpy(reply_reply_str0, nameArray[current->dest]);
        reply_reply_para0 = pathID;
        reply_reply_para1 = ++nodeID;
        reply_reply_para2 = listPos->length;
        reply_reply_no = last_request;
        reply_idx = post_reply(&process_env, &reply);
    }
    current = current->next;
}
reply_idx = reply_last_id(&process_env);
reply_reply_para0 = 0;
strcpy(reply_reply_str0, "NextPath");
reply_reply_no = last_request;
reply_idx = post_reply(&process_env, &reply);
listPos = listPos->next;
}
printf(" n");
reply_idx = reply_last_id(&process_env);
reply_reply_no = last_request;
reply_reply_para2 = 1;
reply_reply_para1 = candPaths;
reply_reply_para0 = pathSearched;
strcpy(reply_reply_str0, "EndPaths");
reply_idx = post_reply(&process_env, &reply);

} /* If paths are posted to blackboard. */

if ((option == PRINT_ARCS) || (option == PRINT_ALL))
{
    /* "arcQ" is pointer to queue listing all arcs found in
       the specified number of paths in descending frequency
       of occurrence of the arcs. frequency is stored in
       field "current->dest" End of transmission is
       signalled by "EndArcs" */

    arcQ = countArcsInPaths(pathList, numberOfPaths);
    nodeID = 0;
    current = arcQ->head;
    while ((current != NULL) && (nodeID < numberOfArcs))
    {
        reply_idx = reply_last_id(&process_env);
        strcpy(reply_reply_str0, nameArray[current->orig]);
        strcpy(reply_reply_str1, nameArray[current->flag]);
        reply_reply_para0 = current->dest;
        reply_reply_para1 = ++nodeID;
        reply_reply_no = last_request;
        reply_idx = post_reply(&process_env, &reply);
        current = current->next;
    }
    reply_idx = reply_last_id(&process_env);
    reply_reply_no = last_request;
    reply_reply_para2 = 2;
    reply_reply_para1 = candPaths;
    reply_reply_para0 = pathSearched;
    strcpy(reply_reply_str0, "EndArcs");
    reply_idx = post_reply(&process_env, &reply);
    killQ(arcQ);
    free(arcQ);

} /* If arcs are posted to blackboard. */

if ((option == PRINT_NODES) || (option == PRINT_ALL))
{
    /* "nodeQ" is pointer to queue listing all nodes found in
       the specified number of paths in descending frequency
       of occurrence of the nodes frequency is stored in
       field "current->dest" End of transmission is signalled
       by "EndNodes" */

    nodeQ = countNodesInPaths(pathList, numberOfPaths);
    nodeID = 0;
    current = nodeQ->head;

```

```

while ((current != NULL) && (nodeID < numberOfNodes))
{
    reply_idx = reply_last_idx(&process_env);
    strcpy(reply.reply_str0, nameArray[current->id]);
    strcpy(reply.reply_str1, typeArray[current->type]);
    reply.reply_para0 = current->dest;
    reply.reply_para1 = ++nodeID;
    reply.reply_no = last_request;
    reply_idx = post_reply(&process_env, &reply);
    current = current->next;
}

reply_idx = reply_last_idx(&process_env);
reply.reply_no = last_request;
reply.reply_para2 = 3;
reply.reply_para1 = candPaths;
reply.reply_para0 = pathSearched;
strcpy(reply.reply_str0, "IndNodes");
reply_idx = post_reply(&process_env, &reply);
killQ(nodeQ);
free(nodeQ);

} /* If nodes are posted to blackboard */

/* Delete current list of paths and get ready for next run */
killAllPaths(pathlist);

} /* ELSE if no error condition exists */
} /* If request function is 10 to 13 */
} /* If request posted (tempLong == 0) */

/* For this program, the ELSE should never be reached */
else printf("ERROR: process called and no message waiting\n");

} /* WHILE waiting for change */

} /****** npath server program *****/

```


3 OF/DE 3

PM-1 3 1/2" x 4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

Appendix G

Node Compression Functions – bb_compfun.c

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "bb_pktype.h" /* Paul's data structure definitions */
#include "bb_compfun.h" /* node compression functions */
#include "bb_pkfun.h" /* general functions */

/*****

list_t *instlist(char *name)
/* Creates a new node of linked list for node compression list.
PRE: None.
POST: New list node has been created. */
{
    list_t *newlist;

    newlist = MALLOC(list_t);
    if (newlist == NULL) {
        printf("\nERROR - null pointer returned by MALLOC in instlist.\n");
        printf("====> ABORTING <====\n");
        exit(0);
    }
    strcpy(newlist->name, name);
    newlist->next = NULL;
    return(newlist);
} /** instlist **/

*****/

void insInList(tree_t *treeNode, name_t name, int *arcCount)
/* Inserts node "name" into linked list attached to tree node
Linked list remains sorted in lexicographic order
This function is used only by the compression functions since network
topology is now being stored in listArray.
PRE: Tree node to which list is being attached must exist.
POST: Constructs device connectivity for network, one arc at a time */
{
    list_t *newlist, *current, *previous;

    if (treeNode->list == NULL) {
        treeNode->list = instlist(name);
        (*arcCount)++;
        return;
    }
    current = treeNode->list;
    if (strcmp(name, current->name) < 0) {
        newlist = instlist(name);
        newlist->next = current;
        treeNode->list = newlist;
        (*arcCount)++;
        return;
    }

    while ((current->next != NULL) &&
           (strcmp(name, current->name) > 0))
    {
        previous = current;
        current = current->next;
    }
    if (strcmp(name, current->name) > 0)
        current->next = instlist(name);
    else if (strcmp(name, current->name) < 0) {
        newlist = instlist(name);
        previous->next = newlist;
        newlist->next = current;
    }
    else return;
    (*arcCount)++;
    return;
} /** insInList **/

```

```

/*****/

void killList(tree_t *aTree)
/* Deletes connectivity list attached to tree node.
   Function is recursive and performs a post-order tree traversal.
   PRE: Tree node must exist and have a list attached to it.
   POST: List pointer of tree node is set to NULL and list is deleted. */
{
    list_t *current, *next;

    if(aTree != NULL) {
        killList(aTree->left);
        killList(aTree->right);
        current = aTree->list;
        aTree->list = NULL;
        while (current != NULL) {
            next = current->next;
            free(current);
            current = next;
        }
    }
} /* killList */

/*****/

void printList(list_t *aList, FILE *out_file)
/* Prints linked list (device connectivity) to output file.
   PRE: The output file must be set for writing and list should exist.
   POST: List is written to output file. */
{
    list_t *current;

    current = aList;
    while (current != NULL) {
        fprintf(out_file, "%s", current->name);
        current = current->next;
    }
    fprintf(out_file, "\n");
} /* printList */

/*****/

void printTree(tree_t *aTree, int *lineCount, FILE *out_file)
/* Recursive function which performs an in-order tree traversal and
   prints the tree node names and attached linked lists (entire
   network connectivity.
   PRE: LineCount counter must be initialized to 0 and output file
        must be set for writing.
   POST: Network connectivity is printed to output file. */
{
    if (aTree != NULL) {
        printTree(aTree->left, lineCount, out_file);
        if (aTree->list != NULL) {
            (*lineCount)++;
            fprintf(out_file, "%d= %s %s:", *lineCount, aTree->name, aTree->type);
            printList(aTree->list, out_file);
        }
        printTree(aTree->right, lineCount, out_file);
    }
} /* printTree */

/*****/

void scanTree(tree_t *aTree, name_t name, type_t type, int *counter)
{
    /* Prints node name and type if it matches the search criteria.
       PRE: Topology must be stored in binary search tree whose root is aTree.
            Counter must be initialized to zero.
       POST: None name and type printed to screen. */
    if (aTree != NULL) {
        scanTree(aTree->left, name, type, counter);
        if (((strcmp(name, "$all$")==0) && (strcmp(type, "$all$")==0)) ||
            ((strcmp(name, "$all$")==0) && (stringstr(aTree->type.type))) ||
            ((strcmp(type, "$all$")==0) && (stringstr(aTree->name.name))) ||
            ((stringstr(aTree->name.name) && (stringstr(aTree->type.type))))
        {
            printf("%s %s: ", aTree->name, aTree->type);
            (*counter)++;
            if ((*counter % 10) == 0) printf("\n");
        }
        scanTree(aTree->right, name, type, counter);
    }
} /* scanTree */

```

```

/*****
void listDevices(tree_t *aTree)
/* Requests search parameters for printing device names and types
   contained in the current topology.
   PRE: Topology must be stored in binary search tree whose root is alree
   POST: All nodes matching search parameters are printed to screen */
{
    name_t name;
    type_t type;
    int counter;

    printf("\nDevice name substring or <CR> for all device names:\n");
    gets(name);
    if (!strlen(name)) strcpy(name, "$all$");

    printf("\nDevice type or <CR> for all device types:\n");
    gets(type);
    if (!strlen(type)) strcpy(type, "$all$");

    counter = 0;
    scanTree(aTree, name, type, &counter);
    printf("\n%d nodes found.\n", counter);
} /** listDevices **/

/*****
/***** COMPRESSION FUNCTIONS *****/

void devCount(tree_t *aTree, name_t name, comp_t *device)
/* Counts devices according to type which are contained in compressed node
   PRE: Device counters must be initialized to zero prior to calling
        this function. Initialization is automatic when node of linked
        list is created by function makeComprList or addtoComprList.
   POST: Device type field of each node of the compression list contains
        the total number of that type of device contained in compressed
        node. */
{
    if (aTree != NULL)
    {
        devCount(aTree->left, name, device);
        if ((strlen(name) > 0) && (stringstr(aTree->name, name)))
        {
            if (stringstr(aTree->type, "BR")) device->br++;
            else if (stringstr(aTree->type, "DC")) device->dc++;
            else if (stringstr(aTree->type, "FP")) device->fp++;
            else if (stringstr(aTree->type, "HU")) device->hu++;
            else if (stringstr(aTree->type, "LN")) device->ln++;
            else if (stringstr(aTree->type, "MB")) device->mb++;
            else if (stringstr(aTree->type, "RT")) device->rt++;
            else if (stringstr(aTree->type, "WS")) device->ws++;
            else device->sp++;
        }
        devCount(aTree->right, name, device);
    }
} /** devCount **/

/*****

comp_t *getCompNode(comp_t *comprList, name_t name)
/* Functions returns the first location which has name as the substring
   of its full name. If node does not exist in comprlist, function returns
   NULL.
   PRE: comprlist points to the head of the linked list of nodes to be
        compressed.
   POST: function returns pointer to location of node in list or NULL if
        node not found. */
{
    comp_t *current;
    int notfound;

    notfound = 1;
    current = comprList;
    while((current != NULL) && (notfound == 1))
    {
        if (stringstr(name, current->n_old)) notfound = 0;
        else current = current->nex;
    }
    return(current);
} /** getCompNode **/

```

```

/*****/
comp_t *makeComplst(FILE *in_file)
/* function builds the linked list of nodes to be compressed from the
input file in_file.
PRE: in file must contain the node compression information in the
correct format.
POST: Returns a pointer to the head of a linked list which contains
the old name and new name of nodes to be compressed. */
{
    char *buf, *old, *newItem;
    char buffer[BUFSIZ];
    name_t n_new, n_old;
    int rename;
    comp_t *current, *newNode;

    /* Initialize pointer and return NULL if in_file does not exist. */
    current = NULL;
    if (in_file == NULL) return(current);

    /* Parse each line of input file. Look for the colon delimiter which
indicates that any name substring following colon should be replaced
by string preceding the colon. If colon not present, old and new name
are the same. */

    while (fgets(buffer, BUFSIZ, in_file) != NULL) {
        buf = strtok(buffer, "\n");
        if (stringstr(buf, ":")) rename = 1;
        else rename = 0;
        if (rename) {
            newItem = strtok(buf, ":");
            strcpy(n_new, newItem);
            if (strlen(newItem) > NAMELEN - 1) {
                printf("\nERROR - Replacement node name '%s' too long.\n", newItem);
                printf("====> ABORTING <====\n");
                exit(0);
            }
            old = strtok(NULL, " ");
            strcpy(n_old, old);
        }
        else {
            old = strtok(buf, " ");
            strcpy(n_old, old);
            if (old != NULL) strcpy(n_new, n_old);
        }
        if (old == NULL) {
            printf("\nERROR - incorrect format of node compression data file.\n");
            printf("====> ABORTING <====\n");
            exit(0);
        }

        /* First string in line has been decoded and presence of colon checked.
DO LOOP parses the remaining text strings in line and if strings
should be renamed, string preceding colon is assigned as new name.
Otherwise, old and new name are the same. Device type fields of
compression list are initialized to zero. List is built up in
reverse order. */
        do {
            if (old != NULL) {
                strcpy(n_old, old);
                if (rename == 0) strcpy(n_new, n_old);
                newNode = MALLOC(comp_t);
                newNode->br = 0;
                newNode->dc = 0;
                newNode->fp = 0;
                newNode->hu = 0;
                newNode->ln = 0;
                newNode->mb = 0;
                newNode->rt = 0;
                newNode->ws = 0;
                newNode->sp = 0;
                strcpy(newNode->n_old, n_old);
                strcpy(newNode->n_new, n_new);
                newNode->next = current;
                current = newNode;
            }
            old = strtok(NULL, " ");
        } while (old != NULL);
    }
    return(current);
} /* makeComplst */

```

```

/*****
comp_t *compNodes(tree_t *root, comp_t *compStart, FILE *compOut_file)
/* Function which records the total number of devices, listed according
to type, contained in each node to be compressed.
PRE: compOut_file must be open and ready for writing, otherwise
function is bypassed.
POST: breakdown of nodes to be compressed according to type is printed
to file compOut_file. */
{
    int tot, lineCount;
    comp_t *dev;

    if (!comp: t_file) return(compStart);

    dev = compStart;
    printf("Recording nodes to be compressed ...\n");
    lineCount = 0;
    while (dev != NULL)
    {
        lineCount++;
        fflush(stdout);
        devCount(root,dev->n_old,dev);
        tot = dev->br + dev->dc + dev->fp + dev->hu + dev->ln + dev->mb
            + dev->rt + dev->ws + dev->sp;
        fprintf(compOut_file,
            "%2d %s: BR=%2d DC=%2d FP=%2d HU=%2d LN=%2d MB=%2d RT=%2d WS=%3d",
            lineCount, dev->n_old,dev->br,dev->dc,dev->fp,dev->hu,dev->ln,
            dev->mb,dev->rt,dev->ws);
        fprintf(compOut_file," SP=%2d TOT=%4d\n",dev->sp,tot);
        dev = dev->next;
    }
    return(compStart);
} /** compNodes */

/****
/***** BUILDING DATA STRUCTURES *****/

tree_t *buildCompTree(FILE *in_file, comp_t *complist)
/* Function builds a binary search tree of compressed nodes. It is
similar to the more general buildTree, but old names of nodes are
replaced by new names as specified in the compression list.
PRE: in_file must be ready for reading and must be in the proper
format. Comlist is a linked list which contains the old and
new names of nodes to be compressed.
POST: Binary search tree of compressed nodes is constructed and pointer
to root node is returned by the function */
{
    tree_t *root;
    comp_t *curlist, *compNode;
    name_t n_name;
    int lineCount;
    char buffer[BUFSIZE];
    char *buf, *name, *type;

    curlist = complist;
    lineCount = 0;

    /* Extract device name and type from input file, check occurrence of
this node in the compression list, substitute new name for old name
as required and get the pointer for this root node. If node is
compressed, change its type to "COMPR".*/

    if (fgets(buffer,BUFSIZE,in_file) != NULL)
    {
        buf = strtok(buffer,"\n");
        name = strtok(buf,"");
        name = strtok(NULL,"");
        strcpy(n_name,name);
        type = strtok(NULL,"");
        lineCount++;
        if (curlist)
        {
            compNode = getCompNode(curlist,name);
            if (compNode)
            {
                strcpy(n_name,compNode->n_new);
                strcpy(type,"COMPR");
            }
        }
        root = instTree(n_name,type);
    }
}

```

```

/* Read the remainder of the file, construct the rest of the tree and
   substitute names as required. */

while (fgets(buffer,BUFSIZE,in_file) != NULL)
{
    buf = strtok(buffer,"\n");
    name = strtok(buf,"=");
    name = strtok(NULL," ");
    strcpy(n_name,name);
    type = strtok(NULL,":");
    lineCount++;
    if (curlist)
    {
        compNode = getCompNode(curlist,name);
        if (compNode)
        {
            strcpy(n_name,compNode->n_new);
            strcpy(type,"COMPR");
        }
    }
    addleaf(root,n_name,type);
}

return(root);
} /** buildCompTree **/

/*****/

void buildList(tree_t *root, FILE *in_file, comp_t *complist)
/* Constructs linked lists attached to each node of the compression tree.
   PRE: Binary search tree containing all node names must have been
        constructed, complist which was used to compress the nodes must
        be the same as the one used for the compression tree, and in_file
        must be ready for reading from the beginning and must be in the
        proper format.
   POST: Linked lists describing connectivity are attached to appropriate
        nodes of the search tree. */
{
    tree_t *srcNode;
    name_t n_dest;
    comp_t *curlist, *compNode;
    int lineCount, arcCount;
    char buffer[BUFSIZE];
    char *dest, *buf, *name, *type;

    curlist = complist;
    lineCount = 0;
    arcCount = 0;

    /* Extract first device name of each line and determine which devices
       are connected to it. Check if node was compressed. */

    while (fgets(buffer,BUFSIZE,in_file) != NULL)
    {
        lineCount++;
        buf = strtok(buffer,"\n");
        name = strtok(buf,"=");
        name = strtok(NULL," ");
        type = strtok(NULL,":");
        if (curlist)
        {
            compNode = getCompNode(curlist,name);
            if (compNode)
            {
                strcpy(name,compNode->n_new);
            }
        }

        /* If desired node is not present in tree, abort and stop. */

        srcNode = nameInTree(root,name);
        if (srcNode == NULL)
        {
            printf("\nERROR - unable to find node '%s'.\n",name);
            printf("*****> ABORTING <*****\n\n");
            exit(0);
        }
    }
}

```

```

/* Construct linked list attached to tree node. Replace node names
   as necessary to account for node compression. */
do
{
    dest = strtok(NULL, " ");
    if (dest != NULL)
    {
        strcpy(n_dest, dest);
        if (curlist)
        {
            compNode = getCompNode(curlist, n_dest);
            if (compNode)
            {
                strcpy(n_dest, compNode->n_new);
            }
        }
        if (strcmp(srcNode->name, n_dest) != 0)
            insInList(srcNode, n_dest, &arcCount);
    }
} while (dest != NULL);
} /* buildList */

/*****

comp_t *sortComplList(comp_t *complList)
/* Sorts the compression list in ascending order of complist->n_new field.
   PRE: Pointer to head of unsorted compression list is complist
   POST: List is sorted and pointer to its head is returned */
{
    comp_t *current, *last;
    int flag;

    if (complList == NULL) return(complList);
    if (complList->next == NULL) return(complList);

    /* Perform bubble sort on the list. */

    do {
        flag = 0;
        current = complList;

        /* Placing smallest element at start of list */

        if (strcmp(current->n_new, current->next->n_new) > 0)
        {
            last = current;
            complist = current->next;
            last->next = current->next->next;
            complist->next = last;
        }
        last = complist;
        current = complist->next;

        /* Swapping with smallest element in remainder of list */

        while (current->next != NULL)
        {
            if ((strcmp(current->n_new, current->next->n_new) > 0) ||
                ((strcmp(current->n_new, current->next->n_new) == 0) &&
                 (strcmp(current->n_old, current->next->n_old) > 0)))
            {
                last->next = current->next;
                current->next = current->next->next;
                last->next->next = current;
                flag = 1;
            }
            last = current;
            if (current->next != NULL) current = current->next;
        }
    } while (flag);

    return(complist);
} /* sortComplList */

```



```

/*****
void printComplst(comp_t *complst)
/* Function which prints the compression list to the screen.
PRE: Head of compression list is pointed to by complst.
POST: Compression list is printed to screen. */
{
    comp_t *current;
    int counter;

    current = complst;
    counter = 0;
    printf("\nlist of nodes being compressed (new_name old_name):\n\n");
    while (current != NULL)
    {
        printf("%s %s: ", current->n_new, current->n_old);
        counter++;
        if ((counter % 10) == 0) printf("\n");
        current = current->next;
    }
    printf("\n");
} /** printComplst **/

/*****
comp_t *addToComplst(comp_t *complst)
/* Adds new nodes read from terminal to compression list.
PRE: Head of compression list is pointed to by complst.
POST: New nodes are read from the terminal and added to compression
list until a blank line is entered. Pointer to updated compression
list is returned. */
{
    comp_t *newNode;
    char buffer[BUFSIZE];
    char *newItem, *old;

    printf("\nEnter 'new_name' ['old_name'] for nodes to be compressed.\n");
    printf("Terminate by entering blank line.\n");

    while(strlen(gets(buffer)) > 0)
    {
        newNode = MALLOC(comp_t);
        newNode->next = complst;
        newNode->br = 0;
        newNode->dc = 0;
        newNode->fp = 0;
        newNode->hu = 0;
        newNode->ln = 0;
        newNode->mb = 0;
        newNode->rt = 0;
        newNode->ws = 0;
        newNode->sp = 0;
        complst = newNode;
        newItem = strtok(buffer, " ");
        old = strtok(NULL, "\n");
        strcpy(newNode->n_new, newItem);
        if (old == NULL)
        {
            strcpy(newNode->n_old, newItem);
        }
        else strcpy(newNode->n_old, old);
    }
    complst = sortComplst(complst);
    return(complst);
} /** addToComplst **/

/*****
comp_t *removeFromComplst(comp_t *complst)
/* Removes nodes from compression list.
PRE: Head of compression list is pointed to by complst.
POST: New nodes are read from the terminal and removed from compression
list until a blank line is entered. Pointer to updated compression
list is returned. */
{
    comp_t *current, *last, *delNode;
    char buffer[BUFSIZE];
    char *newItem, *old;
    name_t n_new, n_old;

    printf("\nEnter 'new_name' ['old_name'] for nodes to be removed\n");
    printf("or 'all_nodes' to remove all.\n");
    printf("Terminate by entering blank line.\n");

```

```

while(strlen(gets(buffer)) > 0)
{
    current = complist;
    newItem = strtok(buffer, " ");
    strcpy(n_new, newItem);

    if (strcmp(n_new, "all_nodes") == 0)
    {
        while (current != NULL)
        {
            last = current;
            current = current->next;
            free(last);
        }
        complist = NULL;
        return(complist);
    }

    old = strtok(NULL, "\n");
    if (old == NULL)
    {
        strcpy(n_old, newItem);
    }
    else strcpy(n_old, old);

    while (current != NULL)
    {
        if ((strcmp(current->n_new, n_new) == 0) &&
            (strcmp(current->n_old, n_old) == 0))
        {
            if (current == complist)
            {
                complist = current->next;
            }
            else
            {
                last->next = current->next;
            }
            delNode = current;
        }
        last = current;
        current = current->next;
        free(delNode);
    }
    return(complist);
} /* removeFromComplist */

/*****/
/***** MAIN COMPRESSION FUNCTION *****/

int compressNodes()
/* Function prompts user to enter option for manipulating the compression
list, performs the appropriate action, and returns the option number.
If option = 0, compression was aborted and topology remains unchanged
PRE: Network topology must be present in in_file and must be in the
correct format. If compression information is read from file, it
must be present in the correct format in compIn_file.
POST: New topology has been written to file "topology.cmp". If present,
breakdown of compressed nodes will be contained in file
compOut_file. */
{
    int lineCount, option;
    FILE *in_file, *out_file, *compIn_file, *compOut_file,
    file_spec input_f, compIn_f, compOut_f;
    char buffer[BUFSIZE];
    tree_t *root, *compTree;
    comp_t *complist;

    printf("Input file containing topology data: \n");
    gets(input_f);

    in_file = fopen(input_f, "r");
    if (!in_file)
    {
        printf("\nERROR - cannot open file '%s'. \n", input_f);
        printf("====> ABORTING <====\n");
        return(0);
    }
    out_file = fopen("topology.cmp", "w");
    printf("Input file describing node compression: \n");
    gets(compIn_f);
    compIn_file = fopen(compIn_f, "r");

```

```

if (!compIn_file) {
    complist = NULL;
    printf("Compression information not read from file.\n\n");
}

printf("Output file for compression statistics:\n");
gets(compOut_f);
if (strlen(compOut_f) == 0)
{
    compOut_file = NULL;
    printf("Compression statistics will not be written to file.\n");
}
else
{
    compOut_file = fopen(compOut_f, "w");
}

/***** Call to various functions building data structures *****/

printf("\nReading original topology ...\n");
root = buildCompTree(in_file, NULL);
complist = makeComplis(compIn_file);
complist = sortComplis(complist);

do {
    printf("\nSelect compression option:\n");
    printf("    0 - ABORT without compressing nodes\n");
    printf("    1 - ADD nodes to compression list\n");
    printf("    2 - REMOVE nodes from compression list\n");
    printf("    3 - DISPLAY nodes in compression list\n");
    printf("    4 - LIST nodes in original data file\n");
    printf("    5 - EXIT and compress nodes\n");
    fgets(buffer, BUFSIZE, stdin);
    sscanf(buffer, "%d\n", &option);
    if (option == 0) return(0);
    else if (option == 5) break;
    else if (option == 1) complist = addToComplis(complist);
    else if (option == 4) listDevices(root);
    else
    {
        if (complist)
        {
            if (option == 2) complist = removeFromComplis(complist);
            else if (option == 3) printComplis(complist);
        }
        else
            printf("Compression list is empty.\n");
    }
} while (option);

/* Write compression statistics. If compOut_file does not exist,
function will not be executed. */

complist = compNodes(root, complist, compOut_file);

/* Build internal topology of compressed nodes and write to out_file. */

rewind(in_file);
printf("\nCompressing nodes ...\n");
compTree = buildCompTree(in_file, complist);
rewind(in_file);
buildList(compTree, in_file, complist);
lineCount = 0;
printf("Saving new topology ...\n");
printTree(compTree, &lineCount, out_file);

/* Delete normal and compressed topology, release memory, and close all
files. */

killList(compTree);
killTree(compTree);
killTree(root);

fclose(in_file);
fclose(out_file);
fclose(compOut_file);
printf("\n\n");
if (compOut_file)
{
    printf("Compression statistics written to file '%s'.\n", compOut_f);
}
return(option);
} /* compressNodes */

```

Appendix H

NP Program

```

#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <time.h>

#include <sysent.h>      /* getpid(), system(), exit(), open(), read(), close() */

#include "bbtypes.h"      /* blackboard data structure definitions */
#include "bbfunctions.h"  /* blackboard functions */
#include "pbase_types.h"  /* PegasusBase data structure defs */

#include "bb_pktype.h"    /* Paul Kalab's structure definitions */
#include "bb_pkfun.h"     /* Paul's general functions */
#include "bb_pathfun.h"   /* Paul's path-related functions */
#include "bb_compfun.h"   /* Paul's node compression functions */

#define KS_NAME      "NP"      /* Pegasus knowledge source name */
#define KS_TYPE      11       /* Pegasus KS type */
#define KS_SUBTYPE   1        /* Pegasus KS subtype */
#define PRINT_PATHS  10       /* option for posting list of paths only */
#define PRINT_ARCS   11       /* option for posting list of arcs only */
#define PRINT_NODES  12       /* option for posting list of nodes only */
#define PRINT_ALL    13       /* option for posting paths, arcs, nodes */

/***** GLOBAL VARIABLES *****/

struct request_record request;
struct reply_record reply;
struct process_record process_env;

/*****

void changeTopology(int option)
/* Signals NPATH that topology has changed but expects no response.
PRE: NPATH knowledge source must be running.
POST: NPATH constructed new topology as read from "topology.cmp" file. */
{
    long indx;

    request.function_no = option;
    indx = post_request(&process_env, &request);
} /** changeTopology **/

/*****

void pathInfoToBB(name_t origin, name_t dest, int numberOfPaths,
                  int numberOfArcs, int numberOfNodes, int option,
                  long *record_id)
/* Posts request for number of paths, arcs, and nodes to be sent back by
NPATH to blackboard.
PRE: NPATH must be running.
POST: Request posted to blackboard, and read and acted upon by NPATH. */
{
    request.function_no = option;
    strcpy(request.request_str0, origin);
    strcpy(request.request_str1, dest);
    request.request_para0 = numberOfPaths;
    request.request_para1 = numberOfArcs;
    request.request_para2 = numberOfNodes;
    *record_id = post_request(&process_env, &request);
} /** pathInfoToBB **/

```

```

/*****
int pathInfoFromBB(int option, struct process_record *process_e,
                  long request_index, FILE *out_file,
                  long *candPaths, long *pathSearched)
/* Receives replies to request via the blackboard, decodes replies, and
   prints the lists of paths, arcs, and nodes, or error message if an
   error condition occurred.
   PRI: NPATH must be running.
   POST: Lists of paths, arcs, and nodes printed to screen and out_file
        if file was opened. */
{
    long last_reply;
    int prPathsFlag, prArcsFlag, prNodesFlag, pathIDflag;

    prPathsFlag = 0;
    prArcsFlag = 0;
    prNodesFlag = 0;

    if ((option == PRINT_PATHS) || (option == PRINT_ALL)) prPathsFlag = 1;
    else if (option == PRINT_ARCS) prArcsFlag = 1;
    else if (option == PRINT_NODES) prNodesFlag = 1;
    else
    {
        printf("ERROR - invalid option %d.\n", option);
        return(0);
    }

    /* Print headers */
    if (prPathsFlag)
    {
        printf("\n\nShortest paths:\n\n");
        if (out_file) fprintf(out_file, "\n\nShortest paths:\n\n");
        pathIDflag = 1;
    }
    if (prArcsFlag)
    {
        printf("\n\nCritical arcs:\n\n");
        if (out_file) fprintf(out_file, "\n\nCritical arcs:\n\n");
    }
    if (prNodesFlag)
    {
        printf("\n\nCritical nodes:\n\n");
        if (out_file) fprintf(out_file, "\n\nCritical nodes:\n\n");
    }

    last_reply = reply_last_id(process_env);

    /* Wait for changes to reply area of blackboard. If reply was for this
       program, start decoding replies. */
    for (;;)
    {
        if (last_reply == reply_last_id(process_env))
        {
            wait_change(process_env);
        }
        else {
            last_reply++;
            get_reply(process_env, last_reply, &reply);
            if (reply.reply_no == request_index)
            {
                if (strcmp(reply.reply_str0, "ERROR") == 0)
                {
                    printf("Error condition detected by server program.\n");
                    return(0);
                }

                /* If only list of paths was requested and list is complete then exit.
                   otherwise get ready for printing list of arcs. */
            }
        }
    }
}

```

```

if (prPathsFlag) {
    if (strcmp(reply.reply_str0,"EndPaths") == 0)
    {
        *candPaths = reply.reply_para1;
        *pathSearched = reply.reply_para0;
        if (option == PRINT_PATHS) return(0);
        else
        {
            printf("\nCritical arcs:\n\n");
            if (out_file) fprintf(out_file, "\nCritical arcs:\n\n");
            prPathsFlag = 0;
            prArcsFlag = 1;
            pathIDFlag = 0;
        }
    }

    /* End of path delimiter read, print <K> and get ready for next output */

    if (strcmp(reply.reply_str0,"NextPath") == 0)
    {
        pathIDFlag = 1;
        printf("\n");
        if (out_file) fprintf(out_file, "\n");
    }
    else
    {
        if (pathIDFlag)
        {
            printf("%d [%d]: ", reply.reply_para0,
                reply.reply_para2);
            if (out_file)
                fprintf(out_file, "%d [%d]: ",
                    reply.reply_para0,
                    reply.reply_para2);
            pathIDFlag = 0;
        }
        if (strcmp(reply.reply_str0,"EndPaths") != 0)
        {
            printf("%s ", reply.reply_str0);
            if (out_file)
                fprintf(out_file, "%s ", reply.reply_str0);
        }
    }
}

/* Once paths have been printed and arcs were requested, and arc
   information is being posted, print it. */

if (prArcsFlag)
{
    if (strcmp(reply.reply_str0,"EndArcs") != 0)
    {
        if (strcmp(reply.reply_str0,"EndPaths") != 0)
        {
            printf("%d [%d]: %s-%s\n",
                reply.reply_para1,
                reply.reply_para0,
                reply.reply_str0,
                reply.reply_str1);
            if (out_file)
                fprintf(out_file, "%d [%d]: %s-%s\n",
                    reply.reply_para1,
                    reply.reply_para0,
                    reply.reply_str0,
                    reply.reply_str1);
        }
    }
    else
    {
        /* Path information is done, print final statistics and exit */

        *candPaths = reply.reply_para1;
        *pathSearched = reply.reply_para0;
        if (option == PRINT_ARCS) return(0);
        else
        {
            printf("\nCritical nodes: n n"),
            if (out_file) fprintf(out_file, " nCritical nodes n n"),
            prArcsFlag = 0;
            prNodesFlag = 1;
        }
    }
} /* ELSE path information is done */
} /* print arcs flag set */

```

```

/* Paths and arcs are done, process node list posted to blackboard. */
if (prNodesflag)
{
    if (strcmp(reply.reply_str0,"EndNodes") != 0)
    {
        if (strcmp(reply.reply_str0,"EndArcs") != 0)
        {
            printf("%d [%d]: %s (%s).n",
                reply.reply_para1,
                reply.reply_para0,
                reply.reply_str0,
                reply.reply_str1);
            if (out_file)
                fprintf(out_file, "%d [%d]: %s (%s)\n",
                    reply.reply_para1,
                    reply.reply_para0,
                    reply.reply_str0,
                    reply.reply_str1);
        }
    }
    else
    {
        *candPaths = reply.reply_para1;
        *pathSearched = reply.reply_para0;
        return(0);
    }
} /* IF print nodes flag set */
} /* IF reply number and request index match */
} /* ELSE reply was posted to blackboard */
} /* endless FOR loop waiting for replies */
/ ***** pathInfoFromBB *****/

/*****
***** MAIN PROGRAM *****/

main()
{
    long hr, min, sec, request_index, last_reply;
    int output, counter;
    long candPaths, pathSearched;
    name_t origin, dest;
    int numberOfPaths, numberOfArcs, numberOfNodes, repeat, option;
    char buffer[BUFSIZE];
    time_t start_time, finish_time;
    file_spec output_f;
    FILE *out_file;

    /* Get access to the blackboard. */
    if (get_access_to_bb(&process_env) == -1)
        exit(1);

    /* Register process and print header. */

    strcpy(process_env.KS_name,KS_NAME);
    process_env.KS_type = KS_TYPE;
    process_env.KS_subtype = KS_SUPTYPE;

    if (register_KS(&process_env) == -1)
        exit(1);

    last_reply = interested(&process_env, REPLY_ARRAY);

    printf("\n** KS TO REQUEST n CRITICAL PATHS, ARCS, AND NODES. **\n\n");
    printf("2.0 - 7/15/92 - Upgraded for compatibility with BB 3.1.\n");
    printf("2.1 - 8/05/92 - User interface modified.\n\n");
    printf("Select one of the following options:\n");
    printf("    0 - display output on screen only\n");
    printf("    1 - display on screen and save to file\n");

    /* Ask user if log file should be kept. If results written to log file,
       prompt for name and delete any file of that name. File will then be
       opened later in append mode. */

    fgets(buffer,BUFSIZE,stdin);
    sscanf(buffer,"%d\n", &output);
    if (output != 0) {
        printf("Enter output file name:\n");
        gets(output_f);
        if (strlen(output_f)==0) strcpy(output_f,"paths.out");
        out_file = fopen(output_f,"w");
    }
    else out_file = NULL;

```

```

counter = 0;
fclose(out_file);

/* Outer DO LOOP repeats until user wishes to quit.
   Inner DO LOOP repeats while the option is to compress nodes
   Print list of options and prompt user to select one */
do
{
    do
    {
        printf("\nEnter NPATH option:\n\n");
        printf("    0 - EXIT PROGRAM\n");
        printf("    10 - print paths only\n");
        printf("    11 - print arcs only\n");
        printf("    12 - print nodes only\n");
        printf("    13 - all of the above\n");
        printf("    15 - compress nodes in network\n");
        fgets(buffer,BUFSIZE,stdin);
        sscanf(buffer,"%d\n", &option);
        printf("\n");

        if (option == 15)
        {
            if (compressNodes()) changeTopology(option);
            if (out_file)
            {
                out_file = fopen(output_f,"a");
                fprintf(out_file,
                    "***** TOPOLOGY CHANGED *****\n");
                fclose(out_file);
            }
        }
        else if (option == 0)
        {
        }
        else if ((option > 9) && (option < 16) && (option != 14))
        {
        }
        else
        {
            printf("Invalid option: %d\n", option);
            option = 15;
        }
    } while (option == 15);

    /* Option requested some information from NPATH. Prompt for
       origin and destination nodes and the number of paths, arcs,
       and nodes which should be examined. Check values entered
       and set minimum limits if numbers negative. */

    if (option != 0)
    {
        numberOfArcs = 0;
        numberOfNodes = 0;
        printf("\nEnter origin node:\n");
        gets(origin);
        printf("Enter destination node:\n");
        gets(dest);
        printf("Number of paths, arcs, nodes:\n");
        fgets(buffer,BUFSIZE,stdin);
        sscanf(buffer,"%d %d %d\n", &numberOfPaths, &numberOfArcs,
            &numberOfNodes);
        if (numberOfPaths < 1) numberOfPaths = 1;
        if (numberOfArcs < 0) numberOfArcs = 0;
        if (numberOfNodes < 0) numberOfNodes = 0;

        if (out_file) out_file = fopen(output_f,"a");

        if(out_file)
        {
            fprintf(out_file,"***** > RUN %d *****\n",
                ++counter);
            fprintf(out_file, "%s to %s Paths = %d, Arcs = %d, Nodes = %d\n",
                origin, dest, numberOfPaths, numberOfArcs, numberOfNodes);
        }
    }
}

```



```

/* Initialize path counters and timer, post request, and wait for
reply. */

candPaths = 0;
pathSearched = 0;
start_time = time(NULL);
pathInfoToBB(origin, dest, numberOfPaths, numberOfArcs,
              numberOfNodes, option, &request_index);
pathInfoFromBB(option, &process_env, request_index, out_file,
               &candPaths, &pathSearched);
finish_time = time(NULL);

/* Separate function decoded and printed replies, print numbers of
paths searched and time elapsed. */

hr = (finish_time - start_time)/3600;
min = ((finish_time - start_time) % 3600) / 60;
sec = ((finish_time - start_time) % 3600) % 60;
printf("\nElapsed time: %02d:%02d:%02d\n", hr, min, sec);
printf("Candidate paths found: %4ld\n", candPaths);
printf("Total paths examined: %4ld\n", pathSearched);

if (out_file)
{
    fprintf(out_file, "\nElapsed time: %02d:%02d:%02d\n", hr, min, sec);
    fprintf(out_file, "Candidate paths found: %4ld\n", candPaths);
    fprintf(out_file, "Total paths examined: %4ld\n\n", pathSearched);
}
fclose(out_file);

/* If user wishes to look for more paths, repeat DO LOOP, otherwise
stop. */

printf("\nEnter 0 to repeat, any number to quit:\n");
fgets(buffer, BUFSIZE, stdin);
sscanf(buffer, "%d\n", &repeat);
}
} while ((repeat == 0) && (option != 0));

if (out_file)
{
    fclose(out_file);
    printf("\nResults printed to file %s'.\n", output_f);
}

/* Post special request for NPATH to stop execution, deregister, and stop. */

pathInfoToBB(origin, dest, -1, 0, 0, 10, &request_index);
uninterested(&process_env, REPLY_ARRAY);

check_out(&process_env);
printf("\n====> NORMAL TERMINATION <=====\n");
} /***** MAIN PROGRAM *****/

```

END

16-12-93

FIN