JPL PUBLICATION 85-67

Topology Design and Performance Analysis of an Integrated Communication Network

Victor O.K. Li, Y.F. Lam, and T.C. Hou
University of Southern California

Joseph H. Yuen
Jet Propulsion Laboratory

September 15, 1985

National Aeronautics and
Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

TECHNICAL REPORT STANDARD TITLE PAG

1. Report No. JPL Pub. 85-67	2. Government Accession No.	3. Recipient's Catalog No.

4. Title and Subtitle

Topology Design and Performance Analysis of an Integrated Communication Network

5. Report Date
September 15, 1985

6. Performing Organization Code

7. Author(s)
Victor O.K. Li, Y.F. Lam, T.C. Hou, and Joseph H. Yuen

8. Performing Organization Report No
JPL Publication 85-67

9. Performing Organization Name and Address

JET PROPULSION LABORATORY
California Institute of Technology
4800 Oak Grove Drive
Pasadena, California 91109

10. Work Unit No.

11. Contract or Grant No.
NAS7-918

13. Type of Report and Period Covered

JPL Publication

12. Sponsoring Agency Name and Address

NATIONAL AERONAUTICS AND SPACE ADMINISTRATION
Washington, D.C. 20546

14. Sponsoring Agency Code
BG-310-30-71-84-06

15. Supplementary Notes

16. Abstract

This is a research study on the topology design and performance analysis for the Space Station Information System (SSIS) network. We begin with a survey of existing research efforts in network topology design. Then a new approach for topology design is presented. It uses an efficient algorithm to generate candidate network designs (consisting of subsets of the set of all network components) in increasing order of their total costs, and checks each design to see if it forms an acceptable network. This technique gives the true cost-optimal network, and is particularly useful when the network has many constraints and not too many components. The algorithm for generating subsets is described in detail, and various aspects of the overall design procedure are discussed. Two more efficient versions of this algorithm (applicable in specific situations) are also given. Next, we discuss two important aspects of network performance analysis: network reliability and message delays. A new model is introduced to study the reliability of a network with dependent failures. For message delays, a collection of formulas from existing research results is given to compute or estimate the delays of messages in a communication network without making the Independence Assumption. The design algorithm coded in Pascal is included as an appendix.

17. Key Words (Selected by Author(s))	18. Distribution Statement
Spacecraft communications, command, and tracking Communications Mathematical and Computer Sciences Operations Research	Unlimited

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	v + 47	

JPL PUBLICATION 85-67

Topology Design and Performance Analysis of an Integrated Communication Network

Victor O.K. Li, Y.F. Lam, and T.C. Hou
University of Southern California

Joseph H. Yuen
Jet Propulsion Laboratory

September 15, 1985

NASA
National Aeronautics and
Space Administration

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

The research described in this publication was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

Table of Contents

Figures

ABSTRACT

This is a research study on the topology design and performance analysis for the Space Station Information System (SSIS) network. We begin with a survey of existing research efforts in network topology design. Then a new approach for topology design is presented. It uses an efficient algorithm to generate candidate network designs (consisting of subsets of the set of all network components) in increasing order of their total costs, and checks each design to see if it forms an acceptable network. This technique gives the true cost-optimal network, and is particularly useful when the network has many constraints and not too many components. The algorithm for generating subsets is described in detail, and various aspects of the overall design procedure are discussed. Two more efficient versions of this algorithm (applicable in specific situations) are also given. Next, we discuss two important aspects of network performance analysis: network reliability and message delays. A new model is introduced to study the reliability of a network with dependent failures. For message delays, a collection of formulas from existing research results is given to compute or estimate the delays of messages in a communication network without making the Independence Assumption. The design algorithm coded in Pascal is included as an appendix.

1. INTRODUCTION

Topology design of large scale networks has been investigated intensively in the last decade [3]. The basic problem is to find the optimal locations (and maybe capacities) of communication channels within a network with respect to a specified measure of performance and subject to a set of constraints on various parameters. The design variables include network topology, channel capacities, and flow assignment. The major design constraints are channel capacities, network reliability, transmission delay, and network costs. One or more of these constraints may serve as the measure to be optimized. Thus topology design is actually a class of distinct but related optimization problems. Optimal algorithms have been found for some topology design problems [1], but they are computationally inefficient due to the inherent complexity of the problems. Near-optimal heuristics have also been developed for a few types of problems [8], [23].

The Space Station Information System (SSIS) communications network is a complex, mixed media, and time-varying network [17]. It links both space-borne and ground-based elements together by providing the necessary communications, data processing, data storage, and data distribution requirements. Its topology design is further constrained by various technical feasibilities, thus making many ordinary topology design techniques not directly applicable. Section 2 surveys existing research results in network topology design, and investigates the applicability of such results to the SSIS topology design problem. Section 3 presents a new design technique which can explicitly consider every important performance measure of a communication network, and at the same time take into account the constraints due to various technical feasibilities as in the case of the SSIS. Section 4 discusses two of the most important aspects of network performance: network reliability and message delays. It introduces a new model to study the reliability of a communication network in which link failures are statistically dependent. Then it surveys existing research results on network message delays, and discusses those which are

applicable (with modifications) to the SSIS communication network. Concluding remarks and an example are given in Section 5, and the design algorithm coded in Pascal is included as an appendix.

2. SURVEY OF LARGE SCALE NETWORK TOPOLOGY DESIGN TECHNOLOGY

The first step of our research is to survey existing results in network topology design and investigate whether they may be applicable to the SSIS network.

2.1. PROBLEM DEFINITION

The topology design problem for a large scale network can be formulated as follows:

Given:
- terminal and host locations
- traffic matrix
- cost matrix

Over the design variables:
- topology
- channel capacities to be assigned
- flow assignment

Subject to:
- link capacity constraint
- reliability constraint
- delay constraint
- cost constraint

One or more of the above constraints can be considered as the measure to be optimized, and the goal of the design problem is to find the topology that optimizes the specified measure subject to the given constraints. Network cost is usually the most important measure to be optimized in general purpose communications networks.

There are two general classes of networks, namely, centralized networks and distributed networks. The topology design algorithms for both classes of networks are discussed below.

2.1.1. CENTRALIZED NETWORK DESIGN ALGORITHM

The centralized network is characterized by many geographically dispersed terminals which are connected to one or more central computers. These central computers perform the data processing function or serve as data switches. There may also be concentrators which merge data flows so that lines can be used more efficiently.

Consider a three level hierarchical network consisting of terminals, concentrators, and the central computer. Given the locations of the terminals and the central computer, one may be asked to find the optimal locations of the concentrators and the layout of the terminals. The concentrator location problem can be formulated as an integer linear programming problem [3, 34]. Suppose there are n terminals (numbered 1 to n), m potential concentrators (numbered 1 to m), and the central site (which we will designate as concentrator location 0). The cost of connecting terminal i to concentrator j is C_{ij}. Let X_{ij} be 1 if we assign terminal i to concentrator j, and 0 otherwise. Also let Y_j be 1 if at least one terminal is using concentrator j, and 0 if no one is using it. The cost of concentrator j is F_j. Then the total cost of a particular assignment is

$$\text{total cost} = \sum_{i=1}^{n} \sum_{j=1}^{m} C_{ij} \cdot X_{ij} + \sum_{j=1}^{n} F_j \cdot Y_j$$

Since each terminal must be connected to exactly one concentrator, and concentrator j can handle a maximum number (K_j) of terminals, we have the constraints that

$$\sum_{j=1}^{m} X_{ij} = 1, \qquad i = 1,\ldots,n$$

$$\sum_{i=1}^{n} X_{ij} \leq K_j, \qquad j = 1,\ldots,m$$

$$\text{and} \quad Y_j = 1 - \prod_{i=1}^{n}(1 - X_{ij}), \qquad j = 1,\ldots,m$$

The problem is to assign 0's and 1's to X_{ij}'s to satisfy the constraints and minimize the cost. This formulation is very similar to warehouse location problems which are studied in operations research literature [1]. The resulting integer linear programming problem can then be solved either by exhaustive search or by branch-and-bound procedures [6, 7, 20]. Although the formulation of the problem is easy, the solution takes time exponential in the size of the given network. Moreover, this technique cannot be easily extended to layouts other than hierarchical.

There are also heuristic algorithms [19, 27, 35] for solving the concentrator location problems, but they do not give optimal results. Most of them originate from the ADD [23] and DROP [8] algorithms. The ADD algorithm starts with all the terminals connected to the central site, and concentrators are added one at a time. Concentrator 1 is introduced first. Terminals are then assigned to either the central site or to concentrator 1. If the resulting cost is cheaper than having all terminals connected to the central site, concentrator 1 is a possible candidate. Next, all terminals are reassigned back to the central site, concentrator 1 is deleted, and concentrator 2 is inserted. The cost of this two-concentrator system is computed. This process is repeated for all m potential concentrators, and the one that gives the minimum cost is found. This concentrator, say l, has now been chosen and will appear in the final configuration. Now consider the three concentrator problem consisting of concentrators i, l, and the central site. Do this for each of the m-1 remaining concentrators. Select the best triple and use it to solve the four concentrator problem. Keep adding one concentrator at a time to the best possible location as long as the cost continues to decrease. The procedure is stopped when no further improvement can be made by adding more concentrators.

The DROP algorithm is the exact reverse of the ADD algorithm. It starts with all possible concentrators in use. Assign each terminal to one of the m+1 concentrators and

compute the cost. Then the concentrators are discarded one at a time to maximize reductions in cost, and the procedure is stopped when no additional deletions can further reduce the cost.

The terminal layout problem asks to find a tree that connects all terminals and has the minimum cost. This is the minimum spanning tree (MST) problem for which many efficient algorithms exist. The most well known ones are Prim's algorithm [29] and Kruskal's algorithm [22]. Prim's algorithm starts with the central site in the minimum spanning tree. It then connects terminals to the existing tree one at a time in the order of increasing costs. On the other hand, the Kruskal's algorithm examines each potential link in the order of increasing costs and takes in the link provided it does not form a circuit with the links that have already been included. Both algorithms give spanning trees of the same optimal cost. However, these two algorithms will not work when the terminal layout is constrained. For example, there may be a limit on the maximum degree of the spanning tree, or on the traffic on each link. No practical algorithm for finding large constrained minimum spanning tree is known. However there are many heuristic algorithms for doing so and they result from modifications of Prim's and Kruskal's algorithms. For example, Kruskal's can be modified as follows: Each time a new link is about to be added, it is checked to see if the two components being merged satisfy the constraint. If not, do not merge them. Whenever a component grows to size k, it must be immediately connected directly to the central site. Kershenbaum and Chou [18] have given a unified heuristic that can be specialized to yield Kruskal's, Prim's, and several other heuristics as well.

2.1.2. DISTRIBUTED NETWORK DESIGN

The structure of a large distributed network is generally a multilevel hierarchical structure with a backbone network at the highest level, and local access networks at lower levels. Packet switching is a reliable and cost effective solution for distributed data communication requirements consisting of a mixture of interactive voice and bulk data traffic. In designing a distributed computer communications network, we are mostly concerned with cost, throughput, response time, and reliability. All these are direct results of the routing and topological structure of the network. In this section we will concentrate on the topology design aspect.

One basic design technique is called branch exchange. It is a search procedure which optimizes network topology by making a series of changes to small sections of a large network. The procedure is to pick two links that are not too far apart and remove them. Then two new links are added to connect the four affected nodes in a different way, and the result is checked to see if cost is reduced. This process is repeated until a local optimum is achieved. This technique was first discussed in [9], and its application to large network optimization problems was discussed in [10] and [11].

The concave branch elimination method [13] is a computationally more efficient design method. It approximates linear costs as continuous concave functions and assigns traffic to links based on derivatives of delay functions. However, its application is limited to cases where the discrete costs can be reasonably approximated by concave functions.

An evolution of the branch exchange method is called the "cut-saturation" algorithm [14]. Instead of adding and deleting links arbitrarily, it adds links only across saturated cuts and deletes links only from the sub-network that are separated by saturated cuts. First, it sorts all the links by percent utilization (i.e., traffic carrried/capacity). Starting with

the most utilized links, it removes links until the network has been separated into two parts. Those links which are thus removed form a cut. In order to minimize this cut, each link is in turn tentatively put back into the network. If putting a link back does not reconnect the network, it is not part of the minimum cut. The nodes adjacent to the minimum cut are called primary nodes. The nodes adjacent to a primary node are called secondary nodes. The remaining nodes, which are called tertiary nodes, can be divided into the left tertiary and the right tertiary nodes, depending on which of the two components they are in. The algorithm provides for both adding and deleting links. One may start either with a sparse topology and add many links, or a rich one and delete links, or any other acceptable topology and do both. When adding a link, one end should be a left tertiary and the other a right tertiary node. The guidelines for making the choice are choosing the cheapest link (getting the cost down) or the pair of nodes whose best path is the most saturated (improving the performance). When removing a link, the one which is the least utilized is chosen, without regard to its position in relation to the cut. If a topology has a smaller delay time than is required, a link is deleted to reduce the cost. If it has a higher delay than desired, a new link is added to reduce the delay. Studies have shown that the cut-saturation method gives better solutions than the branch exchange method and the algorithm is also computationally more efficient. It is a near-optimal algorithm. This cut-saturation algorithm is somewhat biased to the ARPANET[1] environment. Chou and Sapir [5] presented a generalized cut-saturation algorithm which improves its flexibility and effectiveness.

[1]The Department of Defense Advanced Research Projects Agency Network

2.2. TOPOLOGY DESIGN OF THE SSIS NETWORK

The SSIS communications network is built around two existing National Aeronautics and Space Administration (NASA) communications networks: the NASA Communications (NASCOM) and the Tracking and Data Relay Satellite System (TDRSS). Ground-to-ground communications use NASCOM, which is centrally controlled by the Goddard Space Flight Center (GSFC). Ground to low earth orbit communications and tracking services will be provided by the TDRSS, which is centrally controlled at White Sands by GSFC. This is not a traditional satellite network with the satellite serving as a big relay in the sky for ground-to-ground data transfer. It is rather a network which is responsible for data transfer between space and the Earth. The Space Station communicates with GSFC primarily through TDRSS. A backup of the Space Station is provided by the Space Station Operations Support Center (SSOSC), which is conceived of as a duplicate Space Station. It will not use its full capacity in normal operations. Broadcast TV, facsimile, telephone, and TWX are delivered between the Space Station and the Earth through the Geosynchronous Communications Satellite (GCS). Thus, bulk data traffic and interactive voice (video) traffic are sent through different channels. We also have objects like the Space Transportation System (STS) orbiter, the Manned Maneuvering Unit (MMU), the Teleoperator Maneuvering System (TMS), and the Orbital Transfer Vehicle (OTV) moving around the Space Station and communicating directly with it.

Circa 1990, the Space Station will act like a concentrator with terminals: TMS's and STS orbiters. It will be connected to GSFC via TDRSS and GCS (primary) or via SSOSC (backup). The former is part of the existing NASA communications network. So, we are confronted with a concentrator (SSOSC) location problem and a terminal (TMS, STS orbiter) layout problem. The SSOSC location can be determined by trivial (one terminal: Space Station) linear integer programming. Since TMS's and STS orbiters are mobile, the terminal layout algorithms mentioned in Section 2.2 are not applicable. We have to come

up with a new method to tackle this problem. Circa 2010, in addition to TMSs and STS orbiters, we will have OTVs and MMUs which form the local access network. The scenerio is almost the same as that circa 1990, except that the number of terminals will increase. The Space Station, however, cannot be thought of as just a concentrator any more. We believe it is reasonable to treat the Space Station, the Space Platform, the SSOSC, the Telecommunications and Data Acquisition System (TDAS), etc. as backbone nodes of a distributed network. To meet the reliability and the autonomy requirements, it has to be a distributed network. Besides, future expansions (e.g. more Space Stations, integration with other networks) will certainly make SSIS a distributed network. In designing this backbone network topology, we are also constrained by the existing NASCOM. And we have to consider the mobile feature of some of its component nodes. So, we will be facing the problem of determining the locations of SSOSCs and the problem of interconnecting backbone nodes.

It is now obvious that no existing network topology design algorithm is directly applicable to the SSIS topology design problem. Although the literature abounds in topology design techniques, most of them are primarily tailored for traditional wireline networks, i.e., networks where node locations are fixed and messages transmitted along different channels do not interfere with each other. Furthermore, existing algorithms invariably try to optimize the cost of the topology subject to simple connectivity constraints. Other important performance measures of a network, like reliability and delay, are seldom taken into consideration. The SSIS network topology is also constrained by technical feasibilities. For example, a space-ground link cannot be simply added at will. We therefore conclude that existing network topology design techniques are not directly applicable to the SSIS network topology design problem.

3. NETWORK TOPOLOGY DESIGN BY EFFICIENT ENUMERATION

Since existing network topology design algorithms usually only optimize network costs subject to simple connectivity constraints without considering other important performance measures of a network such as reliability and delay, the algorithms are not directly applicable to the SSIS topology design problem. In this section, we propose a new design technique which considers all such important performance measures explicitly, and at the same time takes into account the constraints due to various technical feasibilities of the SSIS.

3.1. A NOVEL APPROACH

We use an efficient algorithm to generate candidate network designs (consisting of subsets of the set of all network components) in increasing order of their total costs. Technical constraints are taken care of at this stage by properly forming the starting set of candidate components (for example, nonfeasible links are simply not included). For each subset generated, we test to see if it forms an acceptable network by checking whether all other requirements are satisfied. Thus the first feasible subset encountered gives the cost-optimal topology satisfying all given constraints. This section discusses in detail an efficient algorithm that can generate subsets of a given set of elements in increasing order of their total costs. Two modified versions of this algorithm, which are more efficient in some situations, are given. Various aspects of the overall design procedure using this new approach are also discussed.

3.2. THE ALGORITHM

We describe an efficient algorithm which can generate subsets of a given finite set of elements in increasing order of their total costs. Subsets are generated in the correct order independent of those not yet generated, so that the algorithm can be stopped at any time to yield an ordered list of the lowest-cost subsets.

We are given a set $S = \{e_1, e_2, ..., e_n\}$ of n elements, sorted such that $w(e_i) \geq w(e_j)$ for all $i > j$, where $w(e_i)$ is the (non-negative) weight of element e_i. We want to generate subsets SS_1, SS_2, ..., such that $SS_i \subseteq S$ for all i, and $w(SS_i) \geq w(SS_j)$ for all $i > j$, where $w(SS_i)$ is the total weight of all elements in subset SS_i. The elements in each subset are also listed in order of increasing weights.

A priority queue is used in our algorithm to store candidate subsets. One practical implementation of a priority queue is a heap [15]. A heap is a complete binary tree with the property that the value of each node is no larger than the value of its children nodes (if they exist). Thus the root of a heap always has the minimum value. In a heap, a node can be deleted or inserted in O(log k) operations without affecting the ordered structure of the binary tree, where k is the number of nodes in the tree. In our algorithm, every node of the heap is a candidate subset, and its value is just the weight of that subset.

We adopt the following notations:

$e_L(SS_i)$ = last element (the one with largest weight) in subset SS_i.

$n(e_i)$ = the next element after e_i in set S, that is, e_{i+1}.

Tree = the heap for storing candidate subsets.

SSR_i = the subset at the root of the heap.

$SS_i - \{e_j\}$ = the subset SS_i with element e_j deleted.

$SS_i + \{e_j\}$ = the subset SS_i with element e_j added.

Algorithm ORDER-II is as follows:

```
Initialize:  i := 1;  SS_1 := {e_1};  Tree := φ;
Repeat
  if e_L(SS_i) ≠ e_n then
    begin
      add SS_i - {e_L(SS_i)} + {n(e_L(SS_i))} to Tree;
      add SS_i + {n(e_L(SS_i))} to Tree;
    end;
  SS_{i+1} := SSR_i;
  delete SSR_i from Tree;
  i := i + 1;
Until enough subsets have been generated.
```

THEOREM 2.1: The above algorithm correctly generates subsets in increasing order of their weights.

PROOF: From the algorithm, the two subsets obtained from SS_i have more weight than SS_i. From Tree, only the currently best subsets are picked. Thus subsets are generated in increasing order of their weights. Since the two subsets obtained from SS_i are distinct and are different from SS_i, no duplicates are generated. It is also obvious that the algorithm will generate all possible subsets if allowed to run to completion. Therefore subsets are generated correctly in increasing order of their weights (Q.E.D.).

To find the computational complexity, we note that obtaining a new subset takes O(log k) steps for deletion from and insertion into Tree [15], and O(n) steps for listing its elements, where k is the current number of subsets in Tree and n is the total number of given elements. Therefore to generate m subsets (each with O(n) elements) takes O(mn + m log m) = O(mn) steps. This is a linear algorithm in m and n.

3.3. NETWORK DESIGN BY ORDERED ENUMERATION

We now describe how one may use the ORDER algorithm to solve the network topology design problem.

3.3.1. DESIGN PROCEDURE

There are two main steps in using the above algorithm to design the topology of the SSIS communication network:

STEP 1. Obtain data (costs, reliabilities, capacities, and others) of all "feasible" links of the SSIS network. A link between two network nodes is feasible if it already exists or can be implemented technically. For example, it may be highly impractical to provide a direct link between the Manned Maneuvering Unit (MMU) and a ground unit. For existing links, data are readily available; for other links, we have to compute or estimate data.

STEP 2. Use the above algorithm to generate subsets of these feasible links in increasing order of their total costs. Test each subset generated to see if it forms a network that satisfies all given constraints (for example, connectivity, reliability, delay, and others). The first satisfactory subset gives the cost-optimal network topology.

3.3.2. SOME TECHNICAL CONSIDERATIONS

The SSIS network has only a moderate number of components but quite a handful of technical feasibility constraints. The result is that the total number of "feasible" links will not be too big, so that the ordered enumeration technique just described is not impractical. Actually a large part of the SSIS network (especially the ground network) already exists, thus further reducing the search space of the topology design problem. It might also be possible to decompose the network into parts and design each separately. This design technique gives us the true cost-optimal topology satisfying all constraints, while many other algorithms are just heuristics.

In the overall design procedure, the most important part is the screening of each subset of feasible links generated by the algorithm. The screening actually consists of several different tests, like the connectivity test, the reliability test, the delay test, and so on. The order of these tests is significant because a wise choice may greatly reduce the amount of work spent in unnecessary testing. In general the easier tests should precede the harder ones. However we should also consider the screening power of each test, since we would to like to reject unsatisfactory designs in as few tests as possible.

The connectivity test, which checks whether the whole network is properly connected, seems to be the best candidate to head the list of all tests. It is a very easy test and takes linear time in the number of links present [15]. Its screening power is also good since a subset of feasible links can pass this test only if the subset forms a connected component. This test can easily be included in the computer program of the algorithm. On the other hand, the reliability test is a computationally difficult one. The problem of computing network reliability has recently been shown to be NP-hard, so that all reliability evaluation algorithms run in exponential time in the worst case [2]. Therefore this test should be placed at the bottom of the list. The delay test is a relatively simple one (depending on how many delay parameters are of interest), and so may be placed right after the connectivity test. Other additional tests should be evaluated in a similar manner and then positioned appropriately in the list of all tests.

We can also skip some of the tests in the first round of the design, and let the computer generate several good candidate topologies. Then we can choose the most preferable one from these candidates based on any other technical factors we might be confronted with.

3.3.3. MODIFYING THE ALGORITHM FOR BETTER EFFICIENCY

The algorithm given in the last section generates all possible subsets, including those that contain just one or two elements. It is obvious that subsets that contain too few links cannot form a connected network. So it would be desirable to modify the algorithm such that it skips these obviously useless subsets.

We now describe a modified version of Algorithm ORDER-II which generates subsets that contain a fixed number (k) of elements, still in increasing order of their total costs. The same notations are used, and a priority queue is still used to store candidate subsets. Every subset SS_i will have its elements listed also in increasing order of weights, and we let $SS_i[j]$ denote the jth element in subset SS_i. For a given set $S = \{e_1, e_2, \ldots e_n\}$ of n elements, we introduce an additional element e_{n+1} in the algorithm just for convenience, and this fictitious element will not appear in any generated subsets.

Algorithm ORDER-II-FIX is as follows:

Initialize: i := 1; $SS_1 := \{e_1, e_2, \ldots, e_k\}$; $jumper_1 := k$; $stopper_1 := e_{n+1}$;
Repeat
  if $n(SS_i[jumper_i]) \neq stopper_i$ then
    add $SS_i - \{SS_i[jumper_i]\} + \{n(SS_i[jumper_i])\}$ to Tree;
    (* This new subset has the same jumper and stopper values as SS_i. *)

  if ($jumper_i \neq 1$) and $n(SS_i[jumper_i-1]) \neq SS_i[jumper_i]$ then
    add $SS_i - \{SS_i[jumper_i-1]\} + \{n(SS_i[jumper_i-1])\}$ to Tree;
    (* This new subset has jumper := $jumper_i-1$ and stopper := $SS_i[jumper_i]$. *)

  $SS_{i+1} := SSR_i$;
  delete SSR_i from Tree;
  i := i + 1;
Until enough subsets have been generated.

The proof of correctness for the above algorithm is similar to the previous proof, although the above algorithm looks more complicated. Moreover, the above algorithm

has the same computational complexity. That is, it takes O(mk) steps to generate m subsets each of which contains k elements.

Algorithm ORDER-II-FIX only generates subsets of a fixed size. However, it is not difficult to observe that it can be easily merged with the previous algorithm to generate subsets which contain at least k elements. The idea is that now each k-element subset will not only give rise to new k-element subsets, but will also grow, in a manner governed by Algorithm ORDER-II, until it contains n elements. In other words, for every k-element subset SS_i generated, we use Algorithm ORDER-II to generate subsets from elements $n(SS_i[k])$ through e_n and append them to SS_i. Every candidate subset in Tree will be in either one of two statuses: fixed or growing. If a candidate subset is in the fixed status, that means it has k elements, and once it is picked from Tree, it can give rise to as many as three new candidate subsets. Two of them will still be of size k, while the third one will be of size k+1. (Of course, if the last element in a candidate subset is already e_n, then it will not give rise to any new subsets.) If a candidate subset is in the growing status, that means it has more than k elements, and once it is picked from Tree, it can only give rise to two new candidate subsets according to Algorithm ORDER-II. This combined algorithm still enjoys the same complexity as before.

4. RELIABILITY AND DELAY ANALYSIS

An important step of the design technique given in the last section is the testing of candidate topologies generated by the enumeration algorithm. It is actually a series of different tests for the various aspects of network performance. In this section we will concentrate on two of the most important aspects: network reliability and message delay.

4.1. TWO IMPORTANT TESTS IN THE ALGORITHM

We have pointed out that the connectivity test, which checks whether a network is properly connected, shall be first applied to each candidate topology. This is because connectivity is a very basic requirement of every communication network, and the test is simple and has good screening power. Numerous efficient connectivity-testing algorithms have been published, and therefore will not be repeated in this section. For a good reference see [15].

The first part of this section deals with network reliability. In particular we present a new model to study the reliability of a communication network in which link failures are statistically dependent. In the second part, we survey existing research results on network message delays, and discuss those which are applicable (with modifications) to the SSIS communication network.

4.2. RELIABILITY OF COMMUNICATION NETWORKS WITH DEPENDENT FAILURES

One important performance measure of a communication network is reliability. Reliability analysis of networks or other complex systems has been studied for many years, and numerous algorithms and evaluation techniques have been proposed (see [16] for a general review). However, almost all of them make the assumption that component failures are statistically independent. For most real world situations this assumption of independence does not hold. In this subsection we shall study the reliability of networks with dependent failures.

4.2.1. BACKGROUND

There have been very few known attempts to study the reliability of communication networks with interdependent components. One approach is to specify statistical dependencies between network components by conditional probabilities of failure, so that the joint probability of failures of two (or more) dependent components can be evaluated using chain rule expansion. A major problem with this approach is that the number of parameters to be dealt with is exponential in the number of failure-prone components. Furthermore, the set of conditional probabilities has to satisfy a consistency requirement (see [24] for a discussion of these problems).

A q-ψ model was developed in [33] to simplify certain types of failure dependencies between the communication links of a network. Unfortunately, this model is not consistent [24]. More recently, a new ε-model was developed in [28] to incorporate more general types of failure dependencies. It still employs conditional probabilities to specify dependencies, but the authors made use of standard rules of probability and the consistency constraint to reduce the total number of parameters that have to be initially specified for the model. However, the minimum number of parameters required is still exponential in the number of failure-prone communication links.

Since using conditional probabilities to specify failure dependencies presents such inherent problems, a totally different approach was taken in [24] to avoid them. A simple Colored Network Model (CNM) was used to model a specific kind of failure dependencies between communication links. The CNM can be easily transformed to a network whose links are perfectly reliable and whose nodes fail independently, so that its reliability can be evaluated using numerous existing techniques. The restriction of the model is that *links incident to a communication center have to fail in mutually exclusive groups.*

A more recent research result is the development of a new model called the Event Network Model (ENM) in [25]. It incorporates more general cases of interdependent component failures of communication networks without using conditional probabilities. The model is simple and flexible, and network reliability can be computed using a known and very efficient algorithm with a minor modification.

4.2.2. THE EVENT NETWORK MODEL

One major reason why the links of a communication network do not fail independently is that there exist events which can cause the simultaneous failures of several links. For example, in a communication center, several out-going links may share a significant amount of common equipment, in which case the assumption of independent failures obviously does not hold. Also, links within the same geographic vicinity are likely to be affected simultaneously by the same environmental impacts. Furthermore, there may be some atmospheric and cosmic effects which can disrupt radio communication in certain frequency bands. The ENM models such situations in the following manner: Communication centers and links are represented as usual by vertices and edges of a graph respectively, while failure-causing events are modeled by "event elements" which are added to the affected edges (links). An event element is said to be in the "down" mode when the corresponding failure-causing event occurs, and is said to be in the "up"

mode otherwise. All failure-causing events are assumed to be independent and occur with known probabilities.

A simple example is shown in Figure 4-1. The network consists of 4 centers (A, B, C, and D) connected in a bridge configuration, and there are 9 event elements scattered on the links. The relationship between event element failures and link failures is not difficult to visualize. Consider link A-B. It is governed by 3 event elements (1, 3, and 9), and so the link operates if and only if all these 3 event elements are in the "up" mode. If an event element is in the "down" mode, then all links affected by that event element will fail.

The use of ENM in reliability modeling and analysis of networks with dependent failures has the following advantages:

1. Since event elements of the ENM are statistically independent, the model is always consistent. The number of parameters to be handled is also greatly reduced.

2. Event elements and their probabilities of occurrence are physically more meaningful than conditional probabilities of failure. The ENM gives a better understanding of component correlations and causes of component failures, which are of crucial importance to the maintenance and improvement of network performance.

3. The ENM is a very flexible model. A special case of the model is when every event element falls on only one link, and this corresponds to the traditional assumption of independence. As new information is gained or observed, the model can be updated by simply adding new event elements, with no or minimum changes to existing parameters. This makes the ENM adaptive to changes in network operating conditions.

4. Although the ENM is a more general model, its reliability computation is not more difficult than that of the traditional model which assumes independent failures. There is a known and very efficient algorithm for computing network reliability which can be applied to the ENM with a minor modification and no significant increase in computational complexity (for more details see [25]).

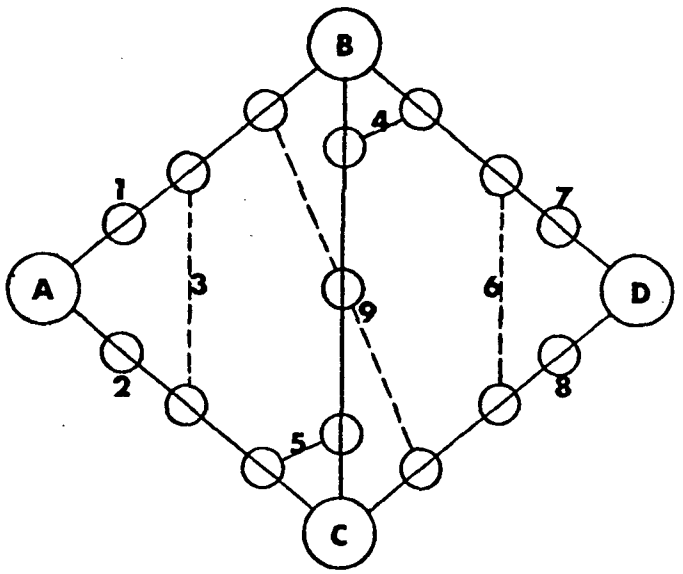


Figure 4-1: An Example of the Event Network Model

Parameter identification and estimation for this new model are interesting and practical problems which have to be solved before this model can be put to actual use. Sources of information and data shall include past history of network failures, and detail knowledge of the communication equipment used and the environmental conditions under which the system operates. The latter is perhaps the principal source of information for the SSIS communication network. The ENM itself is a very simple model, but applying it to real world situations like the SSIS requires detail and complete knowledge of all practical factors involved which are beyond the scope of this study. In the SSIS communication network, the ENM can explicitly model such failures as those caused by jamming, atmospheric and cosmic conditions on certain frequency bands, and other similar natural or man-made factors. These usually affect more than one communication link simultaneously, and are often the major causes of dependent failures.

4.3. MESSAGE DELAYS IN COMMUNICATION NETWORKS

Message delay is an important performance measure in a communications network. Most existing delay analysis makes the "Independence Assumption" [21]. We do not believe this assumption is valid in the SSIS network, and we will discuss an approach to find message delays without using it.

4.3.1. BACKGROUND

Analysis for a single queue has been studied extensively, with the usual assumption that service times and customer interarrival times are independent. For a network of queues, since messages preserve their lengths as they traverse the network, the interarrival and service times at each internal queue are dependent. The distribution of the delay is thus mathematically intractable. One way of tackling this difficulty is to make the "Independence Assumption" [21]. Simulation studies have shown that the independence assumption gives acceptable results only when the number of incident edges at a vertex is large. In the SSIS network, stations have only small numbers of incoming channels, and the packet length is fixed and not random. These are the two reasons why we cannot make the independence assumption in the delay analysis of SSIS network.

4.3.2. ESTIMATION OF MESSAGE DELAYS

In SSIS, the packet (referred to as a frame in SSIS terminology) is of fixed length. We will assume that the input messages to the network are governed by Poisson statistics [12]. A number of results concerning message delays are presented by Calo in [4]. These include: ordering relations for the successive waiting time in the channel, waiting time properties under extreme conditions, and simple bounds for systems with uniformly bounded service processes. However, these results are not useful to us. Rubin [30, 31, 32] has derived many useful formulas for this kind of network. They will be our basic tools in the delay analysis of the SSIS network topology design.

4.3.2.1. Tandem Queues With Single Message Stream

We first consider a path v_1-v_{n+1} as shown in Figure 4-2. The capacity and transmission time on edge v_i-v_{i+1} are C_i and $\tau_i = \alpha/C_i$ respectively, where α is the packet length. Suppose packets arrive at v_1 with rate λ and depart at v_{n+1}, and there are no other packets being transmitted over any segment of the path. The average (steady state) packet waiting time at v_i, $W(v_i)$, when $\rho_i = \lambda\tau_i < 1$, is

$$W(v_i) = \frac{1}{2}\frac{\rho_i(max)}{1-\rho_i(max)}\tau_i(max) - \frac{1}{2}\frac{\rho_{i-1}(max)}{1-\rho_{i-1}(max)}\tau_{i-1}(max) \qquad (1)$$

where $\tau_i(max) = max(\tau_1, \tau_2, ..., \tau_i)$ and $\rho_i(max) = \lambda\tau_i(max)$.

By summing $W(v_i)$ over all i's, we can obtain the overall average waiting time for the n-channel path,

$$W = \sum_{i=1}^{n} W(v_i) = \frac{1}{2}\frac{\rho_{max}}{1-\rho_{max}}\tau_{max}$$

where $\tau_{max} = max(\tau_1, \tau_2, ..., \tau_n)$ and $\rho_{max} = \lambda\tau_{max}$.

The overall average delay is equal to

$$T = W + \sum_{i=1}^{n} \tau_i$$

The above formulas are useful when the message traffic is sporadiac so that there is only one message stream passing through the network. Therefore, given any source-destination pair, we can always obtain the end-to-end delay [26] if we assume there are no simultaneous message flows.

4.3.2.2. Joint Queues With Simultaneous Message Streams

Next, we consider three basic flow interference configurations (Figure 4-3).

(i) One Internal Stream and One External Stream

Consider a vertex into which single internal and external streams arrive (Figure 4-3a). The external stream is Poisson with intensity λ_3, while the internal stream is the departure process from a channel with transmission time τ_1, $\tau_1 = \alpha/C_1$, whose input is a Poisson process with intensity λ_1 (either an external stream or a Poisson approximation for an internal stream). These two streams arriving at v are to be transmitted by a common channel with transmission time τ_3, $\tau_3 = \alpha/C_3$. The approximate average waiting time at v is given by

$$W(v)= \begin{cases} \frac{1}{2}\frac{\rho_3}{1-\rho_3}\tau_3 - (1-\rho_{33})\frac{1}{2}\frac{\rho_1}{1-\rho_1}\tau_1 & \tau_1 < \frac{\tau_3}{1-\rho_{33}} \\ \frac{1}{2}\frac{\rho_{33}}{1-\rho_3}\tau_3 & \tau_1 \geq \frac{\tau_3}{1-\rho_{33}} \end{cases} \qquad (2)$$

where $\rho_1 = \lambda_1\tau_1$, $\rho_{33} = \lambda_3\tau_3$, and $\rho_3 = (\lambda_1 + \lambda_3)\tau_3 < 1$.

(ii) Two Internal Streams

We now consider a vertex v with two input streams departing from channel 1 and channel 2, with transmission times τ_1 and τ_2, respectively, and then being transmitted through a common channel 3 with transmission time τ_3 (Figure 4-3b). The input streams

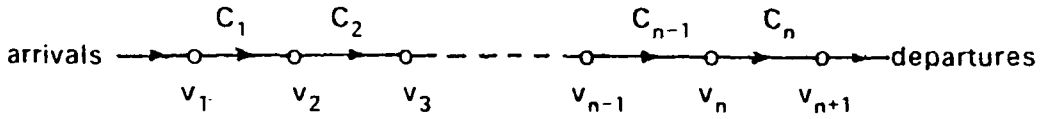
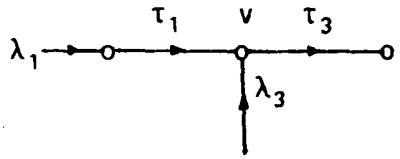
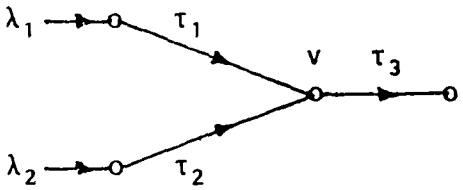


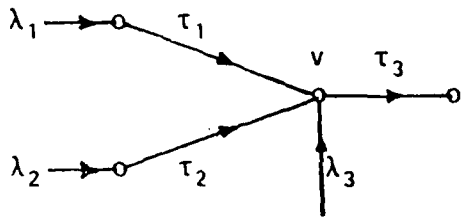
Figure 4-2: Tandem Queue



(a)



(b)



(c)

Figure 4-3: Three Flow Interference Configurations

to channels 1 and 2 are Poisson processes with intensity λ_1 and λ_2, respectively. The approximate average waiting time at v is

$$W(v) = W_1(v) + W_2(v) \tag{3}$$

where

$$W_1(v) = \begin{cases} \frac{1}{2}\frac{\rho_{13}}{1-\rho_3}\tau_3 - \frac{1}{2}\frac{\rho_1}{1-\rho_1}\tau_1(1-\rho_{23}) & \tau_1 < \frac{\tau_3}{1-\rho_{23}} \\ 0 & \tau_1 \geq \frac{\tau_3}{1-\rho_{23}} \end{cases}$$

$$W_2(v) = \begin{cases} \frac{1}{2}\frac{\rho_{23}}{1-\rho_3}\tau_3 - \frac{1}{2}\frac{\rho_2}{1-\rho_2}\tau_2(1-\rho_{13}) & \tau_2 < \frac{\tau_3}{1-\rho_{13}} \\ 0 & \tau_2 \geq \frac{\tau_3}{1-\rho_{13}} \end{cases}$$

and $\rho_1 = \lambda_1\tau_1$, $\rho_{13} = \lambda_1\tau_3$, $\rho_2 = \lambda_2\tau_2$, $\rho_{23} = \lambda_2\tau_3$, $\rho_3 = (\lambda_1+\lambda_2)\tau_3$, $\rho_3 < 1$.

(iii) Two Internal Streams and One External Stream

The third flow configuration is shown in Figure 4-3c. The approximate average waiting time at v is given by

$$W(v) = W_1(v) + W_2(v) + W_3(v) \tag{4}$$

where

$$W_1(v) = \begin{cases} \frac{1}{2}\frac{\rho_{13}}{1-\rho_3}\tau_3 - \frac{1}{2}\frac{\rho_1}{1-\rho_1}\tau_1(1-\rho_{23}-\rho_{33}) & \tau_1 < \frac{\tau_3}{1-\rho_{23}-\rho_{33}} \\ 0 & \tau_1 \geq \frac{\tau_3}{1-\rho_{23}-\rho_{33}} \end{cases}$$

$$W_2(v) = \begin{cases} \frac{1}{2}\frac{\rho_{23}}{1-\rho_3}\tau_3 - \frac{1}{2}\frac{\rho_2}{1-\rho_2}\tau_2(1-\rho_{13}-\rho_{33}) & \tau_2 < \frac{\tau_3}{1-\rho_{13}-\rho_{33}} \\ 0 & \tau_2 \geq \frac{\tau_3}{1-\rho_{13}-\rho_{33}} \end{cases}$$

$$W_3(v) = \frac{1}{2}\frac{\rho_{33}}{1-\rho_3}\tau_3$$

and $\rho_1 = \lambda_1\tau_1$, $\rho_{13} = \lambda_1\tau_3$, $\rho_2 = \lambda_2\tau_2$, $\rho_{23} = \lambda_2\tau_3$, $\rho_{33} = \lambda_3\tau_3$, $\rho_3 = (\lambda_1+\lambda_2+\lambda_3)\tau_3$.

Note that if we make the Poisson assumption that the superimposed input stream at v is Poisson with rate equal to the sum of the rates of the individual input streams, then we have a M/D/1 queue at v for all three flow interference configurations. The average waiting time at v is, accordingly,

$$W'(v) = \frac{1}{2}\frac{\rho}{1-\rho}\tau \qquad (5)$$

where $\tau = \tau_3$ and $\rho = \rho_3$.

It is easy to verify that W'(v) is larger than W(v).

Now, we can apply the above formulas to calculate message delays in a network. First, consider an arbitrary edge (v_1,v_2) in a network. We want to find the approximate packet waiting time at v_1 while waiting to be transmitted over (v_1,v_2). The procedure is as follows:

STEP 1 : Construct a subnetwork $SN(v_1,v_2)$ consisting of edge (v_1,v_2) and all edges incident at v_1 which contribute packet flows on (v_1,v_2).

STEP 2 : For each vertex v in $SN(v_1,v_2)$, excluding v_1 and v_2, assign an external Poisson stream whose rate is equal to the rate of flow along (v,v_1) and continuing to v_2. The external flow into v_1 is included as well.

STEP 3 : Calculate the approximate average delay along (v_1,v_2) by the formulas given above.

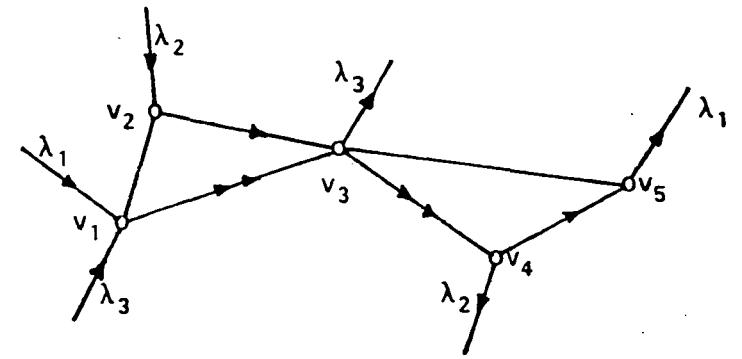
As an illustrative example, consider the network and flows as shown in Figure 4-4a. The reduced subnetworks for all edges which have packet flows are shown in Figures 4-4b to 4-4e. The approximate waiting time is then readily found by using the M/D/1 delay formula (Equation 5) in Figures 4-4b and 4-4c, approximate formula (Equation 3) in Figure 4-4d, and the tandem queueing formula (Equation 1) in Figure 4-4e.

After we have found the packet waiting time along edge b_i, $W(b_i)$, the average delay (waiting time + service time) caused by edge b_i is just

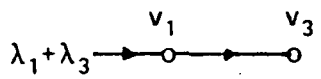
$$T(b_i) = W(b_i) + \tau_i$$

Finally, the end-to-end delay along a path can be found by summing over all delays on its component edges

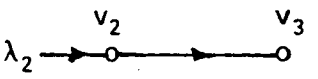
$$T = \sum_i T(b_i)$$



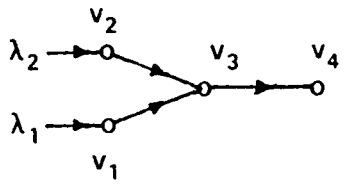
(a)



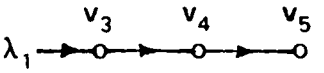
(b)



(c)



(d)



(e)

Figure 4-4: Example for Approximate Average Delay Calculation in a Network

5. CONCLUDING REMARKS AND EXAMPLE

In our survey, we found that existing network topology design algorithms are not directly applicable to the SSIS topology design problem. Most of them do not accommodate multiple design constraints, and thus some important performance measures of a communication network, such as reliability and delay, are seldom taken into consideration. In Section 3, we presented a new approach for network topology design by an efficient ordered enumeration. The technique gives the true cost-optimal topology which satisfies all given design constraints and requirements. In Section 4, we discussed some tools for studying two important aspects of network performance. In particular, we presented a new Event Network Model for studying the reliability of a communication network with dependent failures, and discussed some formulas which can be used to compute or estimate delays in a communication network. These tools can be incorporated in our design algorithm to obtain the topology that meets our needs.

The design algorithm presented in Section 3 has been implemented in Pascal, and a complete listing of the computer program can be found in the Appendix. In the program, the number of stations is fixed by a statement in the constant declaration section. Changes can be easily made by editing only one line of the program. The number of link elements is a variable in the program to make it more flexible for the user. A link element is allowed to connect more than two stations together. (For example, a radio transmitter may connect multiple nodes.) So this program can be used in more general design situations than those consisting only of point to point communication links. The program is written only to generate topologies that are simply connected. Tests on reliability, delay, and others are omitted in order to limit the size of the program.

To use the program, the user has to first sort his set of "feasible" link elements in increasing order of their costs. The program will ask for the total number of link

elements. Then, in increasing order of element costs, the program will ask for the following information for each link:

1. cost,
2. connectivity (the number of stations connected by the link element), and
3. the identities of the stations connected by the link element (stations are to be identified by integers 1, 2, 3, etc.).

We will now give a simple example. In Figure 5-1, the network consists of 5 stations and a total of 8 feasible link elements. All the link elements are shown in the figure together with their costs. For simplicity, each link element connects only two stations. After sorting, the link elements should appear in the following order:

LINK #	COST	CONNECTIVITY	STATIONS CONNECTED
1	10	2	1, 2
2	20	2	2, 3
3	30	2	3, 4
4	50	2	1, 3
5	60	2	2, 5
6	80	2	1, 4
7	90	2	3, 5
8	200	2	4, 5

After complete information has been entered, the program will give the first connected network which consists of link elements 1, 2, 3, and 5 with a total cost of 120. The next topology will be link elements 1, 2, 3, and 7 with a total cost of 150, and so on. Unless asked to stop, the program will generate all possible connected networks in increasing order of their total costs.

So we have presented a new approach for network topology design which can account for any given amount of design constraints and requirements. The technique gives the true cost-optimal topology, and is superior to other existing design algorithms which are usually only heuristics and take very few design factors into consideration.

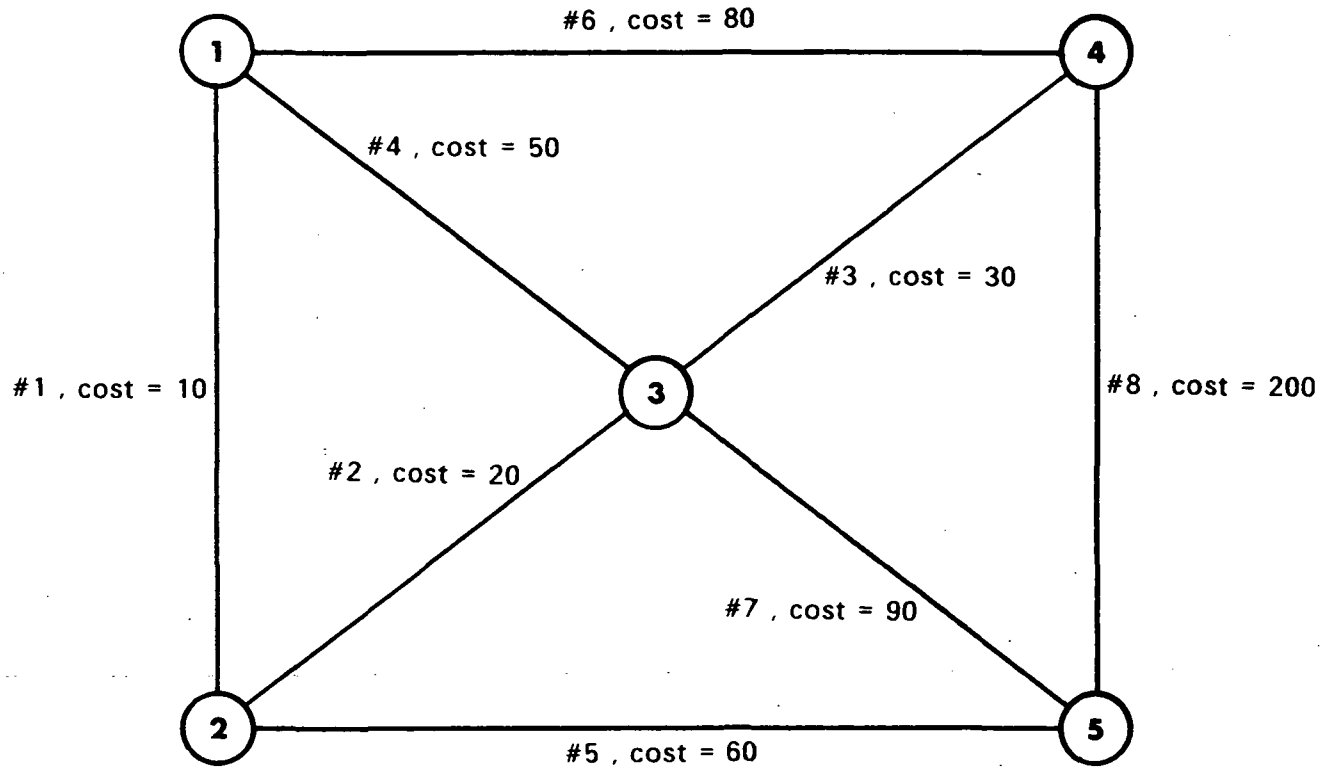


Figure 5-1: An Example of Network Design

The design algorithm might be time-consuming for a network with a large number of feasible link elements, but for the SSIS communication network which has only a moderate number of components due to the many technical feasibility constraints, this approach is applicable.

ACKNOWLEDGEMENT

The authors would like to thank Drs. Charles Wang and Tsun-Yee Yan of the Jet Propulsion Laboratory, California Institute of Technology, for helpful comments on an earlier draft of this report.

References

1. Bahl, L. R. and Tang, D. T. Optimization of Concentrator Locations in Teleprocessing Networks. Proc. Symp. on Computer-Communications Networks and Teletraffic, Brooklyn, New york: Polytechnic Press, April, 1972, pp. 355-362.

2. Ball, M. O. "Complexity of Network Reliability Computations." *Networks 10*, 2 (Summer 1980), 153-165.

3. Boorstyn, R.R., and Frank, H. "Large-Scale Network Topological Optimization." *IEEE Trans. on Commun. COM-25*, 1 (January 1977), 29-47.

4. Calo, S.B. "Message Delays in Repeated-Service Tandem Connections." *IEEE Trans. on Commun. COM-29*, 5 (May 1981), 670-678.

5. Chou, W. and Sapir, D. L. A Generalized Cut-Saturation Algorithm for Distributed Computer Communications Network Optimization. Proc. IEEE ICC, New York: IEEE, June, 1982, pp. 4C.2.1-4C.2.6.

6. Cooper, L. "Location-Allocation Problems." *Operations Research* (1962), 331-343.

7. Efroymson, M. A. and Ray, T. L. "A Branch-Bound Algorithm for Plant Location." *Operations Research* (May-June 1968), 361-368.

8. Feldman, E., Lehner, F. A., and Ray, T. L. "Warehouse Location Under Continuous Economies of Scale." *Management Science 12*, (May 1966), 670-684.

9. Frank, H., Frisch, I. T., and Chou, W. Topological Considerations in the Design of ARPA Network. Proc. 1970 Spring Joint Comput. Conf., Montvale, New Jersey: AFIPS Press, May, 1970, pp. 581-587.

10. Frank, H. and Chou, W. Topological Optimization of Computer Networks. Proc. IEEE, November, 1972, pp. 363-373.

11. Frank, H., Gerla, M., and Chou, W. Issues in the Design of Large Distributed Computer Communication Networks. Proc. IEEE NTC, New York: IEEE, November, 1973, pp. 37A1-37A8.

12. Fuchs, E. and Jackson, P. "Estimates of Distributions of Random Variables for Certain Computer Communications Traffic Models." *CACM 13*, 12 (December 1970), 752-757.

13. Gerla, M. The Design of Store-and-Forward (S/F) Networks for Computer Communications. Tech. Rept. UCLA-ENG-7319, School of Engineering and Applied Science, University of California, Los Angeles, California, 1973.

14. Gerla, M., Frank, H., Chou, W., and Eckl, J. A Cut-Saturation Algorithm for Topological Design of Packet Switched Communications Networks. Proc. IEEE NTC, New York: IEEE, December, 1974, pp. 1074-1085.

15. Horowitz, E. and Sahni, S.. *Fundamentals of Computer Algorithms*. Rockville: Computer Science Press, 1978.

16. Hwang, C.L., Tillman, F.A., and Lee, M.H. "System-Reliability Evaluation Techniques for Complex/Large Systems - A Review." *IEEE Trans. on Reliability R-30*, 5 (December 1981), 416-423.

17. *Space Station Information Systems (SSIS) FY'83 Study Report.* 1983. Tech. Rept. JPL D-1045, Jet Propulsion Laboratory, Pasadena, Calif. (internal document).

18. Kershenbaum, A. and Chou, W. "A Unified Algorithm for Designing Multidrop Teleprocessing Networks." *IEEE Trans. on Commun. COM-22*, 11 (November 1974), 1762-1772.

19. Kershenbaum, A. and Boorstyn, R. R. Centralized Teleprocessing Network Design. Proc. IEEE NTC, New York: IEEE, December, 1975, pp. 27.11-27.14.

20. Khumawala, B. M. "Warehouse Location Problems, Efficient Branch and Bound Algorithm." *Management Science 18*, (August 1972), B718-B731.

21. Kleinrock,L.. *Communication Nets: Stochastic Message Flow and Delay.* New York: McGraw-Hill, 1964.

22. Kruskal, J. B. "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem." *Proc. Amer. Math. Soc. 7*, 1 (February 1956), 48-50.

23. Kuehn, A. A. and Hamburger, M. J. "A Heuristic Program for Locating Warehouses." *Management Science 9*, (July 1963), 643-666.

24. Lam, Y. F. and Li, V.O.K. On Reliability Calculations of Network with Dependent Failures. Proc. IEEE GLOBECOM, New York: IEEE, December, 1983, pp. 1499-1503.

25. Lam, Y. F. and Li, V. O. K. Reliability Modeling and Analysis of Communication Networks with Dependent Failures. Proc. IEEE INFOCOM, New York: IEEE, 1985, pp. 196-199.

26. Li, V.O.K. End-to-End Delay in a Communication Network. Tech. Rept. Report CSI-83-02-02, University of Southern California, Communication Sciences Institute, February, 1983.

27. McGregor, P. and Shen, D. "Network Design: An Algorithm for the Access Facility Location Problem." *IEEE Trans. on Commun. COM-25*, 1 (January 1977), 61-73.

28. Pan, S.N. and Spragins, J. Dependent Failure Reliability Models for Tactical Communications Networks. Proc. IEEE ICC, New York: IEEE, 1983, pp. 765-771.

29. Prim, R. C. "Shortest Connection Networks and Some Generalizations." *Bell Syst. Tech. J. 36*, 6 (November 1957), 1389-1401.

30. Rubin, I. "Communication Networks: Message Path Delays." *IEEE Trans. on Information Theory IT-20*, 6 (Nov 1974), 738-745.

31. Rubin, I. "Message Path Delays in Packet-Switching Communication Networks." *IEEE Trans. on Commun. COM-23*, 2 (Feb 1975), 186-192.

32. Rubin, I. "An Approximate Time-Delay Analysis for Packet-Switching Communication Networks." *IEEE Trans. on Commun. COM-24*, 2 (Feb 1976), 210-222.

33. Spragins, J. and Assiri, J. Communication Network Reliability Calculations with Dependent Failures. Proc. IEEE NTC, New York: IEEE, 1980, pp. 25.2.1-25.2.5.

34. Tanenbaum, A. S.. *Computer Networks.* New York: Prentice-Hall, 1981.

35. Woo, L. S. and Tang, D. T. Optimization of Teleprocessing Networks with Concentrators. Proc. IEEE NTC, New York: IEEE, November, 1973, pp. 37C1-37C5.

6. APPENDIX

```
(*****************************************************************************)
(*                                                                           *)
(*            OPTIMAL NETWORK TOPOLOGY DESIGN  (BASIC VERSION)               *)
(*                              December  1984                               *)
(*                                                                           *)
(* This program accepts a set of communication link elements (together with *)
(* the stations they connect), and outputs candidate topologies (those that *)
(* connect all the stations) in increasing order of their total costs.       *)
(*                                                                           *)
(*****************************************************************************)


program JPL84 (input, output, fin, fout);

label 99;

const

  N = 5;  (* This is the total number of stations in the network. *)

type

  stations = array[1..N] of integer;

  vector   = array[0..N] of integer;

  station_link = ^station;

  station =
    record
      index   : integer;
      next    : station_link
    end;

  element_link = ^element;

  element =
    record
      index        : integer;
      weight       : real;
      connectivity : integer;
      stations     : station_link;
      next         : element_link
    end;

  list_link = ^list_data;

  list_data =
    record
      content_source : list_link;
```

```
      size           : integer;
      weight         : real;
      connection     : stations;
      next           : list_link;
      end_element    : element_link
    end;

  next_list_link = ^next_list_data;

  next_list_data =
    record
      size           : integer;
      weight         : real;
      connection     : stations;
      new_element    : element_link;
      source         : list_link;
      content_source : list_link
    end;

  node_link = ^tree_node;

  tree_node =
    record
      parent         : node_link;
      left_child     : node_link;
      right_child    : node_link;
      left_neighbor  : node_link;
      right_neighbor : node_link;
      content        : next_list_link
    end;


var
  fin, fout                                                    : text;
  answer, answer2, answer3, connected                          : char;
  size, index, count, count1                                   : integer;
  v                                                            : vector;
  se, sce                                                      : station_link;
  element_base, e, ce                                          : element_link;
  list_base, print_base, list_end, list_buffer, pp, pq, op     : list_link;
  next_list_buffer, np                                         : next_list_link;
  node_base, tree_end, parent, node_buffer, p                  : node_link;


procedure INSTRUCTIONS;
  begin
    writeln(' ');
    writeln('The number of stations has been fixed to be', N:3, '.');
    writeln('This number is determined by the value N in the const');
    writeln('statement at the beginning of this program, and can be');
    writeln('changed by editing that part of the program alone.');
    writeln(' ');
    writeln('User has to enter the total number of link elements,');
    writeln('then enter their weights in INCREASING order, together');
    writeln('with the stations that they connect.  A link element is');
    writeln('allowed to connect more than two stations.');
    writeln(' ')
  end;
```

```
(*********************************************************)
(* This procedure accepts input data from the terminal. *)
(*********************************************************)
procedure INPUT_DATA_TERMINAL;
  begin
    writeln(' ');
    write('How many link elements do you have ? (must be greater than 1)  ');
    readln(size);
    writeln(' ');
    new(element_base);
    element_base^.index := 0;
    element_base^.weight := 0;
    ce := element_base;
    for index := 1 to size do
      begin
        new(e);
        e^.index := index;
        writeln(' ');
        write('Enter weight of element #', index:3, '     ');
        readln(e^.weight);
        write('How many stations are connected together by this element ? ');
        readln(e^.connectivity);
        new(e^.stations);
        write('Enter identity of station #  1 : ');
        readln(e^.stations^.index);
        e^.stations^.next := nil;
        sce := e^.stations;
        for count := 2 to e^.connectivity do
          begin
            new(se);
            write('Enter identity of station #', count:3, ' : ');
            readln(se^.index);
            se^.next := nil;
            sce^.next := se;
            sce := se
          end;
        e^.next := nil;
        ce^.next := e;
        ce := e
      end
  end;


procedure INITIALIZE;
  begin

    new(list_base);
    list_base^.content_source        := nil;
    list_base^.size                  := 0;
    list_base^.weight                := 0;
    list_base^.next                  := nil;
    list_base^.end_element           := element_base;
    for count := 1 to N do
      list_base^.connection[count]   := 0;

    new(next_list_buffer);
    next_list_buffer^.size           := 1;
    next_list_buffer^.weight         := element_base^.next^.weight;
    next_list_buffer^.new_element    := element_base^.next;
    next_list_buffer^.source         := list_base;
```

```
    next_list_buffer^.content_source := nil;
    for count := 1 to N do
      next_list_buffer^.connection[count] := 0;
    se := next_list_buffer^.new_element^.stations;
    for count := 1 to next_list_buffer^.new_element^.connectivity do
      begin
        next_list_buffer^.connection[se^.index] := 1;
        se := se^.next
      end;

    new(node_base);
    node_base^.parent         := nil;
    node_base^.left_child     := nil;
    node_base^.right_child    := nil;
    node_base^.left_neighbor  := nil;
    node_base^.right_neighbor := nil;
    node_base^.content        := next_list_buffer;

    list_end := list_base;
    parent   := node_base;
    tree_end := node_base
  end;


(*******************************************************)
(* This procedure adds the least-weight subset in the *)
(* tree to the end of the list of ordered subsets.    *)
(* The least-weight subset in the tree is always at   *)
(* the root of the tree.                              *)
(*******************************************************)
procedure ADD_TO_LIST;
  begin
    new(list_buffer);
    list_buffer^.content_source := node_base^.content^.content_source;
    list_buffer^.size           := node_base^.content^.size;
    list_buffer^.weight         := node_base^.content^.weight;
    for count := 1 to N do
      list_buffer^.connection[count] := node_base^.content^.connection[count];
    list_buffer^.next           := nil;
    list_buffer^.end_element    := node_base^.content^.new_element;
    list_end^.next              := list_buffer;
    list_end                    := list_buffer
  end;


(*******************************************************)
(* This procedure updates the connectivity information *)
(* of a newly formed subset of link elements.          *)
(*******************************************************)
procedure CONNECTIVITY_UPDATE (cv:vector; Var ccon:stations);
  var ci, cj, ch, cs, flag : integer;
  begin
    flag := 0;
    ch := 1;
    for ci := 1 to N do
      if ccon[ci] > ch then ch := ccon[ci];
    ch := ch + 1;
    for ci := 1 to cv[0] do
      begin
        if ccon[cv[ci]] = 0 then ccon[cv[ci]] := ch;
```

```
        if ccon[cv[ci]] > 1 then
          begin
            cs := ccon[cv[ci]];
            for cj := 1 to N do
              if ccon[cj] = cs then ccon[cj] := ch
          end;
        if ccon[cv[ci]] = 1 then flag := 1
      end;
    if flag = 1 then
      for ci := 1 to N do
        if ccon[ci] = ch then ccon[ci] := 1
  end;


(*****************************************************************)
(* When the subset which has just been added to the ordered      *)
(* list is not the last child of its parent, this procedure      *)
(* makes its parent's next best child the new root of the tree. *)
(*****************************************************************)
procedure ADD_TO_TREE_FROM_OLD_SOURCE;
  begin
    new(next_list_buffer);
    next_list_buffer^.size   := node_base^.content^.source^.size;
    next_list_buffer^.weight := node_base^.content^.weight -
                                node_base^.content^.new_element^.weight +
                                node_base^.content^.new_element^.next^.weight;
    next_list_buffer^.new_element    := node_base^.content^.new_element^.next;
    next_list_buffer^.source := node_base^.content^.source;
    for count := 1 to N do
      next_list_buffer^.connection[count] :=
                                  next_list_buffer^.source^.connection[count];
    se := next_list_buffer^.new_element^.stations;
    v[0] := next_list_buffer^.new_element^.connectivity;
    for count := 1 to next_list_buffer^.new_element^.connectivity do
      begin
        v[count] := se^.index;
        se := se^.next
      end;
    CONNECTIVITY_UPDATE(v, next_list_buffer^.connection);
    next_list_buffer^.content_source := node_base^.content^.content_source;
    node_base^.content       := next_list_buffer
  end;


(*****************************************************************)
(* When the subset which has just been added to the ordered      *)
(* list is the last child of its parent, this procedure          *)
(* removes that subset from the tree and moves the subset at     *)
(* the end of the tree up to become the new root of the tree.    *)
(*****************************************************************)
procedure RESTORE_ROOT;
  begin
  if node_base <> tree_end then
    begin
      node_base^.content := tree_end^.content;
      if tree_end^.parent^.right_child = nil
        then
          tree_end^.parent^.left_child := nil
        else
          begin
```

```
            tree_end^.parent^.right_child := nil;
            parent := tree_end^.parent
          end;
        tree_end := tree_end^.left_neighbor;
        tree_end^.right_neighbor := nil
      end
    end;


(*******************************************)
(* Since a new root has just been made, this *)
(* procedure reorders the tree from its root *)
(* to preserve its ordered structure.        *)
(*******************************************)
procedure REORDER_TREE_FROM_ROOT;
  label 99;
  begin
  if node_base <> tree_end then
    begin
      p := node_base;
      while p^.right_child <> nil do
        begin
          if ((p^.content^.weight > p^.left_child^.content^.weight) or
              (p^.content^.weight > p^.right_child^.content^.weight))
            then
              begin
                if  p^.left_child^.content^.weight >
                    p^.right_child^.content^.weight
                  then
                    begin
                      np := p^.content;
                      p^.content := p^.right_child^.content;
                      p^.right_child^.content := np;
                      p := p^.right_child
                    end
                  else
                    begin
                      np := p^.content;
                      p^.content := p^.left_child^.content;
                      p^.left_child^.content := np;
                      p := p^.left_child
                    end
              end
            else goto 99
        end;
      if p^.left_child = nil
        then goto 99
        else
          begin
            if p^.content^.weight > p^.left_child^.content^.weight
              then
                begin
                  np := p^.content;
                  p^.content := p^.left_child^.content;
                  p^.left_child^.content := np
                end
              else goto 99
          end;
      99 : np := nil
    end
```

```
  end;


(*********************************************************)
(* The subset which has just been added to the ordered *)
(* list now becomes a parent.  This procedure adds its *)
(* best child to the end of the tree, and reorders the *)
(* tree from its end to preserve its ordered structure.*)
(*********************************************************)
procedure ADD_NEW_TO_TREE_AND_REORDER;
  label 999;
  begin
    new(next_list_buffer);
    next_list_buffer^.size           := list_end^.size + 1;
    next_list_buffer^.weight         := list_end^.weight +
                                        list_end^.end_element^.next^.weight;
    next_list_buffer^.new_element    := list_end^.end_element^.next;
    next_list_buffer^.source         := list_end;
    next_list_buffer^.content_source := list_end;

    for count := 1 to N do
      next_list_buffer^.connection[count] :=
                                  next_list_buffer^.source^.connection[count];
    se := next_list_buffer^.new_element^.stations;
    v[0] := next_list_buffer^.new_element^.connectivity;
    for count := 1 to next_list_buffer^.new_element^.connectivity do
      begin
        v[count] := se^.index;
        se := se^.next
      end;
    CONNECTIVITY_UPDATE(v, next_list_buffer^.connection);

    new(node_buffer);
    node_buffer^.parent          := parent;
    node_buffer^.left_child      := nil;
    node_buffer^.right_child     := nil;
    node_buffer^.left_neighbor   := tree_end;
    node_buffer^.right_neighbor  := nil;
    node_buffer^.content         := next_list_buffer;

    if parent^.left_child = nil
      then parent^.left_child := node_buffer
      else
        begin
          parent^.right_child := node_buffer;
          parent := parent^.right_neighbor
        end;
    tree_end^.right_neighbor := node_buffer;
    tree_end := node_buffer;

    p := tree_end;
    while p^.parent <> nil do
      begin
        if p^.content^.weight < p^.parent^.content^.weight
          then
            begin
              np := p^.content;
              p^.content := p^.parent^.content;
              p^.parent^.content := np;
              p := p^.parent
```

```
          end
        else goto 999
    end;
  999 : np := nil
end;


(************************************)
(* This procedure outputs a feasible *)
(* topology to the terminal.         *)
(************************************)
procedure OUTPUT_SINGLE;
  begin
    writeln(' ');
    writeln('SUBSET   # ', count1:5);
    writeln('WEIGHT   =  ', list_end^.weight);
    write('ELEMENTS :');
    print_base := nil;
    pq := list_end;
    while pq <> nil do
      begin
        new(op);
        op^.end_element := pq^.end_element;
        op^.next := print_base;
        print_base := op;
        pq := pq^.content_source
      end;
    op := print_base;
    while op <> nil do
      begin
        write(op^.end_element^.index:4);
        op := op^.next
      end;
    writeln(' ')
  end;


begin     (* MAIN PROGRAM *)
  writeln(' ');
  writeln(' ');

  repeat
    write('Do you need instructions (Y/N) ? ');
    readln(answer)
  until ((answer = 'Y') or (answer = 'y')) or
        ((answer = 'N') or (answer = 'n'));
  if (answer = 'Y') or (answer = 'y') then
    begin
      INSTRUCTIONS;
      repeat
        write('Are you ready (Y/N) ? ');
        readln(answer)
      until (answer = 'Y') or (answer = 'y')
    end;

  writeln(' ');
    INPUT_DATA_TERMINAL;


  INITIALIZE;
```

```
  count1 := 1;
  while  (count1 < (2**size)) do
    begin
      ADD_TO_LIST;
      (**********************************************)
      (* A new subset has just been generated and added to *)
      (* the ordered list.  The following few lines test   *)
      (* to see if this subset forms a connected network.  *)
      (* Additional tests can be inserted at this point.   *)
      (**********************************************)
      connected := 'Y';
      for index := 1 to N do
        if list_end^.connection[index] <> 1
          then connected := 'N';
      if connected = 'Y' then
        begin
          OUTPUT_SINGLE;
          repeat
            write('Do you want to continue (Y/N) ? ');
            readln(answer3)
          until ((answer3 = 'Y') or (answer3 = 'y')) or
                ((answer3 = 'N') or (answer3 = 'n'));
          if (answer3 = 'N') or (answer3 = 'n') then goto 99
        end;

      count1 := count1 + 1;
      if node_base^.content^.new_element^.next <> nil
        then ADD_TO_TREE_FROM_OLD_SOURCE
        else RESTORE_ROOT;
      REORDER_TREE_FROM_ROOT;
      if list_end^.end_element^.next <> nil
        then ADD_NEW_TO_TREE_AND_REORDER
    end;

  99:
  writeln(' ');
  writeln('END OF EXECUTION.');
  writeln(' ')
end.
```