(2)

# NAVAL POSTGRADUATE SCHOOL

## Monterey, California






# THESIS

DESIGN AND IMPLEMENTATION OF A TOOLBOX OF MODULARIZED C PROGRAMS TO CONSTRUCT, ANALYZE AND TEST NETWORK OPTIMIZATION ALGORITHMS

by

Homero Fernandes Oliveira

September, 1991

Thesis Advisor: Gordon H. Bradley

# REPORT DOCUMENTATION PAGE

| | |
|---|---|
| 1a REPORT SECURITY CLASSIFICATION<br>UNCLASSIFIED | 1b RESTRICTIVE MARKINGS |
| 2a SECURITY CLASSIFICATION AUTHORITY Multiple Sources | 3 DISTRIBUTION/AVAILABILITY OF REPORT<br>Approved for public release; distribution is unlimited. |
| 2b DECLASSIFICATION/DOWNGRADING SCHEDULE | |
| 4 PERFORMING ORGANIZATION REPORT NUMBER(S) | 5 MONITORING ORGANIZATION REPORT NUMBER(S) |
| 6a NAME OF PERFORMING ORGANIZATION<br>Naval Postgraduate School<br>6b OFFICE SYMBOL (*If applicable*)<br>55 | 7a NAME OF MONITORING ORGANIZATION<br>Naval Postgraduate School |
| 6c ADDRESS (*City, State, and ZIP Code*)<br>Monterey, CA 93943-5000 | 7b ADDRESS (*City, State, and ZIP Code*)<br>Monterey, CA 93943-5000 |
| 8a NAME OF FUNDING/SPONSORING ORGANIZATION<br>8b OFFICE SYMBOL (*If applicable*) | 9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER |
| 8c ADDRESS (*City, State, and ZIP Code*) | 10 SOURCE OF FUNDING NUMBERS<br>Program Element No. / Project No. / Task No. / Work Unit Accession Number |

11 TITLE (*Include Security Classification*)
DESIGN AND IMPLEMENTATION OF A TOOLBOX OF MODULARIZED C PROGRAMS TO CONSTRUCT, ANALYZE AND TEST NETWORK OPTIMIZATION ALGORITHMS

12 PERSONAL AUTHOR(S) Oliveira, Homero F.

| 13a TYPE OF REPORT | 13b TIME COVERED | 14 DATE OF REPORT (*year, month, day*) | 15 PAGE COUNT |
|---|---|---|---|
| Master's Thesis | From To | September, 1991 | 81 |

16 SUPPLEMENTARY NOTATION
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

| 17 COSATI CODES | | | 18 SUBJECT TERMS (*continue on reverse if necessary and identify by block number*) |
|---|---|---|---|
| FIELD | GROUP | SUBGROUP | Networks, Graph Theory, Shortest Path, Coloring, Minimum Cost Spanning Tree, Random Graphs. |
| | | | |
| | | | |

19 ABSTRACT (*continue on reverse if necessary and identify by block number*)

A portable computer system to construct, test and analyze algorithms for large-scale network and graph problems was designed and partially implemented. The system provides an analyst with high-level easy-to-use constructs to specify network and graph algorithms. It produces efficient computer implementations of the algorithms, and constructs large scale unstructured and structured random network problems to test and analyze algorithms. The system is written in the computer language C and has been tested on personal computers and workstations. The present implementation includes algorithms for graph coloring, minimum spanning tree and shortest path problems and some tools to analyze the results.

| 20 DISTRIBUTION/AVAILABILITY OF ABSTRACT | 21 ABSTRACT SECURITY CLASSIFICATION | |
|---|---|---|
| ☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS REPORT ☐ DTIC USERS | Unclassified | |
| 22a NAME OF RESPONSIBLE INDIVIDUAL<br>Gordon H. Bradley | 22b TELEPHONE (*Include Area code*)<br>408-646-2359 | 22c OFFICE SYMBOL<br>OR/BZ |

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete
SECURITY CLASSIFICATION OF THIS PAGE

Approved for public release; distribution is unlimited.

Design and Implementation of a Toolbox of Modularized C Programs
to Construct, Analyze and Test Network Optimization Algorithms

by

Homero Fernandes Oliveira
Major, Brazilian Air Force
B.S., Brazilian Air Force Academy

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN OPERATIONS RESEARCH

from the

NAVAL POSTGRADUATE SCHOOL
September, 1991

Author: _____
Homero Fernandes Oliveira

Approved by: _____
Gordon H. Bradley, Thesis Advisor

_____
R. Kevin Wood, Second Reader

_____
Peter Purdue, Chairman
Department of Operations Research

## ABSTRACT

A portable computer system to construct, test and analyze algorithms for large-scale network and graph problems was designed and partially implemented. The system provides an analyst with high-level easy-to-use constructs to specify network and graph algorithms. It produces efficient computer implementations of tne algorithms, and constructs large scale unstructured and structured random network problems to test and analyze the algorithms. The system is written in the computer language C and has been tested on personal computers and workstations. The present implementation includes algorithms for graph coloring, minimum spanning tree and shortest path problems and some tools to analyze the results.




TABLE OF CONTENTS

# I. INTRODUCTION

## A. PURPOSE

A system that provides tools for an operations research analyst to build, test and evaluate network algorithms in a practical way with the minimum effort has been designed and partially implemented. The analyst works with a "high level" language that hides the complicated operations and data structures. The system can also be used for instructional purposes to help instructors introduce students to some concepts of network and graph theory in a language that allows a good understanding of the algorithms implemented.

## B. NETWORKS AND GRAPHS IN OPERATIONS RESEARCH

Discrete systems or organized collections of objects are frequently encountered in many areas of interest, such as computer science, mathematics, engineering, operations research, industrial management and others. Analysts often need to represent arbitrary relationships among those data objects. Graph theory and network flow theory provide simple techniques for constructing models of systems of this kind, and powerful methods for their analysis and optimization.

Graph theory has developed into a very active area of operations research. A major impetus for this growth has certainly been the wide applicability of graph theory to solve problems in the areas mentioned above. Many monographs have been written in

recent years covering such specialized areas as connectivity, colorability, extremal graphs, random graphs, Ramsey theory, and groups and surfaces.

Many kinds of "real-world" problems can be represented by graphs or networks and can be solved using existing algorithms. For example, to develop a railroad system or a communication network, the concept of connectivity is very useful, either to design the network with a minimum cost or to determine the smallest set of stations whose removed will disrupt the system. If a company that manufactures chemicals wants to partition its warehouse into compartments to store incompatible elements the concept of colorability provides the least number of compartments needed. Many other problems, such as timetabling, job assignment, job sequencing and others, can be modeled as network problems and solved by existing algorithms that, in general, are very efficient. In general, networks and graph theory are a very interesting, important and timely topics.

## C. MOTIVATION FOR THE SYSTEM

The increase in the capacity of computers is a factor that is leading analysts into continued study of networks and graph theory because it is now possible and affordable to implement and solve much larger problems. The solution of these problems is helping the military and many companies to improve their productivity.

Analysts usually have the problem that the algorithms are designed in a high-level language(or pseudo language), and then programs have to be implemented in low-level programming language.

There is a need for a system that helps them design algorithms in a high-level language that could be implemented immediately; that is, a high-level language algorithm that is executable. In addition, in order to get a correct assessment of the efficiency of a network algorithm it is necessary to construct high quality computer programs using effective data structures. It is also necessary to test the codes on a variety of large problems and to analyze the performance of the code in order to improve the algorithm and hence the code.

In operations research, network problems are usually very large, so it is very important to have the ability to generate and solve many large-scale test problems and it is also important to provide an efficient implementation of algorithms to ensure that large problems can be analyzed successfully.

A system designed to support the research in this area must be a compromise among all these goals mentioned above, in order to be useful and effective.

## D. CHOICE OF THE PROGRAMMING LANGUAGE C

Currently most of the algorithms that solve network problems are written in Fortran. Usually they are implemented using different compilers and they are tested in different machines, giving results that are difficult to compare. Those algorithms have their theoretical complexity analysis, but it is very difficult to make practical comparisons.

The choice of the C language was made considering these characteristics:

### 1. Portability

The C language is widely available on many machines, such as PCs and workstations, and most workstations have C as the primary language. With the recent adoption of the ANSI standard for C, programs written in C are very portable among computers. This allows any system written in C to run without modification on a wide variety of machines.

### 2. Dynamic Memory Allocation

Dynamic memory allocation is an important feature in the C language which allows the construction of very flexible and efficient data structures that do not depend on the size of the network. Using dynamic memory allocation, programs of any size up to the limits of the host machine can be run without any modification to the code itself.

### 3. Pointers to Variables

The use of pointers to variables allows very efficient memory management. This is a very important characteristic, especially when dealing with large network structures. Pointers also allow many direct operations on them, making the structure of the database implemented very flexible and access to it very rapid.

### 4. Macro Definitions Associated with C

The possibility of using macros to substitute for functions is very useful. It allows the software designer to build capabilities that instead of generating a call to a function, will expand the code in the location that it is called, avoiding the overhead incurred by calling a function.

Using macro definitions we will be able to build algorithms that hide the complex part of the structure of the language, without making the code inefficient.

### 5. Conclusion

A combination of all the features mentioned above allows a very efficient implementation of the system, and will support good software design, both to make the code a high-level language and to hide the data structures.

## E. BASIC DESIGN DECISIONS

In designing the system, there are some basic decisions that must be made to ensure that the system will be able to work with large-scale problems and will be portable.

### 1. Large-Scale Problems

One important decision is that the system must support networks of all sizes, either in number of nodes or in number of edges. The limitation in the size of the network is given only by the total storage available in the hardware used or by the size of memory given by the operating system.

### 2. Portability

The system is written using only standard structures and commands of **ANSI C**, that is the standard defined by the American National Standards Institutes(ANSI) for the C language. Graphical interfaces and any other specialized user interface is not part of the original system. This decision was made to guarantee that the system will not be dependent on any machine or operating system.

### 3. Hide the Data Structures

The data structures that store the network and related data are hidden from the user. The system provides functions that give the user access to this information. This decision was made to facilitate the work of the analyst who does not have a profound knowledge of the structure of the C language. It also enables the system to be tested with different data structures by just modifying the functions that recover the data from the database.

## F. DESIGN AND IMPLEMENTATION

The design of the system is complete. It is very flexible. It provides a capability to solve very large network problems and allows the analyst the ability to easily construct and test algorithms. The analyst can easily make improvements and modifications that could be necessary in order to get the information he desires about any experiments he wants to do.

The system is fully implemented regarding the network's data structure and also provides a random network generator that is described in Chapter III. The generator is capable of generating structured and unstructured random networks and also produces a database to permit the reproduction of any problem previously generated.

Using the system an analyst can construct and test algorithms for network and graph problems. While not strictly part of "the system", the design of the system includes developing algorithms for some important graph and network problems. This set of algorithms provides examples of the use of the system and

demonstrates that the system has sufficient capabilities to construct many algorithms. The system is partially implemented regarding these algorithms. The present version of the system has several algorithms for the coloring, shortest path and minimum cost spanning tree problems. The implementation can be expanded by providing more algorithms and some other types of data structures, like Fibonacci heaps, to make the system more powerful.

## G. OTHER WORK

### 1. UNIX-Based System - NETPAD

NETPAD[Ref. 1:p. 1] is a system developed by Belcore(Bell Communications Research) as an interactive color graphics program for studying networks. It consists of an adaptable user interface and an expandable toolkit of network algorithms. This interface can be used like an electronic pencil and notepad to interactively create, modify, save, recall and delete networks and their attribute values(cost, names, capacities, etc.).

This system is designed to support small problems with a graphical interface. The specific choice of a graphical interface makes it dependent on a Unix system with X-Windows. The interface can also be used to access a library of NETPAD algorithms for manipulating or analyzing networks via customizable pop-up menus.

### 2. IBM PC-based Programs

#### a. INDS

INDS (Interactive Network Design System) is a software package which solves various network design and optimization problems[Ref. 1:p. 11]. In particular it uses an

iterative graphical interface for applying heuristics to solve the traveling salesman problem and to find minimum cost 2-connected networks.

**b. CARDD**

CARDD (Computer-Aided Representative Graph Determiner and Drawer)[Ref. 1:p. 12] is an expert system that constructs a graph with properties defined by the user. The properties are specified by setting values for any subset of an available set of invariants, such as: number of nodes, number of edges, maximum degree, minimum degree, and others.

## H. TERMINOLOGY

The term **network** is frequently used to refer to such physical objects as a transportation network or communications network, whereas the term **graph** has a precise mathematical meaning. A **graph** G is defined as a set of **nodes** (nodes) together with a set of **directed** or **undirected edges** (**arcs**) where each **edge** consists of two (not necessarily distinct) nodes. Some authors define a directed graph as a **digraph**, a set of nodes and a collection of ordered pairs of distinct nodes(the edges). The term **network** is defined as a set of nodes and a collection of undirected edges where each edge or vertex has some attributes or values, such as costs, capacities, distances, etc. A directed network is a network with directed edges.

## I. OUTLINE OF THE THESIS

Chapter II describes a user's view of the system, how it works and what information is necessary to use the system. Chapter III describes the Random Network Generator. Chapter IV shows the software designer's view of the system. It has the information about the code and how it was implemented. Chapter V gives some examples generated by the system, with some comparisons and statistics about the algorithms already implemented. The Appendices are examples of the use of the system.

# II. USER'S VIEW OF THE SYSTEM

## A. INTRODUCTION

The system is designed to provide tools to the user. These tools allow the user to easily and quickly build and modify efficient functions and then to test them with randomly generated test problems.

This chapter describes how the user sees the system and gives a brief description of each function. Also included is an overview of several network algorithms that demonstrate the use of the system.

## B. FUNCTIONS PROVIDED BY THE SYSTEM

This section describes the functions provided by the system. These functions are the core of the system and were designed in such way to make it very easy to write programs and construct algorithms. They also make the programs very readable, hiding the most complicated operations that manipulate the network. These functions can not be modified by the user.

The functions are grouped into several modules. The modules are described and followed by a brief explanation of the purpose and operation of the functions.

### 1. Network Generation

There are four functions directly related to the generation of the network. The purposes of these functions are to

get the information about the network and to generate it. The random network generator is described in detail in Chapter III.

- **Get_info_from_file(input_file,&data)** - This is used to read the information about the network from a file. The first parameter, **input_file**, is a pointer to a string that has the name of the input file. The second parameter, **&data**, is a pointer to a structure that stores the data read from the file. It has the structure that is used by the function **create_graph**. The function does not return any value.

- **Get_info_from_screen(input_file,&data)** - This has the same purpose as the function described above. The only difference is that this one gets the information from the screen, interactively. The user defines the network by answering questions about it. The function also creates a file with all the data provided by the user. If the user wants to repeat the same design, the file can be used by the function **get_info_from_file**. This avoids the necessity of answering the same questions again. In this case, the variable **input_file** is the file that will receive the information about the network that can be used later on. The second parameter, **&data**, has the data that will be used by the function **create_graph**. The function does not return any value.

- **Create_graph(data)** - This uses the information provided by either one of the previous functions and generates the network. The parameter **data** contains all the data about the network. The function generates a network and stores it in an **adjacency list**. All the attributes of the network are stored inside its structures

and can be recovered by some functions that will be described in the next section. The function returns a pointer to the structure that stores the network.

- **Read_data(input_file)** - This function reads a file that contains a specific network. The file has all the data about the network, such as the number of nodes and all the edges with their attributes. It reads the data and creates the adjacency list. The parameter **input_file** is a pointer to the string that has the name of the input file. The function returns a pointer to the structure that contains the network.

2. **Network Manipulation**

These functions provide all the tools that are necessary to allow the navigation through the network. They are used in every algorithm and completely hide the data structures with which the network is represented.

- **first_node(graph)** - This function returns the first node of the graph.

- **next_node(node)** - This function returns the node following the node that is passed as a parameter.

- **exist_node(node)** - This function returns true if the node exists and false if node points to the nonexistent node following the last node of the network.

- **first_adj_node(node)** - This function returns the first adjacent node of the node passed as a parameter.

- **next_adj_node(node,adj_node)** - This function returns the adjacent node of **node** that is subsequent to **adj_node** in the list of adjacent nodes of **node**.

- **next_seq_adj_node(node)** - This function returns the subsequent adjacent node of **node**. It means, every time that the function is called, it returns the succeeding adjacent node of the one returned previously by the same function. This function is useful when the user knows in advance that the nodes will be taken sequentially. It is more efficient than the function **next_adj_node** described before.

- **first_edge(node)** - This function returns the first edge incident to **node**. The variable **edge** contains all the attributes of the edge including the adjacent node associated with that edge.

- **next_edge(edge)** - This function returns the edge following the one that is passed as a parameter.

- **exist_edge(edge)** - This function returns true if the **edge** exists, and false otherwise.

- **adj_node(edge)** - This function returns the adjacent node associated with **edge**.

- **total_of_nodes(graph)** - Returns the total number of nodes in the **graph**.

### 3. Node and Edge Manipulation

There are several functions to provide access to the attributes of the nodes and edges of the network.

For the nodes, these attributes are: number, color, distance to the start node, degree, parent, etc. Most of these

attributes are related to some specific algorithm and they are meaningless outside their respective algorithms. For that reason, some of these functions are described in the section describing their respective algorithms. The functions that recover attributes that are intrinsic to each node are:

- **number(node)** - It returns the number of the **node**.
- **degree(node)** - It returns the degree of the **node**.

The attributes that can be assigned to the edges are length and capacity. Sometimes, length is used as a cost; that means that we can not have an edge with two different values for length and cost. The functions related to these attributes are:

- **edge_length(edge)** - It returns the length of the **edge**.
- edge_cap(edge) - It returns the capacity of the edge.
- **dist(node,adj_node)** - It returns the length of the edge between **node** and **adj_node**.
- **cap(node,adj_node)** - It returns the capacity of the edge between **node** and **adj_node**.

### 4. Permutation Functions

The functions **first_node** and **next_node** described in Section 2 imply a sequential ordering of the nodes in the network. The functions described here, that are called **permutation functions**, can be used when the user does not want to access the network in its natural(or sequential) order.

To create a permutation vector that contains a random ordering of the nodes of the network, the system has two functions: **create_array_perm** and **construct_perm**. The function **get_item_perm**

allows access to the permutation. To access the network in the random order there are two functions: **first_node_perm** and **next_node_perm**. They are substitutes for the functions **first_node** and **next_node**. Below is a description of each function.

- **create_array_perm(vector,length)** - This function creates a permutation array to be used to reorder the nodes in an algorithm. The parameter **vector** is a pointer to an array and **length** is the length of the desired vector. The function does not return any value.

- **construct_perm(vector,type,seed)** - This function creates a new permutation ordering in the array that has been created by the previous function. The parameter **vector** is an array that has already been created and that will store the new permutation. The second parameter, **type**, defines the type of the permutation desired. Type can be **IDENTITY**, **REVERSE** and **RANDOM**. For the RANDOM type the user can define a **seed**, which is the third parameter. This is the seed that is used by the random number generator. The random number generator is described in Section 5. If the value of **seed** is non-positive, the current system seed will be used.

- **get_item_perm(vector,item)** - This function retrieves information about the permutation vector that has been created. It returns the value of item which can be **length**, **type** or **seed**. It is useful when the user wants to later recreate the same permutation.

- **first_node_perm(graph,permutation)** - This function returns the first node of the network defined by the permutation

vector. The first parameter is the graph in question and the second is the permutation vector to be used.

- **next_node_perm(node,permutation)** - This function returns the next node defined by the permutation vector. The parameters follow the same rules described earlier.

5. **Random Number Generator**

The structures randomly generated, such as networks and permutation vectors, need a uniform random number generator. The system has implemented a function that provides a random number, and two other functions to set and recover the seed used by the generator. These functions are:

- **rand_int(number)** - Returns a uniformly distributed integer random number between 0 and **number**-1.

- **set_seed(seed)** - Set the current **seed** of the system. This seed is the next seed that will be used by the random number generator. The function does not return any value.

- **get_seed()** - Returns the current system seed.

6. **Special Data Structure Functions**

The system provides four different types of structures to the user. They are: **heap, queue, deque** and **2queue**. These structures are designed to be very portable among algorithms, since the elements that they store point to the nodes of the network. The user can access any attribute of the node through an element of these structures.

a. Heap

The heap implemented in the system is a binary heap. The binary heap is a balanced tree where each element points to one of the nodes of the network with some attribute associated with it[Ref 2:p. 33]. The elements in the heap are stored so that the attribute of the node pointed to by each element is less than, or equal to, the attributes of the nodes pointed to by its descendants. Clearly, the root of the heap points to a node with the minimum attribute.

Next, we have a list of the functions to work with a heap and a brief description of them.

- initialize_heap(heap) - Initialize the heap.

- insert_heap(heap,node) - Insert a node in the heap.

- delete_min(heap) - Returns the node pointed by the element with minimum attribute of the heap and delete this element from the heap.

- siftdown(heap,node) - Reorders the heap by sifting down the elements after a modification of the attribute of node or after an insertion of a new node in the heap.

- siftup(heap,node) - Reorders the heap by sifting up the elements after a modification of the attribute of node or after an insertion of a new node in the heap.

- not_in_heap(heap,node) - Returns true if node is in the heap, false otherwise.

b. Queue

The queue provided by the system is a structure that observes the "first element in is the first element out" rule. These are the functions related to the queue structure:

- initialize_queue(queue) - Initialize the queue.

- enqueue(&queue,node) - Enqueue node in the queue.

- dequeue(queue) - Delete the first element(head) of the queue. It does not return any value.

- head(queue) - Returns the node that is the first element in the queue without removing it from the queue.

- not_empty(queue) - Returns true if the queue is not empty and false otherwise.

- not_inqueue(node) - Returns true if node is not in any queue and false otherwise.

- never_inqueue(node) - Returns true if node is not in any queue and has never been in any queue before. It returns false otherwise.

c. Deque

The structure deque is a double-ended queue in which additions and deletions are possible at either end[Ref 3:p. 10]. The structure implemented in the system is a variant of the deque, where additions are possible at both ends (head and tail), while deletiors are made only at the head. It looks like a stack and a queue connected in series(see Figure 1). A stack is a structure that observes the "first element in is the last element out" rule. To access the structure implemented in the system, the user can

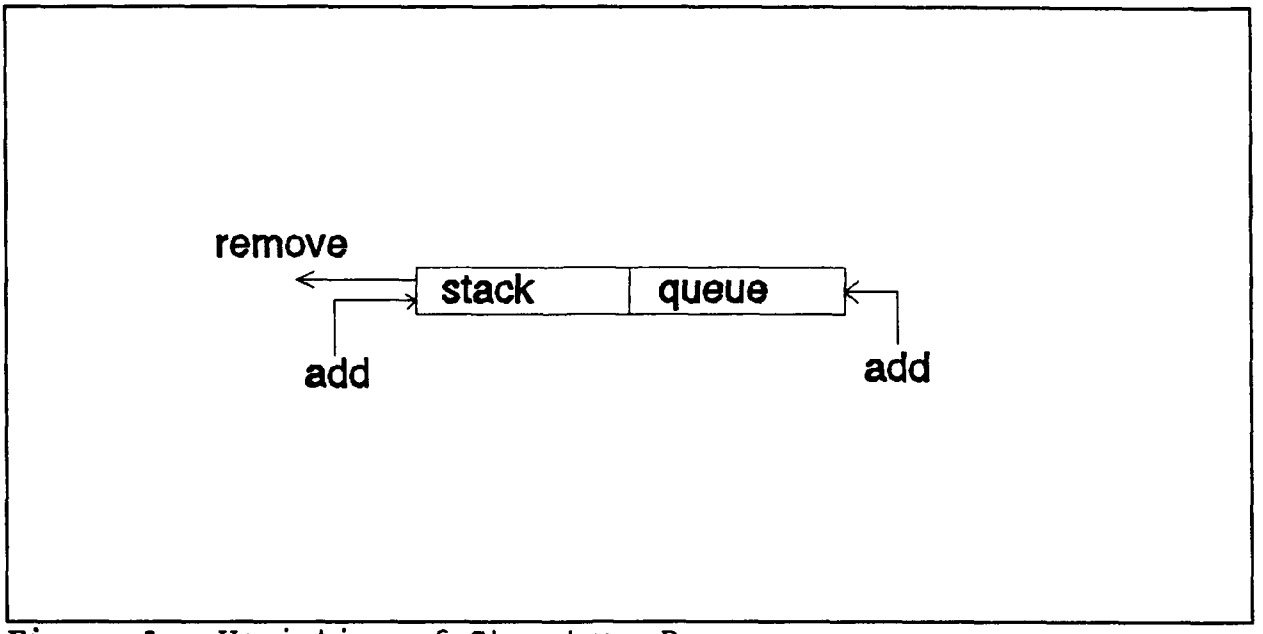


**Figure 1** - Variation of Structure Deque

utilize the same functions described for the queue, except to insert a node in the head of the structure. To do this the user must use the function:

- **deque(&queue,node)** - Inserts a **node** in the head of the **queue.**

**d. 2Queue**

The structure **2queue** is a list structure that combines two queues in series[Ref 3:p. 10]. It allows additions at the end of both queues, while deletions are made only at the head of the first one(see Figure 2). The function that makes the addition in the first queue is the following:

- **ddeque(&queue,node)** - Inserts a **node** in the tail of the queue number 1 of **queue**(see Figure 2).

All the structures described above, **queue, deque** and **2deque** are implemented as linked lists. The policy that defines how

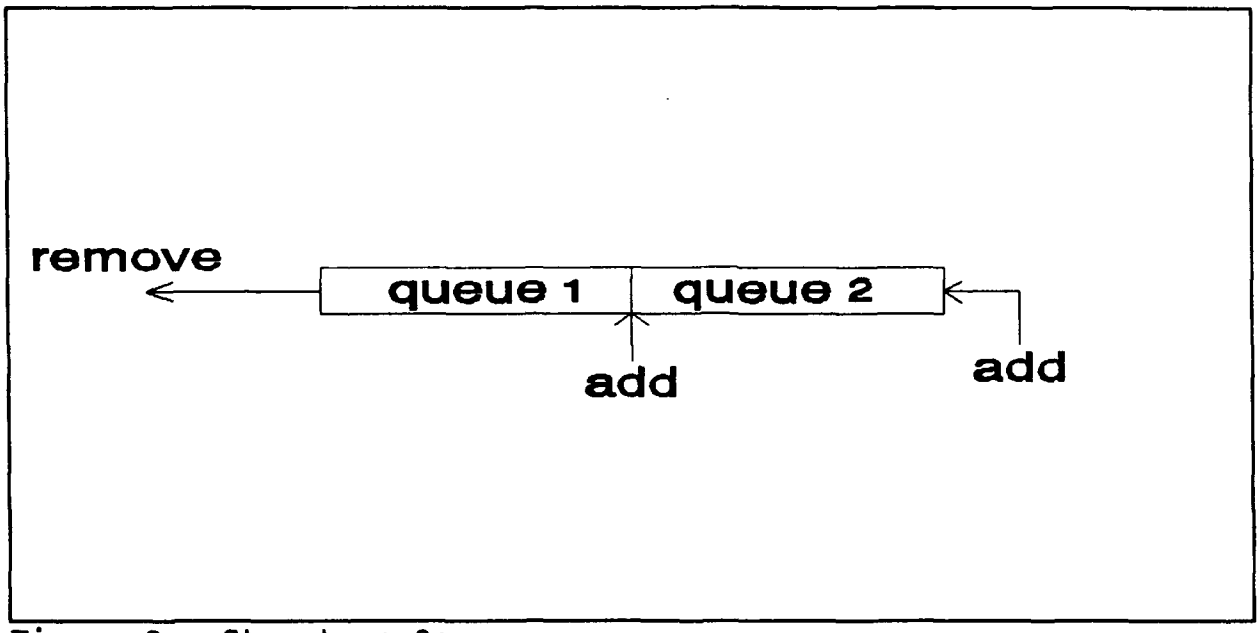

**Figure 2** - Structure 2queue

the nodes will be stored is described in each algorithm that uses these functions. They are described next.

## C. ALGORITHMS IMPLEMENTED USING THE SYSTEM

The main purpose of the system is to build algorithms to solve network problems. This section describes the algorithms that have already been implemented using the system described above. The problems solved by the algorithms are the following: coloring, shortest path and minimum cost spanning tree. There are also programs that performs some analysis of the networks. These programs are examples of how the system can be used. They can be modified and improved by the user.

### 1. Coloring Algorithms

A coloring of a graph is an assignment of a color to each node of the graph so that no two nodes connected by an edge have the same color. The coloring problem consists of coloring a graph

with as few colors as possible. This problem is one of a large class of problems that are called **NP-complete** problems. This means that there is no known algorithm to find the optimal solution for this problem in polynomial time. It is conjectured that there can be no polynomial-time algorithms for NP-complete problems. Here we developed several polynomial-time **heuristics** for coloring. **Heuristic** algorithms produce good solutions but are not guaranteed to construct optimal solutions.

We have three heuristic coloring algorithms implemented using the system. The colors are represented by positive integers(1,2,3...).

**a. Packing Algorithm**

The packing algorithm consists of coloring as many nodes as possible with the first color. Then, it colors as many of the uncolored nodes as possible with the second color, and so on. The packing algorithm is implemented using the functions that manipulate the network structure and these two additional functions:

- **color(node)** - Returns the color of the node.
- **set_color(node,color)** - Sets the **color** of the node.

The syntax to call the function that perform the packing algorithm is the following:

- **color01(graph);**

The function returns the number of colors used to performing the coloring. After the execution of **color01**, each node

has a color assigned to it. To access the colors allocated to each node the user just needs to use the function **color(node)**. The code of the function can be seen in Appendix A.

**b. Sequential Algorithm**

The sequential coloring heuristic consists of coloring every node with the minimum color possible. The nodes are taking in their natural sequence in the network.

The sequential algorithm is implemented using functions previously defined. The function that invokes the algorithm is the following:

**- color02(graph);**

The function returns the number of colors used to performing the coloring. The procedures to access the results are the same described above for the packing algorithm. The code of the function can be seen in Appendix B.

**c. Creating Permutations for the Sequential Algorithm**

One feature of the sequential coloring heuristic is that if we change the order in which the nodes are chosen to be colored, it may produce a different result. In order to look at different ordering of the same network, we produce a permutation vector that is a vector with all the nodes of the network but not in their natural order. To generate the permutation vector and to navigate through the network using that vector the user can utilize the functions described in Section B.4 of this chapter.

### d. Coloring Bipartite Networks

A bipartite graph is a graph whose set of nodes can be divided into two disjoint groups such that each edge has one end in each group[Ref. 4:p. 14]. If a network is bipartite, it can be colored using only two colors, one for each group of nodes.

An optimal algorithm to color a bipartite graph in polynomial time takes the nodes ordered by a **breadth-first search** that is described in Section 2, and assigns the minimum color possible to each one. To make the code more efficient, instead of running the breadth-first search and then running the coloring algorithm using a permutation vector, that is described in Section B of this chapter, the program does the following: while performing the breadth first search, the minimum color possible is assigned to each node, every time it comes out of the queue (see Section C.2 of this chapter).

To invoke the function that performs the coloring, the user invokes:

**- colorbip(graph,number(start_node));**

The function does not return any value. To access the colors of the nodes, use the same method used by the preceding functions.

## 2. Shortest Path Algorithms

The shortest path problem is a fundamental component in many real-life large-scale network models. This explains why, although the problem is quite simple, it is widely studied. Given an initial ncde, the problem is to find the shortest path to every

other node in the network. For this problem, we assume that each edge of the network has an attribute, length, associated with it.

Three algorithms are implemented in this thesis to solve this problem optimally. They derive from a single prototype procedure with the main difference between them being the data structures used to implement a set of candidate nodes. The structures used here, queue, deque and 2queue, are described in Section B.6 of this chapter.

The algorithms presented here are described by Gallo and Pallotino[Ref. 3]. They consist in the following: initially, set the attribute **dist_s**, that is the distance from the node to the start-node, as zero for the start-node and as infinity for every other node. Then, send the start-node to the list of candidate nodes (queue, deque or 2queue). After that, it takes the first candidate node from the list, called node **r**. For every adjacent node of **r**, if **dist_s** of the adjacent node is greater than **dist_s** of **r** plus the length of the edge between **r** and the adjacent node, then it updates **dist_s** of the adjacent node sends it to the list of candidate nodes. It keeps doing that until an optimal solution is found.

To implement the algorithms, several more functions were added to the system. They are:

- **dist_s(node)** - Returns the distance of **node** to the start-node.

- **set_dist_s(node,distance)** - Assigns the **distance** to the attribute dist_s of **node**.

- **parent(node)** - Returns the node that precedes **node** in the resulting shortest path tree.

- **set_parent(node,node_aux)** - Assigns **node_aux** as the nodes that precedes **node** in the tree.

- **set_not_in_queue(node)** - Sets a flag that means that the **node** is not in the queue.

To access the results obtained by the three polynomial-time algorithms that are described next, the user has to utilize the functions **dist_s(node)** and **parent(node)**.

a. **Single Queue Based Algorithm**

This algorithm uses a single queue to implement the set of candidate nodes. It corresponds to a breadth-first search strategy. The syntax of the function that invokes the algorithm is the following:

**short01(graph,number(start_node));**

The function does not return any value.

b. **Deque Based Algorithm**

This algorithm was implemented using a deque structure. The policy to insert a node in the deque is the following: the first time that the node is inserted in the deque, it is inserted in the tail. If a node, after being removed from the deque, again becomes a candidate for insertion, it is inserted in the head of the deque. The function that invokes the algorithm is:

**shodeque(graph,number(start_node));**

As before, this function does not return any value.

### c. 2Queue Based Algorithm

Here, a 2queue is used to represent the set of candidate nodes. The policy to insert a node in the 2queue is the following: the first time a node is inserted in the structure, it is inserted in the tail. After being removed, if the node becomes again a candidate for insertion, it is inserted after all nodes that are there for the second time and before all the nodes that are there for the first time. Looking at the structure as two combined queues, the procedure can be described as follows: the first time the node is a candidate for insertion, it goes into the second queue. After that, every time it becomes a candidate again, it goes into the first queue. The first queue has precedence over the second one.

The function that invokes this algorithm is:

```
sho2deqe(graph,number(start_node));
```

As before, it does not return any value.

## 3. Minimum Cost Spanning Tree Algorithms

A spanning tree of a network is a tree (connected acyclic graph) that connects all the nodes of the network. The cost of a spanning tree is the sum of the costs of the edges in the tree. Obviously, we assume that every edge has an attribute "cost" associated with it. The minimum cost spanning tree problem consists of finding a spanning tree of the network that has minimum cost.

There are two classical optimal, polynomial-time algorithms to find the minimum cost spanning tree. The first one is known as Prim's method and the second one as Kruskal's method. They

are both implemented using the system. After the execution of the algorithms, the edges that are in the minimum cost spanning tree are marked, and can be recognized with the following function:

- **edge_in_tree(edge)** - Returns true if the edge belongs to the minimum cost spanning tree and false otherwise.

a. **Prim's Algorithm**

The algorithm starts with one node as a initial tree. Then it takes the edge with minimum cost that is adjacent to the initial node and adds it to the tree. After that, it takes the edge with minimum cost that is adjacent to the tree (either node) and adds it to the tree. It keeps doing this (growing the tree) until all nodes have been connected.

This algorithm has two new attributes associated with each node: **light blue edge** and **key**. The **light blue edge** associated with a node is the edge that is candidate to enter the tree. That means, it is the edge with the minimum cost that connects the node to the tree. The **key** of the node is the cost of the light blue edge associated with it. To implement the algorithm some functions have been added to the system. They are:

- **initialize_graph_prim(graph)** - Initializes the graph specifically for the function prim.

- **set_node_in_tree(node)** - Sets a flag that mean that the **node** is in the tree.

- **key(node)** - Returns the value of the key of the **node**.

- **set_key(node,key)** - Assigns the **key** to **node**.

- **set_light_blue(node,adj_node)** - Assigns the edge between **node** and **adj_node** as a light blue edge relative to **node**. A **light blue** edge is an edge that is candidate to enter the tree.

The function that runs the algorithm is:

- **prim(graph);**

It does not return any value.

**b. Kruskal's Algorithm**

The algorithm starts with a graph consisting of all nodes and no edges. Every isolated vertex is considered a tree. Then it takes the edges in order of increasing cost. If an edge joins two different trees, then the two trees become a unique tree. If an edge connects two edges in the same tree, it is discarded, since it would create a cycle. The algorithm stops when there is only one tree left. The resulting tree is the minimum cost spanning tree. As before, to implement the algorithm, some functions have been added to the system. They are:

- **ordered_list(graph)** - Returns an ordered list of the edges of the **graph** ordered by increasing cost.

- **set_node_not_in_tree(node)** - Sets a flag in the **node** to indicate that it had not been connected to any tree, that means, it had not been connected to any other node. It does not return any value.

- **set_edge_not_in_tree(node,adj_node)** - Same as before, but in this case, the edge between **node** and **adj_node** is set not in a tree. It does not return any value.

- **initialize_list(list,length)** - Initialize the **list** of size **length** to be used by the algorithm.

- **first_arc(ord_list)** - Returns the first arc of **ord_list** which is the ordered list of edges in the network.

- **exist_arc(arc)** - Returns true if **arc** exists and false otherwise.

- **next_arc(arc)** - Returns the next **arc** of the ordered list of edges.

- **not_in_tree_arc(arc)** - Returns true if **arc** does not belong to any tree yet and false otherwise.

- **set_in_tree_arc(arc,list)** - Sets the nodes adjacent to **arc** in the tree determined by the parameter **list** that has the current tree number.

- **nodes_in_diff_tree(arc)** - Returns true if the nodes adjacent to **arc** are in different trees and false otherwise.

- **set_min_tree_arc(arc)** - This function does the following: if the endnodes of **arc** are in two different trees, then it assigns to all the nodes of the tree with the greater number, the number of the tree that has the smaller number. That means, the **arc** joins two different trees that now becomes only one, the one that has the smaller number.

The function that invokes the algorithm is:

- **kruskal(graph);**

It does not return any value.

# III. THE RANDOM NETWORK GENERATOR

## A. INTRODUCTION

A very important tool to analyze algorithms is a good random network generator. Since we have many different kinds of networks, we need a generator that creates random networks but with some desirable characteristics, or better, that gives to the user some control over the structure of the network that will be generated. In this chapter we will discuss some ideas about how to generate a network and will describe the implementation of the random network generator used by the system.

## B. SOME TYPES OF GENERATION

### 1. Unstructured Random Networks

One possible approach to implement a random network generator is to define the number of nodes and the number of edges and start generating edges between these nodes until we reach the desired number. In this case, the only test needed is to reject repeated edges and to be sure that the network has the number of edges desired.

This approach can be useful if there are few requirements about the structure of the network, because at the end of the process, little can be guaranteed the network. For example, there is no guarantee that the network will connected, or if the degrees of its nodes will follow a sample distribution, etc.

Usually we need a network with some characteristics, and one of the most important is connectivity. Most of the problems associated with networks assume that the networks are **connected**. A network is **connected** if there is a collection of edges joining every pair of nodes in the network[Ref. 4:p. 15].

In the following we will discuss some ways to generate a network that is guaranteed to be connected.

## 2. Generating Connected Graphs

One way to generate a connected network is to generate a **tree** first and then add some additional edges until we have the desired number. A **tree** is a network that is connected and **acyclic**.

### a. First Approach

To generate a tree we do the following: start generating edges in such way that each isolated edge is considered a tree. If an edge joins two different trees, we then consider the whole set of edges a unique tree. If the edge makes a cycle, we discard it. We finish the process when we have only one tree left and the total number of nodes is the number that was defined for the network. After that, we keep generating additional edges until we reach the desired number.

This method can guarantee that the network is connected, but we may have to generate many useless edges until the first tree is constructed. The implementation of this method is not very easy either.

### b. Second Approach

The second approach is similar to the first one. It first generates a tree and then adds some more edges until we have the desired network. It works in this way: we start with a disconnected network that has all nodes we want and no edges. From a vertex defined as a start vertex, we generate an edge to any other vertex. Then we generate a new edge from either one of these nodes to another one that has not been connected yet. We keep doing this until we have the all nodes connected. Then we add additional edges until we complete the network. The biggest problem in this approach is to keep track of the nodes that have already been connected to the others and the ones that still have to be connected.

### c. Third Approach

This model is very similar to the one described above, with the difference being that the edges are generated taking the nodes in ascending order.

We start with a network that has all the nodes and no edges. Then, an edge is created between vertex one and two. After that, a new edge is generated between vertex three and either vertex one or two. We keep doing the same process until all the nodes are connected. In other words, every new vertex is chosen in ascending order and it is connected to one of the previous ones. This eliminates the necessity of keeping track of the nodes that have already been connected. When the last vertex is connected, a tree has been created. After that, we keep generating edges until we reach the desired number.

This model seems to be the simplest and most efficient, so it was chosen to be implemented.

## C. ANALYSIS OF UNSTRUCTURED RANDOM GRAPHS

It is possible to do a theoretical analysis of the graph constructed by the third approach.

### 1. Original Problem

The first idea to create this generator came from the following problem proposed by Eric Wepsic[Ref. 5]: Suppose that we place n balls (numbered from 1 to n) into n urns (numbered from 1 to n) in the following way: the ith ball is placed in an urn chosen randomly from the first i urns. Let P(n,k) be the probability that an urn chosen at random contains exactly k balls. The problem is to find $\mathrm{Lim}_{n->inf}P(n,k)$ when n goes to infinity for a fixed number k.

Let's define $P_j(n,k)$ as the probability that urn j contains k balls when a total of n balls have been cast into the urns.

It is easy to see that

$$P_j(n,0) = \frac{j-1}{n}$$

when $0 < j \leq n$ and zero otherwise. Then if we select one of the n urns randomly, the equation for total probability is:

$$P(n,0) = \frac{1}{n}\sum_{j=0}^{n}\frac{j-1}{n} = \frac{1}{2}\frac{n-1}{n}$$

This is the probability of having no balls in the selected urn. The limit is then:

$$\lim_{n\to\infty} P(n,0) = \frac{1}{2}$$

For $P_j(n,1)$, the easiest way to proceed is by simple enumeration of the possibilities. Thus we achieve:

$$P_j(n,1) = \frac{1}{n}\left[1 + (j-1)\sum_{m=j}^{n-1}\frac{1}{m}\right]$$

and again by total probability:

$$P(n,1) = \frac{1}{n^2}\sum_{j=1}^{n}\left[1 + (j-1)\sum_{m=j}^{n-1}\frac{1}{m}\right]$$

Then if we exchange and solve summations, the above reduces to:

$$P(n,1) = \frac{1}{n} + \frac{1}{4n^2}(n-1)(n-2)$$

and

$$\lim_{n\to\infty} P(n,1) = \frac{1}{4}$$

Developing $P_j(n,2)$ in the same fashion, but with more algebraic manipulation:

$$P_j(n,2) = \frac{1}{n}\sum_{m=j}^{n-1}\left(\frac{1}{m}\right) + \frac{j-1}{n}\sum_{(k=j)}^{n-2}\sum_{m=k+1}^{n-1}\frac{1}{mk}$$

and similarly:

$$P(n,2) = \frac{n-1}{n^2} + \frac{(n-1)(n-2)}{8n^2}$$

and taking the limit,

$$Lim_{n\to\infty} P(n,2) = \frac{1}{8}$$

For greater values of k, proceeding in this method becomes exceedingly difficult, but we can see that for k=0,1,2 we have the values $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}$ which suggests the following general solution:

$$Lim_{n\to\infty} P(n,k) = \frac{1}{2^{k+1}}$$

This hypothesis is confirmed by simulation and can be proved by induction. The proof is not presented as it is beyond the scope of this paper.

## 2. From the Problem to the Generator

Using the problem stated above, we modeled a generator that provides some information about the network that will be generated.

Suppose that we place n-1 balls (numbered from 2 to n) into n urns (numbered from 1 to n-1). The ith ball is placed in an urn chosen randomly from the first i-1 urns. We assume that if ball i goes to urn j, we have an edge between i and j.

In other words, we are taking a new vertex in the network and are connecting it to one of the previous vertex already in the

network. In doing this, we can guarantee that the network is connected with n-1 nodes and n-2 edges, which means that it is acyclic. Since it is acyclic and connected, it is a tree.

One analysis that can be made about the network is its **degree sequence**. The degree of a vertex is the number of edges incident with it and the degree sequence of a network is a list of the degrees of the nodes in nonincreasing order.

Instead of listing all degrees of the nodes of a network, which is not practical for large networks, we can list the number of nodes with degree k(k=1,2,3,4,...). This list gives us an idea about the network we are generating, and we will refer to it as a degree distribution (Figures 3 and 4).

We can associate the degree of each vertex with the number of balls in each urn in the original problem. Since each ball(from 2 to n) is associated once to one urn, the degree of each vertex is at least one, even for vertex one, since vertex two will be associated with it. Every time that one ball is placed in urn j, it means that vertex j has one more edge. So, if urn j has k balls in it, this means that vertex j has degree k+1.

Using the result of the original problem stated above, for large networks, the expected number of urns containing exactly k balls is $n/(2^{(k+1)})$. Then we conclude that, for a network with n nodes, the number of nodes with degree d is $n/(2^d)$, (d=1,2,3,...,max_d). Using simulation, we found that a network with 1000 nodes is big enough to come close to these values. We can see in Figure 3 a graph that shows the proportion of nodes with the

corresponding degree. It was calculated using the average of 10,000 networks with 10,000 nodes each.

### 3. Repeating the Process Backward

Using the third approach, the tree that is first generated to make the graph connected is not random; on the average, the degrees of the first nodes connected are greater than the degrees of the last nodes. This can be partially compensated

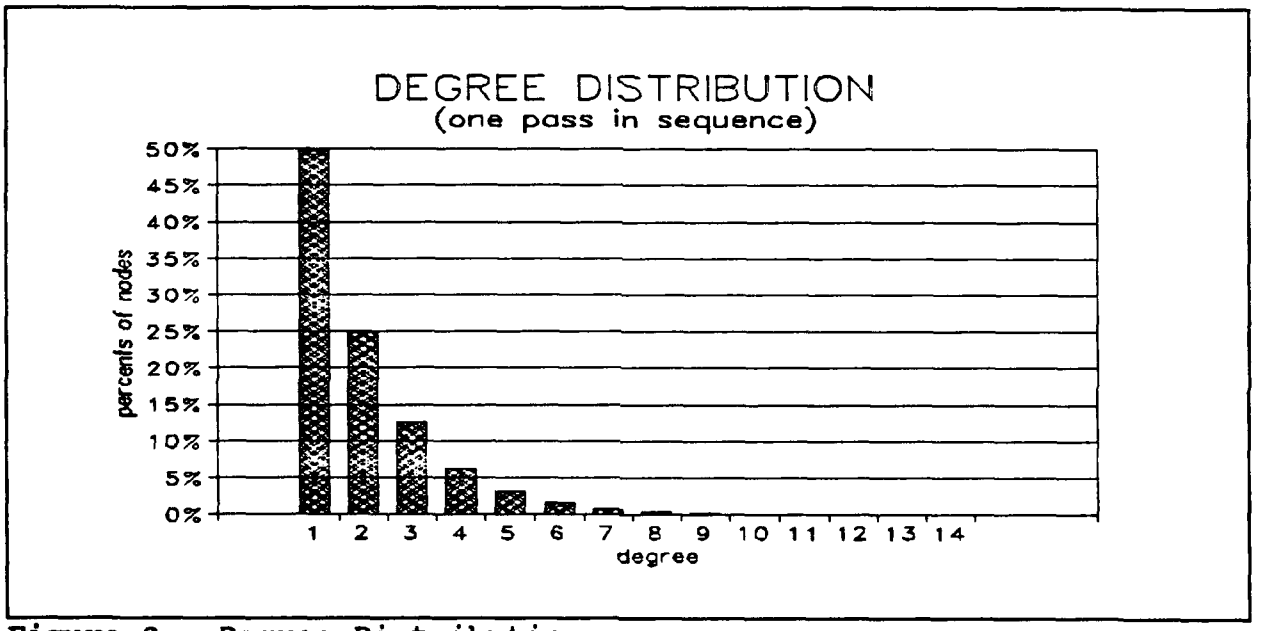


**Figure 3 - Degree Distribution**

for by repeating the process with the nodes in the reverse order. After doing the original process, we take the vertex i(i goes from n-1 to 1) and connect it to a vertex chosen randomly from the last n-i nodes. This procedure will make the network cyclic and every vertex will have degree at least 2. An alternative process is, instead of choosing a vertex sequentially from the last n-i nodes, we can choose it randomly from any nodes from 1 to i-1 and from i+1

to n. In Figure 4 we can see the average percentage of the nodes with corresponding degrees using the two processes described above. We have also in Figure 4, the sequential process with n/2 additional edges created completely at random.

In Figure 5 we can see the average of the degree of each vertex, calculated using 1,000 networks containing 1,000 nodes each. The Figure shows that the degree of the nodes that are close

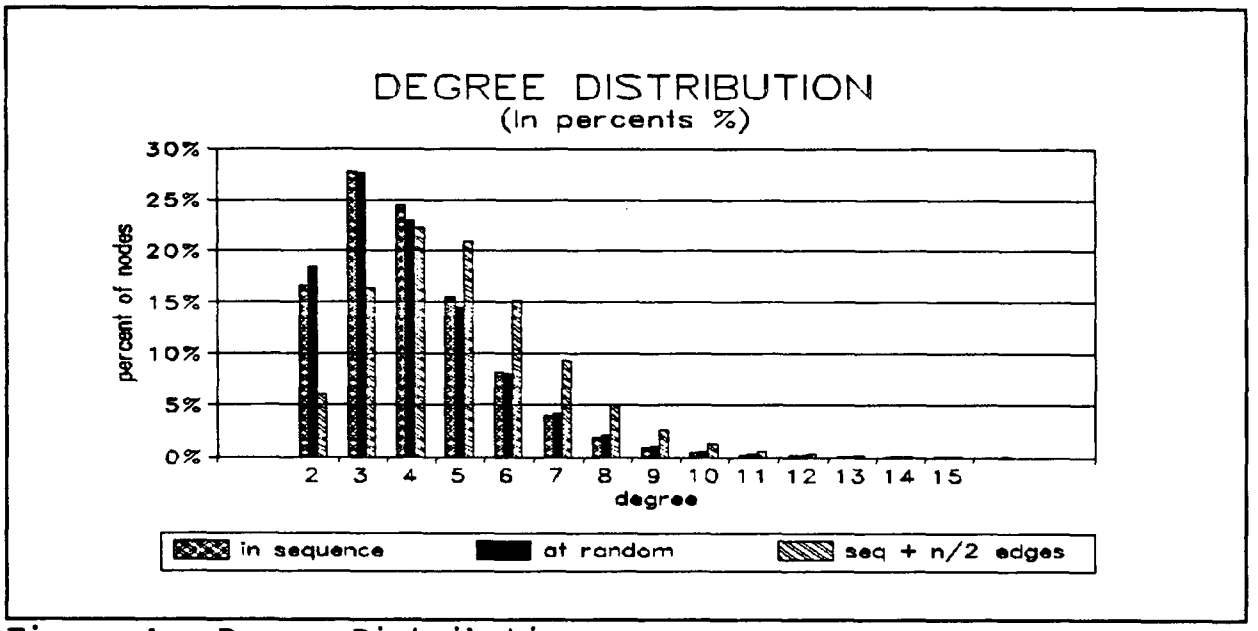


**Figure 4 - Degree Distribution**

to the first and to the last ones are bigger than the ones that are in the middle of the network. This is predictable, since these nodes that are in the extreme of the network have more chances to get new edges than the other ones.

### 4. Adding a Range to the Generator

One way to make the curve of Figure 5 flatter is, instead of for each vertex i randomly choosing a vertex from 1 to i-1 (or

from i+1 to n, if in the backward process), we can define a range r in which the vertex i can be linked. This means that the vertex i can be linked to a vertex chosen randomly from the previous r nodes, i.e., from vertex i-1 to i-r. In the backward process the idea is analogous.

This analysis shows that the random graphs constructed by the third method are quite close to random. As shown the Figure 5,

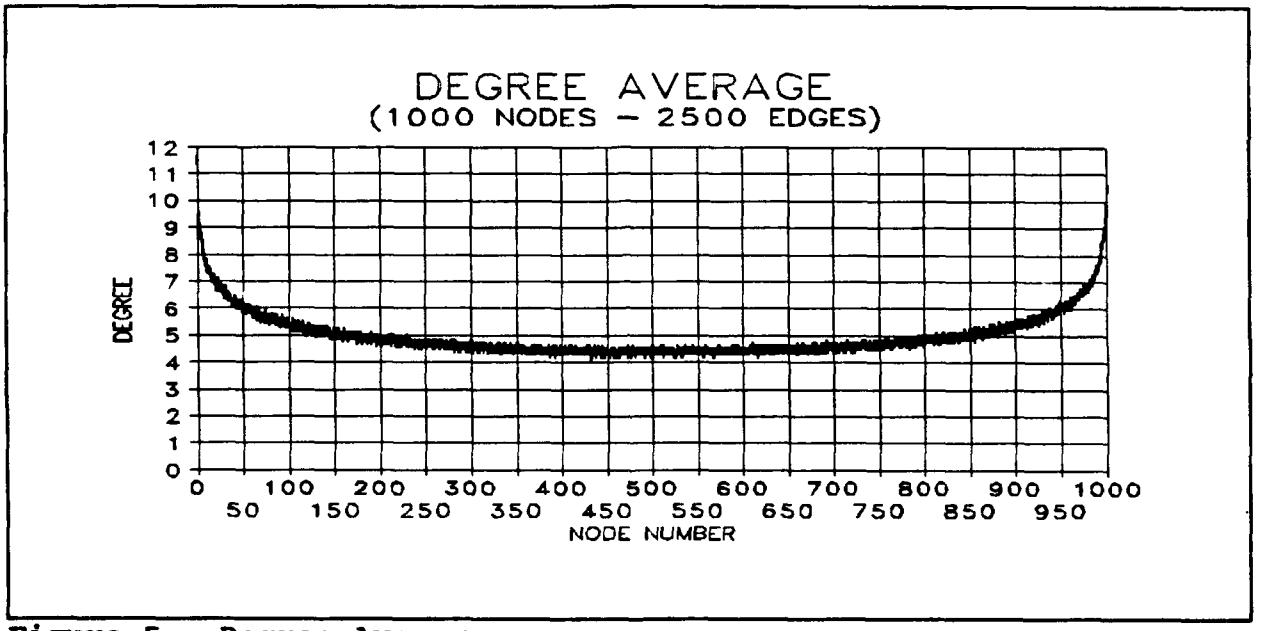


**Figure 5 - Degree Average**

on the average the first few and the last few nodes have higher degree than the others. For algorithms that are influenced by the order of the nodes, the system has capability to generate a random permutation of the nodes that can be used to limit the bias.

### 5. Defining Attributes for the Edges

As we saw before, networks are graphs where the edges have some attributes, so it is necessary to generate edges with these attributes.

The system will support two different types of attributes for the edges: length and capacity. When designing the network, the user must define the minimum and maximum length and the minimum and maximum capacity that the edges will have. The system will generate the value of the attributes randomly in the defined interval.

These values must be defined in the following way: the maximum length must be greater than or equal to the minimum length. If they are equal, all the edges will have the same length. If they were negative, the length will be negative as well. This is useful because some network problems have edges with negative attributes. The definitions for capacities follow the same rule described above.

## D. STRUCTURED NETWORKS

Networks found in the real world usually have some structure associated with them, so it is very important to provide a generator that can generate structured random networks.

To illustrate the concept of a structured network, we need to define component. A component of a graph is a maximal connected subgraph. If a network has more than one component, it is disconnected.

To define a structured network, we first define its components and then connect them with additional edges to complete model. Each component will be designed as a unstructured network.

### 1. Designing the Components

The design of the components follow the same process described in Section C for unstructured networks. After defining the total number of components we need to specify the characteristics associated with each one of them.

The characteristics that have to be defined for each component are the following: Total number of nodes, total number of edges, minimum length, maximum length, minimum capacity, maximum capacity, range and type of the network. All of these characteristics are described in Section C, except the last one: type of the network. "Type" will be a number which defines how the backward process of the generation of the component will be made. If the type is 1, then the backward process, will be made sequentially. If the type is 2, it will be made randomly. Backward process is defined in Section C.3 of this chapter.

### 2. Linking the Components

After the definition of the components, the user must define how they will be linked. The components are numbered by the system assuming the order that they are defined in the definition file. To define how they will be linked, the user defines a pair of components that will be linked and for each pair defines how many edges will exist between them and the attributes of these edges. The attributes are the same as defined before: minimum and maximum length and the minimum and maximum capacity.

The system does not verify that all components have been linked, so if the user wants the network to be connected he must be sure that from every component there is a path to any other in the network.

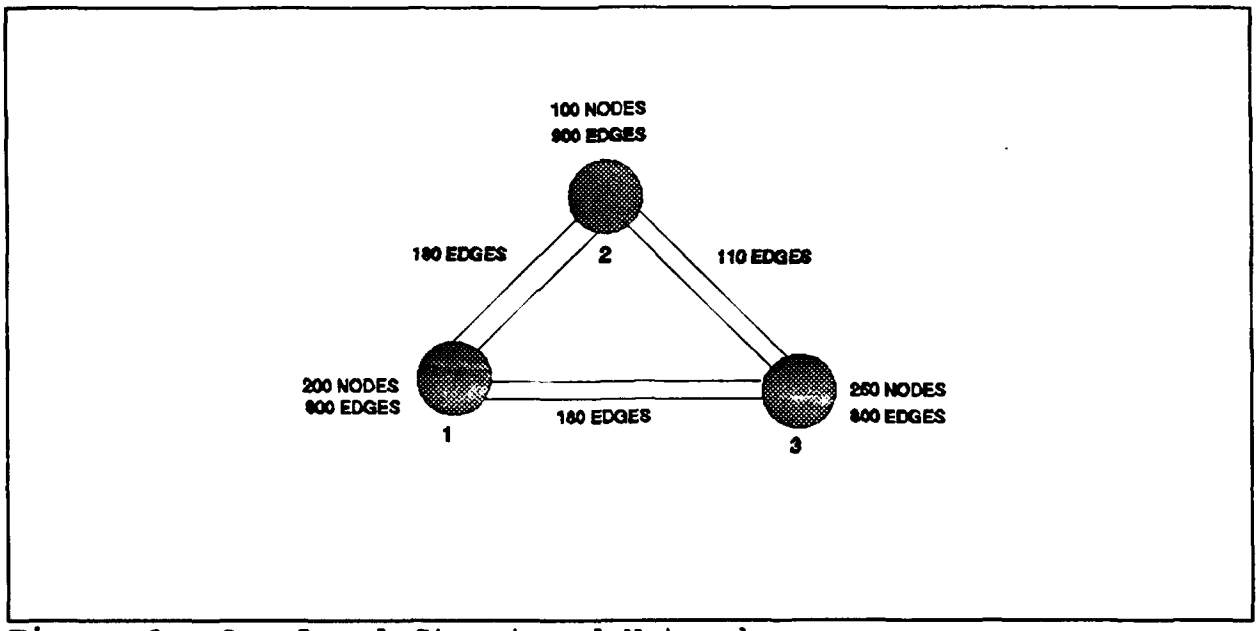


**Figure 6** - One-Level Structured Network

### 3. Numbering of the Nodes

The nodes in the network will be numbered sequentially from 1 to the total number of nodes designated for the network. Inside each component the nodes will be numbered in the following way: in the first component (component #1) they will be numbered sequentially from 1 to n, where n is the total number of nodes of the first component. The nodes of the subsequent components will also be numbered sequentially, and the numbers will start from the last number of the previous component plus 1. Figure 6 has an example of a structured network.

## E. THE MULTI-LEVEL STRUCTURED MODEL

The structure described above can be enhanced in order to provide a multi-level design of a network.

The idea is to provide a system that is able to generate networks with more than one level of structure. To understand the idea of multi-level design, consider the following: define a structured network as described in the preceding section, with components and edges connecting these components. Now, define some other networks like this one. Then, view these structured networks as components of a bigger network, and define more edges to link these components(or sub-networks), the result is a network that could be called a two-level structured network. An example of a two-level structured network can be seen in Figure 7. If the process is repeated recursively, the result is as many levels of structure as desired.

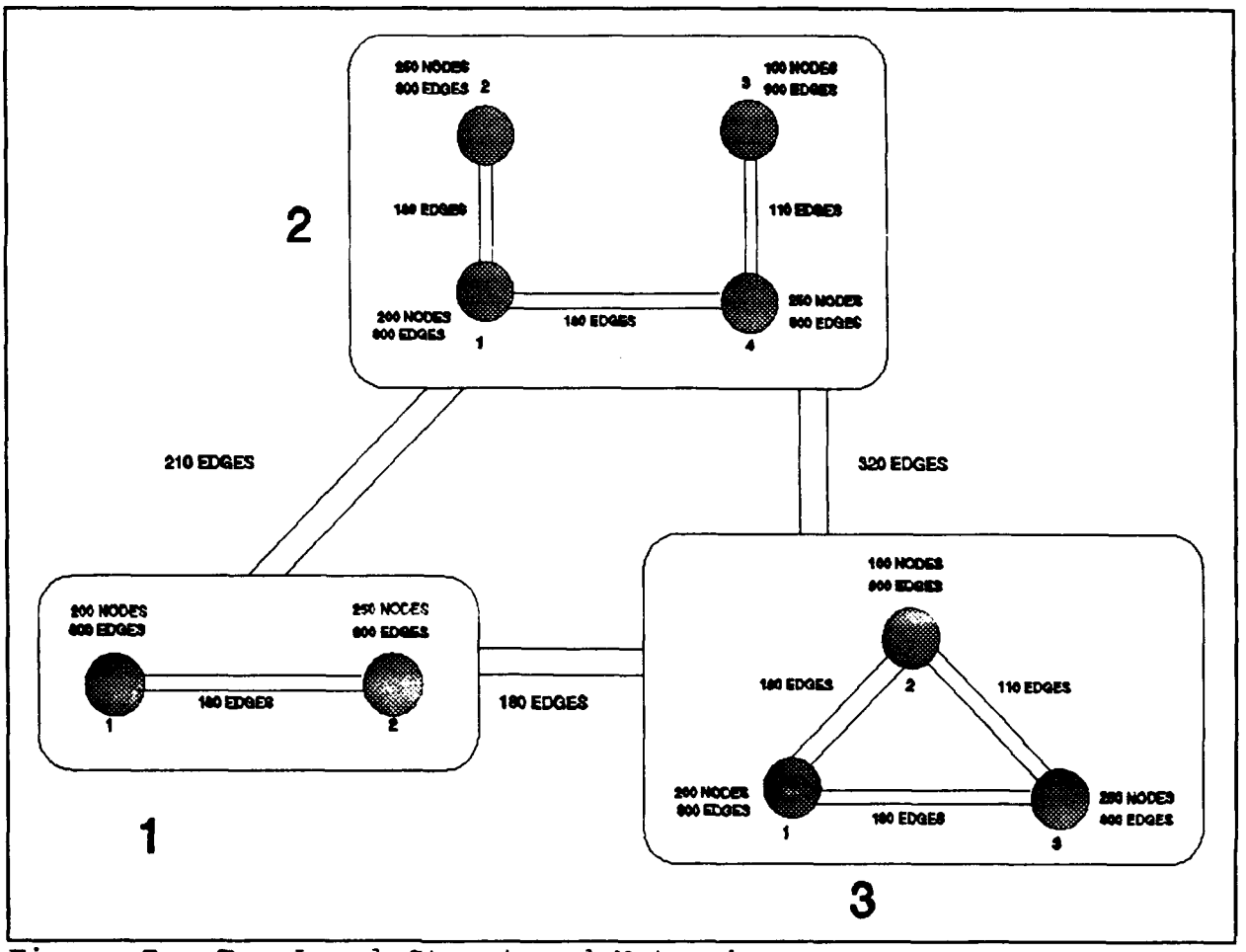


Figure 7 - Two-Level Structured Network

# IV. PROGRAMMER'S VIEW OF THE SYSTEM

## A. INTRODUCTION

This chapter describes some aspects of the system that would be important to anyone who would like to modify or extend the system. It includes a general idea of how some critical parts of the programs are implemented. It also describes some peculiarities inherent to some specific operating systems and some specific hardware that can influence the system.

It is assumed that anyone who modifies the system (referred from now on as the "programmer") is familiar with C.

A description of each function is presented in the code itself. The programmer must be familiar with the functions described in Chapter II, and with the data structures implemented. The programmer must also have some experience with macros substitutions and operations with pointers in C language.

## B. NETWORK DATA STRUCTURES

The data structure used in the system to store the network is a linked adjacency list. The structure is implemented using dynamic allocation for each structure(vector of nodes and a linked list for the edges).

The attributes associated with nodes and edges are stored inside its respective structures. These structures can be easily modified, just adding to or deleting internal fields from them. It allows the programmer to adapt the data base to different problems or situations that could emerge.

The implementation is based in the use of pointers to access the elements of each structure(nodes and edges).

The nodes are stored in a vector, in increasing ordering, and are numbered from 1 to n, where n is the number of nodes of the network. The vector has n+1 elements, and the last one has number 0, to indicate that there is no more nodes in the list.

The list of edges related to each node is stored in a linked list pointed to by the node. Each element of the list has the adjacent node and the attributes of the respective edge. The end of the list is indicated by a NULL pointer.

Figures 8 and 9 describe the data structure.

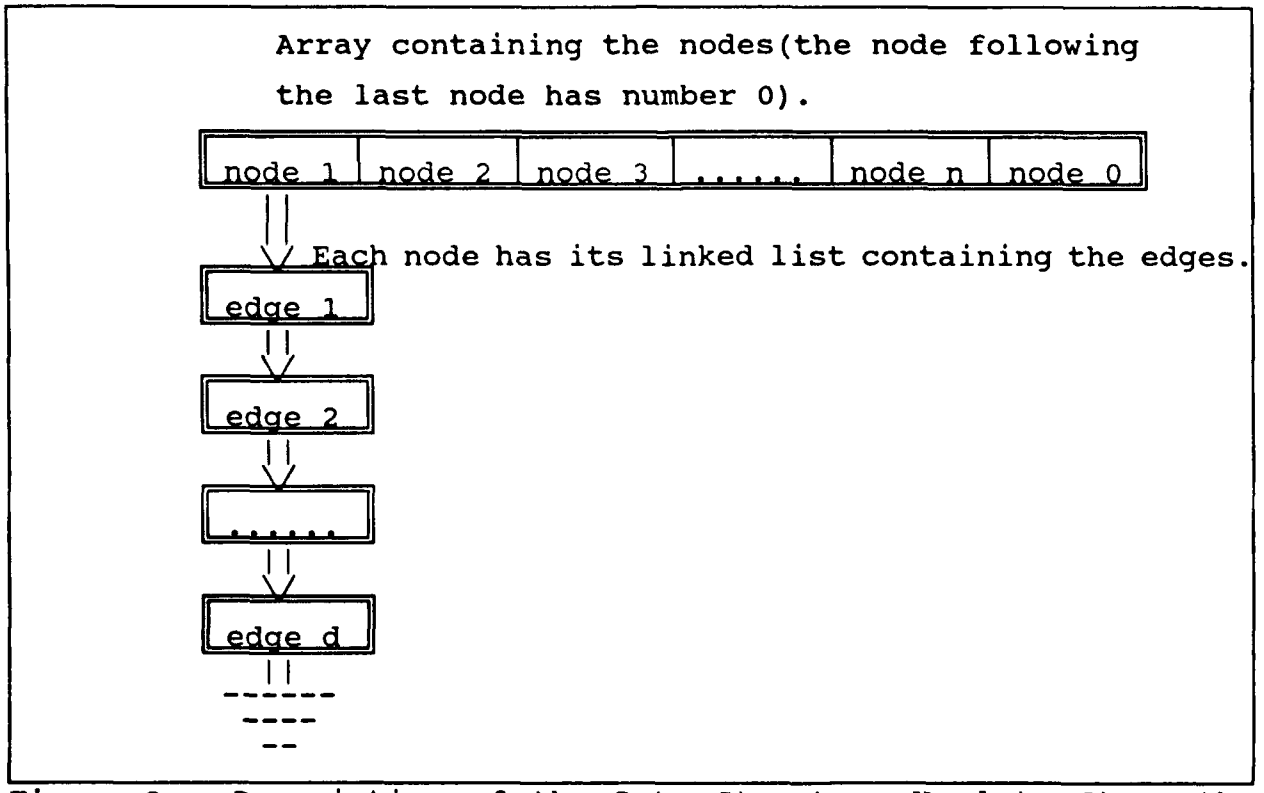


**Figure 8** - Description of the Data Structure Used to Store the Network.

STRUCTURE OF EACH ELEMENT (NODE) OF THE ARRAY.

| NUMBER : Number of the node |
|---|
| DEGREE : Degree of the node |
| These fields may have different uses for each algorithm. |
| ... ... ... ... ... ... ... ... |
| LIST OF EDGES: a pointer to the list of adjacent nodes. |

STRUCTURE OF EACH ELEMENT (EDGE) OF THE LINKED LIST.

| NODE_ADJ : Pointer to the adjacent node associated with this edge. |
|---|
| These fields may have different uses for each algorithm. |
| ... ... ... ... ... ... ... ... |
| NEXT_EDGE : Pointer to the next edge. |

**Figure 9** - Description of the Node and Edge Structure Used in the Data Structure.

## C. CONSIDERATIONS ABOUT FUNCTIONS AND MACROS

To provide a high-level language to the user and to hide the data structure, the system is implemented making use of functions and macros substitutions. This section discusses some of the most important functions and macros and some intrinsic characteristics of them.

### 1. Efficiency

The use of a function results in some overhead cost to the system every time that it is called. For this reason, the use of functions is avoided and a macro is used instead. But, for some complicated operations and for operations that are not executed very often, a function is used. Functions are also preferred when it needs to return a value. For complicated operations, a macro can not return a value, and a parameter is required to do that.

### 2. Visibility

An disadvantage of using macros instead of functions is that it is much more difficult to debug a program with complicated operations performed by macros. The operations that are made inside the macros are not so visible by the debugger as they are when a function is used. For this reason, the programmer should first use functions, and after making sure that the program is working as desired, then transform the function into a macro, if necessary.

### 3. Parameters

The programmer must pay attention in the parameters used by macros and functions. Functions in C only have parameters passed by value, not by reference. So, if the program needs to return a

value in a parameter, the parameter passed must be the address of the desired variable, and the same parameter must be declared as a pointer inside the function. This problem does not happen in macros. Since they are text substitutions, the code of the macro is placed in the location where it is called. The variables passed as parameters have their values modified directly.

### 4. Portability

Functions in C can be used in any place to substitute a variable or a command. The same is not true for macros. If a macro has more than one command, and has a semicolon between the commands, it can not be placed inside some structures of the language, such as a **for loop**, or a **printf** command. Macros that return values like "true or false" must be implemented without semicolons.

## D. RESTRICTIVE CHARACTERISTICS

This subsection describes some unique characteristics inherent to some functions or to the system itself. The programmer must be aware of these important characteristics in order to avoid some problems that could occur during any modification of the system.

### 1. Random Number Generator

The random number generator implemented in the system is a generator described by Robert Sedgewick[Ref. 6]. It is designed to work on 32-bit machines or in machines that stores integers in a four-byte word. The function that returns the number generated is described in Chapter II, Section B.5, and is called **rand_int(n)**. It returns an integer between 0 and n-1. This generator does not run

in PCs with the DOS operating system. To run the system on these machines or on any other that is not compatible with this random number generator, the function rand_int(n) must be modified. One example is to modify the function to use the random number generator that is provided by Turbo C++.

This is the only part of the system that needs to be modified from machine to machine. The programmer working in the system must be sure that the random number generator implemented is compatible with the hardware that is being used.

### 2. Random Network Generator

The random network generator uses dynamic allocation to creates the data structure that stores the network. Depending on the size of the network and on the size of the structure utilized for the nodes, the vector containing the nodes can be very big. Some systems will not accept memory allocation bigger than 64 kbytes, and will ask for special declaration of the variables or some special compilers options. This happens when using Turbo C++ on DOS machines. The programmer must check the result of every attempt to allocate memory, and be sure that it succeeds.

## E. SPECIAL DATA STRUCTURES

The special data structures implemented in the system are described in Chapter II, Section 6. This section gives some important detail of their implementation.

### 1. Queue Structure

The structure used to implement the queue and its variations(deque and 2queue) is a linked list. Each element of the

list has a pointer that points to its respective node and a pointer to the next element of the list. This design was adopted to allow the use of the structure for a wide variety of problems, since the attributes associated with the nodes are in the node itself.

2. Heap Structure

The structure used to implement the heap is a doubly linked list; that means, each element of the heap has a pointer to its descendent(two in this case), and a pointer to its predecessor. The other field is a pointer to its respective node. The only peculiarity inherent to this structure is that each node of the network has a pointer to the element in the heap that points to it. It is useful when performing operations in the heap, because there is no need to look for the node's respective element in the heap. This makes the code very efficient.

# V. USING THE SYSTEM

## A. INTRODUCTION

This chapter has some examples of the utilization of the programs implemented using the system. It has some analysis about the algorithms that have already been implemented using data collected from randomly generated networks. It shows a comparison among different versions of algorithms that solve specific problems. The algorithms used here are described in Chapter II, Section C.

This chapter includes some discussion about the use of the system by the students of a seminar, OA-4203, that has been offered in the summer of 1991 at the Naval Postgraduate School.

## B. COLORING ALGORITHMS

Two versions of the coloring algorithm had been implemented. They are the "sequential" and "packing" algorithms. To make an analysis of the performance of these algorithms, we counted how many searches each algorithm makes to color the nodes of the network. We consider it a search every time that an algorithm checks a color of a node or checks the vector of colors to find the minimum color possible to be assigned to a node (sequential algorithm). The result of the comparison is shown in Figure 10. The figure shows the numbers obtained from 30 networks randomly generated with 2,000 nodes and 5,000 edges each. Figure 11 shows

the same comparison, but now generating networks with 550 nodes and 1,770 edges. The results show that the sequential algorithm is more efficient than the packing algorithm in both cases.

Another comparison that was made, is between the sequential algorithm and the algorithm used to color bipartite graphs (colorbip algorithm). The idea is to see which algorithm uses few colors. Regarding the number of colors used to color the networks, Figure 12 shows that both algorithm are equivalent. The figure shows the data obtained from 140 networks randomly generated (70 with 500 nodes and 1,500 edges and 70 with 500 nodes and 4,000 edges). The numbers in the horizontal axis are the number of colors obtained by the colorbip algorithm minus the number of colors obtained by the sequential algorithm.

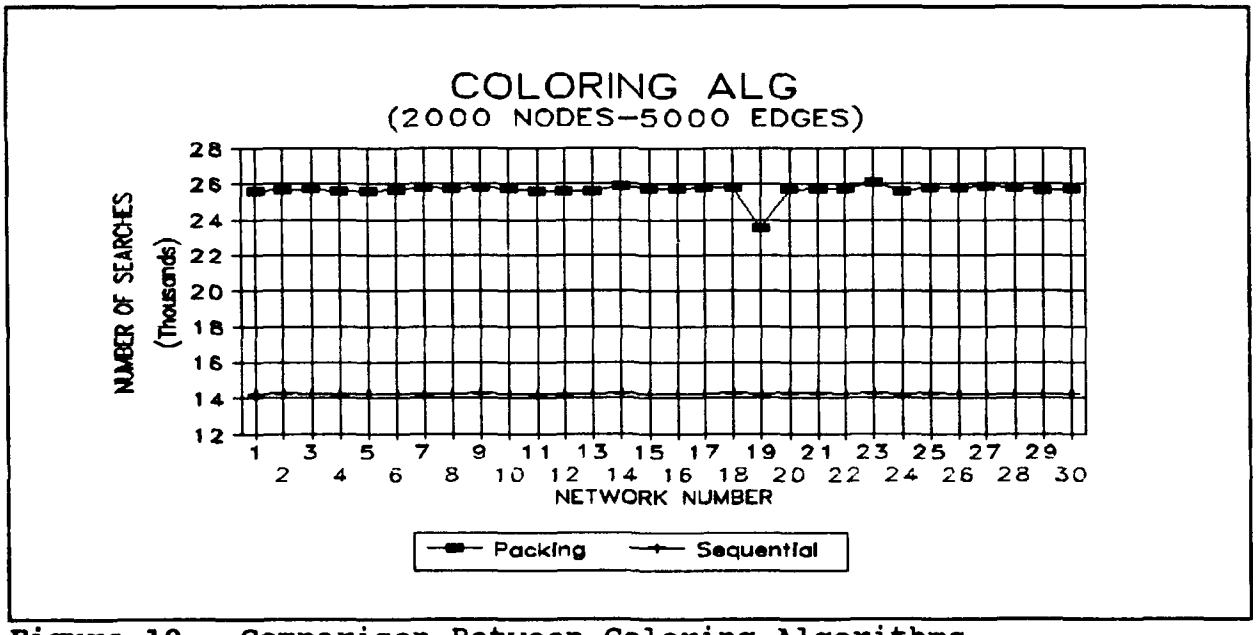


**Figure 10** - Comparison Between Coloring Algorithms

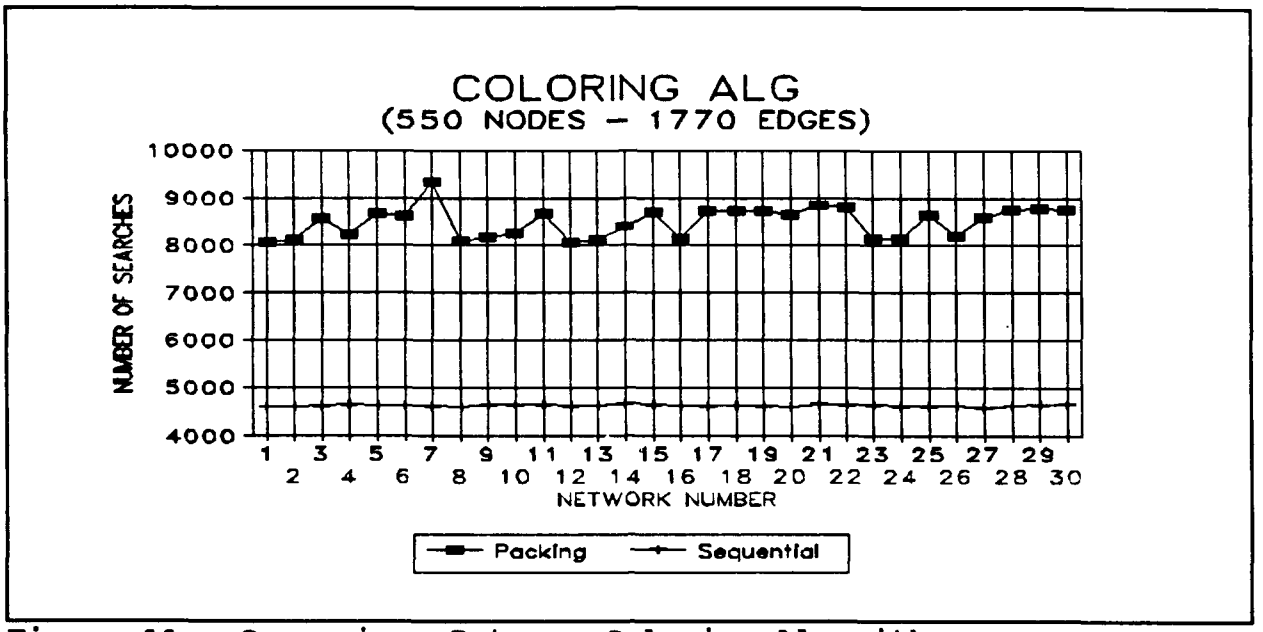


**Figure 11 - Comparison Between Coloring Algorithm**

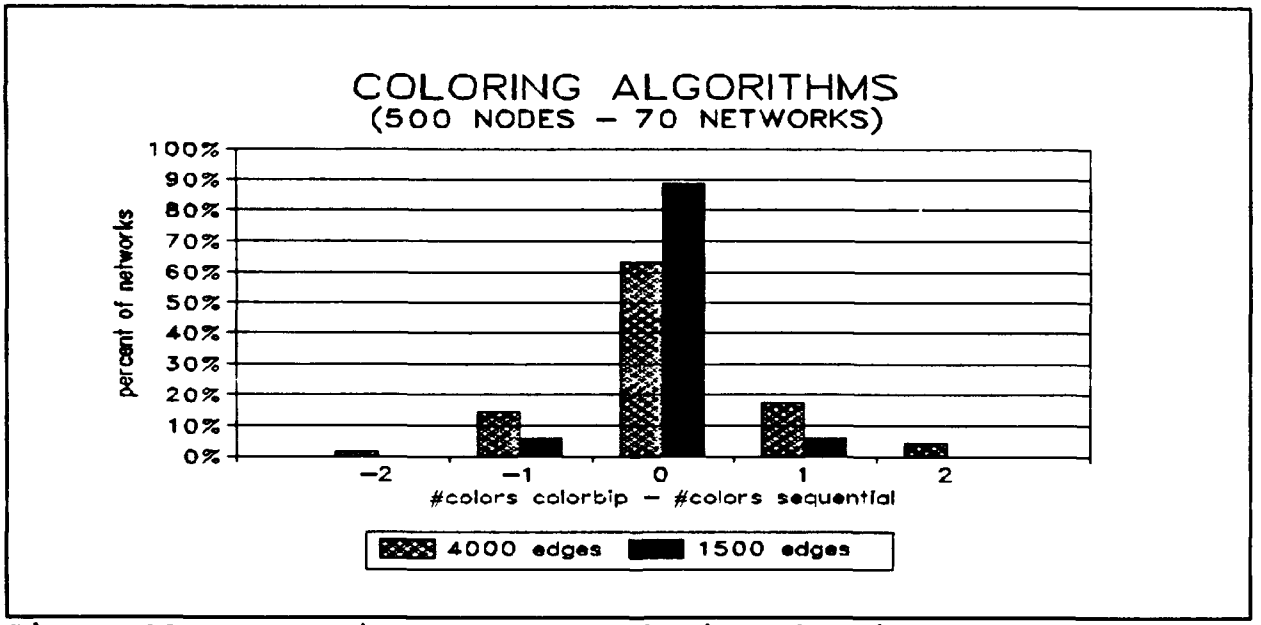


**Figure 12 - Comparison Between Coloring Algorithms**

## C. SHORTEST PATH ALGORITHMS

There are three versions of algorithms to solve the one-to-all shortest path problem. The algorithms are basically the same, but implemented with different data structures. To make an analysis of the efficiency of each one, we counted the number of nodes sent to the queue (or to the structure adopted for the algorithm). Figure 13 has the data obtained from 30 networks randomly generated with 2,000 nodes and 5,000 edges each, and the maximum length of the edges is 50. Figure 14 has the results obtained generating 30 networks with 550 nodes and 1,770 edges each. We can see that the algorithm that uses the Deque structure is more efficient than the other two and the algorithm that uses the 2queue structure is more efficient than the algorithm that uses the queue structure.

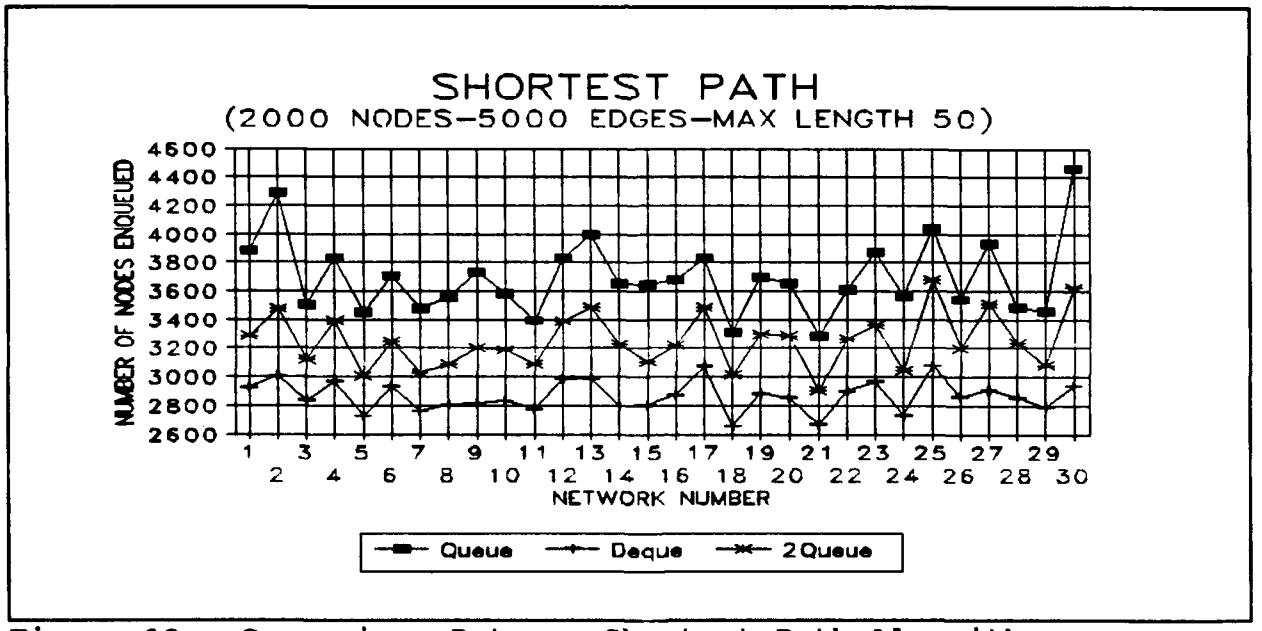


**Figure 13** - Comparison Between Shortest Path Algorithm

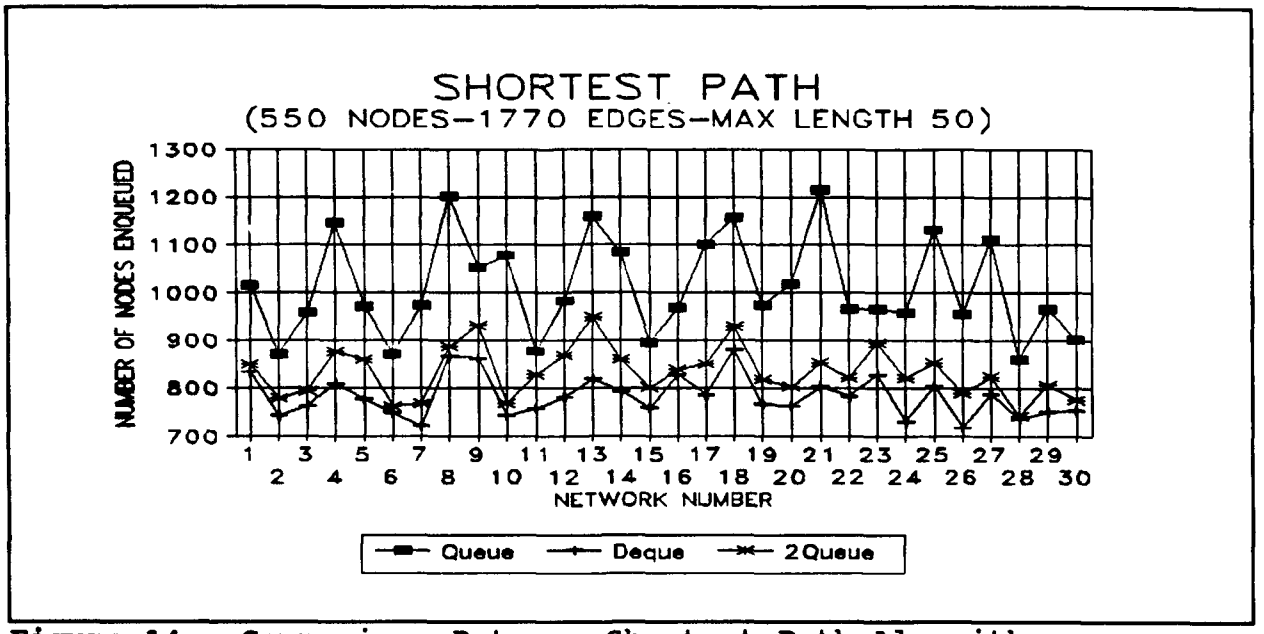


**Figure 14** - Comparison Between Shortest Path Algorithm

## D. MINIMUM COST SPANNING TREE ALGORITHMS

Two algorithm to solve the minimum cost spanning tree problem are implemented. Since they are very different from each other, an analysis that can be done between them is a comparison of the time that they spent to give the minimum cost spanning tree. The time measured is the time from the call of the function that performs the algorithm until it returns to the main program. The hardware used was a PC-compatible with a 80286 processor running at 16 Mhz. The program used 50 networks randomly generated with 550 nodes and 1,470 edges each. We can see in Figure 15 that Prim's algorithm is more efficient than Kruskal's. However, Kruskal's algorithm spent most of the time to sort the edges by increasing cost. If for some reason, the user has the edges already sorted, then Kruskal's algorithm can be faster than Prim's. In Figure 16 we have a comparison between the two algorithms without the time spent to

sort the edges, and Figure 17 shows the time of the two algorithms compared to the time to sort the edges.

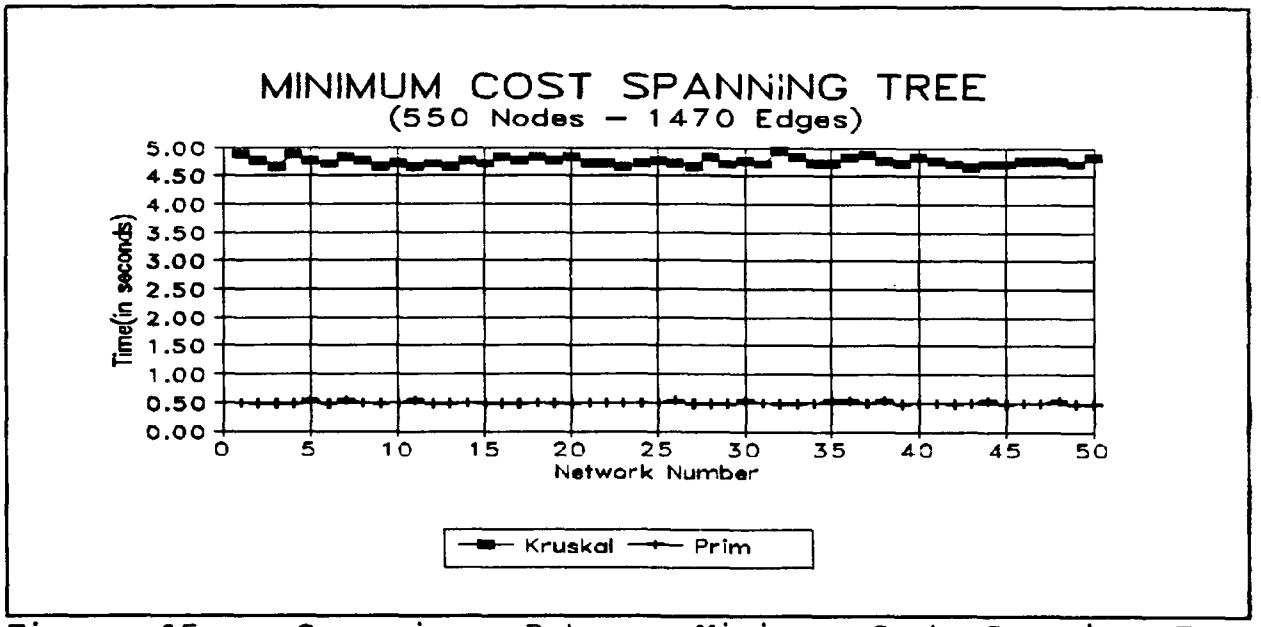


**Figure 15 - Comparison Between Minimum Cost Spanning Tree Algorithms**

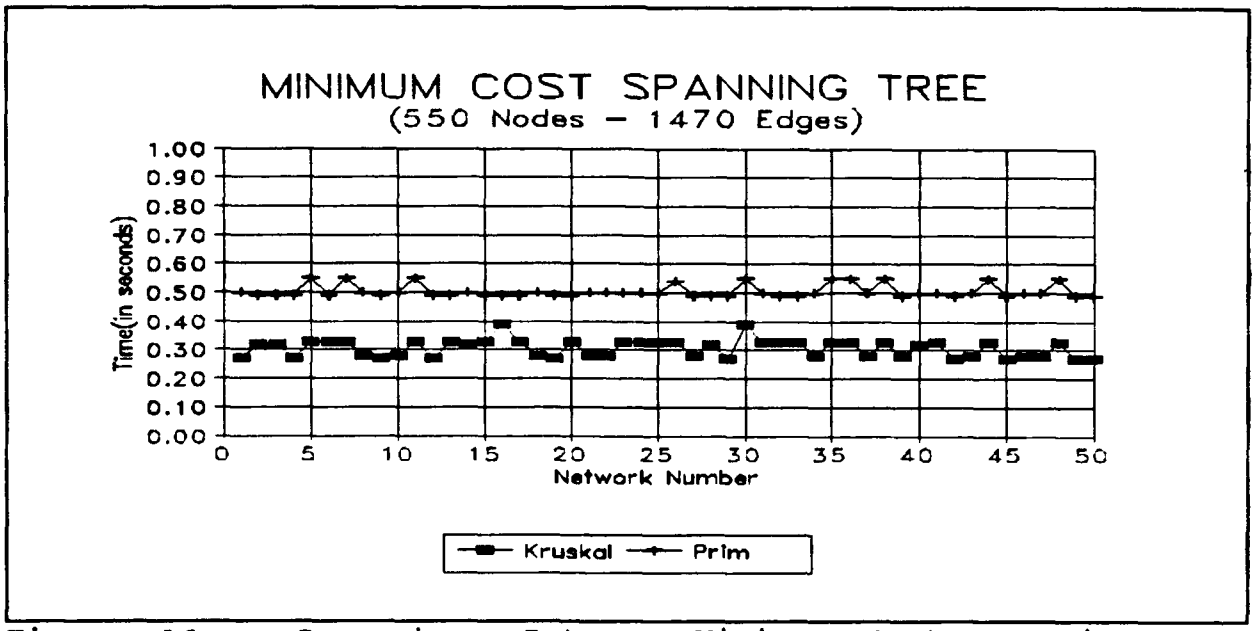


**Figure 16 - Comparison Between Minimum Cost Spanning Tree Algorithms**

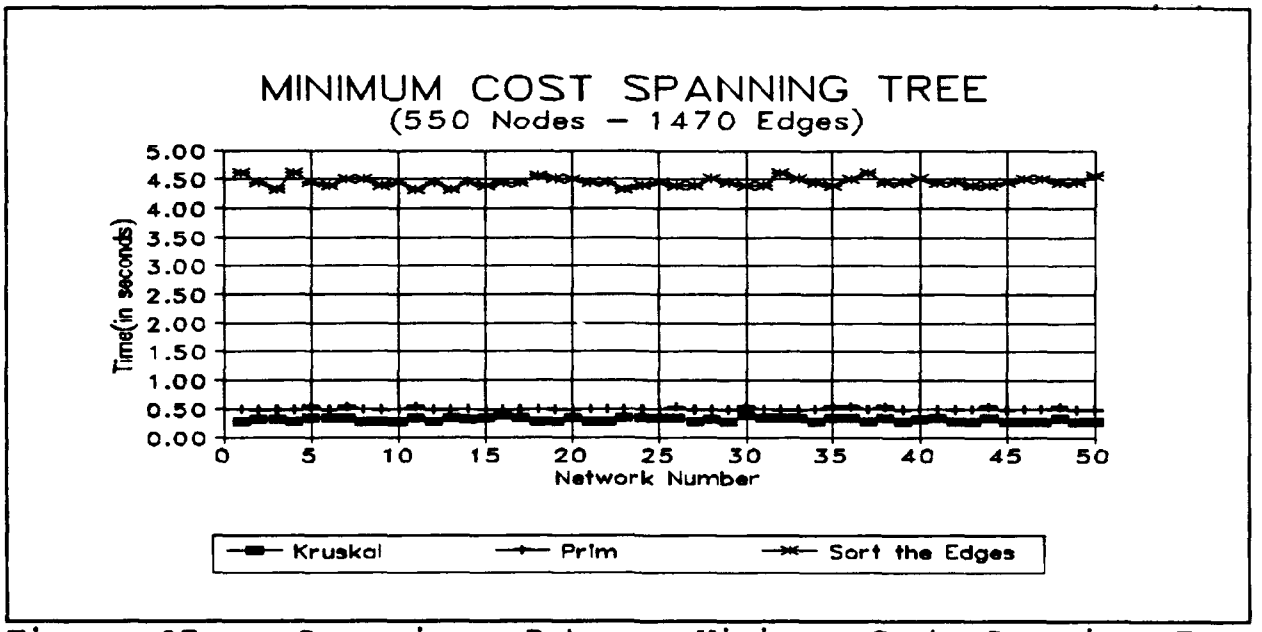


**Figure 17** - Comparison Between Minimum Cost Spanning Tree Algorithms

## E. USING THE SYSTEM IN THE OA-4203 SEMINAR

The system has also been used by the students in the seminar OA-4203 offered in the summer of 1991. The seminar had eleven participants and was a study of graph and network algorithms. The system was used to conduct experiments with coloring, shortest path and minimum cost spanning tree algorithms using randomly generated networks.

### 1. Equivalency Between Packing and Sequential Coloring Algorithms

Experimental results, obtained from programs made using the system, suggested that packing and sequential algorithms provide the same coloring of the network. After several tests involving thousands of networks of different type and size, the result shown identical coloring for both algorithms. Motivated by this fact, two students of the seminar developed formal proofs of

the equivalency of these algorithms. The code of the program that runs both algorithms is shown in Appendix C as an example of the utilization of the system.

### 2. Other Experiments Conducted Using the System

Several coloring algorithm modifications were developed and tested, including a generation of many random permutations to run these algorithms.

A study of the distribution of the number of colors used to color the network and the distribution of node colors using the sequential algorithm was made. The result, that can be seen in Figures 18 and 19, suggests that the distribution is about the same for both sizes of networks and the number of colors used to color them is also about the same. The figures also show some information about the time to generate and to color the networks on a specific computer.

Most of the students participating on the seminar were Fortran programmers. They had no experience with C, but they were able to use the system to implement their ideas, with just a basic introduction to C and the system itself.

## F. HARDWARE AND SOFTWARE USED

The system was developed in a PC-compatible machine with a Intel 80286 processor running at 16 Mhz, with 1 Megabyte of memory and 42 Megabytes of hard disk. The compiler used was Turbo C++, version 1.0 and the operating system used was MS-DOS version 4.01. Part of the tests were executed on this machine.

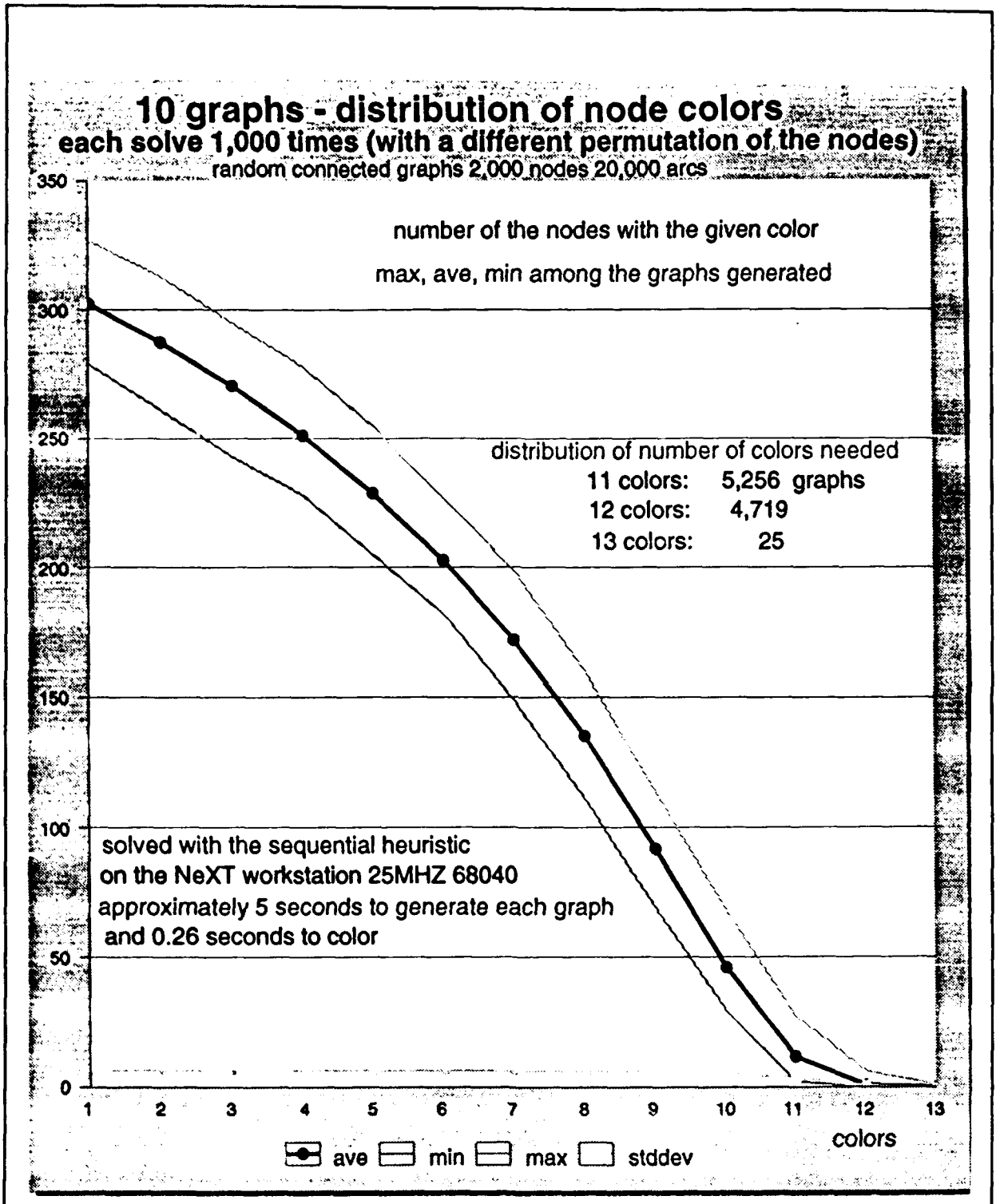


Figure 18 - Distribution of Node Color - 2,000 Nodes - 20,000 Arcs

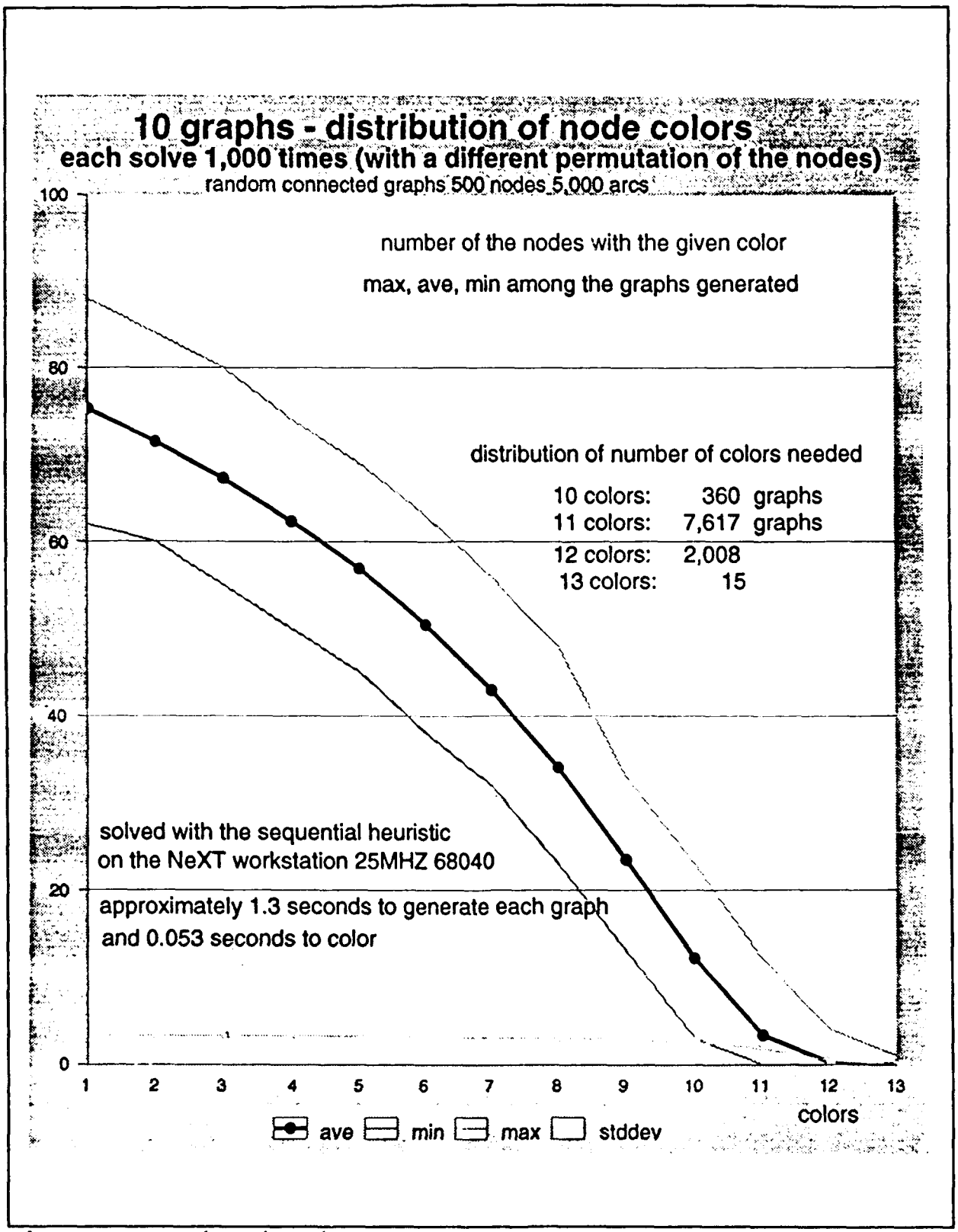


Figure 19 - Distribution of Node Color - 500 Nodes - 5,000 Arcs

Experiments conducted by the students in the seminar and some of the tests were executed using a Next computer with a Motorola 68040 processor running at 25 Mhz and with 16 Megabytes of memory and 330 Megabytes of hard disk. The operating system was the OS Nextstep version 2.0 and the compiler was Objective C based on Free Software Foundation's GNU compiler.

The graphs seen in previous chapters were made in a PC-using Quattro Pro version 3.0 spreadsheet. The graphs seen in this chapter were made using Lotus Improv spreadsheet on a Next computer. The use of different spreadsheet to generate these graphs was made for portability reasons, since they are specific for each hardware and operating system used. For this reason, the system was designed to provide a flexible output that allows produced data be used for any spreadsheet or data analysis program.

# VI. CONCLUSION

Many kinds of "real-world" problems can be represented by graphs and networks and can be solved using existing algorithms. The increase in the capacity of computers now makes it possible and affordable to implement algorithms to solve these problems. This fact is leading researchers and analysts into cc inued study of graph and network theory.

To help analysts construct, test and analyze network problems, a portable computer system was designed and partially implemented. The system was designed to allow analysts to design and implement algorithms in a high level language. The system also provides a random network generator that is able to generate structured and unstructured random networks to permit analysts to test and analyze the algorithms.

The system was implemented in ANSI C. This language was chosen because it is very efficient and very portable. This is important because the system must be portable in order to allow its use on any machine that has a compiler compatible with ANSI C. The system was run on several machines and performed well on every one.

The system was used to implement several algorithms to solve network problems and many other programs to test it were developed. The system has been used and extensively tested in a seminar to study graph and network algorithms. The students participating at the seminar were able to implement different versions of

algorithms, even without any experience in C. They were able to build algorithms and programs to conduct experiments using randomly generated networks. The results obtained with the experiments lead to some important conclusions about the algorithms and the behavior of random networks. It also helped the students in the design and implementation of the programs.

The system is partially implemented and can be used to construct other algorithms and programs that could be interesting. The system can also be expanded, allowing the implementation of new features, such as new data structures and new functions.

## APPENDIX A

```
/*************************************
 *   COLOR01.C                       *
 *************************************
 *   PACKING HEURISTIC FOR COLORING  *
 *   MAJ HOMERO F. OLIVEIRA          *
 *   COMPILER : TURBO C++ 1.0        *
 *************************************/

/*************************************************************
 * This program perform the Packing Heuristic for Coloring.*
 *************************************************************/

/************************ COLOR01 ***************************/

int color01(struct node *graph)
{
 struct node *node,*adj_node;
 int ncolor;  /* number of the color */
 int ncolored;  /* number of nodes colored */
 int v;

 v = total_of_nodes(graph);
 node = first_node(graph);
 while(exist_node(node))
 {
     set_color(node,0);
     node = next_node(node);
 }

 ncolor = 0;
 ncolored = 0;
 while(ncolored < v)
 {
     ncolor++;
     node = first_node(graph);
     while(exist_node(node))
     {
      if(color(node) == 0)
      {
          adj_node = first_adj_node(node);
          while(exist(adj_node) && (color(adj_node) != ncolor))
          {
           adj_node = next_seq_adj_node(node,adj_node);
          }
          if(!(exist(adj_node)))
          {
```

```
                set_color(node,ncolor);
                ncolored++;
              }
          }
          node = next_node(node);
        }
    }
    return(ncolor);
}
```

APPENDIX B

```
/*************************************
 *  COLOR02.C                        *
 *************************************
 *  SEQUENTIAL HEURISTIC FOR COLORING *
 *  NEW VERSION                      *
 *  MAJ HOMERO F. OLIVEIRA           *
 *  COMPILER : TURBO C++ 1.0         *
 *************************************/

/*****************************************************************
 * This program perform the sequential heuristic for coloring.*
 *****************************************************************/

/************************* COLOR02 ****************************/

int color02(struct node *graph)
{
  struct node *node,*adj_node;
  int color;
  int ncolor = 0;
  int *color_vector;
  int v;
  int i;

  v = total_of_nodes(graph);
  node = first_node(graph);
  while(exist_node(node))
  {
    set_color(node,0);
    node = next_node(node);
  }
  color_vector = create_array(v,int);
  node = first_node(graph);
  set_color(node,1);
  node = next_node(node);
  while(exist_node(node))
  {
    adj_node = first_adj_node(node);
    while(exist_node(adj_node))
    {
        if(color(adj_node) != 0)
        {
         *(color_vector + color(adj_node)) = number(node);
        }
        adj_node = next_seq_adj_node(node,adj_node);
    }
```

```
    color = 1;
    while(*(color_vector + color) == number(node))
    {
        color++;
    }
    if(color > ncolor)
    {
       ncolor = color;
    }
    set_color(node,color);
    node = next_node(node);
  }
  erase_vector(color_vector);
  return(ncolor);
}
```

## APPENDIX C

```
/*************************************
 *   TESTCOR.C                       *
 *************************************
 *   MAIN PROGRAM                    *
 *   MAJ HOMERO F. OLIVEIRA          *
 *   COMPILER : TURBO C++ 1.0        *
 *************************************/

/*****************************************************************
 * This program runs the sequential and packing heuristics for *
 * coloring and compares the results.If there is any difference*
 * between the results it will print a message, the difference *
 * found and the nodes where the difference was found.         *
 *****************************************************************/

 #include "randtur.c"
 #include "macros.c"
 #include "creat2.c"
 #include "functions.c"
 #include "color01.c"
 #include "color02.c"


/********************** MAIN ************************/

 void main()
 {
     struct node *graph, *node;
     int tot_comp;
     int tot_nodes;
     int tot_networks;

     int ncolor;
     int *color_vector,*point;
     int i;
     int seed;
     FILE *ofp1;
     char input_file[12];

     printf("Enter the name of the input file.\n==>");
     scanf("%s",input_file);
     get_info_from_file(input_file,&data);
     printf("Enter the seed:\n==>");
     scanf("%i",&seed);
     setSeed(seed);
     printf("Enter tot_networks:\n==>");
     scanf("%i",&tot_networks);
```

```
    for(i=1;i<=tot_networks;i++)
    {
     seed = getSeed();

// *** generates the network
      graph = create_graph(data);
      printf("created network # %5i;\n",i);

// *** runs the packing heuristic
      color01(graph);

// *** saves the coloring obtained
      tot_nodes = total_of_nodes(graph);
      node = first_node(graph);
      color_vector = create_array(tot_nodes,int);
      point = color_vector;
      while(exist_node(node))
      {
          *point = color(node);
          node = next_node(node);
          point++;
      }

// *** runs the sequential heuristic
      color02(graph);

// *** Test if there is any difference between the two results
      node = first_node(graph);
      point = color_vector;
      while(exist_node(node))
      {
          if(*point != color(node))
          {
             printf("******** color different in node %i",
                    number(node));
             printf(" %i %i  ;max color :%i\n",
                     color(node),*point,ncolor);
          }
          node = next_node(node);
          point++;
      }

// *** erases the graph to release memory to new one
      erase(graph);
    }
  }
```

## LIST OF REFERENCES


1. Bell Communications Research, Report R3001, An Overview of NETPAD, by N. Dean, M. Mevamkamp, and C.L. Monma, 7 February 1991.

2. Tarjan, R.E., Data Structures and Network Algorithms, Society for Industrial and Applied Mathematics, 1983.

3. Simeone, B., and others, Fortran Codes for Network Optimization, Scientific Publishing Co., 1988.

4. Buckley, F., and Harary, F., Distance in Graphs, Addison-Wesley Publishing Co., 1990.

5. Wepsic, E., American Mathematical Monthly, p. 18, 18 November 1990.

6. Sedgewick, R., Algorithms in C, pp. 509-519, Addison-Wesley Publishing Co., 1990.

## INITIAL DISTRIBUTION LIST

| | | |
|---|---|---|
| 1. | Defense Technical Information Center<br>Cameron Station<br>Alexandria, Virginia 22304-6145 | 2 |
| 2. | Library, Code 52<br>Naval Postgraduate School<br>Monterey, California 93943-5002 | 2 |
| 3. | Professor Gordon H. Bradley, Code OA/BZ<br>Naval Postgraduate School<br>Monterey, California 93943-5000 | 4 |
| 4. | Professor R. Kevin Wood, Code OA/WD<br>Naval Postgraduate School<br>Monterey, California 93943-5000 | 3 |
| 5. | Maj Av Homero F. Oliveira<br>Centro Técnico Aeroespacial<br>São José dos Campos, São Paulo, Brazil, 12225 | 4 |