Edge Disjoint Spanning Trees and Failure Recovery in Data Communication Networks

by

James Anthony Roskind

S.B., S.M. Massachusetts Institute of Technology (1980)

Submitted to the Department of
Electrical Engineering and Computer Science
in Partial Fulfillment of the
Requirements of the
Degree of

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1983

c James A. Roskind 1983

The author hereby grants to M.I.T. permission to reproduce and to distribute copies of this thesis document in whole or in part.

Signature of Author____________________________________
Department of Electrical Engineering and Computer Science September 9, 1983

Certified by____________________________________
Professor Robert G. Gallager
Department of Electrical Engineering and Computer Science

Approved by____________________________________
Chairman Arthur C. Smith
Department of Electrical Engineering and Computer Science

Edge Disjoint Spanning Trees and

Failure Recovery in Data

Communication Networks

by

JAMES ANTHONY ROSKIND

Submitted to the Department of Electrical Engineering and Computer Science on September 12, 1983 in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in Electrical Engineering and Computer Science

ABSTRACT

In this thesis we discuss a variety of ways in which information about topological changes (failures and restorations of links of a network) can be disseminated to the nodes of a packet switching network. Nodes need this information as soon as possible, if they are to avoid wasting the communications resources of the network by misrouting of packets. At the time of a failure we therefore view communications resources to be at a premium. We demonstrate how the use of precomputed (before the failures) structures can aid in providing swift and communications efficient methods of achieving the necessary notifications.

One example of such a precomputed structure is a set of k edge disjoint spanning tree. One major result of this thesis is a new and computationally efficient method for finding such spanning trees (O(knkn) time to find k edge-disjoint spanning trees in a network with n nodes). We also describe directions which might reduce this time complexity further.

Inherent in any data communication network are variable communications delays and failures of the communication channels. These two factors combine to make the task of having all nodes route packets in a consistent way impossible in a rapidly changing network, as no node can know what is occurring in a distant part of the network. When the rate of change in the network topology (re: failures and restorals) is slow enough, the task of synchronizing the nodes is complex but achievable. The second major result that we provide is a way in which precomputed edge disjoint trees can be used by a distributed algorithm to effectively synchronize the nodes so that they use the same topology as a basis for routing.

Thesis Supervisor: Dr. Robert G. Gallager

Title: Professor of Electrical Engineering and Computer Science

Table of Contents

Chapter 1- INTRODUCTION 6

EXISTING NETWORKS 10

PERFORMANCE OF RECOVERY ALGORITHMS 12

DETOURS 13

SPANNING TREES 16

Chapter 2- DIRECTED NETWORKS 20

RELATING DIRECTED NETWORKS TO UNDIRECTED NETWORKS 31

UNDIRECTED NETWORKS 34

Chapter 3- FINDING MULTIPLE EDGE DISJOINT SPANNING TREES IN UNDIRECTED NETWORKS 38

PERTURBING EXISTING TREES 38

AUGMENTING EXISTING SUBTREES 46

WHY IT WORKS 50

IMPLEMENTING THE INDEPENDENCE TEST 58

SET UNION FUNCTIONS 64

ARE TWO NODES IN THE SAME TREE 65

FINDING A PATH IN A FOREST 67

PRECOMPUTATION OF THE DIRECTION TO THE ROOT 71

COMPUTATIONAL COMPLEXITY 76

THE "CLUMP" STRUCTURE 80

COMPUTATIONAL COMPLEXITY - USING CLUMPS 86

CONSTRAINED MINIMAL AUGMENTATION SEQUENCES 86

REMOVING WASTE 89

MODULO K FORESTS	89
CORRECTNESS PROOF	91
COMPUTATIONAL COMPLEXITY - MODULO K FORESTS	93
PREVIOUS RESULTS	93
FUTURE IMPROVEMENTS	94
Chapter 4- NETWORK SYNCHRONIZATION VIA EDGE DISJOINT SPANNING TREES	95
RELIABILITY OF K EDGE-DISJOINT SPANNING TREES	95
NETWORK MODEL	97
DISSEMINATING EDGE FAILURE INFORMATION	98
"SURE FIRE" RECOVERY METHOD	100
CORRECTNESS PROOF OF THE SF ALGORITHM	121
MAKING USE OF THE SPANNING TREES	138
CORRECTNESS PROOF OF THE kSTRA	173
ACTUAL IMPLEMENTATION OF THE kSTRA	190
RECOVERY	194
Appendix A- MORE EDGE DISJOINT SPANNING TREES	196
SMALL TREES	196
PLACING kn/2 LINKS IN O(m+kn) TIME	199
WISHFUL EXTENSIONS	203
FINDING LINKS THAT TOUCH SMALL TREES	204
TIGHTER BOUNDS ON AUGMENTATION SEQUENCE LENGTHS	210
MAINTAINING INTERSECTION DATA STRUCTURES	214
INTERSECTION DATA STRUCTURE IN THE MULTIPLE TREE ALGORITHM	218

Appendix B- FIND-UNION ALGORITHMS 221

O(1) FIND, O(n) UNION 224

MAINTAINING SIZES OF EQUIVALENCE CLASSES 226

O(n) FIND, O(1) UNION 230

O(1) FIND, O(log n) UNION 234

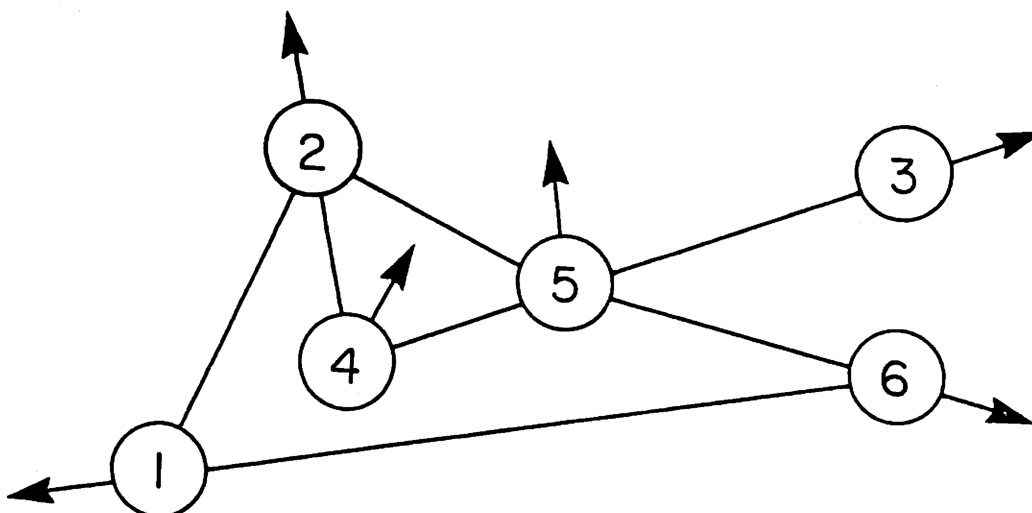
References 239

Chapter 1 - INTRODUCTION

WHAT IS A DATA COMMUNICATION NETWORK?

A data communication network (network for short) consists of a set of computers (nodes) and a set of communication channels (links). The links we will consider connect distinct pairs of nodes. The functional goal of a network is to allow data to enter the network along with a destination address (a specified node), via an external channel, and have the the same data transmitted out of the network (at the specified node) via the specified node's external channel.

We can represent a network by drawing a graph to show its topology (i.e. what is connected to what). We offer as an example:



The numbered circles represent the 6 functional nodes of the network. The lines between circles indicate the presence of a functional link between two nodes. The arrows leading away from the circles correspond to the external channels which carry data into and out of the network. In the future we will not bother drawing these external channels, their presence is assumed, and they have no real significance to the thesis.

Within the context of this thesis we will only be considering packet switching networks. In such networks the data passed along by the network is assumed to enter and exit the network as discrete bundles, which are called packets. It is then the "job" of each node, to decide where each packet it receives should be sent next. The nodes performance of this job is referred to as routing.

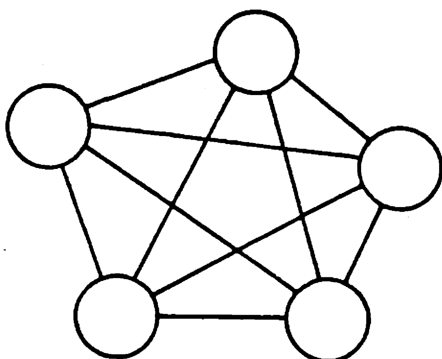
Within a network, there are two types of failures that may be considered: link failures and node failures. In general, packets of data are transmitted between nodes by one transmitter-receiver pair, and acknowledged by a second transmitter-receiver pair in the reverse direction. For an edge to continue to consistently and accurately convey data between nodes, both such transmitter-receiver pairs must function. A link is considered to have failed if either of the directed links (transmitter-receiver pairs) that make up that link has failed to function. A node failure has occurred when a node is no longer capable of processing packets that it receives along

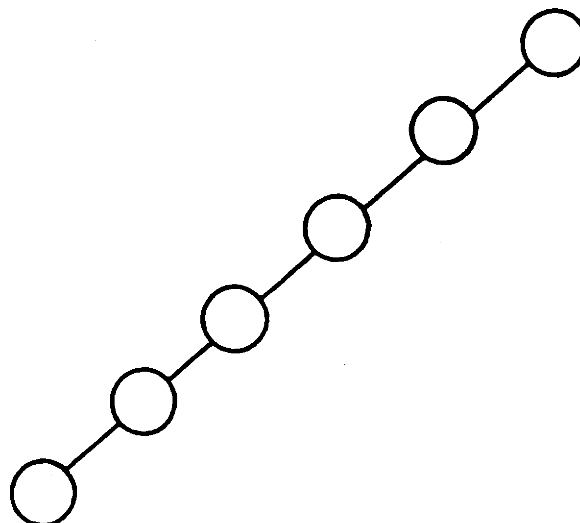
any of its links (ie: redirecting them). Note that a node failure would be perceived by each of the failed node's neighbors as an edge failure, but packets could be lost in the process.

The topology of the network is defined by the links and nodes in the network that are functional. After any failure has occurred, the topology has changed, and hence routing decisions made at nodes must be modified. For example, once an edge has failed, neither of the adjacent nodes should attempt to transmit data across it, and they must in fact find alternate routes for their packets (assuming the destination is still connected to the rest of the network!). It is the consistent modification of routing policies at all nodes in the network that is termed recovery.

We have defined what we mean by data communication networks, but we have actually grouped together a wide variety of quite distinctive "types" of networks. The distinction is apparent in that some networks find certain algorithms useful, while other networks find the same algorithm to be almost useless. For example, some networks have a topology of links that allows for direct communication (ie: one link) between every pair of nodes (we would describe such a network as having diameter 1, 1 being the greatest number of links necessary to communicate between any two nodes). Other networks are comprised of a chain, where there are two end nodes that have

exactly 1 link, and some in between nodes that have exactly 2 links. The diagram below illustrates the two types of networks.





Diameter 1

Diameter 5

Note that the second network shown has 6 nodes, and is labeled "diameter 5" because the furthest apart pair of nodes (the end nodes) are 5 hops apart. Notice then that algorithms that require notifying all nodes of an event (eg: a failure) that is observed by a single node, would proceed very slowly in a large chain like network, and very swiftly in a network of diameter 1.

The example just given shows how the topology of a network might effect the performance of an algorithm. There are several other details about networks that impact similarly. These include node processing speed, node storage capacity and link delay. These factors interact heavily with the topology to

determine the performance of algorithms in a given network.

EXISTING NETWORKS

As examples of existing link failure recovery algorithms, we will look briefly at and contrast some existing architectures. The first example is incorporated in the ARPA net [20]. The routing in the ARPA net is dynamic. The routing is varied to attempt to take advantage of low delay links and to attempt to avoid links with large waiting queues. Unfortunately, massive rerouting can quickly congest low delay paths and cause dramatic oscillations in the choice of minimum delay paths. This phenomenon was examined by Bertsekas [3], along with methods of damping (slowing or lessening) the response of the system. These methods include asynchronous updating, fading memory and biasing of routing toward the minimum hop route. The current ARPA routing algorithm includes all of these elements to maintain stability.

The ARPA algorithm, hinted at above, automatically deals with link failures. The failure appears as a bottleneck with a large queue of packets waiting to traverse the "down" link. The routing algorithm shifts flow around such an edge and adapts to the effective change in link capacity (a "down" link has capacity 0). As mentioned in the last paragraph this change must be slow to prevent instability.

A second example is the network architecture proposed by IBM. In this architecture there are a list of fixed routes that are allowed for each source-destination pair. This list of routes is static, and decided on when the network is created. When an edge failure occurs, all nodes are are notified, and a source node deletes from its list any routes that traverse the the failed edge. This process would proceed quite quickly as all that is necessary is notification of all nodes of the failure.

It is worth noting that the two examples just given lie at opposite ends of a scale in several respects. The static prearranged routing scheme recovers quickly, whereas the dynamic scheme must by its nature adapt slowly. The static scheme has very limited adaptability to flow variations whereas the dynamic scheme is constantly attempting to track the (in some sense) optimal routing policy. The static scheme makes little use of run time computation and relies heavily on the long ago precomputed and prearranged routes, while the dynamic scheme relies entirely on run time computation.

Given the two extreme strategies just discussed, it is interesting to explore strategies along the continuum between them. Such strategies would employ modest precomputation of

what to do in the event of a failure, and result in a relatively swift recovery after the fact. This is to be the direction the thesis will take, and the question is then what, if any, strategies are "workable", and what are their computational complexity.

PERFORMANCE OF RECOVERY ALGORITHMS

We discussed earlier how the specifics of the construction of a network (topology, edge delays etc.) can impact upon an algorithm's performance. In some networks, the nature of the data packets may imply performance constraints that will impact on the usefulness of various recovery algorithms.

When packetized speech is being transmitted, minimal delay and delay variance is critical. Speech reconstruction is a real-time operation that requires the next packet to arrive by a prescribed time, or else it is useless. Packets that are delayed due to a failure, and then arrive late are of no use.

When interactive communications are carried out between a user and a time sharing system, the mean delay is very significant and variance can be tolerated. In this situation packets that are delayed by a failure are very welcome upon

arrival, but only if the order of packets can be maintained.

In a network where large files are transmitted between nodes, the throughput is a key performance criteria. Delay and variance of delay are of little significance. When recovering from failures there are two possible scenarios. The first is that "any failure is very rare." Hence a complete restart of transmission of the files would not effect the long term throughput. Therefore any recovery algorithm that "recovered" would be fine. The second scenario is "failures are common." This would make it useful or necessary to recover without having to restart transmission of the entire file.

The above list of examples shows clearly why a variety of "recovery" algorithms are useful. This is the motivation, to a great extent, of this work.

DETOURS

As a first approach we will consider how traffic on the roadway deals with failures (ie: bridge out, road being paved, etc.). Presumably, there is some intelligence behind the wheel of every car to establish basic routes (head in the right direction, use high capacity edges such as superhighways). When a failure occurs (a semi jackknifes and blocks the road) a

secondary routing policy takes place as police decide on a detour. The decisions made by the police would override the basic routing that the driver had decided on.

There are several facts worth noting about such detours. The first point is that many detours can be precomputed. It is possible that state police have decided in advance upon detours around every stretch of state highway. With such detours already worked out, there is very little computation necessary when an accident occurs. Little computation implies very swift implementation when necessary.

A second point to note about the use of detours is inherent in the word detours. Implicit is a method of circumventing the difficulty when the driver gets close enough to begin the detour. The point here is that information about such difficulties is broadcasted just beyond the perimeter of the congested area. (this way drivers can avoid the congested area, rather than being forced to backtrack out of the congested area, and hence increase the congestion). A major point is that the police who enact the detour don't have to travel very far.

Leaving aside now the analogous situation of traffic on

the roadways, we will look briefly at the pros and cons of using detours in networks. In the area of communication networks, the use of detours would involve precomputation of paths around edges (or groups of edges) that are likely to fail. Specifically, contingency plans might be established for the failure of any single edge in the network. The contingency plans for the failure of any one edge would be worked out by all the nodes close enough to be effected by such a failure. Hence there would be no more than O(m) (there are m edges) detours to be computed, and not all of these would have to be computed by each node.

As mentioned earlier, detours lend themselves to precomputation. This tends to allow much of the failure recovery computation to be done when the network is not heavily loaded. The second point about detours is that when a failure occurs, only neighboring nodes that participate in the detour need to be notified of the failure. At first glance this appears to be a positive point. Local notification implies very little communication and hence very little use of network resources. However, should a second failure occur near the first failure, the difficulties can become great. In the worst case, the detour around the first failure might use the edge that failed second, and vice-versa.

Notice that the above second failure scenario is only noteworthy if there is a significant probability of multiple failures, and also there is no "centralized intelligence" to catch such a scenario. It should also be realized that installing a "centralized intelligence" would not solve all such problems, but rather forestalls them. A "centralized intelligence" must be notified of network activity via communication edges, and must disseminate orders via similar edges. The problem of network edge failures would end, just as the problem of "centralized intelligence" edge failure would begin.

Within the context of this thesis we decided that we would look for distributed algorithms (ie: no centralized controlling intelligence). We also decided that recovery from multiple failures would be considered. We were unable to find any nice extensions of detours as such under these conditions , and we moved on to other attacks.

SPANNING TREES

We just discussed attempts at generating a "detour" oriented recovery algorithm. The difficulty in such a development centered on the significant possibility of multiple failures, and the fact that nodes would only be notified of a failure if they were close to the resulting congested area.

Hence, when the probability of multiple failures is significant, the only reasonable approach we can see involves global notification of failures (ie.:tell all the nodes about every failure).

With the thought in mind that global notification of an event must take place, we note that the most efficient method, in terms of communications, is to broadcast the occurrence of such an event on a spanning tree of the network. A "spanning tree" of a network with n nodes is most simply defined as any set of n-1 edges in the network that do not include any cycles. There are many equivalent definitions of spanning trees, and they involve the following facts:

1) For every pair of nodes in the network, there is exactly one path between these nodes formed by edges of any specific spanning tree.

2) Every spanning tree is a maximal set of edges that is cycle free (i.e. if any edge is added to the set, a cycle is formed).

3) Every spanning tree is a minimal set of edges that form a connected subnetwork (i.e.: remove any edge from a spanning tree, and the resulting set of edges (with the original nodes) form a disconnected network).

The reason why global notification of events is best accomplished (minimal communication) by broadcasting on a spanning tree is: The broadcast is complete after n-1 transmissions. There are in fact n-1 notifications to be made, so n-1 transmissions is optimal.

A last point to mention about spanning trees is that they are very useful in dynamically establishing a fixed routing. Specifically the work of Friedman [11] seemed to indicate that a broadcast of the topology and edge utilizations of the network by way of a spanning tree is close to optimal (minimizing communication) for the computation of routing tables throughout a network. The use of such a routing algorithm would further allow us to assume that all nodes are aware of the pre-failure topology of the network (i.e.: they used the knowledge to compute the current routing). With this assumption we see that a broadcast of the fact that a given edge has failed is indeed sufficient for recovery (we have already indicated that the broadcast appears to be necessary to cope with multiple failures).

Given then that the existence of a known tree in the post failure network would make a recovery easier, the question is: How can we assure the existence of such a tree? One solution is to establish in advance several trees in the

pre-failure network. To guarantee the existence of at least one of these trees in the post-single-failure network, it is sufficient that the pre-failure trees be edge disjoint. (For now we will be considering edge failures only).

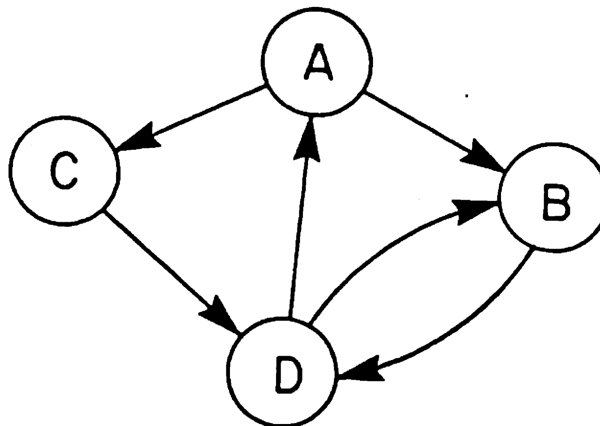
It is interesting to note that one tree in the original network is sufficient to allow efficient notification of the entire network that one failure occurred. Specifically, if a failed edge was NOT part of the lone tree, then broadcasting on the tree is straightforward. If the edge that fails is a part of the tree, then the tree is divided into two subtrees. The union of the nodes in the two subtrees is indeed the totality of the network. Making use of this fact, notification can be broadcast across these subtrees by starting at both ends of the failed edge (the failure is directly visible to both of the adjacent nodes). This result does not extend to multiple failures without the presence of additional (disjoint) trees, and so we will continue in that direction.

The many questions to be examined at this point include: How hard is it to find edge disjoint trees? What topological constraints on the network are necessary for the existence of such trees? How can this process be extended to handle large numbers of failures? Some of these questions will now be addressed.

Chapter 2

DIRECTED NETWORKS

Before looking for edge disjoint trees in a network, we will look at the easier problem of finding edge disjoint rooted spanning trees in a directed network. A directed network consists of a set of nodes, and a set of links. Each link connects two nodes in a fixed order. For example, an edge that goes from node A to node B is a completely different entity from an edge that might go from node B to node A. An example of a directed network is given below.



There are 4 nodes and 6 links in this example. There is an edge from node A to node B, but none in the other direction.

A rooted spanning tree (branching) in a directed network is formed by a subset of the links of the network that satisfy the following:

1) There is a distinguished node called the root.

2) No outgoing link from the root is in the branching.

3) Except for the root, every node has exactly one outgoing link in the branching.

4) There is a directed path in the branching from each node to the root.

There are many other equivalent defining characteristics and properties. These include (for instance) the facts that the branching has no cycles, that there is a unique path from every node to the root, and that ignoring directions on the links yields an undirected spanning tree.

We will now look at the problem of finding branchings in directed networks. First we will consider whether or not a single branching exists in a given directed network with a given root node. A necessary condition for the existence of such a tree is:

For every subset S of the nodes in the network that includes the root, there is at least one link going from S' to S (S' is the complement of S within the set

of nodes in the network).

The necessity of this condition follows from the fact that there is a directed path from every node to the root in a branching.

It turns out that the above condition is also sufficient to guarantee the existence of a branching in a network. The proof of this follows from an algorithm that finds a branching, and has the above condition as the only requirement for its completion. The algorithm builds a progressively larger subtree until it encompasses the entire network and hence is a branching. The algorithm is:

a) Start with the root only as the subtree

b) Let S be the set of nodes already in the subtree (from step a, the root is in S). If S'={}, then we're done.

c) Find an edge going from a node in S' to a node in S. Add that link to the subtree along with the node that it is outgoing from. Repeat steps b and c.

It can be seen that this results in a branching as desired.

Now consider the existence of several directed spanning trees which have a common root, and are such that no link is used by more than one tree. The necessary and sufficient conditions are exactly analogous to the single branching problem, but the algorithms for finding them [6,17,21,22] are far more complicated. Specifically, the conditions for the existence of k mutually edge disjoint branchings with a single given root are: For every set S of nodes in the network that includes the root node, there are at least k links going from node(s) in S' to node(s) in S.

Again the necessity follows directly, but the sufficiency is far from obvious. The necessity of the above conditions follows from the fact that there are k distinct paths (one per branching) from any node (includes those in any S') to the root.

The sufficiency again is based upon an algorithm. For the general case of k branchings the reader is referred to [6,17,21,22]. As a less complex example, we will present the algorithm for the case k=2 (ie:find 2 edge disjoint branchings with a fixed root) in order to show the reader one approach to this problem.

First we will give an outline of the algorithm, then details will be given. The algorithm starts by generating a single branching, to be referred to as the red tree. Then the algorithm tries to find a second directed spanning tree (the white tree) that does not use any links already in the red tree. The white tree starts out as just the root, and it is iteratively increased in size. Eventually, either the white tree touches every node (we are done) or else a temporary state is reached where we can go no further. The reason for this state is that all the links that enter the set of nodes already in the white tree, are already used by the red tree. To overcome this state it is necessary to modify the red tree. The modification consists of taking one of the links away from the red tree (one that would prove immediately useful to the white tree), and then doing what is necessary to patch the red tree without this link. Thus we produce a "larger" white tree and an edge disjoint new red spanning tree. Repeating this process we eventually get the two edge disjoint spanning trees (the white tree can only grow so large, then it must be a branching).

The algorithm:

STEP I: Grow a red tree using the algorithm given to find a single tree.

STEP II:Grow a white tree without using any red tree links as follows:

a) Start with no links in the white tree, and only the root node in the white tree.

b) Let W be the set of nodes in the white tree (note: the root is in the white tree).

c) Try to find an edge that goes from W' to W, that isn't already used by the red tree. If there is no such link, then go to Step III. (Which will steal an edge away from the red tree). If there is such an edge, add it to the white tree and add the node that it came out of to the white tree.

d) If all nodes are now in the white tree, then we have both a red and a white spanning tree, and we are done. Otherwise repeat b, c, and d.

STEP III: Carefully choose an edge in the red tree.

The choice of the link to be taken from the red tree for use in the white tree is rather critical to the success of this algorithm. The "red tree distance to the root" from a specific node is defined to be the number of links on the unique red tree path from that specific

node to the root.

We already know that all the links from W' to W, are in the red tree. From among the set of links that go from W' to W the SELECTED link is the link which emanates from the node with the greatest red tree distance to the root.

This link is selected so as to prevent the case of a red tree path that goes from W' to W, then back to W' and then crosses the selected link.

STEP IV: Prepare to use the SELECTED link in the white tree

a) Remove the SELECTED link from the red tree

The network is now partitioned into :

R=(Def.) The set of nodes still connected via the red tree to the root

and

R'=(Def.) The set of nodes no longer connected via the red tree to the root

The network is also partitioned into:

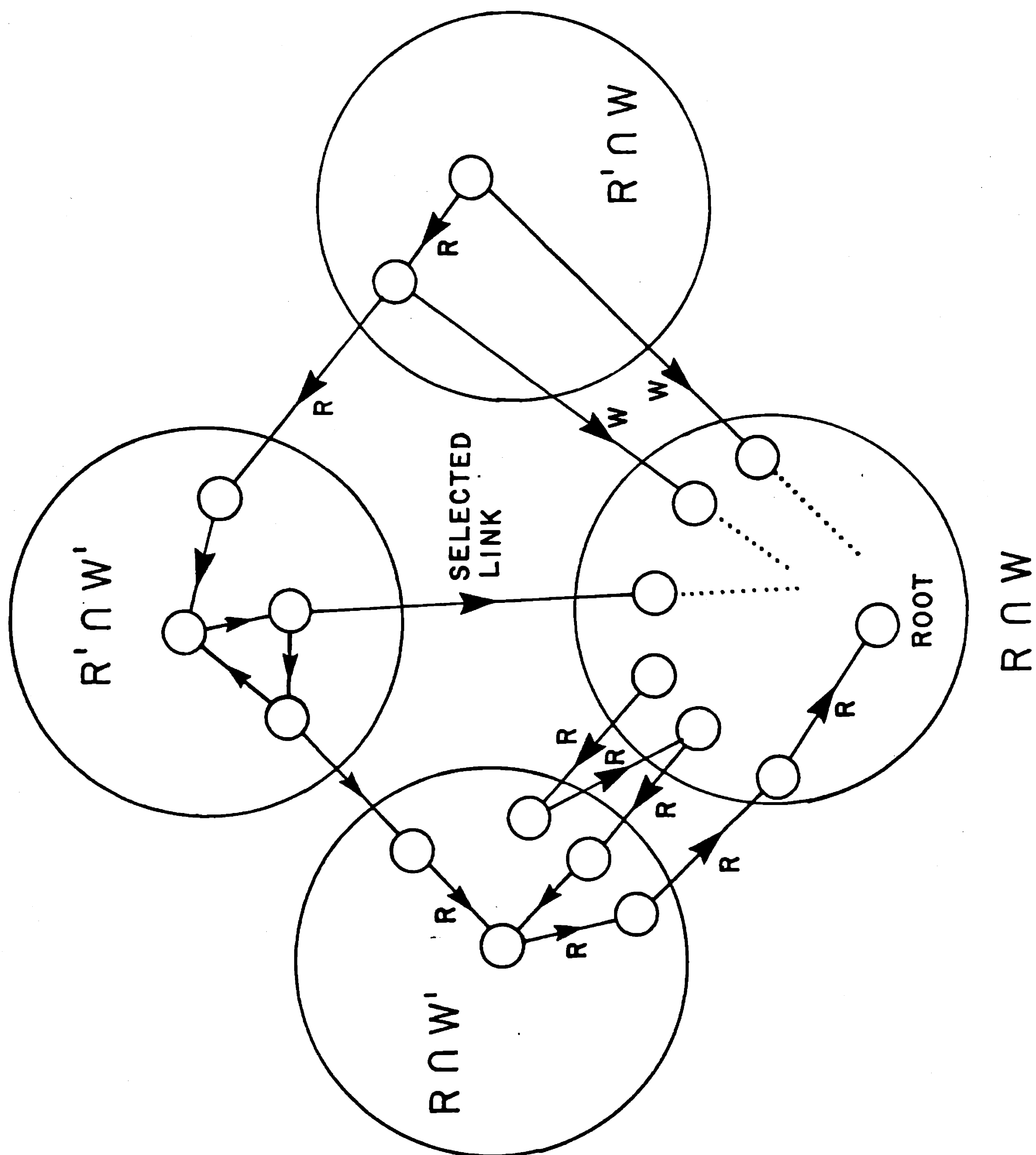
W=(Def.) The set of nodes already connected to the root via the white tree.

and

W'=(Def.) The set of nodes not yet connected to the root via the white tree.

b) One straight forward way to mend the broken red tree back to full tree status (without the use of the just removed selected link or any white links) is: Consider the set of nodes R'W' (read R' intersect W'). It is very significant to note that the careful choice of the selected link guarantees that the links of the red tree joining nodes in R'W', form a spanning tree of R'W'. Assume there are p nodes in R'W'. Remove from the existing red tree the p-1 links that connect pairs of nodes within R'W'. Now to repair the red tree we must add a total of p links to the existing subtrees without creating loops (p= (1 selected link) + (p-1 just removed)).

c) Start the iteration with the set S=R'W'. An example of what the network might look like is:



d) Let L be the set of directed links which start in S and end in a node in S' (since the root is not

in S, there must be at least 2 such links).

e) If the selected link (re:step a)) is in L, remove it from L.

f) There is now at least one link in L. We assert at this point that any link in L:

1) is not in the red tree

2) is not in the white tree

3) has an end node outside S, that is connected via the red tree to the root.

Assertion 1 is true because we have (by the end of step b) removed from the red tree, one outgoing link for every element of S. A property of a directed tree is that there is exactly one outgoing link in that tree from every node (except the root). Hence there cannot remain a red link outgoing from a node in S

Assertion 2 is the case because S is a subset of (its initial value) R'W', which must be a subset of W'. Since nodes in S are not in the white tree, we see by definition that the link in L can not be in the white tree.

The validity of the 3rd assertion rests on two facts. The first fact is that links in L MUST run from nodes in S to nodes in W'. (Note: S is a subset of W'. We are at this point in the algorithm because there are no more links from W' to W that were not red). We further note that all nodes that didn't have red tree paths to the root were in R'. So a proof by contradiction of assertion 3 would proceed: Assume the link in L leads to a node with no red tree path to the root. Then that node must be in R'W'. Hence the node used to be in S. This contradicts the fact that nodes are removed from S only after a red tree path to the root is established (see part g)).

g) Having found an edge (any link in L) that is neither in the red nor the white tree, and leads to a node with a red tree path to the root:

1) We add that link to the red tree

2) We remove the node from S that originates that link (this node now has a red tree path to the root)

h) If S is not empty, repeat steps d) on (iterate)

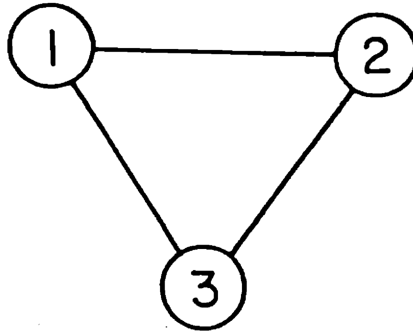
i) Finally now we have a full red tree restored. We have, in the restoration, not made use of the original "selected link". Add the original "selected link " to

the white tree. Add the originating node to the set of white nodes.

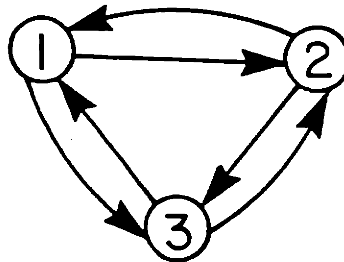
j) Go back to Step II b) and continue to grow the white tree.

RELATING DIRECTED NETWORKS TO UNDIRECTED NETWORKS

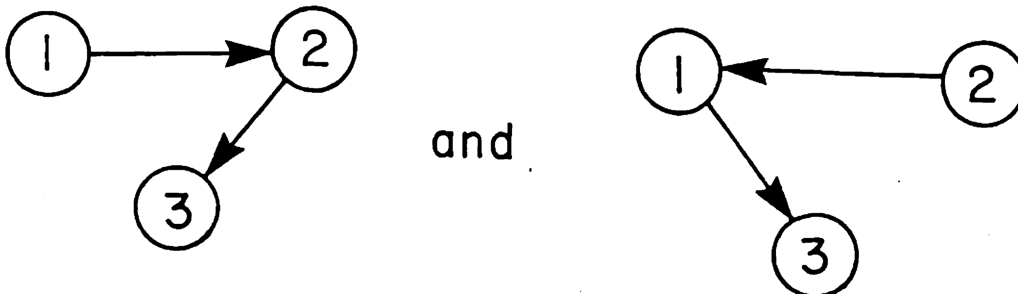
A common reason to look at directed networks is that any undirected network is equivalent to a directed network with twice as many links. This correspondence is based on replacing every undirected link with a pair of directed links in opposing directions. Early in this research it was hoped that this correspondence would lead to a natural extension of results involving directed networks to undirected networks. Unfortunately, neither the questions of existence nor method of finding 2 link disjoint spanning trees in an undirected network follow from the directed network results. The reason for this problem is that link disjoint trees in a derived directed network might share a common link in the original undirected network. For example:



is an undirected network. The derived directed network is

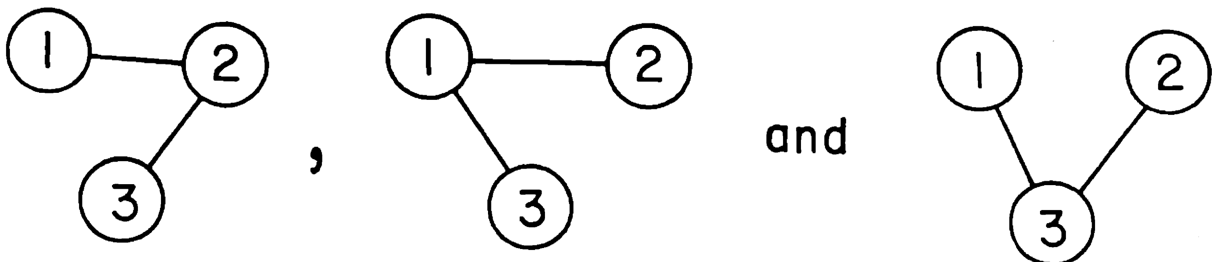


If we are looking for two edge disjoint spanning trees in this directed network rooted at node e, we find:



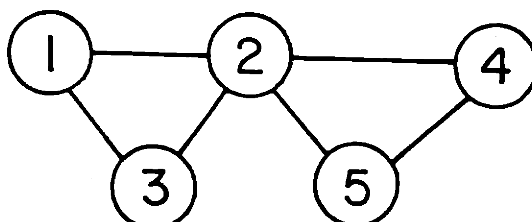
It can be seen by a counting argument that it is impossible to ever have two edge disjoint spanning trees in the original network. Specifically, each spanning tree must have its own two links, but there are only three in total.

We can however look for 3 link disjoint rooted spanning trees in a derived directed network. If there are 3 such trees, then at most two trees can share an edge in the original undirected network. To put it another way, there is no edge in the original undirected network that is common to all 3 trees. If any one edge were destroyed, at least one tree would remain intact. Unfortunately, The most general solution to the problem of finding 3 trees in an undirected network, such that no edge is common to all 3 trees, does not even follow from the directed network results. Consider again the fully connected network of 3 nodes just described. There are indeed 3 trees in the undirected network that have no edge common to all of them, namely:



There are not however, 3 rooted spanning trees on the derived directed network, that have a single common root, and share no edges in common. Hence this solution to the undirected network would not follow from the derived directed network. For the concerned reader, it is interesting to note that the constraint of a "single common root" in the directed network spanning

trees is not always as critical as it was in the example just given. A more general network is:



wherein 3 spanning trees exist with no edge common to all 3, but the derived directed network does not have 3 edge disjoint rooted spanning trees (even with distinct roots!).

The moral of this story is that: if we want general solutions to problems of this sort involving undirected networks, then we must confront them directly. That is what the next section will address.

UNDIRECTED NETWORKS

The fundamental problem to be addressed in this section is that of finding 2 spanning trees in an undirected network. The constraint on these 2 trees is that there is no edge common to both trees.

As we looked at this problem several extensions appeared. The first part of the extension is the constraint on the k trees. The constraint could be:

a) no edge is used by more than 1 tree

b) no edge is used by all k trees

or most generally

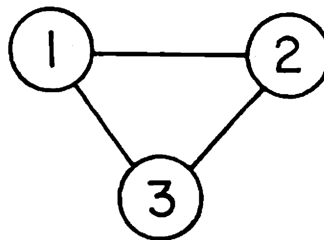
c) no edge is used by more than e trees (with e fixed $0 < e < k$).

The motivation for finding 2 edge disjoint spanning trees was to guarantee that one of the trees would remain intact after any single edge failure. Motivation for looking at k mutually edge disjoint spanning trees would be to guarantee the integrity of at least one tree, even after as many as k-1 edge failures. The usefulness of such things as k spanning trees with no edge common to all of them, lies in the fact that at least one three will survive a single edge failure. Moreover, the conditions for the existence of such a set of trees, grow more and more lax as k is chosen larger and larger.

We offer now a small theorem that addresses the concluding point of the last paragraph. It can be seen that a

necessary condition for the existence of 2 edge disjoint spanning trees is that the network be 2 connected.(ie: Every binary partition of the nodes is traversed by at least 2 edges). The theorem is that: For every 2 connected network, there exists a k>1, such that k spanning trees exist within that network with no edge common to all k trees.

The point that makes the above theorem interesting is that 2 connectedness is necessary, but not sufficient for the existence of 2 edge disjoint spanning trees. Recall for example that the network



does not have 2 edge disjoint spanning trees, despite the fact that it is clearly 2 connected.

The proof of the theorem is as follows: Suppose there are n nodes in the network. Take any spanning tree in the network to be tree T(0). T(0) has n-1 edges in it, call them l(1),...l(n-1). Consider the n-1 spanning trees T(1)...,T(n-1)

that are (nonuniquely) defined by: T(i) is a spanning tree of the network and l(i) is not part of it. The fact that the network is 2 connected guarantees that the removal of one edge (l(i)) from the network will leave a connected network. Indeed, there are spanning trees in every connected network, and hence T(1),...T(n-1) exist. The set of trees T(0),...T(n-1) satisfies the requirement (no edge is in all the trees) and completes the proof. This proof does not always produce the least number of trees, but it certainly demonstrates nicely the existence of some k.

Chapter 3

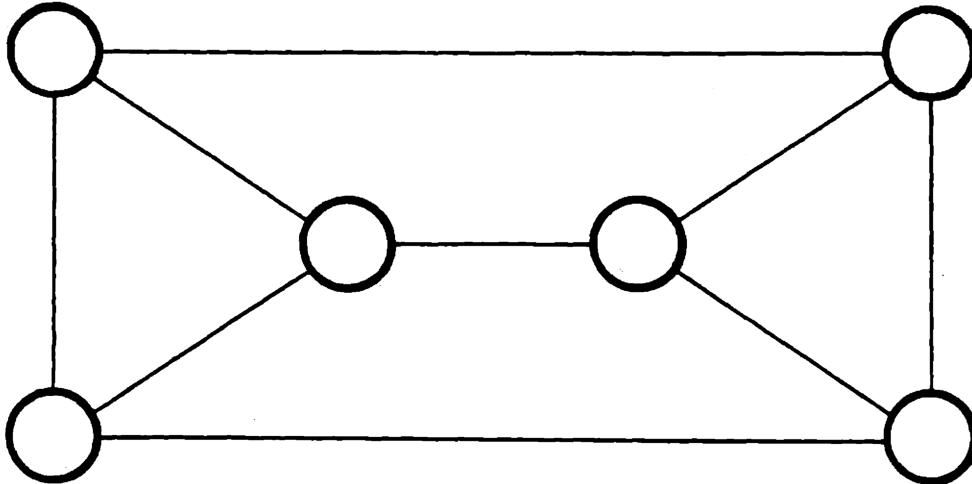
FINDING MULTIPLE EDGE DISJOINT SPANNING TREES IN UNDIRECTED NETWORKS

In this chapter we will focus on the problem of finding multiple edge-disjoint spanning trees in an undirected network. We will start with a method of perturbing a set of overlapping spanning trees into a set of maximally distant (minimally overlapping) spanning trees. The methods involved in this algorithm are fundamental to all the algorithms that follow. Next we will view the problem as a matroid problem and show how the "greedy algorithm" may be applied. This method will involve growing k edge-disjoint spanning trees from scratch. Finally, we will look at the computational complexity of the resulting algorithm and examine ways to improve it. The major result of this chapter is an algorithm for finding k edge-disjoint spanning trees in O(knkn) work (assuming n nodes in the network).

PERTURBING EXISTING TREES

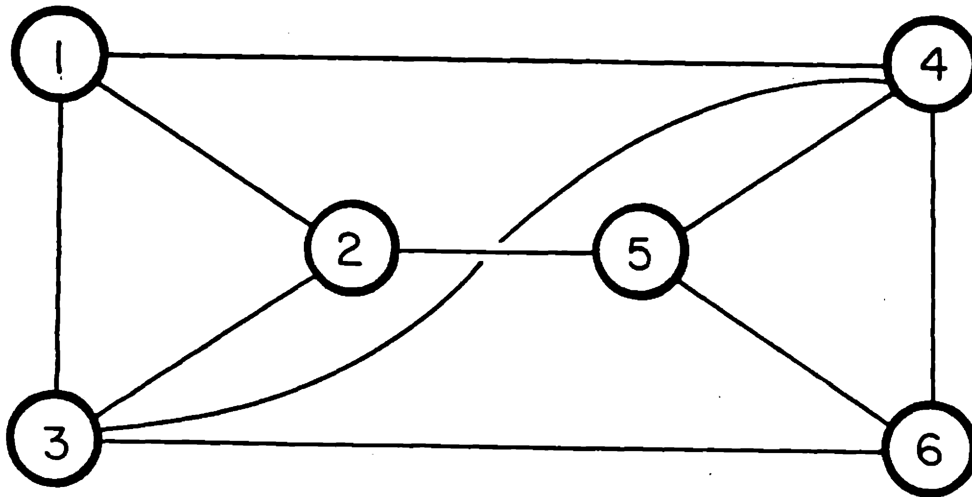
For simplicity, we examine the problem of finding 2 edge disjoint trees in an undirected network. The methods extend directly to the case of k trees, but the generality only complicates the discussion. Consider the question of whether or

not the following network can be broken into 2 edge-disjoint spanning trees:



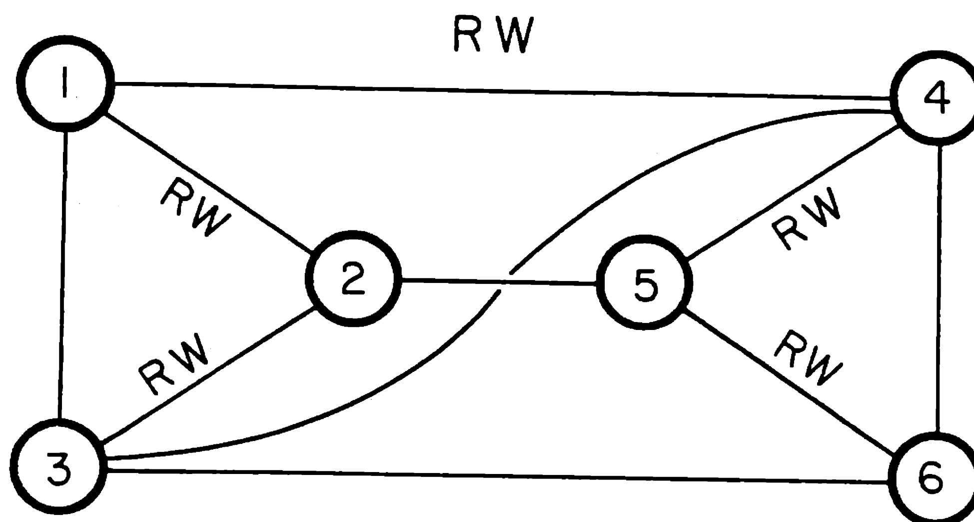
We would like any algorithm to answer this question very quickly with a "NO", using a simple counting argument. The above network has 6 nodes. Any spanning tree must have 5 edges in it. We would then need 10 edges to find 2 such spanning trees (but there are only 9).

Consider then a harder problem: Does the following network contain 2 edge-disjoint spanning trees?



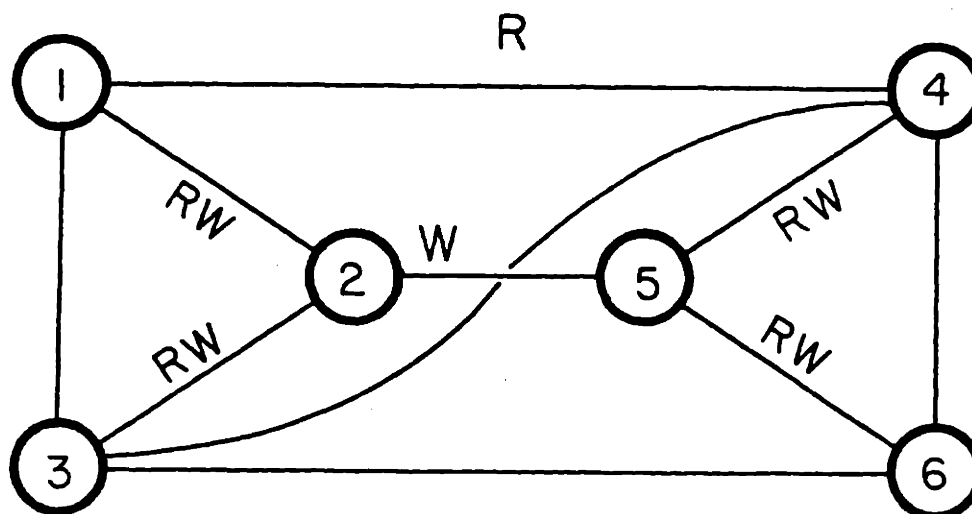
The method we will employ to answer this question will take any 2 spanning trees, and perturb them into 2 new spanning trees that have fewer common edges than their predecessors. By repeating this process, we show that we can eventually get 2 totally edge-disjoint spanning trees.

As an example suppose we started with identical spanning trees as shown below:

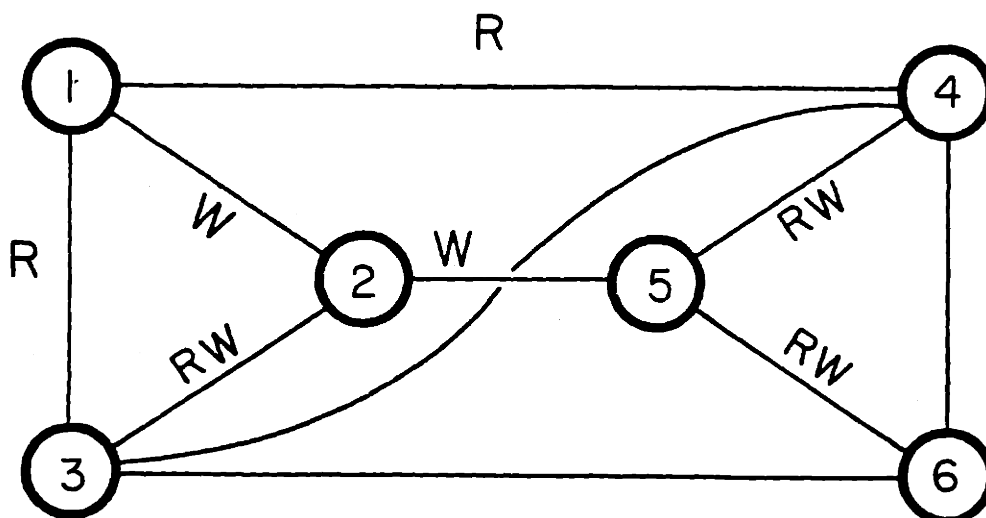


The trees (R for red, and W for white) are defined by putting their labels on edges. Notice that both trees use the same set of edges, namely 12,23,14,45,56.

We wish to incorporate some unused edge into one of the trees so that the overlap can be reduced. Take any edge, say edge 25, and consider what would happen if it were used in the white tree. If we were to use 25 in the white tree we would form a cycle with edges 21,14,45. Since trees can never have cycles, we must then remove one of the forementioned edges from the white tree. As it turns out, all three edges are used by both trees at present, and removal of any of them would be a step in the right direction. If we removed 14 from the white tree we would then have:

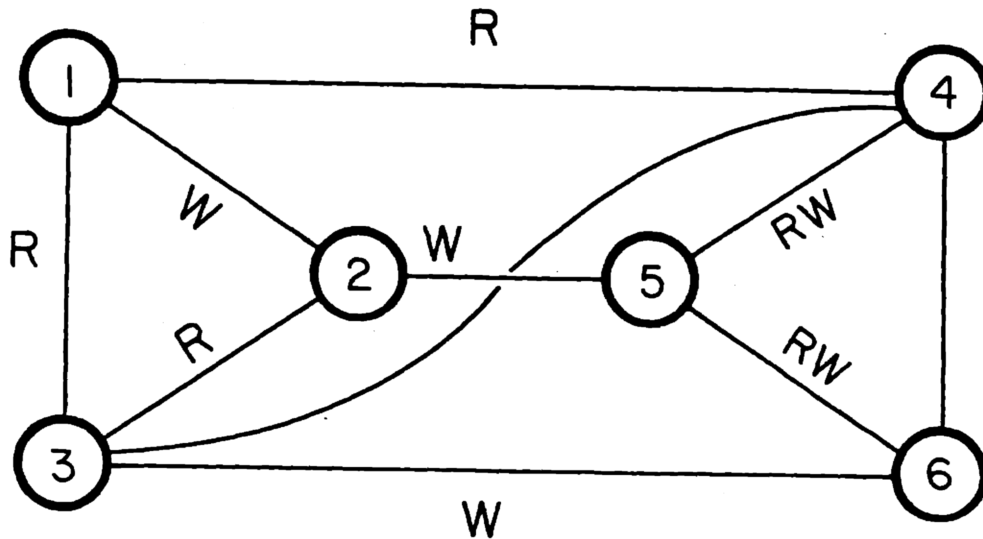


The overlap has been reduced from 5 edges to 4 edges. Suppose we next tried to add edge 13 to the red tree. We would form a cycle with 12,23. Again both edges are doubly used. Suppose we remove 12 from the red tree. The result is:

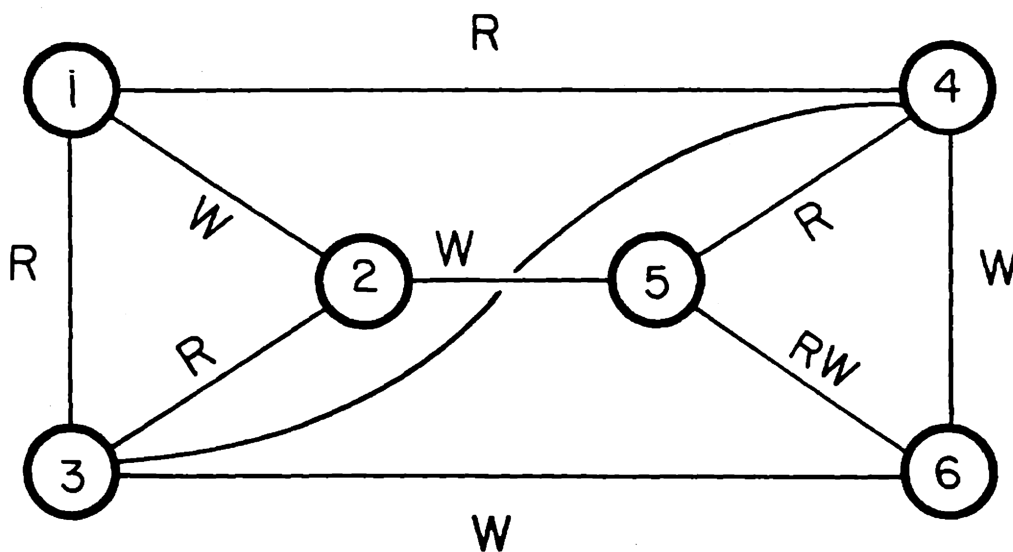


Next we might try adding edge 36 to the white tree. From the resulting cycle 32,25,56, we would want to remove 32 or 56. The

removal of edge 25 would not help us as it is used only once. If we removed 32 then we would have:



Here we might try to add edge edge 46 to the white tree, and remove edge 45 from the cycle of 45,56. The resulting network has only one edge common to both trees, and looks like:



At this point things suddenly become a bit more complicated. If we try to use the remaining unused edge 34 in the white tree, we get cycle 36,64. Both of these edges are used only by the white tree and there would be no apparent progress in using such edges. Similarly, if edge 34 were used in the red tree, we would form cycle 31,14, and again these edges are only used by one tree! All is not lost however, we are just forced to use the full generality of the algorithm. The most general form of this algorithm requires a deeper search of the potential consequences of using an edge in one of the trees.

Consider using edge 34 in the white tree. We have mentioned that no "immediate" progress can be made by removing edges 36 or 64 from the white tree. If however we were to remove the edge 36 from the white tree, it would then become an unused edge, and hence a candidate for use in the red tree. The resulting red tree cycle of 31,14,45,56 does indeed include the doubly used edge 56, which can then be removed from the red tree. Hence, we have found a perturbation sequence (add 34 to the white tree, remove 36 from the white tree, add 36 to the red tree, and finally remove 56 from the red tree) that maintains the tree structures, and decreases the overlap of the 2 spanning trees.

There are several natural questions that might be asked

at this point: Does the above algorithm always find two edge disjoint trees (when they exist)? In a situation where 2 edge-disjoint trees do exist, couldn't the algorithm find two non-disjoint trees for which there is no possible perturbation sequence that will improve matters? How much work does it take to run this algorithm?

In answer to the last question, the complexity of the above algorithm can be brought down to O(knkn) time (to find k trees) by using some noteworthy structure of the problem. We will not present the proof of this complexity result for this specific algorithm, but we will present a proof for a related algorithm that makes use of the same problem structures.

The nature of an argument that shows (by contradiction) that the algorithm always converges to a pair of edge-disjoint trees (when they exist) would be as follows: Assume we have a network and there are two edge-disjoint spanning trees r and w. Suppose we also had two overlapping spanning trees R and W. We can note that the trees r and w make use of a total of 2(n-1) edges, whereas R and W are making use of strictly fewer edges (some of the edges are used twice). We first create a set of edge-disjoint spanning trees r' and w' such that every edge that is in R or W, is definitely in r' or w'. (This pair of trees would be created by a sequence of edge changes performed

on r and w.) Recalling that r' and w' must use a greater number of edges than R and W, there must be an edge e1, that is used by r' or w' and is used by neither R nor W. Without loss of generality assume that e1 is in r'. We can then perturb R by adding e1, and removing one of the edges in the cycle that we just formed. Specifically, we should remove the edge e2 in this cycle that is not in r' (r' has no cycles). We have thus perturbed R to have more edges in common with r' (and maintained the fact that every edge in R or W is still in r' or w'). Next we recall that e2 was in R, so it must have been in w' or r'. We know however that e2 is not in r'. Therefore e2 is in w'. We also note that e2 cannot be in W, as otherwise we would have just found the non-existent perturbation sequence. We can then add e2 to W, and remove the edge e3 (from the cycle that we just formed) that is not in w'. This makes W have more edges in common with w'... Repeating this argument we can eventually find a perturbation sequence that makes R and W identical to r' and w' respectively (which we recall are edge-disjoint). We must then have found a perturbation sequence that made R and W more edge-disjoint (contradiction).

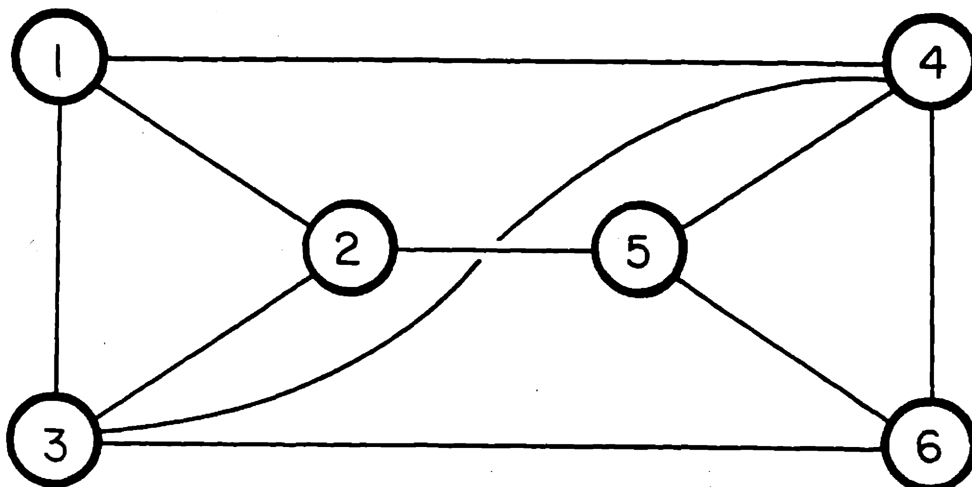
AUGMENTING EXISTING SUBTREES

In this section we will develop a method of adding edges to k mutually edge-disjoint forests (a forest is a subset of edges and nodes in a network that contain no cycles. This method was originally described to the author by Dr. Robert

Tarjan[19]. Our ideas which yielded a computationally efficient algorithm based on finding perturbation sequences extended directly to this method, and are easier to follow in this setting.

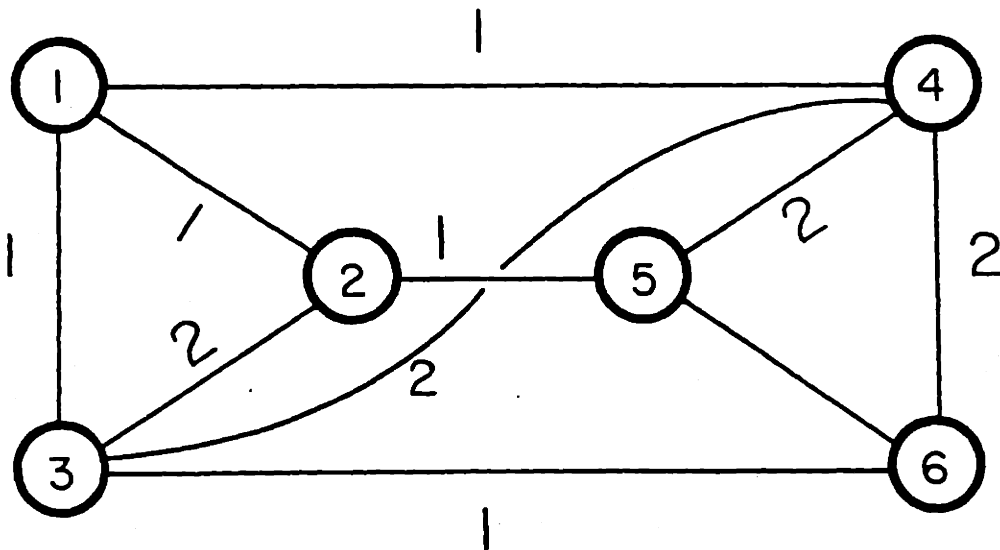
All forests that we will be concerned with will include all the nodes in the network. All further references to forests will discuss only the edge set of that forest). We continue to add edges until each forest has n-1 edges. A forest with n-1 edges must be a spanning tree of the network. Hence we plan to arrive at k edge-disjoint spanning trees by slowly augmenting k forests.

Before we get into the theoretical underpinnings, proof of correctness and details of the efficient use of structures, we consider once again the following network:



EXAMPLE: Finding two edge-disjoint spanning trees.

The list of edges in this network are 12,13,14,23,25,34,36,45, 46,56. We start with the two forests (we are looking for two spanning trees) both being empty sets, and specify this state as forests=({},{}). The first edge 12, can easily be used in the first forest, with the result being forests=({12},{}). Next we try to use 13,and we get forests= ({12, 13},{}). Next 14 gives forests=({12,13,14},{}). The edge 23 cannot be used in the first forest because it would form a cycle, but it can be used in the second forest. The result is forests=({12,13, 14}, {23}). Using 25 gives ({12,13,14,25},{23},and then 34 leads to ({12,13,14,25}, {23,34}). Similarly using 36,45 and 56 results in forests=({12,13,14,25,36},{23,34,45,56}). The network now looks like:



Unfortunately the remaining edge 46, can neither be used directly in forest 1 (as the cycle 41,13,36 would be

formed) nor in forest 2 (as the cycle 45,56 would be formed). As with the perturbational algorithm, the method of making progress involves looking for a sequence of edges. Specifically consider what would happen if we wanted to add edge 46 to forest 1. In order to preserve the cycle free status, we would have to remove either edge 41,13 or 36. Since progress involves strictly increasing the number of edges used by all of the forests, the removed edge should be used in forest 2. Indeed, carefully reviewing the above four edges shows that edge 41 may be directly used in forest 2. Hence we have an augmentation sequence of: add 46 to forest 1, delete 41 from forest 1, add 41 to forest 2. This augmentation gives the new forests ({12, 13,25,36,46},{23,34,45,56,41}), which are indeed examples of two edge-disjoint spanning trees.

With that example under our belt, we can now look at the general question of finding k edge-disjoint spanning trees by such a method. Still to be clarified is: How exactly do we extend this method to an arbitrary k? How we can efficiently look for augmentation sequences? How do we know that such an algorithm will always find a solution when such exists?

WHY IT WORKS

For any fixed k, the edges of the network can be viewed as a matroid. This structure is gotten by defining a set of edges to be independent iff the set can be partitioned into k forests [4,7,14]. With this theoretical view of the problem, we can apply the matroid greedy algorithm [16,24] to get a maximal independent set, which corresponds to a solution to our problem (if the set ends up with k(n-1) elements). We will not rely on these facts, but will rather prove directly that our greedy algorithm works. This will also leave us with a clearer understanding of how and why we may modify the algorithm to improve its computational efficiency.

The greedy algorithm consists of initializing an independent set F as the empty set, and performing the following test with every edge e in the network:

If F+{e} is an independent set, then add e to the set F

("independent set" was defined in the last paragraph) Note that the greedy algorithm would not automatically give the specific forests, but would rather tell us that such forests exist. The beauty (and the name) of the greedy algorithm comes from the fact that once F includes an edge e, it is never removed. Also to be noted is the fact that every edge is considered for

inclusion in F exactly once. This will later play a role in the computational complexity of the entire algorithm.

Having made the algorithm sound very simple, we point out that the difficult part is to decide if the set F+{e} is an independent set. In order to decide this efficiently, we maintain at all times a partition of F into k forests F(1),..., F(k). The fact that we maintain this set also means that when the algorithm terminates, if F has k(n-1) edges, then F(1),.. F(k) are actually the k edge-disjoint spanning trees.

Given an edge e that connects node v to node w, and any forest F(i) which has v and w in the same tree, we will find it useful to define C(e,F(i)) to be the unique set of edges in F(i) that forms a path from v to w. Notice that if e' is in C(e, F(i)), then F(i)+{e}-{e'} is still a forest. Basically, what we are saying is that if we add an edge to a forest that would form a cycle, then we simply have to remove any edge from that cycle to return to a forest status.

For a given set of edge-disjoint forests F(1),...F(k), we shall define a "minimal augmentation sequence" to be a list of edges e(1),....e(p), and a list of indices to forests n n(1),...n(p) (that are not necessarily distinct), such that:

1) $e(1)$ is not in any forest $F(i)$.

2) for $0<i<p$, the end nodes of $e(i)$ are both in the same tree in $F(n(i))$.

3) for $0<i<p$, $e(i+1)$ is in $C[e(i),F(n(i))]$.

4) $e(p)$ has end nodes in separate trees in $F(n(p))$.

5) for $0<i<p$, $0<j<p+1$, $i+1<j$, $e(j)$ is not in $C[e(i),F(n(i))]$.

It should be visible that we are trying to define a sequence of edges that allows us to augment, or increase the size of, the set F (condition 1 says that e(1) is not in F). We are able to add e(i) to F(i) as we delete e(i+1), and still maintain F(i) as a forest by virtue of conditions 2 and 3. The fact that we delete from one forest what we add to another forest (except e(1), which is simply added) guarantees that the set of forests would remain a partition of F. The word "minimal" was used to indicate that no subsequence of edges and corresponding indices could be left out (re: condition 5) and still give us an augmentation sequence.

We have hinted strongly that a minimal augmentation sequence is such that we could run through the sequence i=1,...p; and at each stage:

1) modify F(n(i)) by adding e(i) and, if i<p, also removing e(i+1)

When we are all done, we expect that the resulting F(j) are all forests. In actuality what we have shown is that any ORIGINAL F(n(j)) could be modified by the addition of e(j) and the deletion of e(j+1) and still remain a forest. It remains to be shown that a forest that has been modified several times by the above augmentation process is still a forest. It is actually the minimality condition (condition 5) that allows us to prove that a forest that is modified repeatedly is still a forest.

Theorem: Given a set of k edge-disjoint forests F(1),...,F(k), and a minimal augmentation sequence e(1),... e(p), and n(1),...n(p). The following series of operations:

```
for i=1 to p-1
        F(n(i)) <-- F(n(i)) + {e(i)} - {e(i+1)}
next i
F(n(p)) <-- F(n(p)) + {e(p)}
```

produces a new set of edge-disjoint forests.

The preceding theorem has been proved in the literature [4,7] but a proof will be provided for completeness and to assist the reader with understanding the subtleties of the

theorem.

Proof: We will use induction on the length of the augmentation sequence.

Clearly all minimal augmentation sequences of length 1 produce new edge-disjoint forests.

Assume that all minimal augmentation sequences of length r produce edge-disjoint forests by the above process. Let e(1),...e(r+1) and n(1)...n(r+1) be a minimal augmentation sequence for forests F(1)... F(k). Define F'(1)...F'(k) to be the resulting forests after one stage of augmentation. That is:

$$F'(n(1)) = F(n(1)) - \{e(2)\} + \{e(1)\}$$

$$F'(i) = F(i) \text{ for } i \text{ other than } n(1)$$

The fact that F'(1)...F'(k) are edge-disjoint forests is clear. Now let us define new sequences n'(1),...n'(r) and e'(1),... e'(r) as follows:

$$e'(i) = e(i+1)$$

$$n'(i) = n(i+1)$$

We will now show that the sequences n'(*) and e'(*) are minimal augmentation sequences for F'(1),...F'(k). By the inductive hypothesis we will have that the modification of these new forests by the r length sequence will result in edge-disjoint forests. Hence the original forests F(1),...F(k) can be modified by the r+1 length sequences n(*) and e(*) to yield edge-disjoint forests, which completes the inductive step.

There are 5 conditions to be satisfied to make sure that e'(1),...e'(r) and n'(1),...n'(r) are a minimal augmentation sequence for F'(1),...F'(k).

Condition 1 is that e'(1) is not in any of the forests. We knew that the forests F(*) were edge-disjoint, and that F(n(1)) contained e(2) (=e'(1)). Hence we know that none of the other forests could contain e(2). The definition of F'(*) guarantees that e(2) is not in F'(n(1)), and all the other F'(*) forests are identical to the F(*) forests. So we see that the condition is satisfied.

Condition 2 guarantees that both the endnodes of each edge e'(i) are in the same tree in F'(n'(i)). This fact follows directly for all forests except F'(n(1)), which is the only

forest that changed. Suppose that p and q were in the same tree in F(n(1)), or equivalently, a path from p to q exists in F(n(1)). We will show that a path exists from p to q in F'(n(1)). If this path in F did not use e(2) (the edge that we removed to create F'), then clearly the path still exists in F'(n(1)). If the path did include e(2), then it consists of a path from p to one endnode of e(2), followed by e(2), followed by a path from the other endnode of e(2) to q. Note that the first and third portions of this path are still intact in F'(n(1)). We also note that F'(n(1)) has a path between the endnodes of e(2) which consists of edges in C[e(1),F(n(1)] and e(1). By the transitivity of paths, we must have a path from p to q in F'(n(i)).

Having shown condition 2, the text of condition 3 is well defined. What we must show is e'(i+1) is in C[e'(i),F'(n'(i))]. As with condition 2, this follows directly in all cases except where n'(i)=n(1) (the modified forest). In the case where n'(i)=n(1), we only have difficulty if e(2) (the removed edge) was part of C[e'(i), F(n'(i))]. (If e(2) were not on this path, then the path could not possibly have changed.) If we consider the new path around e'(i) in forest F'(n'(i)) as we did in the proof of condition b, we can view it as the result of "adding" 3 paths. These paths are: the path from the endnode of e'(i) to the endnode of e(2) in F(n'(i)), the path between the endnodes of e(2) in F'(n'(i)), and the path from the other

endnode of e(2) to the other endnode of e'(i) in F(n(i)). The resulting path has all the edges that the path C[e'(i),F(n'(i))] had except for some edges in C[e(1),F(n(1))]. We know that e'(i+1) could not be in C[e(1),F(n(1))] because e(*) and n(*) are a minimal augmentation sequence, and must satisfy condition 5 for the case i=1. Hence e'(i+1) is in the path C[(e'(i),F'(n'(i))].

The next condition is difficult to show only if it is referring to the one forest that is changed in creating F'(*). If n'(p) is not the same as n(1), then condition 4 is immediately true. If n'(p) is the same as n(1), then we must verify that the endnodes of e'(p) are still in separate trees. We note that the endnodes of e(1) were in the same tree in F(n(1)), and hence the addition of e(1) to f(n(1)) could not have combined two distinct trees in F(n(1)). Hence the endnodes of e'(p) must still be in the same distinct trees in F'(n(1)), as they were in F(n(1)) .

As we showed in the proof of condition 2, the only difference between C[e(i),F(n(i))] and C[e(i),F'(n(i))] (i>1) is the possible addition of edges in C[e(1),F(n(1))] and/or e(1). Using the fact that e(*), n(*) are minimal augmentation sequences, and condition 5 with i=1, we get: 2<j<p+1

e(j) is not in C[e(i),F(n(i))]

We can then combine the above original condition 5 to get:

for all i,j such that 0<i<p, 0<j<p+1, i+1<j

e(j) is not in C[e(i),F(n(i))] + C[e(1),F(n(1))]

We also note that e(1) is not in any forest F(*) (condition 1), and hence e(j) can not be the same as e(j) for j>1. This last fact combined with the above condition guarantee that e'(*) and n'(*) satisfy condition 5 in forests F'(*).

IMPLEMENTING THE INDEPENDENCE TEST

The basic algorithm starts with F=F(1)=F(2)=...F(k)={}, and considers each edge e in the network. If F+{e} is independent, then e is added to F, otherwise e is discarded. Using what we have shown thus far, we will be looking for a minimal augmentation sequence that starts with e. If we can find such a sequence, then F+{e} must be independent and we can perform the augmentation. When we finish describing the details of the algorithm, we will prove that when we fail to find such a sequence, then F+{e} is dependent.

```
A) "Initialization
        F <-- {}
        for i=1 to K
```

```
                F(i) <--  {}
          next i


      B) "Look at all edges
          L <-- the set of edges
          for e= each edge in L
               GOSUB C "Test and augment if possible
               If F has k(n-1) elements then STOP "We found
                                                        k trees
          next e
          stop "k edge-disjoint trees don't exist


      C) "Test to see if F+{e} is independent
               Mark every edge in F as unlabeled
               QUEUE <-- e"the edge we're trying to start with
               goto D "the labeling section
```

The forementioned QUEUE will be used by the labeling section to search for an augmentation sequence. It holds names of edges that need to be looked at "further".

```
      D) "labeling step
               If QUEUE is empty then goto G "cleanup, F+{e}
                                                  is dependent
               remove the next e' from QUEUE
               for i=1 to k "try it in all the forests
```

```
            If the endnodes of e' are in different
                                     trees in F(i)
                then goto E "augmentation found
            L' <-- C(e',F(i)) "the path around e'
            Label any unlabeled edges in L' with e'
            Add any edges that we just labeled to
                                             the QUEUE
        next i
        goto D "try this labeling step again
```

The labeling step is performing a breadth first search of the edges that may be exchanged for the original edge e, and the edges that may be exchanged for them, and so forth. When an edge is found to be of immediate use in a forest (i.e. its endnodes are in different trees), then the labels can be traced backwards to reveal the augmentation sequence. The next section illustrates this process.

```
E) "Trace out augmentation sequence
        n(1) <-- i "winning forest in section D
        e(1) <-- e "the winning edge from section D
        j <-- 1 "counter for sequence
        TRACE LOOP:
            if e(j) == e then goto F "perform
                                        augmentation
            j <-- j+1 "increment counter
            e(j) <-- label of e(j-1) "trace back label
```

```
            n(j) <-- forest # that contains e(j-1)
        GOTO TRACE LOOP
```

Now we have the actual augmentation sequence (in reverse order) so all we have to do is use it.

```
F) "Augment F and the forests
    F <-- F + {e} "add the original edge
    AUGMENT LOOP:
        F(n(j)) <-- F(n(j)) + {e(j)} "add the new edge
        if j==1 then return to B "augmentation complete
        F(n(j)) <-- F(n(j)) - {e(j-1)} "swap out the
                                              next edge
        j <-- j-1 "move to next forest
   GOTO AUGMENT LOOP
```

There is now only one point left to be described, namely, what to do if the QUEUE in D becomes empty. We claim that this eventuality implies that F+{e} is dependent.

Lemma: If the above algorithm completely empties the QUEUE without finding an augmentation sequence, then any node that is adjacent to a labeled edge in one forest, is adjacent to a labeled edge in every other forest.

Proof: Every labeled edge was put in the QUEUE. Every edge in the QUEUE was tried in every forest, and the path that was found around that edge in each forest was labeled. Hence, if a node is adjacent to a labeled edge in one forest, then it is adjacent to a labeled path (and so a labeled edge) in every forest.

By virtue of the above lemma, it is reasonable to speak of nodes that are adjacent to labeled edges, without reference to a specific forest. We will now show that the set of labeled edges in each forest forms a tree.

Lemma: If the above algorithm completely empties the QUEUE without finding an augmentation sequence, then the set of labeled edges in each forest forms a tree.

Proof: It is sufficient to show that there is a labeled edge path from any node adjacent to a labeled edge, to the nodes at bo h ends of e (the original edge). Using proof by contradiction, let e' be the first edge that is taken from the QUEUE and has endnodes that will never have a labeled path to one endnode of e (assume that e' is in F(j)). Let e'' be the edge which caused e' to be put on the QUEUE (Note- It is

impossible for e' to be the original e, hence e'' exists). Since e'' was taken from the QUEUE before e', we know that there will eventually be a labeled edge path in F(j) from the endnodes of e to the endnodes of e''. When e'' caused e' to be put on the QUEUE, it was also guaranteed that the other edges in C[e'',F(j)] will be labeled. There will then be an extension of the labeled path from endnodes of e to endnodes of e'', on to endnodes of e' via edges in C[e'',F(j)]. This contradicts the assumption that no such path existed.

Finally now we can show that F+{e} must be dependent in this empty QUEUE case.

Theorem: If the above algorithm completely empties the QUEUE without finding an augmentation sequence, then F+{e} is dependent.

Proof: From the first lemma we have identified there are some number of nodes that are adjacent to labeled edges; assume there are q such nodes. The second lemma tells us that there is a tree in each forest that spans these q nodes. Hence we have identified k(q-1) edges which already interconnect these q nodes. The end nodes of the edge e are already among the q nodes. It is the case that k(q-1)+1 edges that interconnect q

nodes always are a dependent set.

With the validity of the above claim proven, we see that it is sufficient to end the algorithm with:

G) QUEUE was empty in D

return to B "No augmentation is possible

Now that we have a workable algorithm, we will clean up some of the computational details before we evaluate its complexity. In the "labeling step" we have two operations that might pose some difficulty if not carefully done. The first operation is trying to find out if two endnodes are in the same tree in a given forest. The second operation is trying to find the tree path between two nodes, and labeling any of the edges on this path that are unlabeled.

SET UNION FUNCTIONS

As a brief aside, we should describe what a "set union" algorithm is. In many discrete problems a partition of a set must be maintained. This is the case here, where we partition the nodes into subsets that are interconnected in a given forest. Such algorithms are so widely used, that some standard functions are commonly seen in the literature [23] and they are generally referred to as set union algorithms. A set union algorithm is able to manipulate a data structure that

represents the partition so as to combine two separate subsets in the partition into a single subset (via set union). It is also necessary for an algorithm to be able to determine if two elements of the original set are in the same subset in the partition (i.e., ask questions about the actual partition). Generally a set union algorithm is broken into two functional parts that manipulate the same data structure. The first part is a function FIND(i), which returns the name of the subset that currently contains i. The second function is UNION(i,j). UNION(i,j) modifies the existing data structure to combine the subsets that contain i and j. Examples of set union algorithms are given in appendix B. A reader that is unfamiliar with such algorithms would be advised to read appendix b before proceeding further.

ARE TWO NODES IN THE SAME TREE

Note that when any two nodes are connected via a tree in some forest F(j), then for the remainder of the algorithm these nodes remain connected in F(j). The "augmenting step" either supplies an alternate connection between nodes before breaking the old path, or forms connections between nodes that were not previously connected. With this structure in mind we see that we can use a set union algorithm in each forest to keep track of connectivity. When nodes p and q are joined for the first time in forest F(j), we can perform a UNION(j,p,q). (Note: There are k separate UNION functions, which are

distinguished by the first parameter. These correspond to the k distinct forests.) Specifically the following statements would be added to the "augmentation step" just before AUGMENT LOOP:

```
let p,q be endnodes of e(1) "The winning edge
UNION(n(1),p,q) "remember, these sets are connected now
```

With these statements in place, the question "Are the endnodes of an edge already in the same tree in a given forest F(j) ?", is reduced to a pair of FIND(j,*) operations, and a comparison. The FIND(j,p) function returns the value of the canonical node that p is connected to in forest F(j). If both endnodes are already connected to the same canonical node, then they are in the same tree. Specifically, the following statement in the labeling step:

"If the endnodes of e' are in different trees in F(i)"

as found in the labeling is replaced by the sequence of statements:

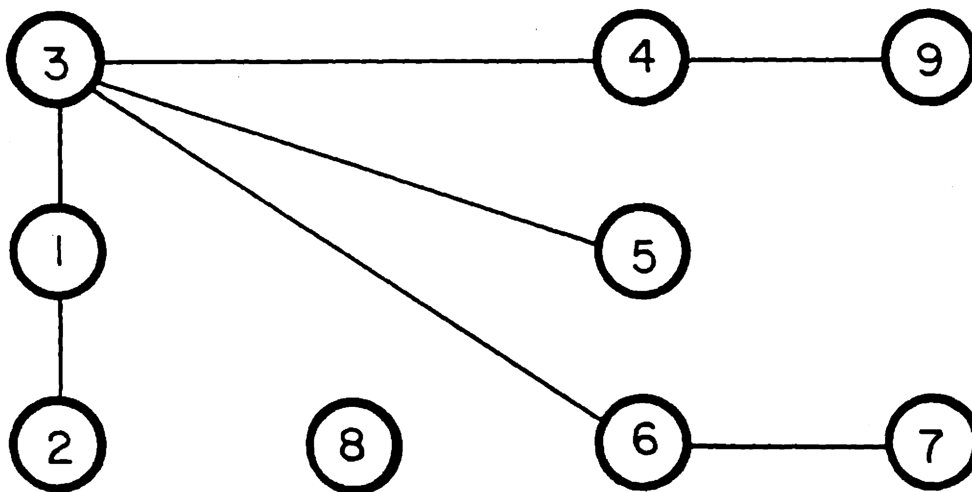
```
Let p,q be the endnodes of e'
IF FIND(i,p) is not equal to FIND(i,q)
```

FINDING A PATH IN A FOREST

Next we consider the problem of labeling all edges in the path between two nodes, excluding those that have already been labeled. This is the second potentially complex task in the "labeling step". In general, such a task might take O(n) time (n is the number of nodes) to perform, as a path might in the worst case be of length n-1. We repeat this task for each edge that we put on the QUEUE, which has a worst case average of O(kn) (except for the original edge, only edges that are in forests can be put on the QUEUE). The QUEUE is restarted every time we try to find a new augmentation sequence, which happens at least O(kn) times. If we are not clever about these operations, this step in the algorithm could easily consume O(nknkn) time.

This specific problem, however, has special structure that may speed up the net complexity. Assuming that we put the edges out onto the QUEUE from C[e,F(j)] ordered by their distance from one endnode of e, then the set of labeled edges in each forest is a tree at each point in the labeling algorithm. A proof of this fact would follow from the method used to show that the empty QUEUE case resulted in labeled edges that formed trees. With this structure in mind, the problem of finding a path around an edge becomes reduced to

tying the endnodes of the given edge into the existing labeled tree. We also should note that the endnodes of e (the original edge that we're trying to base our augmentation on) are always in this labeled tree. To speed our search for the direction to the existing labeled tree, we can precalculate a "Direction to an endnode of e" in each forest, for each node. This computation would be done at the end of section C, just before we called upon the labeling algorithm to do its work. The use of this speed up will allow us to effectively spend, for each forest, a fixed amount of time to process every edge put on the QUEUE. This speed up is most clearly seen by looking at an example.



The above forest has 9 nodes, 7 edges, and 2 trees. Suppose that the initial edge e has endnodes 1 and 7. Before starting the labeling algorithm we would choose one of the endnodes to be a labeling root in all of the forests. Suppose we chose 1 to be the root. We would then calculate the direction to the root

for each node in this forest. Specifically we would calculate 1-(),2-(21),3-(31),4-(43),5-(53),6-(63),7-(76),8-(),9-(94). This data structure shows each node, followed by the first step (edge) toward the root. With all this precalculation done, the labeling could be run.

The first question that the labeling algorithm would ask (relating to paths) would be the path from 7 to 1 (or 1 to 7), as this is the initial edge e. Note that one of these nodes, node 1, is already in the labeling tree as it is the root. Starting from the node that is not in the tree, and using the precomputed path to root information, we find the path 7-6-3-1. This is indeed the path from 7 to 1 as desired. We would then put the corresponding edges on the QUEUE, with the closest to the root going first: that is the order 13, 36,67. With this order strictly enforced, it will always be the case that one of the endnodes of any edge that is removed from the QUEUE is already in the labeled tree. With the above example, by the time 67 is removed from the QUEUE, node 6 MUST have been put in the labeled tree in every forest. The reason is that 36 was already removed from the QUEUE, and the path around that edge put node 6 in the labeled tree in each forest.

Later in the running of the algorithm, the labeling algorithm might ask for the path from 9 to 1. As we claimed,

one of the nodes is already in the labeled tree. We simply start to look at the path from the node that is not in the labeled tree, toward the root. The path then starts at 9, and takes a first step of 94. At this point we note that node 4 is not in the labeled tree, and we continue the path towards the root. The next step takes us from node 4 via 43. Here we notice that node 3 is already in the labeled tree (refer to the previous paragraph) and we stop looking further towards the root. The reason why we can stop here, even though our task was to find a path from 9 to 1, is that we know that the path from this point to the destination node is already fully labeled. We know this because we have tied into a labeled tree that includes the destination node, node 1. We would then put the edges that we traversed onto the QUEUE in order of their nearness to the root: 43,94.

As a last stage of this example, we might be asked to label the edges on the path from 9 to 5. As always, one of the endnodes (9) in this path is in the labeled tree (in this case, 9 is in the labeled tree because the path from 9 to 1 was labeled in the last paragraph). We then begin to head for the root from the other node. Our first step is 53. We note that 3 is in the labeled tree and stop looking any further. The point, of course, is that the remainder of the path from 5 to 9 was labeled previously (The remainder is 34, 49. This was labeled in the last paragraph). We would then put the edge that we

traversed (53) onto the QUEUE.

This example demonstrates that the method given requires a constant amount of processing time for each edge put on the QUEUE, in each forest that processes it. This time is independent of the path lengths, as we can count time both getting onto the QUEUE and being removed from the QUEUE as the service time ascribed to each edge that is processed. This constant amount of processing time is sharply contrasted with the time it might take if we did the following: For every edge that is removed from the queue, find the entire path between the endnodes of that edge, and put any new edges found in that path on the queue. Since a path may be of length n-1 (n is the number of nodes in the network), the processing time (using this inferior method) would be O(n) per edge removed from the queue, per forest that it is processed in.

In order to be able use the above method we must precalculate the directions to the current root in each forest.

PRECOMPUTATION OF THE DIRECTION TO THE ROOT

This section must, every time we choose a new edge to start an augmentation sequence, for each forest, precompute the direction to an arbitrary root node from every node in the

network. The root node that is chosen is always one endnode of the edge that we wish to start the augmentation sequence with. For simplicity, we will only demonstrate the algorithm's use with one of the forests. If this multiple spanning tree algorithm was actually implemented, we would execute this procedure on every one of the forest's list of edges.

We will assume that we have a list of edges E(1),... E(p) for the forest that we are working with. We will show how to create a data base such that, if some node x is connected to the root node in this forest, then a function call of the form DIR_ROOT(x) will return the next node in the path to the root. To remain well defined in all cases, if DIR_ROOT is given the name of the root as its parameter, it will return the name of the root. Since the root is arbitrary in this section, we assume that this data base is produced by a subroutine call of the form PRECOMPUTE_ROOT(y), where y is some node in the network.

We will start by defining the function call DIR_ROOT(x). This function will use the data base to find the direction to the root from the given node (which is specified by an integer from 1 to n). All values of this function are precomputed, and the execution of this function takes constant time, as it is nothing more than a table lookup.

The simplicity of the DIR_ROOT() function call tells us that all the work is done in the precomputing of the table. The basic way that we deduce the table from the list of edges is as follows. First we scan down the list of edges once to create for each node, a list of neighbors of that node. Now that we have the edge information in this form, we just slowly traverse the tree, starting at the root, marking each step we take in the array PATH. We actually perform a breadth first traverse of the tree, and hence we first put all the correct entries in PATH for all nodes that are one hop from the root, then for all nodes that are two hops away, etc. The exact algorithm is as follows:

```
PRECOMPUTE_ROOT(x)
        'For now, just remember that x will be the root.

        DIMENSION PATH[n]  'Declare PATH to be an array with
                                    n entries

        DIMENSION NEIGHBORS[n]  'This is an array of queues
                                              of neighbors
        'Which initially must all be empty.
        FOR i=1 to n
            NEIGHBORS[i] <-- EMPTY
        NEXT i
```

```
'Start by running through the list of edges and
'building up the lists of neighbors
FOR i=1 to p   'There are p edges
     'If an edge e connects node y to node z, then
     'we define N1(e) to be y, and N2(e) to be z.
     ADD N1(E(p)) to the queue NEIGHBORS[N2(E(p))]
          'The first node is the neighbor of the second
     ADD N2(E(p)) to the queue NEIGHBORS[N1(E(p))]
          '..the second node is a neighbor of the first
NEXT i



'Now that we have lists of neighbors, we can start to
'set up the PATH array.
FOR i= 1 TO n
     PATH[i] <-- 0   'Mark each entry as uninitialized
NEXT i

PATH[x] <-- x   'The path to the root is defined to be
                                                  'the root

'Start our queue with the only node that is "no" hops
'from the root...
PATHQUEUE <-- x

LOOP:
```

```
    y <-- next node name in PATHQUEUE  'Get ready to
                         'process a new node. Note that this
                         'operation removes y permanently from
                         'PATHQUEUE (i.e. a pop operation)

  INNER_LOOP:
    IF NEIGHBOR[y] is EMPTY THEN  'if there are no
                                           more neighbors
         GOTO END_INNER_LOOP

    z <-- next node name in NEIGHBOR[y]  'Get the name of a
                                           'neighbor
    IF PATH[z]=0 THEN  ' Make sure that its not
         [                  'yet initialized
         PATH[z] <-- y  'The way to get to the root from
                            'node z is to go to node y.
         ADD z to the PATHQUEUE

  END_INNER_LOOP:
    IF PATHQUEUE is not empty THEN  'See if we've processed
                                           'everything
         GOTO LOOP
```

To analyze the computational complexity of this procedure, we note that we start by running through the list of edges in this forest. Since there can be no more than n-1 edges in a forest with n nodes, this section takes no more than O(n)

work. We also see that there are two additional entries made in the NEIGHBORS list for each edge that we look at. Hence there are no more than 2(n-1) entries ever made in the NEIGHBOR list. We also note that an entry is made into the queue PATHQUEUE during the looping section only when an entry is removed from the NEIGHBOR list. Hence no more than 2(n-1) entries can be added to PATHQUEUE. Every time we pass through the bottom section of the procedure, we delete an entry from the either the PATHQUEUE or the NEIGHBORS list (or both). Hence the bottom section can be executed no more than O(n) times. Hence the total work done to precalculate the entries in PATH is of O(n). We recall that this procedure must be run on each of the k forests, and the grand total of work done in precalculating the direction to roots in all the forests is O(kn).

COMPUTATIONAL COMPLEXITY

Before developing more efficient versions of this algorithm, we should evaluate the time complexity of the algorithm thus far.

Section A performs the initialization of F to an empty set, and starts all k of the forests F(i) also as the empty set. Initially none of the forests have edges, and the greedy algorithm starts F as an empty set (which is by definition

independent). This section takes a time proportional to k, and calls B once.

In section B we pick out unused edges to try to augment F. Section B relies on the fact that once the algorithm finds that the edge cannot be used to augment F, it will never be useful to augment F. For this reason, section B sequentially selects no more than m edges (m is the number of edges in the entire network). Hence this section runs in O(m) time. For every edge that section B looks at, section C is called to see if the edge can be used.

Section C is titled "Test for Independence". Section C initializes all the dynamic data structures for section D to do the actual work. This initialization work consists of marking every edge in every forest as unlabeled, performing the pre-order calculation (to assist labeling section), and clearing the queue to include only the edge that we are trying to augment F with. The number of edges that get marked varies from 0 (at the start of the algorithm) to k(n-1)-1 (near completion of the algorithm). Hence the marking operation takes an average of O(kn) time. The pre-order calculation, as mentioned in its section, takes O(kn) time to compute. The clearing and reinitializing of the queue takes a constant amount of time. Hence EACH run of section C takes O(kn)

processing time in toto. Being called O(m) times by section B, section C runs in total O(mkn) time, and calls D once each run.

Section D (Labeling Step) is where much of the work in this algorithm takes place. For every edge that gets taken from the QUEUE, this labeling section reruns itself. In the worst case, all edges in all forests can be placed on the QUEUE. Since each forest can have as many as n-1 edges, and there k forests, O(kn) edges might be placed on the QUEUE, and hence O(kn) reruns of section D could occur. Within this labeling section, work is done for each edge that is removed from the QUEUE. This work is done for all forests (k of them) and consists of doing a couple of FIND's (O(FIND) work) and some work to label the path around each edge. The work to label the path around an edge can be thought of as constant. (There is a constant amount of needed time to start the labeling process for each edge, and a constant amount of time needed to put each edge onto the queue. So we have that that the processing which is done in this section for each edge that finds it way onto the QUEUE amounts to O(k)O(FIND) for processing as the edge is removed, and some constant processing time as the edge is added to the QUEUE. So we have that each call of section section D by section C will take O(kn)O(k)O(FIND) processing time. Hence the labeling section D takes O(m)O(kn)O(k)O(FIND) time (as section C calls section D a maximum of m times).

Finally now we look at sections E and F. The algorithm executes these sections each time an augmentation sequence is found. The length of the augmentation sequence governs the amount of processing done in these two sections. A worst case scenario would have all augmentation sequences use all the edges in all the forests. As mentioned earlier, the number of edges used by all the forests varies from 0 (at the start of the algorithm) to almost k(n-1) (near the end of the algorithm). So we see that the average augmentation sequence could have length O(kn) edges. This result tells us that the processing done in the looping sections E and F can be no more than O(kn) per call. We should also recall that a UNION operation was added just before the "AUGMENT LOOP" statement in section F (this statement updates the FIND-UNION data structure as to the augmentation). The total work done in sections E and F is then O(kn) +O(UNION) for each call. Each augmentation increases the size of F, and hence these sections can be called no more than O(kn) times (there are k(n-1) edges in k edge-disjoint spanning trees). The total amount of work done in these two sections is then O(kn) (O(kn)+O(UNION)). (The work to do a UNION is O(UNION)).

The total running time for the algorithm is then:

O(kn) [O(kn) + O(UNION) + O(mk)O(FIND)]

As is shown in Appendix B, it is quite easy to get a UNION-FIND pair such that O(FIND) is constant, and O(UNION) is O(n). The above complexity then reduces to:

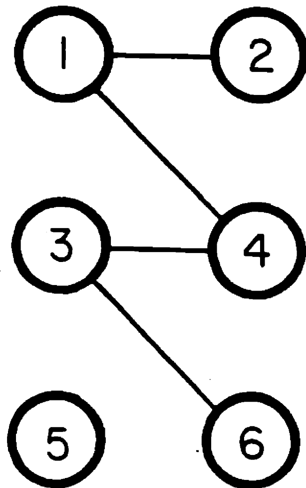
O(knkm).

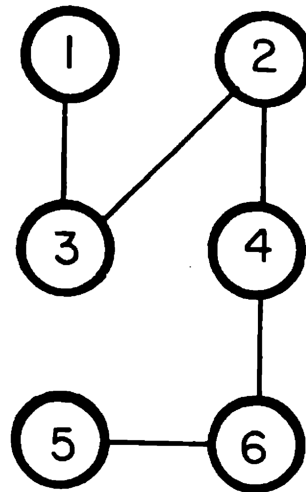
THE "CLUMP" STRUCTURE

Now we will reduce the complexity of the algorithm described up to this point. We start by recalling that whenever the algorithm discovers that an edge cannot be used, it completely empties the QUEUE. The fact that the QUEUE was emptied implies (see the previous theorem) that we have found a set of nodes that are spanned by labeled edges in every forest. This fact is sufficient to guarantee that the original edge e, that started the labeling algorithm, was not independent of the current F. Lets call a set of nodes that are spanned in every forest F(i), by a subtree of that F(i), a "clump". If we think about this clump for a second, we realize that any edge that has both of its endnodes in a clump, cannot be independent of the current F.

As an example of the above structure, consider the following fully connected (every node is connected to every other node) network with six nodes. The algorithm is trying to

find two edge-disjoint spanning trees. The algorithm has already placed nine edges into the forests, and the forests currently look like:





F(1) F(2)

The algorithm just tried to augment the forests using the edge e=26. This resulted in the labeling of edges 12,14,34, and 34 in forest F(1), and edges 13,23,24, and 46 in forest F(2). The algorithm then realized (as it emptied the QUEUE) that the addition of e=26 to the set F (namely: {12,13,14,23,24,34,36, 46,56}) produces a dependent set. The greedy algorithm guarantees that link 26 will not be independent of F in the future. The clever structure to notice is: not only will any link that connects node 2 to node 6 prove to be a dependent addition to F, any node that connects any two nodes in the clump of nodes {1,2,3,4,6} will prove to be a dependent addition to F! In this particular example, a clever algorithm would not even bother to try to augment F with link 16, as it

is totally within the clump. In a more general example, the savings will be shown to be quite significant.

We will now present a few theorems about clumps that allow us to use this structure to our advantage.

Theorem: If a set of nodes C is a clump for the current F, and F is augmented using a minimal augmentation sequence, then C is a clump of the augmented F.

Proof: We simply assert that none of the edges that interconnect any of the nodes of C (in any forest) can be in the augmentation sequence. If we show this assertion, then it will be impossible for any of the edges that interconnect nodes of C to be removed from any forest by the augmentation process. Since these edges remain intact in all forests, C must remain a clump. The proof of our assertion is by contradiction: Suppose a minimal augmentation sequence exists and the i'th edge has both endnodes in C. Since all nodes in C are connected in every forest by edges (there is a subtree in every forest that spans C), we see that all the endnodes of edges in C[e(i),F(j)] are also in C. Hence by induction all later edges in the augmentation sequence have both endnodes in C. This contradicts the definition of an augmentation sequence that states that the

last edge in the sequence has its endnodes in separate trees in some forest.

Now that we have a theorem that states roughly: "once a clump, always a clump", we are in a position to prove a theorem that will allow us to make great use of a clump's structure. Notice that it is no longer necessary to proclaim a set to be a clump relative to the current F, and this fact is used in the next theorem.

Theorem: If two sets of nodes A and B are clumps, and there is a node p that is common to A and B, then the set union A+B is a clump.

Proof: It is necessary to show that there is a subtree in every forest that spans A+B. Since forests have no cycles, it is sufficient to show that a path exists between any two nodes in any forest. Consider any two nodes q and r, that in are A+B. Without loss of generality let us fix our attention on some forest F(j). Since each of our two nodes is in A or B (non-exclusively), and both A and B are clumps, there must be a path in F(j) from q to p, as well as r to p. Using the transitivity of paths, we have a path from q to r in F(j), and we are done.

Having established these properties of clumps, it is clear that we would like to check to see if an edge e is in some established clump before we try to base an augmentation sequence on it. As we have said, if both endnodes of some edge e are in a clump, then F+{e} must be a dependent set. This test would go in section B, just before the "GOSUB C".

Before we actually describe the modifications to the algorithm, we should mention how we can keep track of the clumps. The method is to be a set union algorithm, as we have already shown that existing clumps can be automatically combined by a set union of the clumps. The question: "In which clump are the endnodes of a given edge?", can then be reduced to a FIND of the canonical node for the clump. We then make use of a standard FIND-UNION subroutine pair to do the work.

For the purposes of the complexity bound, it is not necessary to maintain a perfect representation of all sets that can be deduced to be clumps. It is is sufficient to maintain a coarser structure which is easier to maintain. For example, suppose our algorithm found out that the set of nodes {1,2,4,6} formed a clump, as a consequence of trying to use the edge 12. Our algorithm will only bother to record the fact that {1,2} is a subset of a clump. This act of discarding information, (in

this case, that nodes 4 and 6 are also in the clump), will make the computational complexity easier to analyze, and not effect the asymptotic results. Note that having a coarser structure will simply cause the algorithm to look for an augmentation sequence even though perfect deduction (via clumps) would have eliminated the possibility of finding such a sequence. Specifically the union operator is added in section G as:

```
let p,q be the endnodes of e "The original edge
CLUMPUNION(p,q) "Don't bother with the other nodes in
                 the clump
```

Due to the fact that we don't bother looking for all the other nodes to perform the appropriate union, the complexity analysis becomes very simple. Since there are only n nodes, we can do at most n-1 CLUMPUNION's. After that point it must be the case that the entire network is one clump. This would mean that we have found k spanning trees!

The checking of an edge to see if both endnodes are in the same clump is done in section B, just before the "GOSUB C". the exact statements would be:

```
Let p,q be the endnodes of e
IF CLUMPFIND(p) == CLUMPFIND(q) then goto next e
        "don't bother to look at this edge,
              F+{e} is dependent
```

COMPUTATIONAL COMPLEXITY - USING CLUMPS

The new complexity calculation shows that section C, which formerly could be called m times, can now be called only k(n-1) times for successful augmentations, plus n-1 times that clumps are united. Hence section C is only called O(kn) times. The net complexity is then

O(m)O(CLUMPFIND)+

O(n)O(CLUMPUNION)+

O(kn) [O(kn)+O(UNION)+O(nkk)O(FIND)]

Using the FIND-UNION pair that runs in O(1) and O(n) respectively, and identical algorithms for CLUMPFIND-CLUMPUNION, we then get:

$O(m + n^2 k^3)$

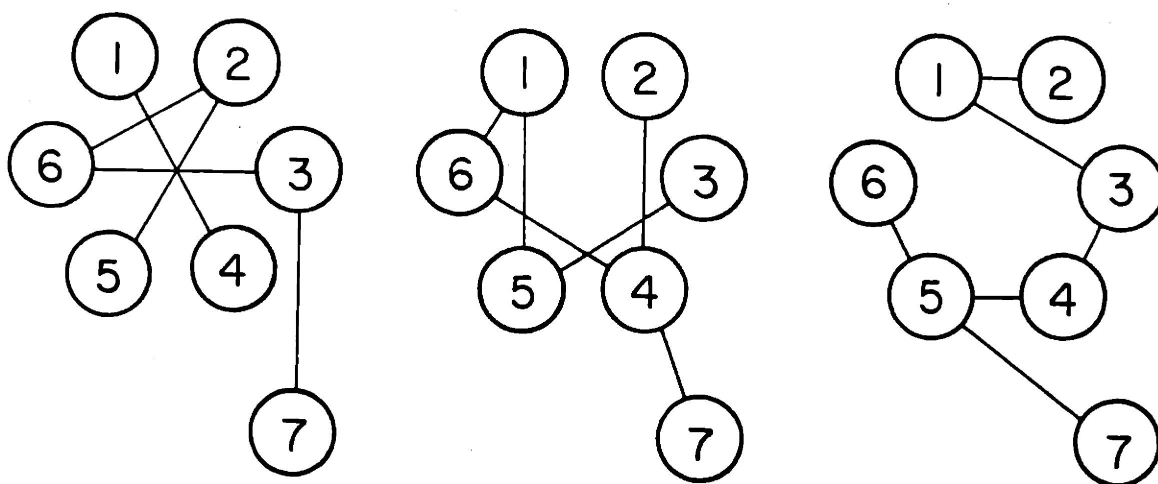
If there is no more than 1 edge connecting any two nodes, we have that:

$O(m) < O(n^2)$, which reduces the above complexity to $O(n^2 k^3)$.

CONSTRAINED MINIMAL AUGMENTATION SEQUENCES

The intent of this section is to achieve a factor of k speedup in the algorithm just presented. As motivation for this potential time savings, we offer the following example of a

segment of our algorithm trying to find 3 edge-disjoint spanning trees. The algorithm is working on a network with 7 nodes. The algorithm has already placed 17 of the necessary 18 (7 nodes imply 6 edges per tree) edges in various forests. We show below the state of the algorithm by showing the 3 forests that are currently being examined.



The next edge to be tried is 23. The algorithm first tries to use 23 in each of the forests. In the first forest the algorithm cannot use 23, so it puts edge 26 and 63 (the path around 23) on the QUEUE. Similarly the algorithm tries to use 23 in the second forest, and adds 24,46,61, 15,53 to the QUEUE. The attempt at using 23 in the third forest yields an addition of 21,13 to the QUEUE.

The QUEUE now contains 26,63, 24,46,61,15,53, 21,13. As we watch the algorithm continue to perform, we will begin to

see the wasted operations. The labeling algorithm now pops the 26 from the QUEUE and tries to use it in all the forests. It is no surprise that the attempted use of 26 in forest 1 is unsuccessful, and causes no other edges to be put on the QUEUE. The edge 26 was generated by forest 1, and the algorithm probably should avoid even trying to use it in such a forest (a small bit of waste). The use of 26 in forest 2 yields no new edges on the QUEUE. Using 26 in forest 3 adds 65,54,43 to the QUEUE (Note: 21,13 are already on the QUEUE).

The QUEUE now contains 63, 24,46,61,53, 21,13, 65,54, 43. This stage of the algorithm then tries to use the edge 63. We can tell in advance that this stage will be a total waste of time! Careful checking of this prediction shows that the edge is not of use in any of the forests, and it causes NO additions to the QUEUE. The reason for our prediction is that edges 26 and 23 were already considered in all of the forests. Hence all the edges on paths from node 2 to 6, and nodes 2 to 3 are already on the queue. By transitivity of paths, all edges on paths from nodes 3 to 6 must already be on the QUEUE!

We will not proceed further with the algorithm, as we have already illustrated our point.

REMOVING WASTE

The question is then raised: How we can keep from performing such "wasteful" steps? there are two possibilities:

1) If we are going to check every edge that comes off the queue in every forest (eg: we checked 23 in every forest), then we should not put ALL the edges that are in a path around that edge (eg: the path around 23 in the first forest consisted of 26,63) on the QUEUE.

2) If when we remove an edge from the QUEUE (eg: 23) we are going to put all the edges in the path (eg:26,63 for the first forest) on the QUEUE, then we shouldn't bother to check to see if the original edge (eg: 23) was useful in all the forests. Instead we should rely on the fact that the edges in the path (eg: 26,63) around our test edge will be effectively tried in those other forests!

MODULO K FORESTS

Of these two possibilities, we will pursue only the latter. It is very easy to implement, and equally easy to analyze. The idea is simply that when the labeling algorithm pops an edge from the QUEUE, it should check to see only if it is useful in the forest after the one it came out of (modulo

k). Instead of checking for usefulness in k forests, we need look only in one forest (a factor of k savings). The modified algorithm would have a labeling algorithm that would look like:

```
D) labeling step
              If QUEUE is empty then goto G "cleanup, F+{e}
              is dependent remove the next e' from QUEUE
              let j be the index of the forest containing e'
                    use j=0 if e' is not in any forest "the
                          original edge isn't in a forest
              let i <-- (j mod k)+1 "the next forest: 0<i<k+1
                 If the endnodes of e' are in different
                                trees in F(i)
                    then goto E "augmentation found
                 L' <-- C(e',F(i)) "the path around e'
                 Label any unlabeled edges in L' with e'
                 Add any edges that we just labeled to
                                      the QUEUE
              goto D "try this labeling step again
```

What we have thus removed is a direct factor of k work from this section of the algorithm. As we saw earlier, this is the key section of the algorithm in terms of complexity.

Although this forementioned change improves the

algorithm's computational complexity, it has changed the algorithm and, therefore, it remains to prove that this modified algorithm will still perform correctly. This algorithm still produces minimal augmentation sequences when it finds a sequence, as we have only further constrained the previous conditions (In addition to the five conditions that define a minimal augmentation sequence, we have the added constraint that n(i) is congruent to i, modulo k) Hence any augmentation sequence that is found can be used to augment the existing forests. It remains to show that when no such constrained minimal augmentation sequence can be found by the above process, then the set F+{e} is dependent.

CORRECTNESS PROOF

The proof of this fact uses the same main theorem used earlier in the thesis. Only the proof of the first lemma needs to be changed slightly.

Lemma: If the above algorithm completely empties the QUEUE without finding an augmentation sequence, then any node that is adjacent to a labeled edge in one forest, is adjacent to a labeled edge in every other forest.

Proof: Every labeled edge was put in the QUEUE. Every edge in

the QUEUE was tried in the next forest, and the path around that edge was labeled in the next forest (as forests are disjoint, and an edge can only be labeled in its own forest). Therefore, if a node is adjacent to a labeled edge in one forest, then it is adjacent to a labeled edge in the next (modulo k) forest. Repeating this process shows that it is adjacent to a labeled edge in all k forests.

Lemma: If the above algorithm completely empties the QUEUE without finding an augmentation sequence, then the set of labeled edges in each forest forms a tree.

Proof: The same proof as was used earlier applies.

Finally now we can show that F+{e} must be dependent in this empty QUEUE case.

Theorem: If the above algorithm completely empties the QUEUE without finding an augmentation sequence, then F+{e} is dependent.

Proof: Same as was used earlier.

COMPUTATIONAL COMPLEXITY - MODULO K FORESTS

We can now recall the complexity of the algorithm that uses the clumps structure, and we get the new complexity:

$$O(m)O(CLUMPFIND) + O(n)O(CLUMPUNION) + O(kn)\ [O(kn)+O(UNION)+O(nk)O(FIND)]$$

Using the FIND-UNION pair that runs in O(1) and O(n), respectively, and identical algorithms for CLUMPFIND-CLUMPUNION, we get a complexity:

$$O(m + n^2 k^2)$$

PREVIOUS RESULTS

Within the literature we have found algorithms [5,13, 14] which do find pairs of spanning trees in an undirected network, when they exist.

The Kameda algorithm algorithm is more complex to describe and is evaluated to run in time $O(\max(n^2 \log(n), mn))$ where: m is the number of links, m is the number if nodes. The Chase algorithm is said to have complexity O(mn log*(n)) where:

$\log^*(n)$=(Def.)least integer i such that $\log_2^i(n) <= 1$

and "log" to the "i" refers to functional composition of log with itself i times, and then applied to n.

In comparison, our algorithm to find two edge edge disjoint spanning trees would run in $O(n^2)$ time.

FUTURE IMPROVEMENTS

There are other potential improvements that may be made to this algorithm. To date, none of these improvements have demonstrably improved the net asymptotic performance. Several of these improvements are presented in Appendix A.

Roskind's Conjecture

We believe that further research in this area, along the lines of the methods of the Appendix A, could produce an algorithm with performance of better than:

$$O(nk^2 n + n^2 k) \log(kn))$$

Chapter 4

Introduction

To review what we have done thus far, we started out by examining a couple of failure recovery schemes. We concluded that there was a gap between algorithms that had post failure decisions completely precalculated (via predominately static routing), and algorithms that left all the work to be done after the failure. To fill that gap we discussed the possibilities of using precalculated detours. This detour method could generally affect a recovery when there was the possibility of only one failure. To accommodate the possibility of more one failure (between major routing updates), we considered methods of creating redundant (edge-disjoint) spanning trees. In chapter 3 we developed highly efficient methods for finding such spanning trees, and now we will discuss how they might be used in recovering from edge failures in network.

RELIABILITY OF K EDGE-DISJOINT SPANNING TREES

The recovery from "several" arbitrary edge failures (using pre-computed tree(s)) would require pre-computing "several" edge-disjoint spanning trees. For example, in order to guarantee that the edges on the pre-computed spanning trees could be used to inform every node in the network of k arbitrary failures, it is necessary to have computed k edge-disjoint spanning trees. The natural question that might

be asked is: For an arbitrary k, what is the mean time to failure of all k trees, as compared to the mean time to failure of a single edge? The mean time to failure of all k trees is significant in that, after all k trees have failed, it is usually impossible to notify all nodes in the network of an additional failure via edges of the spanning trees. What this question addresses is how much failure recovery ability is gained by adding additional spanning trees. To answer the above question, we offer the following analysis.

Assume that the edge failure rates are independent and Poisson with mean time to failure of M time units. We are assuming that we have k edge disjoint spanning trees, spanning the n nodes of the network. We would like to find the mean time to failure of the set of all k trees. Since there are k spanning trees, they contain k(n-1) "spanning tree" edges. The mean time to failure of one of the spanning trees is M/(k(n-1)), as the failure of any of the significant edges would cause a spanning tree to break. When the first spanning tree has failed, there are only k-1 trees left, and hence the are only (k-1)(n-1) relevant spanning tree edges. The mean time to the failure of ALL the trees is then:

$$\frac{M}{k(n-1)} + \frac{M}{(k-1)(n-1)} + \ldots\ldots + \frac{M}{(n-1)}$$

$$= \frac{M}{(n-1)} * \{ \frac{1}{k} + \frac{1}{k-1} + \ldots + \frac{1}{2} + \frac{1}{1} \}.$$

This sum is bounded above by $\frac{M}{n-1} * \log k+1$.

If we have 2 or 3 trees already, this analysis shows that the added complications of using and maintaining additional trees yields little marginal increase in the reliability of the set of trees.

NETWORK MODEL

Before we discuss how the edge-disjoint spanning trees can be put to use in recovery from edge failures, we will define our network model, and what we mean by "failures". The exact model that we will discuss is not the only model in which edge-disjoint spanning trees might be used, but it is meant to be a well defined scenario in which we can define explicit protocols. Several of the assumptions are made to mimic the paper on "Resynch Procedures..." by Finn [10], which serves as the basis of much of our protocol development.

We will assume that nodes in the network have error free processors and storage. We will also assume that all edges in the network are error-free while they are up (operational). All edges are either up or down (non-operational). These last two facts can be guaranteed by a link protocol, at the expense

of having variable transmission delays on each edge. Edges that are down can carry no information, except test messages concerning the status of that edge. Edges that are up are assumed to have some arbitrary capacity and, as we said, variable transmission delay. An edge may change from up to down at any point in time, and we assume that the two endnodes of that edge might not realize this at the same time. We do require that both endnodes agree that a node is down, and both endnodes execute a prescribed link level protocol before an edge can become operational again. The last assumption is that all packets are received on an edge in the order in which they were transmitted (packets can't pass each other on a link).

DISSEMINATING EDGE FAILURE INFORMATION

The original reason for developing multiple edge-disjoint spanning trees was to be able to broadcast the fact that a failure had occurred. The usefulness of this lies in the assumption that each node in the network has knowledge of the pre-failure topology. The statement that "Edge x-y has failed", is sufficient to inform any node of the post failure topology. We are using several spanning trees so that after more than one failure has occurred, at least one of our trees will be predominately intact. It is necessary then to describe a protocol (or distributed algorithm) that will, assuming we have k edge disjoint trees:

1) When fewer than k edge failures occur, all nodes will be notified (in a communications efficient manner) of all failures.

2) When more than k-1 failures occur, but not all of the spanning trees are effected, all nodes will be notified of all failures.

3) When "too many" failures have occurred in the spanning trees (all of the trees are damaged), a "sure fire" backup method is instigated in each connected group of nodes. All nodes in a connected network where a "sure fire" algorithm is active are made aware (or conclude independently) that such a method is to be used.

The protocol should generally be such that likely events (eg: single failures or restorals) cause minimal action (communication), and unlikely events (such as so many failures that the network becomes disconnected) are permitted to instigate a lot of work (communications), but must not fail to perform (no deadlocks or infinite loops, no matter what). An example of a "sure fire" method to deal with massive failures would be an algorithm that would flood the network (every node that receives a "flood message", retransmits that message on all other outgoing edges) and reestablishes whatever connectivity is left in the network. Details of such a "sure

fire" method will given later.

A major difficulty with such protocols is getting all nodes to use the new link level topology information at nearly the same time. If different nodes maintained different views of the topology of the network, they might ping-pong a packet back and forth, having total disagreement as to how the packet should get to its destination. This synchronization of nodes will make for some difficulties in the final explicit definition of our protocol. Many of the complications that arise in the algorithms are based on this synchronization problem.

"SURE FIRE" RECOVERY METHOD

There are three major gaps in the protocol that we have just hinted at. The first question is: How can one reach a state where every node is sure that all nodes are aware of exactly the same topology? (We assume that each node has an internal topology table that all routing is based upon, and this is what we seek to synchronize.) This first question is quite significant, as this synchronization is the starting assumption of the entire protocol. The second question is: What is a concrete example of a "sure fire" protocol? It is hard to be convinced that an entire protocol is correct when one part of it (use "sure fire" method) is not defined. Finally, we need

to very explicitly define the entire protocol, so that we can rigorously prove its correctness. In this section we will address the first two questions. The explicit definition of the entire protocol will be postponed until we have discussed, in looser terms, the way our protocol might use k spanning trees.

The reason for developing this algorithm was to synchronize all nodes in their views of the network topology. Implicit in a link protocol is the fact that each node knows the local link level topology (re: adjacent nodes). We would like to prove that when the algorithm terminates, every node knows the entire link level topology of the network. Unfortunately, this algorithm cannot lay claim to such an incredible feat. The fundamental problem with any algorithm achieving such synchronization is that all information that an individual node has is (by virtue of the delays in the edges) information about how the network used to be in the past. We assume that failures can occur asynchronously throughout the network, and hence a node can never be sure that the topology that it has accumulated has not changed. The best that we can hope to achieve is that we can find a fixed topology that every node will agree with at SOME point in time SINCE the start of the algorithm. The reason behind the "some time" statement is that at least this fixed topology was correct locally (each node agreed potentially at different times). The reason for the "since the start" statement is that the nodes then know that

local correctness was not that long ago.

The fundamental problems that we just mentioned, with respect to what a distributed algorithm may conclude about a changing network, extends to what it means for a distributed algorithm to end. We can talk about when any one node stopped running a given algorithm, or when a specific node found out that all nodes have stopped running, or when an omniscient observer noticed that all the nodes have stopped running, or when all nodes know that all nodes have stopped running, or when all nodes know that all nodes know that all nodes have stopped running, etc. When we refer to the "termination of the sure fire algorithm", we are describing the point in time when an omniscient observer has noticed that ALL of the nodes in the network have stopped running the algorithm. Notice that individual nodes may stop running at earlier points in time.

Our "sure fire" protocol will, loosely speaking, terminate execution with all nodes aware of some fixed topology of the part of the network to which they are connected. More over, every node will be sure that every other node in its connected subset of the network has an identical (and correct in the above sense) view of the topology of the network. Hence, the "sure fire" algorithm will terminate with the nodes having exactly the assumptions necessary to begin to make use of the

multiple spanning trees! When we have finished describing the "sure fire" algorithm, we will have also proved that the situation is reachable in which all the nodes are aware of the entire topology of the network (at least the part they are connected to).

The conditions, again loosely stated, that our "sure fire" (SF) algorithm will satisfy are:

1) The algorithm may be started asynchronously by any node(s) in the network when there is a change in the link level topology around that node. If the topology of the network changes (an edge fails) while the algorithm is running, any node(s) may start a new version of the algorithm.

2) When any node p starts to execute the algorithm, all nodes that can receive messages from p will be forced to start the SF algorithm. The remaining conditions deal with the performance of the algorithm after we have stopped having link level topology changes.

3) No node may stop executing the algorithm until it is aware of a jointly approved topology of its maximal connected subset of the network. By "jointly approved" we mean that for every node p in this connected set, there was a time (since the start of this algorithm) that node p agreed with all aspects of this topology that related to node p (i.e.: what edges are

adjacent to it).

4) When a node x stops executing the algorithm, all of its neighboring nodes will "immediately there after" also stop executing the algorithm. By "immediately there after" we mean they will stop execution before they have time to receive any data packets from node x. (The exception to this would be when a neighbor starts a new version of the algorithm.)

The idea behind running the algorithm is to determine what the connectivity is at each node. Condition 1 guarantees that any node may start the inquiry. Condition 2 guarantees that every node in the connected subset will assist in this inquiry. Every node in the connected subset MUST participate in gathering the information for it to be correct! Condition 2 also guarantees that every node in the connected subset will update his view of the topology to conform to the results of this current inquiry (nodes that participate in the algorithm end up having the same view of the topology). Condition 3 guarantees that each node will wait for the full results of the inquiry before acting upon the information that is being gathered. Finally condition 4 guarantees that once a node begins to act upon the results of the inquiry (by sending out packets in a direction that is reasonable for the "known" topology), all the nodes neighbors will be in agreement in their view of the topology.

The "sure fire" protocol that we will describe is based upon algorithm A1 of Finn [10]. The changes that we have made will allow the nodes to find out the link level topology of the network. The logical structure of the algorithm is due entirely to Finn, and we are making no modifications to it at this point.

The following state information is kept at each node in the network:

a) M = (Def.) mode of this node: either NORMAL or RESYNCH mode

b) R = (Def.) The version number of the last SF algorithm started

c) N(i) = (Def.) i=1,...n A table with entries for each node in the network. This table will eventually tell which nodes in the network are connected to this node.

d) L() = (Def.) A table with entries for each edge that touches this node. The entries contain status information about each of these adjacent edges.

e) E(i,j) = (Def.) i and j =1,...n A connectivity table with entries for every pair of nodes in the network. Eventually this table will contain the view of the topology of the network.

When in the past, we have said that the SF algorithm was "running at a given node", what we meant was that the node was in the RESYNCH mode. This mode indicates that the node is actively participating in the resynchronization of the nodes "view of the topology" and no data packets are being sent . When we said that the algorithm "stopped running at a given node", what we meant is that the node is in the NORMAL mode, and regular data packet switching is also being done.

In order to avoid confusion (in the nodes) between different tries at running this algorithm (that are started at different points in time), a version number is kept at each node. This version is basically an integer counter. If, at any point in time, a node wants to start to run the SF algorithm all over again, it simply picks a version number that is one greater than any version number that it has ever heard used. When a node exchanges information across a link during the running of the SF algorithm, it includes a copy of the version number that this information is related to. Using this protocol, information that would have been pertinent to old

versions (and hence is not part of the most recent version) is ignored.

The node table carries some logical information about the status of other nodes in the running of the SF algorithm. If we examine the contents of this table at node j, and we look at the entry N(p) in this table, we get the following information:

If N(p) is 0: Node j does not yet know that there are any possible paths from j to p.

If N(p) is 1: Node j knows that there is a path from j to p. (We will be more precise about these paths later.)

If N(p) is 2: Node j knows that there is a path from j to p. Node p has received (since the start of this version) responses across all of its adjacent edges that are up. Hence node p knows exactly what nodes it is connected to by crossing exactly one edge. Node j has been made aware of all the nodes that node p can reach in one step, and this is represented in the table at node j (that is: Since there is a path from j to p, and p can reach some set S of nodes in one edge, then there is a path from j to every node in S. Hence the entries of N (kept at node j) for each node in S are either 1 or 2).

As we said, the table L() contains entries for each edge that is adjacent to this node. This table is used to keep records of the status of the adjacent edges. As we will see, the table entries are initially set to either "LINK IS DOWN", or "LINK IS PROBABLY UP". Later in the running of the algorithm, when a response is received across a link, the status "LINK IS PROBABLY UP" will be changed to "LINK IS DEFINITELY UP". This algorithm is made to act in a very conservative fashion, and verifies all its assumptions (such as certain links being up) before it records them as facts.

The E(n,n) table is not significant to the logical flow of the algorithm, in that there is never a test made as to the contents of E. The meaning of an entry E(i,j) in node p, is:

If E(i,j) is 1: There is an edge that connects nodes i and j that "IS DEFINITELY UP".

If E(i,j) is 0: Node p is not aware of an edge that connects node i to node j that is up. If in addition, either of the entries N(j) or N(i) in node p is a 2, then there is no edge between node i and j that is up. Note that the table E(*,*) is symmetric. for purposes of clarity, we will maintain this entire table.

Having given the reader a preview of what the algorithm will maintain as variables, we now present the actual algorithm.

We start by describing the procedure that any node p that wants to start (or restart) the SF algorithm would do. This procedure can only be executed if node p is aware of some change in the link level topology of the network. Until we introduce the spanning tree algorithm that will work in conjunction with this algorithm, we will restrict node p to executing this procedure only if some local topological change takes place. (i.e. The underlying link protocol tells us that an adjacent edge has changed from up to down, or from down to up). Hence when we later prove theorems that assume that there no topological changes during the period of time being discussed, this procedure can not be involved. If some node is in the resynch mode (M=RESYNCH) and there is a change in the status of an adjacent edge, then that node MUST immediately execute this procedure.

INITIATE "Sure Fire"
execute at node p

```
        R <-- R+1  'Get a version number that is bigger than
                    any we've heard of.
```

```
M <-- RESYNCH  'we have to put ourselves into RESYNCH
                mode, thus starting our own running of
                the algorithm


'Now we initialize our tables to run
FOR i=1 TO n
    N(i) <-- 0  'we are connected to no other nodes,
NEXT i
'except...
N(p) <-- 1  'we are connected to ourselves

'We make no assumptions about the link level topology
'of the network and we initialize it to have no edges.
FOR i=1 to n
    FOR j=1 TO n
        E(i,j) <-- 0   'We don't even assume that we
                        are sure of the adjacent
                        (edges) link level topology
    NEXT j
NEXT i

'For the final part of the initiation of SF, we must
'initialize L(), and tell all our adjacent nodes to
'run this version of SF.
```

```
'Assume that there are q links that touch our node p.
'Hence the L() table has q entries. There are also q
'physical edges that we refer to as LINK(1),...LINK(q).
'Note that L(i) is an entry in a table, but LINK(i) is
'the actual link as viewed by the underlying link
'protocol.

FOR i=1 TO q  'run through all these links
    IF LINK(i) is down THEN  'Get the edge status from
                             'the underlying link
                             'protocol system.
        L(i) <-- "LINK IS DOWN"  '..remember it, but
                             'that's all we'll do (ie: look
                             'at the next link)

        ELSE [   'The actual link is viewed as up by
                                        'the link protocol
            L(i) <-- "LINK IS PROBABLY UP"  'remember

                'and test it out by sending a message

            SEND (SF(R,N,E)) ON LINK(i)
            ]  'End the ELSE
NEXT i
WAIT TILL NEXT MESSAGE ARRIVES
```

In the special case where all links are down, we should go back

to the normal mode.

The test message that was sent across each link that was marked as "PROBABLY UP", identifies itself as a part of a specific version of the SF algorithm. Since this message is sent as part of the procedure, we expect that the rest of the network will soon be executing the SF algorithm. This test message then serves double duty, in forcing all of its neighbors to start executing the SF algorithm, as well as testing the status of the edge. We will soon see that upon receipt of this message, neighbors will soon send a message back over the same edge (part of the protocol), and hence verify the functionality of the edge.

The next thing that we must describe is the response of node p to an arbitrary SF(R',N',E'). There are several cases, each of which depends on the version number R, which the node p currently has. The easiest possibility is the case where the version numbered in the received message is older than the current version R ($R'<R$). In this case the SF message can be totally ignored, as there is a more current version running (or a more current version was run).

The next possibility is that the R' in the received

message SF(R',N',E'), refers to a more recent version than node p has heard of (R'>R). In this case, node p must start to run a SF algorithm with version number R' from scratch. The procedure that must be run is then:

node p RECEIVED SF(R',N',E') on its LINK(r) from node s
(LINK(r) corresponds to table entry L(r) at node p)
at node p R<R'
node p executes

```
        R <-- R'   'Update our record of the highest version
                         'number that we've heard of.

        M <-- RESYNCH  'Officially start to execute this
                         'version of SF, no matter what we were
                         'doing.

        'Since we are connected to all the nodes that our
        'neighbor (who sent us the message) is, we can load
        'its connectivity table for a start
        FOR i=1 TO n
            N(i) <-- N'(i)   'Load the table that we received
        NEXT i
        'Also, we are connected to ourselves
        N(p) <-- 1

        'Our view of the topology is restricted to what was
```

```
'just received in the SF message. So we start by
'loading that:
FOR i=1 TO n
    FOR j=1 TO n
        E(i,j) <-- E'(i,j)  'This is what we received
    NEXT j
NEXT i
'We actually have one more bit of information, the SF
'message that we received had to be sent out after the
'start of this version of the SF algorithm started
'running. Since we received this message directly from
'node s, there must an edge that is up from node p to
'node s.
E(p,s) <-- 1  'Record the existence of this link...
E(s,p) <-- 1  '.. in both directions


'The last thing to do is to set up our L() table, and
'propagate the fact that version R if the SF algorithm
'is being run.


'Assume that there are q links that touch our node p.
'Hence the L() table has q entries. There are also q
'physical edges that we refer to as LINK(1),...LINK(q).


FOR i=1 TO q  'run through all these links
    IF LINK(i) is down THEN  'If the actual link is
                             'down,
```

```
L(i) <-- "LINK IS DOWN"  '..remember it, but
                 'that's all we'll do (ie: look
                 'at the next link)

ELSE [   'The actual link is view by the link
                         'protocol to be up

    L(i) <-- "LINK IS PROBABLY UP"  'remember

        'and test it out (and propagate the
        'running of this algorithm) by sending
        'a message.
    SEND (SF(R,N,E)) ON LINK(i)
    ]  'End the ELSE
NEXT i
'There is actually one edge that we are sure is
'working (since the start of this version. As mentioned
'when we initialized the topology E table, LINK(r) is
'definitely up, as the SF message came over it. So...
L(r) <-- "LINK IS DEFINITELY UP"
WAIT TILL NEXT MESSAGE ARRIVES
```

We have now describe the exact actions taken by an arbitrary node p when it receives a message SF(R',N',E'), when the version number R that is maintained by node p is different from R'. The final case to discuss occurs when R'=R. In this

case node p has already sent out some SF messages with this version number, and the received SF message is in some sense "a response" to node p's broadcasts. There are two states that node p may be in, and these will determine node p's response. If node p has finished running this version (M is now NORMAL), then node p can ignore this message, as node p has completed its work on this version. If node is still running this version (M is still RESYNCH), then node p must perform the following series of updates and transmissions. The work that is required is basically to combine any additional information that arrived in this SF message with all the information that node p already had, and then tell all its neighbor all the information that it knows. Note that the underlying link protocol will only deliver messages over edges that it has told us are up. If any edge status (as determined by the link protocol) changes while a node is in the resynch mode, then it must go back and execute the initiate procedure.

```
node p RECEIVED SF(R',N',E') on its LINK(r) from node s
LINK(r) is up
(LINK(r) corresponds to table entry L(r) at node p)
at node p   R = R', M = "RESYNCH"
node p executes


        'Since we are connected to all the nodes that our
        'neighbor (who sent us the message) is, we should
```

```
'combine this information with what we knew in
'our old N.
FOR i=1 TO n
    N(i) <--MAXIMUM (N(i), N'(i))  'This MAX function
        'serves to combine the two arrays. Notice that
        'if either table was a 1 or a 2, then the
        'result was a 1 or a 2. This corresponds to
        'either table saying that there is a path to
        'node i, so we record in our N table that there
        'is a path to node i. Also, if either table has
        'an entry of 2, then a 2 is stored into our N
        'table. This corresponds to the case where one
        'of the N tables carries information that says
        'all the connectivity data about that node is
        'contained elsewhere in that table. Since we
        'are copying all the connectivity data from
        'both tables, it is correct to put a 2 into our
        'N table as well.
NEXT i

'LINK(r) is definitely up, as the SF message came over
'it. So...
L(r) <-- "LINK IS DEFINITELY UP"


FOR i=1 to n  'Run through the entire table
    FOR j=1 to n
```

```
            E(i,j) <-- MAXIMUM (E(i,j),E'(i,j))  'Here we
                        'are simply saying that if either E or
                        'E' has definite information about two
                        'nodes being connected, then we should
                        'remember that those two nodes are
                        'connected.
        NEXT j
    NEXT i
    'We actually have one more bit of information, the SF
    'message that we received had to be sent out after the
    'start of this version of the SF algorithm started
    'running. Since we received this message directly from
    'node s, there must an edge that is up from node p to
    'node s.
    E(p,s) <-- 1  'Record the existence of this link...
    E(s,p) <-- 1  '.. in both directions

    'The next thing we have to do is see if WE know (via E
    'table) about all the topology that touches us. To
    'check on this we have to see if all the entries in L()
    'are "DOWN" or "DEFINITELY UP" (i.e. check to see that
    'none of them are still "PROBABLY UP").

    'Assume that there are q links that touch our node p.
    'Hence the L() table has q entries. There are also q
    'physical edges that we refer to as LINK(1),...LINK(q).
```

```
    FOR i=1 TO q   'Run through the table
        IF L(i)="LINK IS PROBABLY UP" THEN  'If we are not
                      sure of our adjacent edges...
            GOTO BROADCAST_STAGE   'then lets get on
                                              with things

        'Since we haven't found an indecisive edge yet,
                      'try the others
    NEXT i
    'All the L table entries must be decisive, hence we do
    'know all about our local connectivity. We have to
    'record this fact in the N table.
    N(p) <-- 2   'We (node p) are sure of the local
                 'connectivity and have the appropriate
                 'entries elsewhere in N to represent
                 'this.

BROADCAST_STAGE:

    'As with all the responses, we must broadcast all the
    'information that we have to all our neighbors.

    'If we have made no changes to our tables, then the
    'following broadcasts are not necessary.

    FOR i=1 TO q  'run through all these links
        IF LINK(i) is UP THEN  'If we've been acting as if
```

```
                    'this link is up...

            SEND (SF(R,N,E)) ON LINK(i)   ]  'Broadcast
                                 'across that link

NEXT i



'There is now the possibility that all the entries in N
'are zeroes or twos. This would mean, as we will prove
'shortly, that the topology in E is totally correct
'(for our connected subnet), and the set of non-zero
'entries in N correspond exactly to the the set of
'nodes in our connected subset of the network. This
'would also mean that we have completed the algorithm.

FOR i=1 TO n    'Look through
    IF N(i)=1 THEN    'If we find a one, then they're
                        'not all 0 or 2

        GOTO CONTINUE_RUNNING
    'Otherwise, keep checking the list
NEXT i
'All the entries in N must be 0 or 2, so we terminate
'running by going into the normal mode
M <-- NORMAL
```

CONTINUE_RUNNING:

START TO PROCESS NON-SF PACKETS

CORRECTNESS PROOF OF THE SF ALGORITHM

Each of the proofs that follows will start out with some elementary discussion, and progress to the point where a series of very carefully stated theorems can be shown. We have endeavored to exercise the greatest care when a proof was complex, and the least rigor during the preliminary (motivating) discussions. We hope that this format will both prove correctness to the reader that scrutinizes the text, and at the same time explain it to the casual reader.

We will now give a proof of the correctness of the above SF algorithm. We define "correctness" to mean that all four conditions that were given at the start of this section are satisfied (re: any node(s) can start the algorithm,..., all nodes become aware in their E table of the topology of their connected subset, etc.).

It is common, in the proof of an algorithm, to show that the algorithm always terminates. We have defined "terminates", with respect to our algorithm, to mean that all nodes in the connected subset that is running the SF algorithm have returned to the normal mode (M=NORMAL). In the case of our SF algorithm, it is not possible to show that it always

terminates. The algorithm restarts a new version of itself every time there is a link level topology change. If the link level topology continues to change (rapidly) then the algorithm may never terminate. What we do expect from this algorithm is that: if the link level topology doesn't change for some "sufficient" length of time, then the algorithm will terminate.

Lemma: If a node p sends a message SF(R,-,-) on edge p-q, then either: a message of the form SF(R,-,-) will be (or was) received on edge q-p of the form SF(R,-,-) OR node p will run a version of the SF algorithm with a version number greater than R. (We will use "-" in such transmissions to mean unspecified, and of no significance.)

Proof: We start by noting that if edge p-q changes status at node p to down, then the definition of the SF algorithm requires that node p execute the "Initiate" procedure This would imply execution of version R+1 of the SF algorithm, and we would be done.

If on the other hand the edge p-q did remain up as viewed by the link protocol, then the link protocol guarantees that the message will be (or was) received, and node q can still send messages on edge q-p. We then would have the following possibilities. Node q must either be running some

non-SF algorithm, running a version R' of the SF algorithm where R'<R, running version R of the SF algorithm, or running version R''>R of the SF algorithm. In either of the first two cases, the node q will, upon receiving the message from p, immediately start running version R of the algorithm, and send to all its neighbors (including node p) the message SF(R,-,-). If a neighbor q of node p is already running version R, then q must have sent a message SF(R,-, -) to all its neighbors (including p), when it started version R. Hence in these cases, node p must eventually receive messages SF(R, -,-) from node q. Lastly, if node q is executing version R'' of the SF algorithm, then it must have sent a message of the form SF(R'',-,-) to node p. Once node p gets such a message (note that R''>R) it must execute (or be executing) a version of the S algorithm with a higher version than R.

End of proof

If we look back at the definition of the SF algorithm we notice that if some edge e is considered to be down by the underlying link protocol, at node p, when node p first starts version R, then node p will never send (and has never sent) an SF(R,-,-) message on edge e. Suppose then that we have two neighboring nodes, p and q, that have an edge e between them. Further suppose that both p and q are running the same version R, but when they each started that version they were told different things about the status of e by their respective link

protocols (p was told that e was up, but q was told that e was down). From our initial comment in this paragraph we know that node q will never send an SF(R,-,-) message over e. From the definition of the SF algorithm, we know that node p has tried (or will try) to send an SF(R,-,-) message. By applying the above lemma, we then know that node p will eventually run a higher version of the SF algorithm. We are, in our analysis, trying to focus on the version of the SF algorithm that might allow the entire SF to terminate. We would like to avoid being distracted in our proofs by nodes that cannot possibly return to the normal mode during this version (such as node p in the above example) We would also like to avoid the distraction of adjacent nodes that "claim" that they are connected (node p claims that it is connected to q in this version) to the set of nodes that we are concerned with. These "claims" can be made despite the fact (known to an omniscient observer) that they will never interact via that connection during that version (in our example above, q will not reply during this version). For this reason we will consider two nodes to be "directly version R connected" iff they have an edge between them that each node agreed was up when it began version R. We can then consider a "maximally connected subset" of nodes in the network that are all running the same version, based on this notion of "version R connectivity".

Lemma: In a maximal connected subset S of the network, if version R of the SF algorithm is running, then either the SF

algorithm will terminate, or a version of SF with a version number greater than R will be run.

Proof: By contradiction: Assume that version R was run, no greater version ever gets run in S, and the SF algorithm never terminates. Since version R was running at some point, there must have been some node(s) that started this version R. When a node p first starts running version R it sends a message SF(R, -,-) to all its neighbors.

Making use of the previous lemma and the assumption that no version greater than R is run at node p, we can conclude that node p must eventually receive messages SF(R,-,-) from all its neighbors. Note that there is no provision for any node to decrease the version number of the SF algorithm that it is running, hence node p will be running version R throughout the receipt of all these SF(R,-, -) messages. There is no facility to change a "DEFINITELY UP" value of any L() at node p to a "POSSIBLY UP", and each time SF(R,-,-) is received from a new neighbor, another element of L() is changed to a "DEFINITELY UP". Finally, when the last of node p neighbor's (that was sent the original version R message by node p) sends p an SF(R,-,-) message, the last entry of "POSSIBLY UP" in L() will be changed to a "DEFINITELY UP". With all the entries in L() at node p having decisive values, the algorithm calls for

N(p) to be given the value of 2. Node p then transmits to all its neighbors the message SF(R,N,-), with N(p) =2.

We now have that every neighbor of p will eventually receive a message SF(R,N,-) with N(p)=2. The first time each neighbor p' of p receives a message SF(R,N',-) with N'(p)=2, it must set the value of its N''(p) to 2, and must send to all its neighbors a message SF(R,N'',-), with N''(p)=2. Similarly, the neighbors of neighbors of node p must set their N'''(p) to 2, and continue the process. Eventually every node p' in S must be running version R with their N' such that N'(p)=2.

Since the above argument could have been done with any p in the set S, we can conclude that for every p and p' such that nodes p and p' are in S, we will eventually have that N(p')=2 in node p.

Now since we assumed that this algorithm did not terminate, there must be a node p' in S that is not in the normal mode at this point. For p' to not be in the normal mode, there must be an entry N'(q)=1 at node p'.Since all the entries in N' that correspond to nodes in S are 2, we know that node q is not in S. Let node p be the first node in S to have N(q)=1 during the running of version R of the SF algorithm. When node p first started running version R, it initializes all entries

of N to a value of 0 or the values supplied by some adjacent node. Since node p is the first node in S to have N(q) =1, and S is the maximal connected set, we know that N(q) was initially set to 0 at node p. There must then be a subsequent point in the algorithm that allows node p to change N(p) to 1, despite the fact that it only receives SF(R,N',-) with N'(q) =0. The only way node p can change an entry in N when it receives such messages is when it is changing N(p). Hence p must be the same as q, which contradicts the the fact that p in is S and q is not.

End of proof.

Theorem: If there are no changes in the topology according to the link protocol after time t in some maximal connected subset S of the network, and R is the largest version number being used by any node in S at time t, then the algorithm will terminate with a version number of R.

Proof: If there are no further topological changes, then there is no facility for increasing the version number (i.e. we have no reason to execute the "Initiate" procedure). Hence no greater version than R can ever be run after time t. Combining this with the previous lemma gives the desired result.

End of proof.

So we have now shown that if we stop having link level topology changes in some maximal connected set of nodes, then the SF algorithm will "eventually" terminate. It follows then that if we don't have any topological changes in a maximal connected set of nodes for a "sufficiently" long period, then the algorithm will terminate. We see that this follows as: if we wait a "sufficiently" long time without a failure or restoral, then the algorithm will have "eventually" terminated.

Now that we have some proof about the fact that the SF algorithm terminates, it is reasonable to talk about the state of the nodes when it does terminate. Hence forth, in our proof of the correctness of the definition of our SF algorithm, we will center our discussion on version R, which is active when the SF algorithm next terminates. We will also restrict our discussion to the connected set of nodes S that are running that version R. Our proof will now go through the four conditions that we stated at the start of the SF algorithm section.

The first condition stated that it must be possible to start the SF algorithm asynchronously at any node(s), and restart it at that node as a whole new version if there is any link level topology change before it terminates. Looking back

at the algorithm, it is clear that we have provided procedures that start and restart the algorithms. This claim is not a very large point, as we have not yet proved what happens when the algorithm is started.

The second point begins to address what it means to "start" the algorithm. Specifically, the second condition says that once any one node starts to run the SF algorithm, all nodes that can receive messages from p will be forced to start the SF algorithm. This feature follows directly from the definition of the SF algorithm. Whenever a node starts the SF algorithm, it sends out SF(-, -,-) messages on all its edges that are up to all of its neighbors. This message will force the neighbors that actually receive it (those that the starting node can communicate with) to run the SF algorithm (if they aren't already).

The real reason for the previous condition was to guarantee that when we apply the theorems that assume "maximal connected set S...", we will be able to prove that all nodes in S are running the algorithm. The remaining conditions deal with the performance of the algorithm after we have stopped having link level topology changes.

Condition 3, non-rigorously stated is that:

No node may stop executing the algorithm until it is aware of a jointly approved topology (E table) of the network. By "jointly approved" we mean that for every node p that is mentioned in this E table topology, there was a time (since the start of this algorithm) that node p agreed with all aspects of this table that related to node p (i.e.: what edges are adjacent to it)."

The topology that our algorithm produces is E(*,*). We have to show two things about this topology. We must show that the same topology is generated at every node, and we must show that this topology was "approved" by each node in this connected subset.

As before, we will focus attention upon version R of the SF algorithm. We start by examining the values of E in node p, right after it has completed the procedure (except possibly going into normal mode) that changed the value of N(p) to 2. (Note that node p is the first node that can change N(p) to 2. All other nodes must wait till they receive a message SF(R,N', -) with N'(p)=2.)

Since we know that the SF algorithm will terminate with

this version number, we know that there will be no actual link level topology changes around node p, from the time it begins version R, to the time it enters the normal mode. (Otherwise node p would start a new version, and prevent termination with this version.)

By looking at the procedure that changed N(p) to 2, we see that all entries in L() at node p (which correspond to the edges adjacent to node p) must have been marked either "DOWN" or "DEFINITELY UP" to allow this change. Suppose an edge e adjacent to p was up during the running, by p, of version R. We are guaranteed by the way the SF algorithm has node p start version R, that this edge was marked "POSSIBLY UP" in that start up procedure. There is no place in any of SF's procedures that a node could possibly change a "POSSIBLY UP" to a "DOWN" in the same version. Hence this edge must be now marked "DEFINITELY UP". Every time an edge is marked "DEFINITELY UP", the corresponding entry in E is changed to a 1. So we have that as a consequence of edge e being up (for the duration of node p's run of version R), there must be a corresponding entry of 1 in E here at node p.

Suppose that the entry E(p,q) (or E(q,p)) is a 1. The first node that could have had an entry of E(p,q)=1 must be either node p or node q. In either case the first node to

change E(p,q) to a 1 did so after a message of the form SF(R,-,-) was received over the p-q edge. Knowing that edge p-q was up during the running of this version, guarantees that it is still up as when it goes down a new version of SF will start. Hence if an entry in E(p,q) is 1, then the corresponding edge is currently up.

To sum up the results of the last paragraphs; Right after node p has completed (except possibly going into normal mode) the procedure that changed the value of N(p) to 2, we find that the entries in E contain the exact list of edges that are functional and adjacent to node p. We will refer to the contents of E in node p at this point in time as Ep(*,*), and we will refer to this point in time as tp.

Lemma: If in some node q (that is between executing procedures), the value of N(p) is 2, then all entries in node q of the form E(p,p') are exactly identical to the corresponding entries of Ep(p,p').

Proof: Consider some fixed p':

case 1: p=q We are basically addressing the point that here

at node p, the entry E(p,p') cannot change from when it was Ep(p,p'). Since the SF algorithm only increases the values of E during the running of a specific version, we have only to show that there is no possibility that Ep(p,p')=0 and E(p,p')=1.

If E(p,p') is 1, then there must have been a first time in the network that a node had E(p,p')=1. Since the only nodes that can change E(p,p') without copying some received value are nodes p and p', the first occurrence of E(p,p')=1 must have been at node p or node p'. In either case, it was caused by the receipt of a message of the form SF(R,-,-) on edge p-p'. From this we can conclude that nodes p and p' were directly version R connected, and hence Ep(p,p') must be 1.

case 2: p is different from q

First we will show that if Ep(p,p') is zero, then E(p, p') is zero. Using proof by contradiction, assume the contrary: Ep(p,p')=0 and E(p,p')=1. Since E(p,p')=1, there must have been a first node in the network in which E(p,p')=1, and it must have been node p or node p'. In either case this latter E(p,p') had to become one in response to a message being sent across p-p' of the form SF(R,-,-). This then implies that edge p-p' was up during this version, which contradicts the fact that Ep(p,p')=0 (which means p-p' must have been down during this version).

Next we will prove the hard part; If Ep(p,p')=1, then E(p,p') must be one. This basically guarantees that the information that is gathered and stored in node p (as Ep(p,*)) is delivered to every node before that node goes into the normal mode. We are given the key assumption above, that N(p)=2 in node q.

We start by noting that, since N(p)=2 at node q, then there must have been a message received at node q that caused N(p) to become 2. Suppose the message was SF(R, N',E'), and it was received from node q'. Notice that it must have been the case that N'(p) was two when it sent the above message.

We can now repeat the above process and find a list of nodes q,q',q'',... (with no repetitions) such that:
message SF(R,N',E') from q' caused q to make its N(p)=2,
message SF(R,N'',E'') from q'' caused q' to make its N(p)=2

.

.

.

message SF(R,Np,Ep) from p caused q''''...' to make its N(p)=2

The reason why the list must end with a message from node p, is that p does not require a message with N(p)=2 to make its N(p)

into 2. Moreover, since node p is the only node that the SF algorithm allows to change N(p) without receiving a message with N'(p)=2, all other nodes must rely on some such sequence of messages to eventually get N(p)=2. The reason why we know the exact message that node p sent to start this process is that right after node p changed N(p) to two, it sent out the above message to all its neighbors (so this had to be the first time they got an N(p)=2), and then we recorded (defined) Ep as the contents of E in P at that point.

Having formed the above list, we note that by virtue of the way that a received SF(R,-,E) is used to modify the E table at the receiving node (take the MAX of the received E and the current E) we have the following:

message SF(R,Np,Ep) from p caused q''...' to make its E(p,p')=1

.

.

.

message SF(R,N'',E'') from q'' caused q' to make its E(p,p')=1

message SF(R,N',E') from q' caused q to make its E(p,p')=1

Hence E(p,p') at node q must be one, which is the desired result.

End of proof

Theorem: If two nodes q and q' (elements of S) enter the normal state in version R, then every entry in their E tables are identical.

Proof: Consider any entry E(p,p') in node q, and E'(p,p') in q'. Since both nodes have entered the normal mode, we know N(p) =2 in both nodes. By the above lemma, we know that E(p,p')= Ep(p,p'), and E'(p,p')=Ep(p, p'). By transitivity we then have E(p,p')=E'(p,p').

End of proof

We have now shown that all nodes that enter the normal mode in version R have the same exact E table. We must now show that this E table was locally correct with respect to any node p (in S) at some point during the running of the algorithm.

The time that that each node p agrees with all of its local topology in THE final E is what we defined as tp (the time right after node p changed N(p) to 2). We can recall from our discussion of tp that the values of Ep (the E table at node p at time p) contained exactly the correct entries to represent all the local link level topology of node p, at that time. The last lemma tells us that that any final E table (all entries in

N are 2, and this includes N(p)) agrees at all E table entries of the form E(p,p'), with Ep(p,p'). If we add to this the fact that every E table in every SF algorithm is symmetric (E(p,q) =E(q,p)), then we are done.

Finally, to complete this correctness proof, we must show condition 4 is true of our SF algorithm. This condition states that when any node stops running the algorithm, then all of its neighbors immediately there after" also stop running the SF algorithm also (unless they are starting a new version).

We can prove this fact about the SF algorithm rather directly. First we note that we define "stop" to mean "change to normal mode" (M=NORMAL). When ever a node changes to the normal mode, all of its entries in N' must be 2. The last thing that a node p does before it returns to normal mode is to send out an SF message packet to all of its neighbors that is of the form SF(R',N',-). We should recall from our network model, that all packets are received in the order in which they are transmitted. Hence no data packet sent out by node p can be received before they receive this SF message packet.

Notice that the only way that a neighbor can ignore this SF(R',N',-) is if it is running a newer version of the SF

algorithm with a greater version number, and hence it will soon start node p into a new version. Assuming that some neighboring node q is not running a greater version number, it must run the appropriate procedure upon receipt of SF(R',N,-). In all cases, this involves having the node q form a new N table by taking the maximum value of the received N'(i,j) with its internal N(i,j). Since the largest possible value of elements in N is 2, we know that it will end up with 2 as every entry in its N table. From the definition of the SF algorithm, it then follows that the node q will end this procedure by entering the normal mode ("stopping").

We have now shown that all the conditions that we required for a "sure fire" algorithm, are satisfied by our definition of our SF algorithm.

MAKING USE OF THE SPANNING TREES

We assume that there are k spanning trees that are numbered T(1),...T(k). Since all the nodes identified all these trees independently, but using the same algorithm and data (E table topology generated by the SF algorithm), we can assume that all the nodes have the same set of spanning trees, numbered in the same way.

In the SF algorithm, there was no presumed knowledge of the network, and the major mode of communicating in the network was flooding. In a network with n nodes and m edges, it requires between m and 2m transmissions to get a message to all nodes via flooding. By making use of a spanning tree the same message can be delivered to all nodes in exactly n-1 transmissions. These sorts of differences are fundamental to the communications savings that we will realize when we make use of the spanning trees that we have on hand. In addition, it will not be necessary to shut down the network when the spanning tree based algorithm is running, and yet arbitrary topological changes can be communicated.

Now we will turn our attention to the actual use of the k spanning trees to disseminate information on link level topology changes. We will call this algorithm the k Spanning Tree Recovery Algorithm (kSTRA). We start by examining the different parts of the "sure fire" algorithm that we have described. The following parts are intertwined in the algorithm, but separating them out will lead us to an algorithm that makes use of the spanning tree structures that we have identified.

Roughly speaking, the SF algorithm starts out with each node having its own view of the network. The first thing that

happens is that all the nodes are told that their old view of the topology is very bad, they should discard all the old information about the topology, and stop processing packets of data. All nodes are required to stop what they are doing, and begin to run this algorithm. Notice that if the old topology information that the nodes had was not so terrible, they could continue to route packets using it.

The next thing that happens is that all nodes tell all their neighbors everything that they know about the network. By transitivity, eventually every node knows everything about the network's link level topology. There is also a facility in the SF algorithm to allow each node to realize when it has received (although indirectly) a report from every other node. This way the algorithm can assure each individual node when THAT PARTICULAR node knows the entire link level topology of the network.

When a single node is totally sure that it knows everything about the link level topology of the network, it is almost ready to start to route packets. The key step that precedes this routing is that this totally informed node tells all of its neighbors everything that it knows about the network. The basic problem here is that this knowledgeable node doesn't want to be sending packets to unknowledgeable nodes, as

they wouldn't know what to do with them. This well informed node could wait till some other protocol guaranteed that all nodes were well informed before it began to route packets, but the SF algorithm we described is trying not to waste any time. In the interest of this time savings, the entire E table topology is retransmitted over every link twice, once in each direction as each node realizes that it is well informed.

Finally now, the nodes begin routing packets. These packets are sent with the confidence that when they are received by the adjacent nodes, they will be routed in accordance with the mutually known E table topology. The first packet that is sent over each link might as well have a header which reads: "Use the new E table topology to route this packet", as that was what the last SF(*,*,*) message had (it contained all the E table topology, and the receiving node figured this out).

Having broken down the "sure fire" algorithm into its parts, we can discuss the similarities with the task of disseminating failure information. The big differences that we wish to achieve in the kSTRA are:

1) Packets should continue to be routed while the

algorithm is running. These packets should continue using the existing E table (which is known to have been consistently formed at every node) until a new E table topology can be calculated. Note that the nodes then perform their routing based upon this table, and not on any other information (re: failures) that they might deduce had occurred. (Obviously we don't however send packets over failed edges.)

2) Spanning trees should be used instead of flooding to carry information between nodes (when correctness of the protocol will allow).

3) Rather than transmitting the whole E table topology between nodes, only the changes from the previously agreed upon E table topology will be sent. (minimize communication)

When we begin the task of spreading information about a failure, each of the nodes has its own view (E table) of the network's topology. For the most part, nodes in the network have a fairly accurate view of the network's topology, and we don't want them to throw away that information. We would like to inform all the nodes of the change in the link level topology, and then try to "all at once" have all nodes in the network start using the new (more correct) E table topology as their basis for routing. Hence, all nodes will maintain a

topology (E table) that was originally generated by the SF algorithm, while the kSTRA is running, making no change in this topology until a special CHANGE message is received.

We start by looking back at all the pieces of the algorithm that we said constituted the "sure fire" algorithm. The first step is to initiate the algorithm, and be sure that all nodes are involved. The SF algorithm used flooding to accomplish this startup, the kSTRA can certainly use a spanning tree to get every node involved. The spanning tree that is used is selected from the list of precomputed ones. Specifically, it is the least numbered tree that is intact (at least within the knowledge of the given node). Any node may restart the algorithm on a higher numbered tree if it knows that the spanning tree that has been suggested is not intact. If none of the trees are intact, then the SF algorithm is initiated. As with the SF algorithm, version numbers are maintained, and a node may reinitiate a new version of the algorithm if it is necessary. It should be noted that during the running of the kSTRA algorithm, all nodes will continue to route using the old E table topology. In this way, we defer to the final point in the kSTRA the synchronized change to the use of the new E table topology by all nodes.

The notification that a new version of the kSTRA is

running is broadcast out across a spanning tree along with a list of E table topology changes that are required by the originating node to get the E table to conform to the known link level topology. In this way, all the nodes are not only told that the algorithm is running, but they are told the E table topology modifications that go along with this version. If a node must add anything to the E table topology changes that are on the list, it must reinitiate a new version of the kSTRA with all the things on the old list, and any additions that it has to make. Hence, a version of the kSTRA is eventually formed that has all the E table topology changes that are required. Note that if two copies of the same version of kSTRA are formed with different lists of changes, some node will eventually hear about this, and start a new version with all the changes from both the former lists.

At this point in the algorithm, some of the nodes might have received the list of E table topology changes, and some might not have. Before we can make the transition to using the new E table topology, all nodes must be aware of all updates. Since we are using a spanning tree of the network, we have a convenient way to verify that all the nodes have this new version of the kSTRA. The broadcast of the start of this specific version of the kSTRA continues across the tree until it reaches a leaf of the tree (a node that has only one edge in this tree). This leaf node then responds to the node that told

it about this version with an acknowledgement of that specific version. Similarly, nodes that have received acknowledgements for a specific version from all but one of their tree edges, transmit an acknowledgement for that version across that lone edge. This process repeats itself until these acknowledgements all collect at some node. That is, some node receives acknowledgements from all its tree neighbors for a specific version of kSTRA. Any node that receives acknowledgements from all its tree neighbors is then sure that every node in the network has heard the list of changes involved in this version of kSTRA.

Now the moment comes when nodes are about to use their new E table topology information. Once any node is sure that all nodes in the network have heard about all the changes in this version of kSTRA, we are ready to proceed. What we must do now is have a synchronized change in the E table topology used as each node's basis for routing.

As we said in the preface to this chapter, the change in E table topology used for routing should be very nearly "all at once". As with the "sure fire" algorithm, we wish to prevent any node from transmitting a packet to another node, and having the other node not understand why the packet was sent. The SF algorithm flooded the network with an SF message that explains

the entire E table topology to any node that receives it. A flood was sufficient as all packets are received across edges in the same order as they were transmitted. Any method other than using a flood would allow packets that are routed using one E table topology to arrive at neighboring nodes before the neighbors are aware that this new E table topology is in service. We are in the position that every node is definitely aware of this new E table topology, and all we have to say is "Use the new topology". The final step in this algorithm is then the flooding of the network with the statement "Use the new E table topology specified in such and such a version of the kSTRA". This flooding (across all edges) can be initiated by any node that has received acknowledgement messages from all of its tree neighbors. By waiting for this point to perform the flooding operation, we have also succeeded in producing a very small amount of information that needs to be flooded across the network.

As a final point, we note that after the "Use the new E table topology as given in version of kSTRA" has started its flood, all the nodes in the subset of the network that are currently connected to the node that originated the flood will soon be using the new topology. Hence, any later changes in topology that are instigated by future versions of the kSTRA can carry lists of changes from this new E table topology. Care must be taken to express which E table topology the list of

changes modifies, but that information can be carried in the packet with the list.

Having given a casual description of the kSTRA algorithm, we will now give a careful and detailed description of the tasks involved in running it. We start by defining the variables that the algorithm will maintain at each node p:

Variables used by kSTRA

R the version number of the last known SF algorithm

E(i,j) (i and j=1,...n) The currently used E table topology, which was initially set by version R of the SF algorithm.

ECOUNT An integer 0,1,2,.... that tells where E() came from. If ECOUNT is 0, then E() is the E table topology generated by the SF algorithm. If ECOUNT > 0, then ECOUNT is the number of times that the E table topology has changed from the one generated by the SF algorithm.

V the version number of the kSTRA that was (is) most recently run(ning).

U(i,j) (i and j=1,...n) A list of updates to some E table topology. For every possible edge in the network it has an entry of "DON'T CHANGE", "ADD THIS EDGE", or "REMOVE THIS EDGE".

HIST() An array that records the list U that was associated with each version of kSTRA that we have heard about and acknowledged. (We will explain later how we can throw much of this HISTory away.)

T(i) (i=1,...k) A list of k edge disjoint spanning trees, every edge is in the current E table topology E().

TREE An integer from 1 to n, that points to the least numbered tree that we can use the kSTRA on (due to failures, not all the spanning trees are still intact).

LINK(i) i=1,...q, assuming that there are q edges in T(TREE) that are adjacent to node p. This array contains the "names" of these tree edges that this node p is adjacent to.

L(i) i=1,...q A list of logical variables that are used to record what is "going on" over each of the above

LINK(i)'s, in terms of the progress of the kSTRA. 0 means that this node has broadcast the current version of kSTRA over the edge, 1 means that it has received an acknowledgement of the current version.

Before we describe the details of the kSTRA algorithm, we must take care of initializing variables that will be used by the algorithm. The simplest place to put these initialization statements is in the SF algorithm. Specifically, they are placed just before the statement that assigns NORMAL to M (the mode). By inserting the following statements at that point in the SF algorithm, we guarantee that every node in the (connected subset) of the network has the same initial conditions.

```
INITIALIZE kSTRA
at each node p:

        V <-- 0 'Start the system as if version 0 of kSTRA
                        'just ran

        'Use the topology to generate a set of
        'edge-disjoint trees.
        GENERATE ( T(1),... T(k) ) USING E(*,*)

        'We assume that tree 1 can be used by kSTRA
```

```
TREE <-- 1


'Its nice to have the list of links that we are
'adjacent to in tree T(TREE) on hand at all times. This
'variable makes it easier to describe the algorithm.
LINK() <-- links of T(TREE) adjacent to node p


'Remember that the current topology in E(*,*) was
'produced by the SF algorithm.
ECOUNT <-- 0


'Initially, there are no changes desired in the
'topology E(*,*). We start U(*,*) to indicate this fact
FOR i=1 TO n
    FOR j=1 TO n
        U(i,j) <-- "DON'T CHANGE"  'There are no
                                   'changes desired in the
                                   'current topology.
    NEXT j
NEXT i
```

At this point we will mention the other interface between the kSTRA and the SF algorithm. We recall that the SF algorithm was previously allowed to start at any node if there was a local link level topology change at that node. At this point we remove that restriction, and replace it with the restriction that nodes may only execute the "Initiate" SF

procedure if the kSTRA requests it. The correctness of all the theorems will be guaranteed by the fact that the kSTRA will only be run if some node has a local failure. Since that local failure gave free license to that node to start the SF algorithm (under the definition of the SF), it is in keeping with the proven structure that the SF algorithm may be later initiated upon the request of the kSTRA.

As with the SF algorithm, we start by describing the procedure that a node p would execute to instigate the running of a new version of the kSTRA, based solely on a change in local link level topology. To prevent confusion in the algorithm, it will not be possible for a single link to be brought down and up in a single version of the kSTRA. To settle all such disputes, whenever a request to add an edge is made in the same list as a request to remove it, only the request to remove the edge will be entered into the new update request list U(*,*). Later, when a new E is formed, each node will have a chance to update that E table to bring up an edge that it could not (by the above convention) do as part of this update. At this point we will only discuss link level topology changes that are totally within the set of nodes that are reachable under the current topology E(*,*) (We will avoid the problem of bringing up an edge that connects to an unreachable node by assuming that the SF algorithm will be started if such a link level topology change takes place.) With this convention, we

describe the instigating procedure performed by a node p, when it decides that the edge from node p to node p' should change status. (i.e. The underlying link protocol tells us when this is the case.)

Interrupt received from link level protocol
Execute at node p

```
'This procedure is executed because the status of some
'edge (p,p') has changed. Let STATUS = 1 if we are
'running this code because (p,p') has just come up, 2
'if it has just gone down.

'First see if the current update table contains this
'topology change.
IF U(p,p')="ADD THIS EDGE" AND STATUS=1
                OR
   U(p,p')="REMOVE THIS EDGE" AND STATUS=2
                THEN   'If we already have exactly this
                              'update...
         GOTO ALL_DONE  'then there's no work to do

'There is also the chance that the update table says to
'remove the edge, but we wanted to bring the edge up
'(STATUS=1). The result is that this version of the
'kSTRA will only bring the edge down (by leaving the
'update table unchanged), in keeping with the above
```

```
'convention.
IF U(p,p')="REMOVE THIS EDGE" AND STATUS=1
                THEN
        GOTO_ALL_DONE  'then there's no work to do

'We want to change the update list, so start a
'new version
V <-- V+1  'Increment the version number

'Lets add our change to the update table
IF STATUS=1 THEN
        [
        U(p,p') <-- "ADD THIS EDGE"
        U(p',p) <-- "ADD THIS EDGE"
        ]
    ELSE
        [
        U(p,p') <-- "REMOVE THIS EDGE"
        U(p',p) <-- "REMOVE THIS EDGE"
        ]

'Now we have to make sure that there is still a good
'spanning tree
IF STATUS=1 THEN  'We couldn't hurt T(TREE) by
                        'suggesting that a new edge be
                        'added to the topology..
        GOTO TREE_OKAY  'so get on with things
```

```
    'We are asking that an edge be taken down, so see if
    'its in the current spanning tree T(TREE).
    IF edge(p,p') is not in T(TREE) THEN  'If its not...
            GOTO TREE_OKAY  'then get on with things.

    'The spanning tree T(TREE) is no longer intact, so we
    'have to look for a new one (in our list), that is
    'intact even with all the suggestions made in the
    'update list.

NEXT_TREE:
    TREE <-- TREE+1  'Try the next larger tree
    IF TREE > k THEN  'We only have k spanning trees,
         INITIATE NEW VERSION of SF algorithm  'once they're
                     'all broken, use the sure fire
                     'algorithm.

SPECIAL_ENTRY_POINT:
   'Now we have to test to see if any update clobbers
   'this tree
   FOR i = 1 TO n
        FOR j=1 TO n   'Run through all the updates
            IF U(i,j)="REMOVE THIS EDGE"
                THEN  'if the update brings down a link..
                [            'See if the tree uses it..
                IF edge(i,j) is in T(TREE)
```

```
                    THEN  'If it is...
                    GOTO NEXT_TREE  'Try another tree
                ]
        NEXT j
    NEXT i
    'Since we changed which spanning tree we are going to
    'use, we have to reload this list of adjacent edges
    LINK() <-- links of T(TREE) adjacent to node p


TREE_OKAY:
    'Well, none of the updates we know about damage this
    'spanning tree T(TREE).

    'Since we are starting a new version of kSTRA, we have
    'to record the fact that we haven't heard any
    'acknowledgements on any of the edges in the tree.

    'Assume that there are q edges that are adjacent to
    'this node in tree T(TREE)..
    FOR i=1 TO q  'Across every adjacent link in the tree..
        TRANSMIT( kSTRA(ECOUNT,V,TREE,U) ) ON LINK(i)
                                '....the information we have.
        L(i) <-- 0  'No acknowledgements yet
    NEXT i

ALL_DONE:
```

CONTINUE NORMAL PROCESSING UNTIL NEXT kSTRA MESSAGE ARRIVES

Now that we have described how a node instigates the running of a new version of the kSTRA, we have to describe how a node responds to a message of the form that we started sending across the tree. The case that we then discuss occurs when an arbitrary node p receives a message of the form kSTRA(ECOUNT',V',TREE',U') from some node p'.

First we notice that node p may be in NORMAL mode, or in RESYNCH mode (in terms of the SF variables). If a node receives the above kSTRA packet, and it is in the RESYNCH mode, then a new version of the SF algorithm is starting, and the kSTRA packet can be ignored. This follows from the fact that node p' was not in the RESYNCH mode when it sent the packet (a node in RESYNCH mode is not allowed to do anything but run the SF algorithm). Since node p' was not in the RESYNCH mode, the last thing it did before entering the NORMAL mode, was to send a packet to p, that would force p to enter the NORMAL mode UNLESS p WAS STARTING A NEW VERSION OF THE SF ALGORITHM.

Thus there remains the possibility that node p is in the NORMAL mode, which we will assume from here on.

If ECOUNT' is less than ECOUNT (ECOUNT is the variable that is kept in node p), then this kSTRA message pertains to an E table topology that is no longer being used at node p, and should be ignored.

It is not possible for ECOUNT' to be greater than ECOUNT. The last thing that node p' did before it incremented ECOUNT' to its present value was to send a message to node p that would bring ECOUNT up to the same value. Hence we have only to give further concern to the case of ECOUNT=ECOUNT'.

There are several possibilities for the way that V compares to V'. If V' is less than V, then this packet pertains to a kSTRA that has been superseded, and the packet can be ignored. If V<V', then node p must do work that is quite similar to the work that is involved in initiating a kSTRA. If the versions numbers agree, then we have only to verify that these are part of the same exact version of the kSTRA, and not different versions that started up with identical version numbers (update lists should be identical). The following procedure performs these tasks

node p receives kSTRA(ECOUNT',V',TREE',U') from node p'

V<V' or V=V' ; ECOUNT=ECOUNT'

Execute at node p:

```
    'separate the cases where V=V' and V<V'
    IF V=V' THEN  'When the versions agree..
      [
      'Check to make sure that both update tables
          'are identical
      FOR i=1 TO n
          FOR j=1 TO n
              IF U(i,j) differs from U'(i,j)
                  THEN  'If they disagree
                  GOTO NEW_VERSION
          NEXT j
      NEXT i   'Finish the table
      'Since both update tables are the same, we have
          'already broadcast this exact kSTRA....
      GOTO END_OF_PROC   'and we're done
      ]

   'Now for V<V'...

   'Make sure that all the changes that we had in our
   'update table are included in the update list that we
   'received.
   FOR i=1 to n
       FOR j=1 to n
```

```
        IF U'(i,j)="DON'T CHANGE"
                AND
            U(i,j) is not"DON'T CHANGE"
                THEN  'if we knew something that
                                    'node p' didn't...
                GOTO NEW_VERSION  'start a new version
        IF U(i,j)="REMOVE THIS EDGE"
                AND
            U'(i,j) is not"REMOVE THIS EDGE"
                THEN  'again
                GOTO NEW_VERSION  'start a new version

    NEXT j
NEXT i

'We just have to continue the version we were
'told about, so..
V <-- V'

'The table U' has all the updates that we had, and
'probably more, so load this update list into our table
FOR i=1 to n
    FOR j=1 to n
        U(i,j) <-- U'(i,j)  'load our table
    NEXT j
NEXT i
'
```

```
        'Since someone who started this version tested out the
        'fact that the tree T(TREE) is still intact with
        'updates U', all we have to do is load it.
        TREE <-- TREE'

        'We've modernized our list of updates, so we can jump
        'to the point where we set up LINK(), and continue the
        'broadcast.
        GOTO PREPARE_TO_BROADCAST

NEW_VERSION:
        'We knew about some updates that weren't in U', so we
        'should start a new version of the kSTRA with the
        'combination of all the updates.

        'The new version is just one larger than the version we
        'received.
        V <-- V'+1

        'remember that the combination of bringing an edge up
        'and down in the same version implies that we should
        'just bring down the edge.
        FOR i=1 to n
            FOR j=1 to n
                        'load our table with the combination of
                        'the updates in our old table, and
                        'those in U'.
```

```
            IF U'(i,j)="REMOVE THIS EDGE"
                THEN U(i,j) <-- "REMOVE THIS EDGE"
            IF U(i,j) is not "REMOVE THIS EDGE"
                    AND
                    U'(i,j)="ADD THIS EDGE"
                THEN U(i,j) <-- "ADD THIS EDGE"
        NEXT j
    NEXT i
    '
    'Now we have to make sure that there is still a good
    'spanning tree. We do this by using the smallest
    'spanning tree that might be intact under the above
    'updates.
    TREE <-- MAXIMUM(TREE,TREE')
    '...and checking that it is not damaged by any of the updates
    GOTO CHECK_TREE

NEXT_TREE:
    TREE <-- TREE+1   'Try the next larger tree
    IF TREE > k THEN   'We only have k spanning trees,
        INITIATE NEW VERSION of SF algorithm  'once they're
                                   'all broken, use the sure fire
                                   'algorithm.

CHECK_TREE:
    'Now we have to test to see if any update clobbers
    'this tree
```

```
FOR i = 1 TO n
     FOR j=1 TO n    'Run through all the updates
          IF U(i,j)="REMOVE THIS EDGE"
               THEN  'if the update brings down a link..
               [             'See if the tree uses it..
               IF edge(i,j) is in T(TREE)
                    THEN  'If it is...
                    GOTO NEXT_TREE  'Try another tree
               ]
     NEXT j
NEXT i


PREPARE_TO_BROADCAST:
    'Since we changed which spanning tree we are going to
    'use, we have to reload this list of adjacent edges
    LINK() <-- links of T(TREE) adjacent to node p

    'Since we are starting a new version of kSTRA, we have
    'to record the fact that we haven't heard any
    'acknowledgements on any of the edges in the tree. Do
    'this as we broadcast across each edge.

    'Now we can finally start to broadcast this new version
    'across the spanning tree T(TREE).

    'Assume that there are q edges that are adjacent to
    'this node in tree T(TREE).
```

```
'Our job is to notify all of our tree neighbors
'of this version.

'If we haven't started a new version, then we don't need
'to send the broadcast back to node p', but things
'still work if we do send it..

FOR i=1 TO q  'Across every adjacent link in the tree..
    TRANSMIT( kSTRA(ECOUNT,V,TREE,U) ) ON LINK(i)
                'Send notification of the new version.
    L(i) <-- 0  'We have received no acknowledgements.
NEXT i

'Next we have to check to see if we are a leaf of the
'tree. A leaf has the property that it has only one
'neighbor in the tree.
IF q=1 THEN   '...and if we are a leaf, then we start
               'the acknowledgements flowing back
    [
    SEND (ACK_kSTRA(ECOUNT,V)) ON LINK(1)  'The ack
                              'goes to our one neighbor.
    HIST(V) <-- U(*,*)  'remember this update list, we
               'may end up implementing it later.
    ]

END_OF_PROC:
```

CONTINUE NORMAL PROCESSING UNTIL NEXT kSTRA MESSAGE
ARRIVES

We have now described all the parts of the kSTRA that involve the sending of packets containing the update list. The above procedures are then the procedures that guarantee that all nodes hear about a comprehensive update list. Now we will describe how the acknowledgements will propagate back through the tree. The purpose of this section is to guarantee that at least one node can "figure out" that all nodes in the network have been appraised of a specific update list. Once that node draws this conclusion, the final flood stage of the algorithm will commence. We will first describe the procedure by which acknowledgements propagate back from the leaves of the network.

So far the only time that a node has sent an acknowledgement has been when the node is a leaf of a tree. In order to get the acknowledgements to propagate back to some central node in the tree we have to have nodes pass along acknowledgements. The specific rule for passing along these acknowledgements is that whenever all but one of a node's tree neighbors has sent in an acknowledgement, the node sends an acknowledgement to its remaining tree neighbor. With these rules used, a node that receives an acknowledgement from its neighbor can deduce that all nodes beyond the transmitting node

in the tree have heard about this version of the kSTRA. This deduction is also the justification for sending an acknowledgement out when we have received acknowledgements from all but one of our tree neighbors. It follows also then that if a node receives an acknowledgement from every one of its tree neighbors, it can deduce that every node in the network knows about this version of the kSTRA.

As with each of the procedures that we have described up to now, we must discuss the response to various ACK_kSTRA(ECOUNT',V') at node p, depending on the values on ECOUNT' and V'. If ECOUNT>ECOUNT', then this acknowledgement pertains to an outdated E table topology, and can be ignored. For the same reason as was given at the start of the previous procedure, ECOUNT' cannot be larger than ECOUNT.

If V>V', then node p has knowledge of a more recent kSTRA, and simply ignores the message. It cannot be the case that V' is greater than V. To see this, we recall that part of the procedure that node p' (which sent the acknowledgement) executed as it started to run version V' included informing all of its tree neighbors of this new version (except possibly for a tree neighbor that informed p' of ALL the details of this version). Hence either p was already notified by p' of all the details of this version V', or p told p' all the details of the

version V'. In either case, node p must then have a version number of at least V'. Hence not only does V=V', but these version numbers refer to exactly the same version of the kSTRA (ie.: they agree as to the contents of the update list that this version number refers to). We can also conclude that this acknowledgement arrives on some edge that is in T(TREE) at node p, and we can refer to this edge as LINK(t).

The only response to an acknowledgement that we have to describe is what to do when we receive an acknowledgement with exactly the same version numbers as we are maintaining.

node p receives ACK_kSTRA(ECOUNT',V') from node p' on LINK(t)
V=V' and ECOUNT=ECOUNT'
Node p executes:

```
'We start out by recording the fact that we have
'received an acknowledgement over this edge.
LINK(t) <-- 1

'Next we have to count up the number of
'acknowledgements that we have gotten so far.
ACK_COUNT <-- 0  'initialize the counter
FOR i=1 TO q   'there are q adjacent edges in T(TREE)
    IF LINK(i)=1
```

```
        THEN  'If we've gotten an acknowledgement...
        ACK_COUNT <-- 1+ACK_COUNT  'Increment
                                   'the counter
NEXT i


'Check to see if we are missing
'several acknowledgements..
IF ACK_COUNT < q-1 THEN   '..and if we are..
    GOTO END_ACK_PROC  'don't send out anything


IF ACK_COUNT = q-1
    THEN  'If we're missing exactly one, then
    [
    'find out which link hasn't carried an acknowledgement
    FOR i=1 TO q
        IF LINK(i)=0
            THEN  '0 means that there hasn't been one
            [
            SEND (ACK_kSTRA(ECOUNT,V)) ON LINK(i)
                        'send the ack on this edge...
            HIST(V) <-- U(*,*)  'Remember this update
                        'list, we may end up
                        'implementing it later.
            GOTO END_ACK_PROC  'we're done
    NEXT i
    'The above loop will never fall through
        'into this statement
```

```
        ]

    'It must be the case that ACK_COUNT is q. This means
    'that all the nodes have heard about the details of
    'this version of the kSTRA. Hence it is time to
    'broadcast the message to all nodes that they should
    'start using the new E table topology.

    'Since what we must do here is identical to what a node
    'must do when it receives a CHANGE_kSTRA message, we
    'will just reference that part of the algorithm (which
    'will be the next section).
    HIST(V) <-- U(*,*)
    EXECUTE code for receipt of:
            CHANGE_kSTRA(ECOUNT,V)

END_ACK_PROC:
    CONTINUE NORMAL PROCESSING UNTIL NEXT kSTRA MESSAGE
                                             ARRIVES
```

As the last bit of this algorithm we will define what a node does when it receives a CHANGE_kSTRA(ECOUNT',V') message. This message is intended to flood throughout the network across every edge. The purpose of the message is to get the receiving node to change the value of E that it is using, by updating E with the update list contained in version V'.

As in each of the previous procedures, if ECOUNT'<ECOUNT, then it should be ignored, and it is not possible for ECOUNT' to be greater than ECOUNT.

The first node to cause execution of this code must have had evidence (acknowledgements from all its neighbors) that all nodes have heard about this specific version V' of the kSTRA. Hence we are sure that V is greater than or equal to V'. Also this node must have sent an ACK_kSTRA(ECOUNT',V'), and so HIST(V') must contain the update list for version V'.

Node p receives CHANGE_kSTRA(ECOUNT',V') from node p'

ECOUNT=ECOUNT'

V'=< V

HIST(V') has an update list for this version

Node p executes:

```
        'First we should continue the flood of the message
        SEND CHANGE_kSTRA(ECOUNT,V) on ALL adjacent links

        'Since we are going to have a whole new E table
        'topology, we might as well restart the version numbers
        V <-- 0 'Start the system as if_version 0 of
                        'kSTRA just ran
```

```
'Now we have to update the E table topology.
'First get back that update list that we saved.
U(*,*) <-- HIST(V')


'Now run through the entire topology and make the
'changes
FOR i=1 TO n
    FOR j=1 TO n
        IF U(i,j) is not "DON'T CHANGE"
            THEN   'if there might be a change here..
            [
            IF U(i,j)="ADD THIS EDGE"
                THEN  'We should this edge to E
                E(i,j) <-- 1
            ELSE              'We should remove this...
                                   '... edge from E
                E(i,j) <-- 0
            ]
    NEXT j
NEXT i


'Use the topology to generate a set of
'edge-disjoint trees.
GENERATE ( T(1),... T(k) ) USING E(*,*)


'We assume that tree 1 can be used by kSTRA
```

```
TREE <-- 1

LINK() <-- links of T(TREE) adjacent to node p

'Since this is an E table topology change, increment the
'counter that keeps track of the number of changes
'since the SF algorithm ran.
ECOUNT <-- ECOUNT+1

'Initially, there are no changes desired in the new E
'table topology E(*,*). We start U(*,*) to indicate
'this fact
FOR i=1 TO n
    FOR j=1 TO n
        U(i,j) <-- "DON'T CHANGE"  'There are no
                                   'changes desired in the current
                                   'topology.
    NEXT j
NEXT i

'We don't need any of that old HISTory, as it all
'referred to changes in the old topology
CLEAR HIST()

'As a last point, we have to check to see that the E
'table has exactly the edges that the lower level
'link protocol says it should.
```

```
VALID <-- "YES"  'Assume that the E table is valid
FOR i=1 TO n
     IF E(p,i)=1 AND our edge to node i is down
          THEN [    'E says the edge is up, but its
                         'really down
               U(i,p) <-- "REMOVE THIS EDGE"
               U(p,i) <-- "REMOVE THIS EDGE"
               VALID <-- "NO"  'Remember that we have
                         'updates
               ]
     IF E(p,i)=0 AND our edge to node i is up
          THEN [    'E says the edge is down, but its
                         'really up
               U(i,p) <-- "ADD THIS EDGE"
               U(p,i) <-- "ADD THIS EDGE"
               VALID <-- "NO"  'Remember that we have
                         'updates
               ]
NEXT i


IF VALID="NO" THEN  'The E table isn't yet right so..
          GOTO SPECIAL_ENTRY POINT  'in INITIATE_kSTRA
                    'to start up a new version


CONTINUE NORMAL PROCESSING UNTIL NEXT kSTRA MESSAGE
                                        ARRIVES
```

CORRECTNESS PROOF OF THE kSTRA

The conditions that we wish to show that the kSTRA algorithm satisfies are identical to those that we wanted for the SF algorithm. The only differences between them is that the kSTRA algorithm requires a certain amount of knowledge about the network, and as a result of this knowledge the kSTRA is able to complete its task (most of the time) using much less of the network's resources (communication on edges of the network).

As with the SF algorithm, we start by addressing the question of whether the kSTRA terminates. We define "terminates", with respect to the kSTRA, to be reaching the state where either: a new version of SF has been started, -OR- ECOUNT is the same for all nodes in the network and version 0 of the kSTRA is active at every node.(i.e: There are no suggestions yet to change the current E table topology.) The second termination possibility condition really means that the algorithm succeeded in its job, and the first condition means that the kSTRA "gave up" and called on the SF algorithm. As with the SF algorithm, it is impossible to show that the kSTRA algorithm will always terminate. We can however, as we did with the SF algorithm, guarantee that the algorithm will always terminate if no failures or restorals occur for a sufficiently

long time.

The proof will have three basic parts. In the first part we will show that all the nodes that were connected in the original E table (that the SF algorithm produced) will start out at some fixed time with identical values of ECOUNT and E(*, *). All of our references to the network henceforth in our proof will be restricted to the connected nodes in this original E table. Then we will show that if the kSTRA is run (and the SF algorithm doesn't take over) then every time the ECOUNT variable is incremented, it is incremented at every node, and again there is a point in time when all nodes have identical values of ECOUNT and E(*,*) . Finally we will show that if no link level topology changes occur for a sufficient period of time then a point in time will come such that all values of ECOUNT and E(*,*) at all nodes will agree, and the topology represented by E will be the exact link level topology of the network.

Theorem: If the SF algorithm terminates at time t, then the value of of ECOUNT at time t at all nodes in the network is 0, also at time t the E tables at all nodes in the network are identical.

Proof: When each node enters the normal mode, the initializing procedure of the kSTRA sets the value of ECOUNT to 0. We also know that at the time when each node p enters the normal mode (and the SF algorithm is going to terminate at this version R), the E table at node p is set to some value Ef (Ef is the same for all nodes). What we must show is that from the time that each individual node enters the normal mode to the time when all the nodes have entered the normal mode (definition of SF algorithm terminates) that the values of ECOUNT and E cannot change. The only procedure that can change ECOUNT or E (while the kSTRA is in the normal mode) is the one that sends and responds to CHANGE_kSTRA messages. This procedure can only run after ALL nodes have acknowledged some version of the kSTRA. Each node can only run the kSTRA algorithm after that node is in the normal mode. Hence a CHANGE_kSTRA message can only be sent after all nodes are in the normal mode, and ECOUNT and E at each node can only change after ALL the nodes are in the normal mode.

End of proof

For the following definitions, we will equate the following entries in an update list with the following numbers:

"DON'T CHANGE" <----> 0

"ADD THIS EDGE" <----> 1

"DELETE THIS EDGE" <----> 2

Definition: We say that update list U(*,*) is less than U'(*,*) iff:

1) for every i,j in {1,...n}

U(i,j) is less than or equal to U'(i,j)

AND

2) for some i,j in {1,...n}

U(i,j) is strictly less then U'(i,j)

Definition: We define SUM(U(*,*)) to be the sum of the elements of update list U(*,*).

Lemma: If U(*,*) is less than U'(*,*) then SUM(U(*,*)) < SUM(U'(*,*))

Proof follows directly from the definitions.

Lemma: For any update list U(*,*) on a network with e edges, it is always the case that 0 < SUM(U(*,*)) < 4e+1

Proof: The largest value of any entry in U is 2, and the least is 0. The rest of the proof follows from the definition of SUM.

Lemma: Assuming that there are e edges in the network, the kSTRA will never have a version number greater than 4e.

Proof by contradiction: If some node had a version number V greater greater than 4e with some ECOUNT, then there must have been a sequence of versions:

version V had update list U

U'(*,*) is less than U(*,*)

version V-1 had update list U'

U''(*,*) is less than U'(*,*)

version V-2 had update list U''

.

.

.

version 0 had update list U'''...'=all zeroes

The reason for the above list is that the only way the kSTRA can start a new version V at some node p is when some version V-1 doesn't have all the information in its update list that node p knows about. Node p then raises some of the values in the update list, and starts a new version.

In the above list of items we have more than (by

assumption) 4e update lists that are sequentially related by "is less than". If we take the SUM of each of these update lists and apply the previous lemmas, we get a list of more than 4e+1 integers (U(*,*) contains only integers, hence the sum is integral) that are in order and vary from 0 to 4e. This is the desired contradiction.

End of proof

Lemma: Assume that at time t1 all nodes in the network have the same value of ECOUNT and identical E tables. Also assume that at some time after t1 node p still has the same value of ECOUNT (and hence the same E table) and at least one edge in every tree T(1),...T(k) has failed. If node p is running the kSTRA algorithm then either node p will receive a CHANGE_kSTRA message, or else it will be forced to start the SF algorithm.

Proof: Using the method of proof by contradiction, assume that node p never receives a CHANGE_kSTRA message, and never again runs the SF algorithm. (Since no SF or CHANGE_kSTRA message will ever be received, E and ECOUNT will never change.)

Let version V be the highest version that node p will ever run. (Recall that there is a bound on the number of

distinct version numbers.) Let us wait till node p is running version V. We then have at node p some well defined integer TREE which is at most k. We can also assume that the update table U(*,*) at node p has no removal requests for any of the edges in tree T(TREE), and hence all edges that are adjacent to node p in T(TREE) are still up.

Let edge i-j be the closest edge in tree T(TREE) (via a path in T(TREE)) that will ever be labeled down in some nodes update list. We can then assume that the edges in the path from node p to i via tree T(TREE) are all functional, and will remain that way forever. (We are taking i to be on the same subtree of T(TREE) as node p after edge i-j is removed.) We will refer to nodes on the tree path in T(TREE) from p to i as: p', p'', p''',...p'''...',i.

Since node p is still running version V, it must have sent out messages kSTRA(ECOUNT,V,TREE,U) to all its neighbors on edges of tree T(TREE). Let node p' be the closest neighbor in the T(TREE) path from node p to node i (a node that knows that T(TREE) is broken). Once node p' has received this kSTRA message from node p, node p' must be aware that no tree number less then TREE is still functional.

If node p' is not aware that tree number TREE is not functional, then it must send (or have sent) to the next node (p'') on the tree path in T(TREE) from node p to i, the message kSTRA(ECOUNT,V', TREE, U'). Eventually we must reach a node p2 that is both aware that node i-j is not functional and aware (we have traced the notification path) of all the updates that node p was aware of. This node is then aware that none of the trees numbered 1 through TREE are intact, and must be running the kSTRA on some other tree T(TREE') (TREE'>TREE). We should also recall that the edges in the path from p to p2 will, by assumption, never fail. Hence node p2 can never run the SF algorithm or receive a CHANGE_kSTRA message (if p2 did either of these things, then the flood would propagate back to node p).

If we then wait till node p2 runs the highest version it will ever run, then we can repeat the above procedure and find a node p3 that is permanently connected to p2, that must eventually come to know (in its update list) that all trees numbered less than or equal to TREE' are broken.

By performing the above procedure no more than k times, we will find a node which is aware that all k trees have failed edges, and must by the definition of the kSTRA algorithm run the SF algorithm. The permanent connection back to node p which

we have guaranteed would then force node p to participate in the SF algorithm, which is the desired contradiction.

End of proof

Theorem: Assume that at time t1, all nodes in the network have the same value of ECOUNT and identical E tables. If at any node p, since the time node p assumed its current ECOUNT value and E table, the local link level topology around node p differed from that given in the E table, then either the SF algorithm will eventually run at node p, or a CHANGE_kSTRA message will be received (or sent) by node p.

Proof: If the local link level topology around node p is different from the topology given in E, then there must have been a point in time t2 when this difference first existed. Time t2 could not have been before the SF algorithm stopped running at this node. If this difference existed from the time that the current E table was established, then the procedure that changed the E table would have noticed the difference and forced node p to run the Initiate kSTRA procedure. If the change occurred after the table was established, then the definition of the kSTRA requires that the initiate kSTRA procedure be run at that time. In any case, kSTRA must be running at node p.

Now that we know that the kSTRA is running at node p, we know from our last lemma that if all the trees T(i) become damaged, then our theorem is proved. We are then left to show that if one tree remains intact, then eventually a CHANGE_kSTRA message will be received (or sent) by node p.

If we assume that one tree remains intact, then the version of kSTRA that node p initiates will

propagate out on tree T(TREE) to all the nodes,

or else it will reach a node that has an update entry that prevents further use of tree T(TREE),

or else it will be received by a node that is running a higher version number (and hence be ignored),

or else it will be received by a node that has received a CHANGE_kSTRA message,

or by a node that is running the SF algorithm.

In either of the latter two cases, the fact that one tree is intact will guarantee that node p will either get a CHANGE_kSTRA or a SF message. In the second or third cases we are guaranteed that a higher version will be sent and one of

the above five fates will befall it. Since there are no more than 2n distinct version numbers, eventually we are guaranteed that some broadcast will reach all nodes, and no node will create a higher version (or else the CHANGE_kSTRA procedure or SF algorithm will be run. All that follows could only be preempted by a CHANGE message, or the running of the SF algorithm, either of which would end our proof correctly). The nodes that receive this maximal kSTRA broadcast will then start to execute the acknowledgement procedure, and eventually all nodes will have received ACK_kSTRA(ECOUNT,V) on all but one of their edges in the tree that this version V used, and sent out an ACK_kSTRA(ECOUNT,V) on the remaining outgoing edge. Since there are n-1 edges in a spanning tree, and n ACK messages are eventually sent, there must be an edge in this spanning tree that carried this ACK message both ways.Hence there must be a node that received the ACK message on ALL of its edges in the version V tree. This node would then start a CHANGE_kSTRA message moving through the network. If the node that started sending the CHANGE was node p, then we're done. If node p did not start the CHANGE messages moving through the network, then it would eventually reach node p (recall that at least one tree is intact).

End of proof.

Theorem: Assume that at time t1 all nodes in the network have

the same value of ECOUNT and identical E tables. If at time t1, at any node p, the local link level topology around node p differs from that given in the E table, then every node q in the network will eventually either run the SF algorithm, or a CHANGE_kSTRA message will be received (or sent) by node q.

Proof: From the previous theorem we know that this theorem is true if p=q. If node p is not the same as node q, then the last theorem tells us that node p will eventually run the SF algorithm, or receive a CHANGE_kSTRA message. If the network has a path that connects node p to node q, this then guarantees the correctness of the theorem. If the network is no longer connected, then none of the spanning
trees are still intact,
and again the theorem must be true (by our last lemma).

End of proof

Now that we have a theorem that basically says that if the E table differs from what the actual link level topology is, then a CHANGE_kSTRA will eventually be broadcast, we will show that this change of E tables can be done in identical fashion in all nodes, and hence the resulting new E tables will be the same at all nodes.

Lemma: Assume that at time t1 all nodes in the network have the same value of ECOUNT and identical E tables. If after t1 node p receives ACK(ECOUNT,V') from node q, and node p is running version V' of the kSTRA, then the update list U(*,*) at node p is the same as the update list was at node q, when it sent the ACK.

Proof: Before node q could have sent the above ACK message to node p, it must have sent the message kSTRA(ECOUNT,V',U',TREE). Hence node p received this message before it received this ACK_kSTRA message, and was still running version V' when the ACK arrived. By definition of the algorithm that processed the kSTRA(ECOUNT,V',U',TREE), node p must have had exactly U' as its update table when it finished the above kSTRA message. By definition of the algorithm, if the version number at node p didn't change from the time of receipt of the kSTRA message to the time of receipt of the ACK, the update table must not have changed.

End of proof.

So we have that when we receive an ACK(ECOUNT,V'), if we are running version V' (i.e. we don't ignore the message) then we know exactly the update list that the sending node has

saved in its HISTory array under version V'. Looking back at the definition of the kSTRA algorithm, we see that ACK messages are only sent when all the nodes that are "further out" on the tree used by V' have also sent acknowledgements. By repeated application of this last lemma, we then have that all the nodes that are further away on the tree used by version V' have also stored the update table U' in their HISTory array. Finally, if and when some node p receives ACK(ECOUNT,V') on all of its edges, it must be the case that all the nodes in the network have stored the update list U' in their HISTory array.

Lemma: Assume that at time t1 all nodes in the network have the same value of ECOUNT and identical E tables. If before the SF algorithm is restarted, two nodes p and q send CHANGE_kSTRA(ECOUNT,V) and CHANGE_kSTRA(ECOUNT,V'), then V=V'.

Proof: There are two reasons for nodes to send out CHANGE messages; one is as a response to receiving a CHANGE message, and one is as a response to having received ACK messages from all neighbors in T(TREE). Let us consider the first nodes p' and q' that sent out these CHANGE messages. We then have that both nodes p' and q' have received ACK messages from all their neighbors in their respective trees. By the definition of the kSTRA algorithm, we then have that p' has never acknowledged any version of the kSTRA greater than V', and p has never

acknowledged any version greater than V. We also know from the preceding discussion that all nodes have acknowledge both versions V' and V. Hence V=V'.

End of proof

Now we have that if CHANGE message is flooded across the network, only one version can be sent with this specific ECOUNT, and all nodes respond to it by modifying their E tables in the same way (all nodes have identical entries in their HISTory arrays for this version). So we know (if all nodes receive this CHANGE_kSTRA(ECOUNT,V) message) that each node will eventually have some fixed ECOUNT'=ECOUNT+1, and E'. (If all nodes don't receive this message, then the network is disconnected. We then know from an earlier proof that the nodes that did not receive the CHANGE must start the SF. The nodes in a CHANGEd section of the network will eventually realize that all the trees that they form under E' are broken, and they too will start the SF algorithm.) We are now left to show that at some fixed time, all nodes in the network have the same ECOUNT' and E'.

Lemma: Assume that at time t1 all nodes in the network have the same value of ECOUNT and identical E tables. If, before another SF algorithm is restarted all nodes receive

CHANGE_kSTRA(ECOUNT,V) message, then at the point when the last node in the network receives this message, all nodes will have the same ECOUNT'=ECOUNT+1 and E' (the topology E table).

Proof: We already know that under the assumptions of this lemma that every node will,at some point in time after t1, have ECOUNT'=ECOUNT+1 and E' (the same E' for every node). What we haven't shown is that these values cannot change at any node until all the nodes have established them. To see that this correct, we start by noting that the only way that either of these values can change is to have a node receive a CHANGE_kSTRA(ECOUNT',-) message (we are assuming that the SF algorithm has not restarted). We can also recall that the only way that a CHANGE_kSTRA(ECOUNT',-) can be sent, is if ALL nodes have sent out a ACK_kSTRA(ECOUNT',-) message, which cannot occur until all nodes have CHANGEd to ECOUNT'. Hence the values of ECOUNT' and E cannot change until all the nodes have received CHANGE_kSTRA(ECOUNT,-).

End of proof

Finally, now that we have shown that all the E tables are modified in each new ECOUNT in the same way, we have only to show that if there are no failures for a sufficiently long time, then the E tables will eventually represent exactly the

link level topology of the network.

Theorem: Assume that the values of ECOUNT and E are the same at time t, at all nodes in the network. If no more failures occur after time t, then either the SF algorithm will be run, or the kSTRA algorithm will modify the E tables so that they are the same at every node, and they will contain exactly the link level topology of the network.

Proof: If the E tables exactly represent the link level topology of the network, then we are done. If the link level topology of the network is different from that shown in E, then we are guaranteed by our previous theorems that either the SF algorithm will run, or the value of ECOUNT and E will be modified at all the nodes. If the SF algorithm runs then we are also done, so assume that the SF algorithm does not get run after time t. Let t2 be the time at which all values of ECOUNT and E have been modified to ECOUNT'=ECOUNT' and E'.

If E' does not represent the link level topology of the network, then we know that it will be changed again. This time however, it will be impossible for any node to request that an edge be added in the same update list as it already requested that it be removed. Hence the update list that is formed at

every node will contain the correct local changes to E' that are necessary to make E' reflect the local link level topology (in the last update list an edge might have gone down and up during the ECOUNT period. Hence that edge might not be placed in E'.) Within the time that ECOUNT' is being used, nodes will only acknowledge versions that include these local E table topology changes. So we have that any update list that is ACKnowledged by every node must include these changes, and when such a list is used to generate ECOUNT''=ECOUNT'+1 and E'', E'' will have all the corrections to E' that are needed.

End of proof

ACTUAL IMPLEMENTATION OF THE kSTRA

We have described the kSTRA thus far in terms of several types of messages which contain a variety of data. Some of the data that we chose to include in these transmissions was placed there to help to clarify what was going on, and simplify the proofs. In an actual implementation, far less information needs to be exchanged between nodes or maintained at each node.

To begin with, the first message that we talked about was kSTRA(ECOUNT,V,U,TREE). If the a receiving node q knows the E table that corresponds to ECOUNT (which is needed to process this message), then node q can infer (based on which edge this

packet arrived on) which of the k edge-disjoint spanning trees the message was making use of. Hence this piece of data is unnecessary.

The second data item is the update table U(*,*). It is expected that this algorithm will run very swiftly, and hence most changes in E tables will involve very few edge additions or deletions. The contents of such a U table could be expressed as a very small list (eg: "remove edge p-q, add edge r-s") and only that list needs to be sent. We are also causing the most likely changes (exactly one change in the E table) to require very little communications.

As we showed in our correctness proof, the value of v is nicely bounded (by 4e, where e is the number of edges), and its presence will later simplify the communication required by the CHANGE_kSTRA.

The value of ECOUNT is, in the above presentation, an unbounded integer. We can recall from our proofs that (in a connected subset of the network) until all nodes have received the CHANGE_kSTRA(ECOUNT,-), no node in that set can send out a CHANGE_kSTRA(ECOUNT+1,-) message. Hence there are only 2 possible ECOUNT values that we might receive. We argued

repeatedly (before each procedure) that the value of ECOUNT' that comes with a message is always less than or equal to the ECOUNT that a node has internally. It is then sufficient to send only the least significant bit of the integer ECOUNT with each kSTRA message, and no confusion can arise.

So we have that all the data that should be included is one bit (for ECOUNT), one integer between 1 and 4e (for V), and two short lists of integers between 1 and e (which edges should be added, and which edges should be removed).

The next message that we described was the ACK_kSTRA(ECOUNT,V). From the preceding paragraph we see that the only data required in this packet is a bit (for ECOUNT), and an integer between 1 and 4e (for V).

Lastly there is the message that will flood across the network: CHANGE_kSTRA(ECOUNT,V). As with the above acknowledgement message, this type of message would only have to carry a bit and an integer between 1 and 4e.

In data communications, the time necessary for a packet to travel from one end of the network is generally proportional

to the size of the packet times the number of hops taken (each node must receive the whole packet before it can be passed on). The fact that the amount of data required in each of the above packets is so small, guarantees that the algorithm can (assuming that kSTRA packets are given top priority) complete a modification of the E table very swiftly. The swiftness of the algorithm tends to guarantee that the topology will be tracked well by the algorithm. This swiftness then justifies our premise that few changes would be included in the update list.

Looking now at the storage requirements for the algorithm, we start by noting that only the one bit that is transmitted to represent ECOUNT must be saved. The current E table topology must be stored at all times, but this block of memory is certainly bounded in size by the number of edges in the network. The current version number requires space for one integer in the range of 0 to 4e. The update list can be stored, independent of the number of entries, in an area that is roughly no larger than is needed to store the topology.

The HIST array (remembers all the update lists that we have acknowledged) can have as many as 4e entries (one for each version). The clever point to notice however, is that every update list that a given node acknowledges, must contain all the updates in all the previously (within this ECOUNT cycle)

acknowledged update lists. Hence the HIST array can simply maintain a list of all updates that we have heard in the order in which we heard them, and keep a pointer to show at what point in in this sequential list the acknowledgement for a given version was sent out. This array would then take space that was bounded by O(e), as are the memory requirements for the E table and U list.

RECOVERY

We have now explained how the multiple trees can be used to disseminate the edge failure information. Recovery from a failure has several parts. As we have said earlier, recovery involves the consistent modification of routing policies at all nodes in the network. It also involves decisions as to what should be the fate of packets that are midway between their source and destination. (Some packets may even have been transmitted across the failed link before it went down!)

As was mentioned in chapter 1 of this thesis, the exact recovery goals vary from network to network, and application to application. There are only a few basic strategies that might be used in conjunction with our kSTRA algorithm, and we will try to mention them here. Fundamentally there is a choice of whether to maintain packets that were in transit before a failure, or to throw them away. If packets can be thrown away,

then some sort of end to end link protocol (from the point of entry into the network, to the point of exit from the network) must be used if the system is to guarantee end to end transmission. If packets are not thrown away, then care must be taken not to produce duplicates of packets (For example; if a packet was sent across an edge, and then the edge went down, and no acknowledgement was received, should the packet be retransmitted after a new E table is developed?). If it is necessary to prevent such duplication of packets, some packet numbering scheme (perhaps included in an end to end protocol) would have to be used. A second item that must be dealt with, if packets are not thrown away, is that the order in which the a sequence of packets arrive at the destination node is not necessarily the order in which the packets were injected into the network at the source node. If it is necessary to preserve ordering then again a numbering scheme would be needed.

Appendix A

Possibilities for finding edge-disjoint spanning trees, in an undirected network, in less than O(knkn) time.

A simplistic analysis of the running time of our algorithm that we presented in Chapter 3 to find k edge disjoint spanning trees in an undirected network is: we must place O(kn) edges in various forests, and the work required in each placement may involve looking at all the previously placed edges (O(kn)). Hence the total work was at least O(knkn). The careful analysis in Chapter 3 also showed that the forementioned tasks dominated the time complexity of the algorithm, and the net complexity was indeed O(knkn). In this appendix we will explore two methods of holding down the lengths of the augmentation sequences that a related algorithm finds. We hope, but do not prove, that one of these methods will eventually lead to an algorithm that finds such edge-disjoint trees in asymptotically less time than O(knkn).

SMALL TREES

The following theorem places a tighter bound on the amount of work needed to augment the set of forests with an edge.

Let v be a node, and let i (a forest number) be such $0<i<k+1$

Define: $T(i,v)$ = The set of nodes that are in the same tree as v in forest $F(i)$

Given a set S, we define $|S|$ to be the number of elements in S.

Assume that we are trying to augment the forests by adding edge (v,w) (v and w are nodes). We have the following theorem:

Theorem: For any forest $F(i)$, if a minimal augmentation sequence exists that starts with edge (v,w), then a minimal augmentation sequence exists that has less than $k(|T(i,v)|-1)$ edges in the sequence. Moreover, the augmentation sequence can be found by looking at no more than $k(|T(i,v)|-1)$ additional edges (other than (v,w)).

The implication of the above theorem is that we should (assuming we're not looking for minimum weight trees) try to augment the existing forests with an edge that "touches" a small tree in one of the forests. We will then have a bound on the number of edges that must be looked at in order to find

an augmentation sequence.

Proof: Let e(1),e(2),...e(p) and n(1),n(2),...n(p) be an example of the shortest possible minimal augmentation sequence. Using proof by contradiction, assume that $p > k(|T(i,v)|-1)$. It must be the case that both endnodes of every edge e(j), ($j<p$), are contained in T(i,v) (otherwise we could produce a shorter sequence). In each forest, there can be no more than $|T(i,v)|-1$ edges that have both endnodes in the set T(i,v). Hence there are no more than $k(|T(i,v)|-1)$ edges from all the forests combined that have both endnodes in T(i,v). This contradicts the premise that we have a list e(*) of greater than $k(|T(i,v)|-1)$ such edges. (Note: there can be no repetitions in the list e(*), as we are dealing with a minimal augmentation sequence.)

Having shown that an augmentation sequence of this bounded length exists, It remains to be shown that such a sequence can be found by looking at no more than $k(|T(i,v)|-1)$ edges. To get this bound, we would have to modify the labeling step of our multiple tree algorithm. The modification would consist of prechecking every edge that was put on the QUEUE, for use in forest F(i) (i.e. can we end the augmentation sequence by using this edge in F(i)). This would guarantee that no more than $k(|T(i,v)|-1)$ edges would be placed on the QUEUE,

by the argument given in the last paragraph. Note that such a modification to the original labeling section would not change its asymptotic complexity. Hence the whole multiple tree algorithm would run (asymptotically) at least as fast as the original.

PLACING kn/2 LINKS IN O(m+kn) TIME

We will now demonstrate the usefulness of such a theorem. Recall that a necessary condition for the existence of k edge disjoint spanning trees is that the network be k connected. In particular, there are at least k edges touching every node. At the start of the algorithm every T(i,v) is a singleton, and any augmentation sequence that starts by touching such a singleton can be found in constant time (i.e. just put the edge that touches v, such that |T(i,v)|=1, in forest F(i)). At the start of the algorithm there are kn distinct singleton sets. Each edge that is placed in a forest during this process changes at most two singletons into a larger set (worst case unites two singletons) From this it follows that at least kn/2 edges (minimum) will be placed into forests during this process. We will show that this work can be done in O(m+kn) time (m from looking at the list of edges, kn from placing the kn/2 edges with a cost of O(1) work each).

The algorithm: Initialize an array s(p) (p=1,2,...n) to be all

0. The definition of s(p) is to be such that for any node p, F(s(p)) is the least numbered forest in which node p might still be a singleton. If s(p)=k+1, then node p is not a singleton in any forest. One at a time the algorithm looks at every edge (v, w) as follows:

```
"check v"
        1) If s(v)=k, then goto "check w"
        2) s(v) <-- s(v)+1 'point to next forest
        3) if |T(s(v),v)|=1 then v is a singleton in s(v), so..
                a) UNION(s(v),v,w)  'put the edge (v,w) in the forest
                b) get the next edge, and goto "check v"
        4) goto "check v" 'see if v is a singleton in the next forest
"check w"
        5) If s(w)=k+1, then get the next edge and goto "check v"
        6) s(w) <-- s(w)+1 'point to next forest
        7) if |T(s(w),w)|=1 then w is a singleton in s(w), so..
                a) UNION(s(w),w,v)  'put the edge (w,v) in the forest
                b) get the next edge, and goto "check v"
        8) goto "check w" 'see if w is a singleton in the next forest
```

Before we analyze the computational complexity of this algorithm, we should define the UNION-FIND algorithm that will be used in conjunction with it. In the earlier section of this thesis we made use of a lookup table to perform a FIND operation. The UNION(i,v,w) operation had to search the i'th

lookup table (corresponding to the forest F(i)) and change all mentions of the canonical node that represented v (i.e. FIND(i, v)) to the canonical node that represented w. This linear search took O(n) operations. This amount of computation in the above algorithm is intolerable, and unnecessary. Note that the first node that is passed to the above UNION(*,*,*) function call is always a singleton. Knowing that the first node passed to this UNION function is a singleton, we can devise a fast UNION function that runs in O(1) time. This fast UNION maintains a data base (an array that gives the canonical node associated with node) that looks identical to that which the O(n) UNION maintains. The O(1) UNION is defined as follows:

UNION(i,p,q)=(definition)

Modify the data structure by changing the entry that corresponds to a FIND(i,p). Since p is a singleton, this corresponds to a single change.

The new entry should be what is returned by a FIND(i,q)

Note that the UNION(i,v,w) can also update a lookup table which maintains the size of T(i,v). Using such a table, the evaluation of |T(i,v)| can be done in constant time. Specifically, we can set up an array N(i,v), so that the following is true at all times: (for $0<i<k+1$, $0<v<n+1$)

$$|T(i,v)| = N(i,FIND(i,v))$$

N(i,w) gives the size of the tree in forest F(i) that contains canonical node w. By requiring that N(*,*) be correct only at canonical nodes, we make it possible to have a UNION algorithm that wastes little time to update the N(*,*) array. The new UNION'(i,v,w) algorithm would then be:

```
1) If FIND(i,v)=FIND(i,w) then return 'no work to do
2) temp <-- N(i,FIND(i,v)) + N(i,FIND(i,w)) 'add old tree sizes
3) UNION(i,v,w) 'perform UNION algorithm
4) N(i,FIND(i,v)) <-- temp 'remember new size
5) return
```

With the above functions defined, we see that each of the individual statements in our algorithm run in constant time. With a little clever counting we will show that we have indeed placed at least kn/2 edges into various forests in O(m+kn) time.

To get a time bound on our algorithm, we note that steps 2 and 6 can be executed only kn times (O(kn)). This bounds the number of times that steps 3,4,7 and 8 can be executed. Steps 1 and 5 can be executed by virtue of the fact that steps 4 or 8 (respectively) have been executed (O(kn)), plus no more than one additional time for each edge that is looked at (O(m)). Hence no statement is executed more than

O(m+kn) times, and since each statement takes constant time, the algorithm runs in O(m+kn) time.

WISHFUL EXTENSIONS

We have now shown the usefulness of trying to find augmentation paths that start with edges that touch small trees. We do not have a proof that shows the complexity of finding k edge disjoint spanning trees is reduced from O(knkn) when edges are fed to the algorithm starting with the ones that touch the smallest trees. We can show that the cost of providing edges in such order (i.e. the edge (v,w) such that |T(i,v)| is currently minimal) during the running of the algorithm is no more than O(m+kn log(kn)). The hope for a better bound on the overall algorithm would then be that:

a) Half of the kn edges may be placed in time O(m+kn) time

b) Half of the remaining kn/2 edges may be placed in O(m+kn) time (unproven)

c) Half of the remaining kn/4... in O(m+kn) time (unproven)

*

*

*

The total work required would then be:

$O((m+kn)\log(kn))$ for the placements

PLUS

$O(m+kn\log(kn))$ for producing the edges in order

Since m is at least k(n-1), this is just:

$O(m\log(kn))$

If we were very lucky, the placement time for steps b, c,... would only be O(kn), and the net complexity might be $O(m + kn\log(kn))$.

The motivation for thinking along these lines is a paper on finding spanning trees using a distributed algorithm by Gallager, Humblet and Spira [12].

FINDING LINKS THAT TOUCH SMALL TREES

We will now describe an algorithm that will, at each point in the running of the multiple spanning tree algorithm, find the edge (v, w) (that has not yet been shown to be dependent) such that $|T(i,v)|$ is minimal over all forests F(i), and over all nodes v, that have unexamined edges touching them. By "unexamined edges" we mean that the multiple spanning tree

algorithm has not yet tried to place them in any of the forests. This algorithm also gives the value i, for which |T(i, v)| is minimal. The spanning tree algorithm must know the value of i, so that it can try every edge that is put on the QUEUE in forest F(i). It is this focusing of attention upon the forest F(i) that provides the bound on the augmentation sequence length.

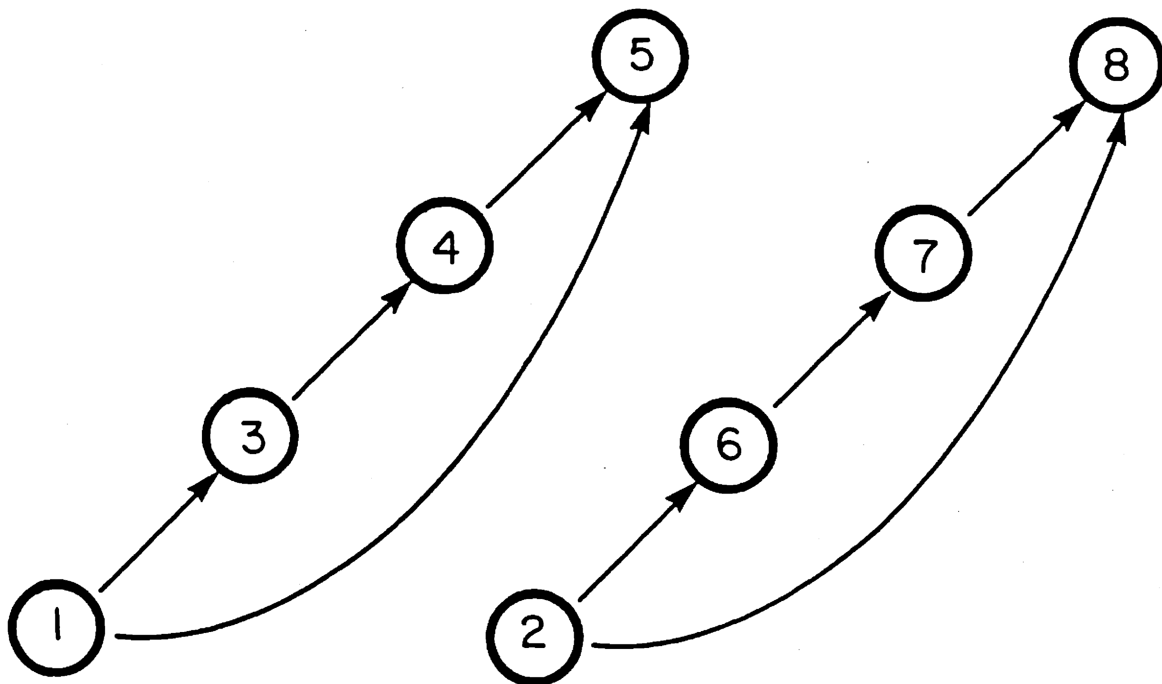
We start by making a list of all the canonical nodes in each forest, along with the size of the tree associated with that canonical node (this takes O(kn) work). This list will be updated as the spanning tree algorithm is run, and the smallest tree in any forest will be found based on this list.

In order to quickly find an unused edge that touches a given node, we form a list of all edges that touch each node. Forming this vector of lists (one list for each node) takes O(m) time. Unfortunately this data structure produces two copies of every edge (one for each endnode). When an edge is found on one node's list and used (given to the spanning tree algorithm to examine) it must be deleted from the other list that it is on. To achieve an efficient deletion (i.e. we can't afford to search for the second listing), we maintain an n by n interconnection matrix. The elements of this matrix tell whether an edge connection between two nodes (that is on the

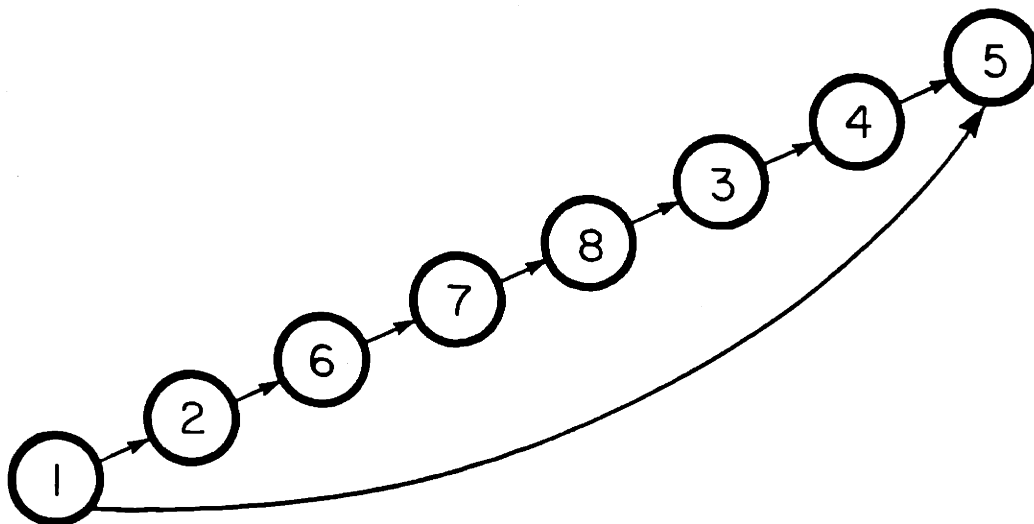
forementioned list) has been examined yet. Note that only m entries in the n by n matrix need be initialized. With these two structures we can, given any node v, find the next unexamined edge that touches node v (or find that there are no more) in constant time. Creating these structures takes O(m) time.

Using the structure described in the last paragraph, we can find edges that touch any given node. Our actual problem is to find an edge that touches a certain tree. Each tree in each forest is referenced by way of a canonical node. We therefore need a fast way to list the nodes in each tree, based on a canonical node. To achieve this fast listing of nodes, we maintain (in addition to the data structure that FIND and UNION must maintain to do their job well) a linear list of all the nodes that are associated with each canonical node for each tree in each forest. To make it possible to combine these lists in constant time (the UNION algorithm will have this added functional chore) we must also maintain a pointer from each canonical node to the last node in the list. A picture amply describes the method, so we offer the following example:

node 1 and 2 were canonical nodes in forest F(i) We have the following structures



A UNION(i,1,2) takes place, and the new canonical node is node 1. The following is the resulting structure



The changes made were as follows:

a) The last node on node 2's list (node 8) was given a pointer to the second node on node 1's list.

b) The pointer from node 1 to the next element in its list (1 to 3) was changed to point to node 2.

Note that the pointer from node 2 to node 8, is no longer shown. Pointers to the end of a list are only significant if the node is a canonical node. It should also be noted that the operation of combining the two lists takes a constant amount of time, independent of the length of the lists (which are determined by the corresponding tree sizes).

There is one more detail that we must add to the above data structure to make it possible to find a useful node in a given tree in constant time. The problem that must be overcome is that the node that "pops out" of the above structure may not have any unused edges touching it. Basically, we drew a member from an equivalence class, and then found out that we couldn't use it. What we are anxious to avoid is having that element pop up again and wasting our time. To speed things up, when ever we find such a node (i.e. all of its edges have been looked at) in one of these linear lists we should delete it from the list. (The canonical node would be marked "empty", where as the other

nodes in the list could be physically deleted.) Note that since the FIND and UNION algorithms rely on separate data structures, we don't damage the original algorithm. Also, since there are n nodes and k forests, the most time we could waste on deletion operations is O(kn). When a deletion does not result, we can find an edge, which touches any tree, in constant time.

Now what remains is to identify the smallest tree in any forest. Using the forementioned structures we can then find an edge which touches that tree in constant time (or be told that none exists). A key point to note here is that the sizes of the trees in the forests vary very slowly. That is, when an augmentation sequence is used, exactly two trees change their "size". All the other trees that are involved in the augmentation sequence get reformed, but the number of nodes in each tree (and the actual nodes in each tree) remain unchanged. As for the two trees that change, one is enlarged to contain the other. The problem is then to find the minimum element (smallest tree) among a slowly changing set of numbers (tree sizes) This can be done efficiently by forming a binary "playoff tree" with kn leaves, and depth log(kn)/log(2). The kn leaves are the initial k forests of n singleton trees. To form this tree initially takes O(kn) work, but updates take only O(log(kn)) work. This tree must be updated when:

1) a UNION takes place (this happens O(kn) times)

2) it is discovered that there are no more edges to be examined that touch the currently smallest tree (again this is bounded by O(kn) occurrences)

The total to maintain this tree is then O(kn log(kn)).

So we have shown that we can (on the fly) find the canonical node and forest that contains the smallest tree. Given the canonical node, we can in constant time find a node in that tree, whereby in constant time we can find an edge that touches that node (the last two "constant time" estimates are amortized over the choice of O(kn) edges). Hence the total time needed to provide the edge we desire is O(kn log(kn)).

TIGHTER BOUNDS ON AUGMENTATION SEQUENCE LENGTHS

We mentioned earlier that the fundamental performance bottleneck in the execution of the multiple spanning tree algorithm is the fact that O(kn) edges must be placed in various forests, and each placement might require an augmentation sequence of length O(kn). In this section we will again attempt to reduce the length of such sequences, in hopes of improving the net performance of our algorithm. No

asymptotic improvements have as yet been shown to result from the ideas that follow, although the ideas seem promising.

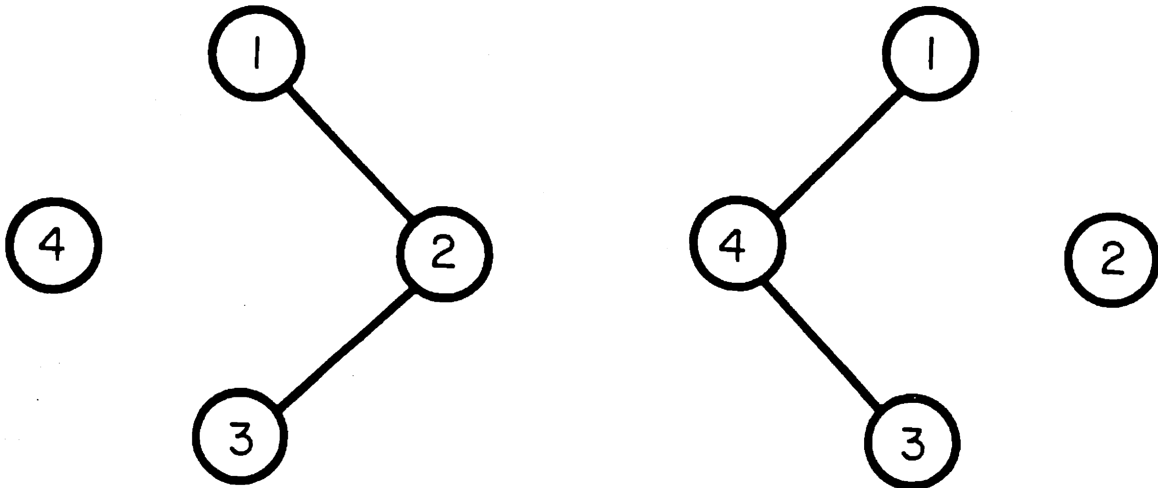
We start by repeating the definition of T(i,v), to be the set of nodes in the tree in forest F(i), that includes node v. In the last section we showed that we could bound the length of an augmentation by O(k |T(i,v)|). Let us define the intersection of all T(i,v), over all forests, for a fixed node:

T(v)=(Def) T(1,v) * T(2,v) * ... * T(k,v)

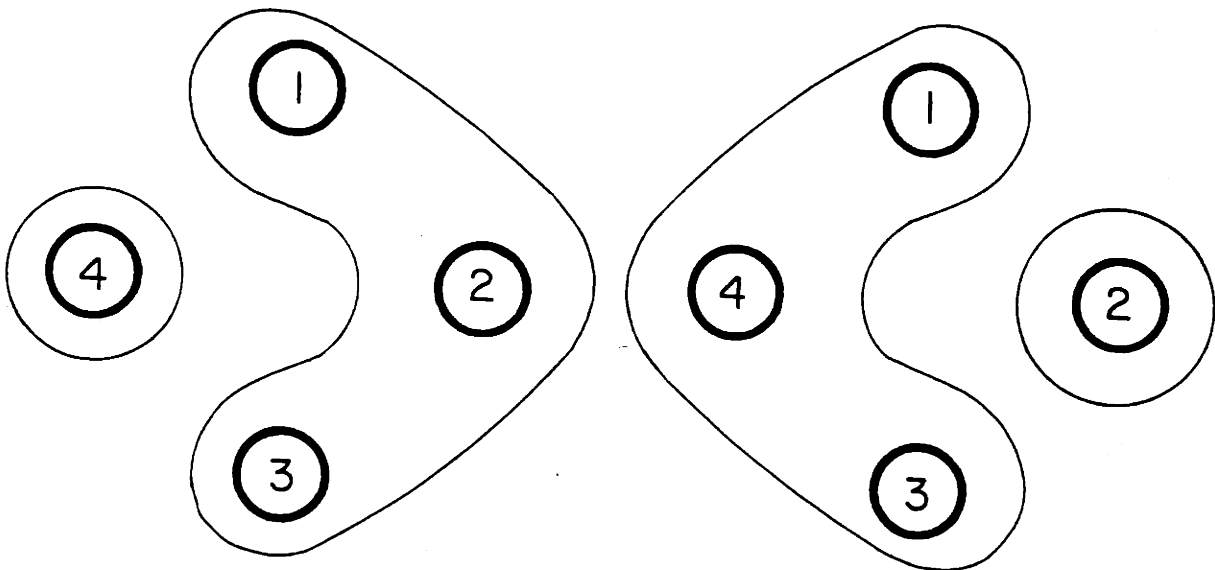
(we remind the reader that "*" in this context means "set intersection"). We will show that we can bound the size of an augmentation sequence that starts with edge (v,w), to be not greater than k(|T(v)|-1). It should be noted that this is a tighter bound than k(|T(i,v)|-1), as |T(v)| must be less than or equal to |T(i,v)| (for all i). As we will see by the example below, it can quite easily be the case that |T(v)| is strictly less than |T(i,v)| (for all i).

We might note that T(v) can be viewed as the set of nodes that are reachable from v in all forests. With that thought in mind, it is clear that T(*) partitions the set of nodes. A bit of thought about this structure will show the interested reader that the sets that result are quite different from the "clump" structure that was discussed in the Chapter 3.

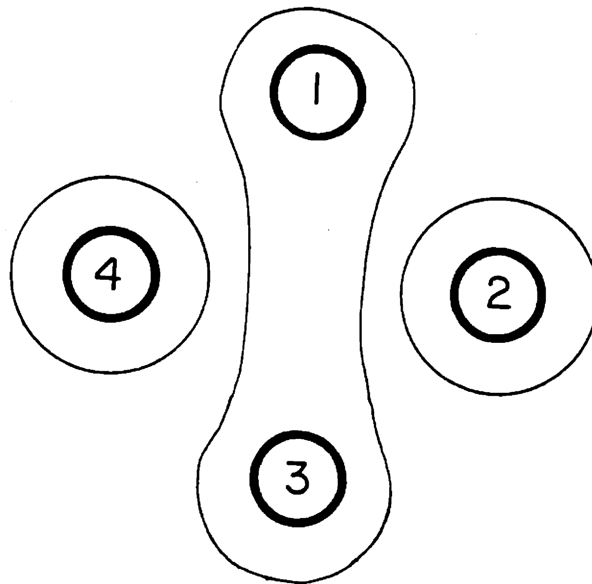
As an example, consider the following two forests:



The partitions induced by T(*,*) would then look like:



Finally the partition induced by T(*):



Notice that $|T(1)|=2$, and this is less than $|T(1,1)|$ and $|T(2,1)|$, both of which are 3.

First we will show that if a minimal augmentation sequence exists and starts with edge (v,w), then a minimal augmentation sequence exists that has no more than $k(|T(v)|-1)$ edges. As with our earlier sequence bound proof, we note that there cannot be more than $|T(v)-1|$ edges in any forest that have both endnodes in T(v). Therefore there cannot be more than $k(|T(v)|-1)$ edges in all the forests that have both endnodes in T(v). Suppose that we had a minimal augmentation sequence that started with edge (v,w), and there were more than $k(|T(v)|-1)$ edges in that sequence. Consider the first $k(|T(v)|-1)$ of

these edges. There must be an edge among these edges that does not have its endnodes in T(v) (if all the edges were in T(v), then T(v) would be a clump, and we could not have found an minimal augmentation sequence). Let edge e(j)=(x,y) be the first edge that does not have both its endnodes in T(v) (as mentioned, j is no more than k(|T(v) |-1)). Since we are dealing with an augmentation sequence, we know that a path around the previous edge C[e(j-1),F(i)] contains e(j). Also, since both the endnodes of edge e(j-1) are in T(v), there must be an edge e=(x', y') in the path C[e(j-1),F(i)], such that x' is in T(v), and y' is not in T(v). By the definition of T(*), there must be a forest F(i) in which y' is not in T(i,v). Also by the definition of T(*), x' must be in T(i,v). Hence a minimal augmentation sequence may be formed with j or fewer edges, that ends with the placement of (x',y') in forest F(i).

All the bounds that we have mentioned thus far in this section can only be used IF we can "quickly" determine the membership of an arbitrary node in T(v). The remainder of this section will focus on this problem.

MAINTAINING INTERSECTION DATA STRUCTURES

We now turn our attention to the task of maintaining T(v), and being able to swiftly determine membership in T(v). As noted, T(*) defines a partition of the set of nodes. We can

therefore use a set union algorithm to maintain the partition, and FIND function calls to determine if two nodes are in the same partition. Sufficient is a set union algorithm that has a FIND algorithm that runs in O(1) time, and a UNION algorithm that runs in O(n) time (See appendix B for details). We will refer to the FIND-UNION pair that maintains this partition T(*), as TFIND(*) and TUNION(*,*), respectively. The hard part is then to decide when to perform a TUNION.

To be able to determine when to perform a TUNION(x,y) (and what x and y should be) we maintain an n by n array of integers:

M(x,y)= (Def) The number of forests in which node x is in the same tree as node y

Note that M(*,*) is a symmetric matrix. The description that follows is easier to give without exploiting this symmetry. An actual implementation would probably take advantage of such structure.

We can update the contents of the array M(*,*) as follows: Whenever two trees in a given forest are to be combined by a UNION(i,v,w), we do the following first:

```
assume  T(i,v)={a(1),a(2),...a(p)}  "Node sets for ..
        T(i,w)={b(1),b(2),...b(q)}   " ... the trees

FOR i=1 to p "Try all elements in T(i,v)..
     FOR j=1 to q "..with all elements of T(i,w)
          M(a(i),b(j)) <-- 1+M(a(i),b(j))
                    "They're connected in another forest
          M(b(j),a(i)) <-- 1+M(b(j),a(i))

          IF M(a(i),b(j))=k then.."If they're connected
                                          in k forests
             IF TFIND(a(i)) not equal TFIND(b(j)) then..
                 TUNION(a(i),b(j)) "Put the sets together
    NEXT j
NEXT i
```

and then we do the actual UNION(i,v,w).

Unfortunately, the set union algorithm that we are using (via FIND(i,v) and UNION(i,w,v)) does not maintain a data structure that is convenient for listing all nodes that are in a given tree. To facilitate such a listing we would maintain a data structure of "linear lists" of nodes in each tree, in each forest, by using a slightly more complex union algorithm (as described above), in addition to our standard UNION-FIND pair.

At first, the evaluation of the computational complexity necessary to maintain T(*) appears as a terrible mess. The combinatoric problem is that of combining all nodes in one tree with all nodes in an other tree, whenever a UNION is done, with the trees varying in size as the algorithm progresses...(very messy). We can however calculate the computational complexity quite simply, by virtue of the fact that we maintain the matrix M(x,y). We note that the entries in M(*, *), are always integers between 1 and k inclusive. Moreover, these entries never decrease. Therefore, there can be no more than nkn changes in values of entries of M(*,*), and hence no more than O(nkn) executions of statements that modify M(*,*). This puts a bound on the total execution time of the above program of O(nkn). (Note that the only statement in the above program that doesn't run in constant time is the TUNION(*,*) operation. The TUNION runs in O(n) time, but can only be executed a maximum of n times. Hence the total contribution of TUNION is O(nn) time, and is negligible.)

We can therefore see that T(*) can be maintained at a cost of no more than O(nkn) work. It should be noted that T(*) is being maintained as a partition, and TFIND operations can be used to see if two nodes are in the same subset of T(*). Using the same methods as we did to find edges that touch small trees, we can then find edges that touch small subsets of the

partition T(*). Since there are only n possible distinct canonical nodes in T(*), the calculation of the smallest subset in the partition would require O(log(n)) time each time T(*) changed. Hence we would spend O(n log(n)) time finding small subsets (as opposed to O(kn log(kn)) time which was spent in the last section to find small trees.)

With the above analysis we see that the O(nkn) time necessary to maintain T(*) dominates the effort of this section and is then the amount of work needed to supply the multiple tree algorithm with edges as desired by this section. We do not claim that this is the best method to provide edges in this order, but since the complexity is less than that of the original multiple tree algorithm (O(knkn)), the impact on the multiple tree algorithm should be further explored.

INTERSECTION DATA STRUCTURE IN THE MULTIPLE TREE ALGORITHM

We have now shown how edges could be supplied (on the fly) to our algorithm for finding multiple edge-disjoint spanning trees in an undirected network. What remains is to show how the multiple tree algorithm can be modified to use T(*) to find minimal augmentation sequences that are constrained in length by k(|T(v)|-1) (where edge (v,w) starts the sequence).

The key point to notice is that if any edge (v,w) that is put in the QUEUE traverses the boundary of the partition T(*) (i.e. TFIND(v) is different from TFIND(w)), then edge (v,w) is immediately useful in one of the forests. This fact stems from the original bounding argument for this "intersection of trees" section. Since a TFIND operation takes only constant time, we can perform this check on every edge that is put on the QUEUE, and have no effect on the time complexity of the original algorithm.

If ever we find an edge that traverses the partition T(*), we could immediately look for the forest in which this edge is directly useful. The search would be among k forests, and would involve doing a FIND(i,v) and a FIND(i,w), both of which take constant time. Hence finding the end of an augmentation sequence by this method would take O(k) time to find each ending. Since there are only k(n-1) edges placed during the running of the program, we see that we spend no more than a total O(knk) time finding these quick endings to augmentations.

Now that we can guarantee that that no augmentation sequence found by our intersection monitoring algorithm can have an edge (except the last edge in the sequence) that

crosses a boundary of the partition T(*), we are sure that the augmentations are less than k(|T(v)|-1) in length. To achieve this result we have added a total of O(knk+nkn) work. The open question is: "What effect does this bound on the augmentation sequence length have on the net complexity of the algorithm?". More specifically: "Can it be shown that the (worst case) average augmentation sequence length is less than O(kn) when this algorithm is used?"

Appendix B

FIND-UNION algorithms

In the problems that we have discussed in this thesis, we have had to maintain a data structure that represents the partition of a set of discrete elements, on several different occasions. One of the first uses was, in developing an algorithm to find k edge disjoint spanning trees in an undirected network, to recall which nodes were already connected to one and other in a given forest. Later on, in enhancing that algorithm, we maintained a partition that recorded the structure of "clumps" for the algorithm. Lastly, in appendix A, there were a variety of partitions of the nodes in the network that we needed to maintain and manipulate. Due to the variety of manipulations that were necessary to perform upon these partitions, and various computational complexity goals, we have developed a range of algorithms to service our needs. In this appendix we are going to present a full list of such algorithms.

A general property of all the partitions (or equivalence classes) that we maintain is: if two elements are put into the same equivalence class, in some partition, at some point during the running of an algorithm, then those two elements will forever more remain in the same equivalence class

in that partition. This condition means that we are not maintaining arbitrarily changing partitions. It precludes, for example, the partition {{1},{2},{3}}, from changing to {{1,2},{3}}, and later to {{1},{2,3}}. This general property of our partitions also reduces the number of primitive manipulations that might be performed to the joining of (or "UNION"ing) of two existing partitions.

An example of the results of such UNION operations on a partition would be:

Start with the partition {{1},{2},{3},{4},{5}}

UNION({1},{3})

resulting in {{1,3},{2},{4},{5}}

UNION({1,3},{5})

resulting in {{1,3,5},{2},{4}}

UNION({2},{4})

resulting in {{1,3,5},{2,4}}

UNION({1,3,5},{2,4})

resulting in {{1,2,3,4,5}}

Since the data structure contains the information that describes the partition, it is sufficient to provide the UNION operator with names of one element in each of the equivalence classes that we want united. For example, when the partition was {{1,3,5},{2,4}}, it is sufficient to say UNION(2,5), to modify the partition to {{1,2,3,4,5}}.

Having a data base that represents a partition is not enough, we need some way to interrogation that data base and gain information about the partition that is represented. The most common question for us to ask is: "Are the elements x and y in the same equivalence class?". A sufficient bit of information is gotten by providing a function that interrogates the data base, and FINDs the canonical name of he equivalence class that contains an arbitrary element x. We could therefore reduce the original question about elements x and y, to the question of whether elements x and y are in equivalence classes with the same canonical name.

An example of the results of various FIND operations when the current data base represents the partition is {{1,3}, {2,4},{5}} is:

FIND(1)=1

FIND(2)=4

FIND(3)=1

FIND(4)=4

FIND(5)=5

The significance of the fact that FIND(2)=FIND(4), is that elements 2 and 4 are in the same equivalence class in the given partition. Note that the names of different equivalence classes must distinct. In order to accomplish this, all of our algorithms will use the name of some element in the equivalence class as the canonical name for the equivalence class. analysis of a nearly linear (almost O(1) UNION and FIND) set union algorithm algorithm is presented.

The two functions that we have introduced are used through out the the literature to describe functions that manipulate partitions [22]. We will now proceed to develop specific FIND-UNION algorithm pairs.

O(1) FIND, O(n) UNION

We saw in the explanation of what the FIND operator does, the way a table of the canonical names for each element completely specifies the partition. We can define a very simple FIND-UNION algorithm that maintains a table (or array) of values for use by the FIND operator. The action taken when a FIND(x) was called for would then be:

1) Get the x'th entry from the table, this is the canonical name for x

The UNION operator would then have to update this table every time it was called upon to act. The update that would be required when a UNION(x,y) was called for would then be:

1) Get the x'th entry from the table, call it X
2) Get the y'th entry from the table, call is Y

We now have X and Y are the canonical names of x and y, respectively.

3) Look through the table for any entries that are X...

3a) ..and change such entries to Y

We have now changed the canonical name of any element that used to be in the equivalence class with x, to have the canonical name of all the elements in the equivalence with y. Hence, all the elements that used to be in either in the equivalence with either x or y, are now all in one equivalence class.

In order to be able to describe the other algorithms that we will present later in this appendix, we be forced to use a more formal language. For consistency, the following is the formal definition of the FIND-UNION algorithm that we have just described:

```
INITIALIZATION1

        DIMENSION M(n)   'reserve space for the array
        FOR i=1 TO n   'Initialize the entire table so that...
                M(n) <-- n   '..the canonical name for each singleton
                            equivalence class is the name of the singleton
        NEXT i

FIND1(x)

        RETURNS (M(x))   'return the x'th entry in the table
```

```
UNION1(x,y)

      X <-- M(x)   'get the canonical name for x
      Y <-- M(y)   'get the canonical name for y
      FOR i=1 to n  'Look at all n entries in the table
              IF [M(i)=X] THEN   'find an entry with
                                      'canonical name X
                      M(i) <-- Y  '...and change its name
                                      'to Y
      NEXT i
```

Assuming that there are n elements in the set that we are manipulating, we can see that the search of all elements done by the union operator takes O(n) time. The FIND operator has only to do an array lookup, and hence performs in constant (O(1)) time.

MAINTAINING SIZES OF EQUIVALENCE CLASSES

Before we discuss the next FIND-UNION pair, we will take a brief diversion to discuss the computational complexity of finding the size of an equivalence class. The goal is to maintain some data structure that allows us to execute a function call, such as SIZE(x), that returns the number of elements in x's equivalence class. There are several uses for such a function. In the later examples of FIND-UNION pairs,

knowledge of such sizes will be critical in constructing efficient algorithms. In appendix A, several of the potential improvements in our multiple spanning tree algorithm are based on being able to figure out the sizes of various equivalence classes.

The algorithm that we present as SIZE(x), is only capable of working when x is the canonical name of the the equivalence class that contains x. Although this seems like a major limitation, the FIND-UNION algorithm pair just given exemplifies a situation where the translation from arbitrary element in a class, to the canonical name of the class takes only constant time. It is also the case, that the algorithm (SIZEUNION(x,y)) that updates this data base when a UNION is performed must be given the canonical names of the equivalence classes, and not just arbitrary elements.

The algorithms are are rather straightforward. First there is the data base initializer:

```
INITIALIZE

      DIMENSION SIZEOF(n)   'there are n elements that we
                              have to remember
      DIMENSION VALIDSIZE(n)   'Which of the above
```

```
                                            'sizes are valid
        FOR i=1 TO n    'run through all entries
                SIZEOF(i) <-- 1  'All classes start out
                                                    'as size 1
                VALIDSIZE <-- TRUE  'All the elements are
                                       'canonical names, so all the
                                       'sizes are valid
        NEXT i
```

Next there is the function that looks onto the data base and returns the size of the given equivalence class:

```
SIZE(x)

        IF VALIDSIZE(x)=FALSE  'make sure that this
                                  element is a canonical name
                THEN STOP 'x is not a canonical name
        RETURNS(SIZEOF(x))  'Since it is a canonical, just
                                         'look it up
```

Finally there is the function that updates the data base when a UNION is performed. The convention that we will use is that the canonical name of the resulting equivalence class, will be the first parameter (the name of the first equivalence class) to

the function call.

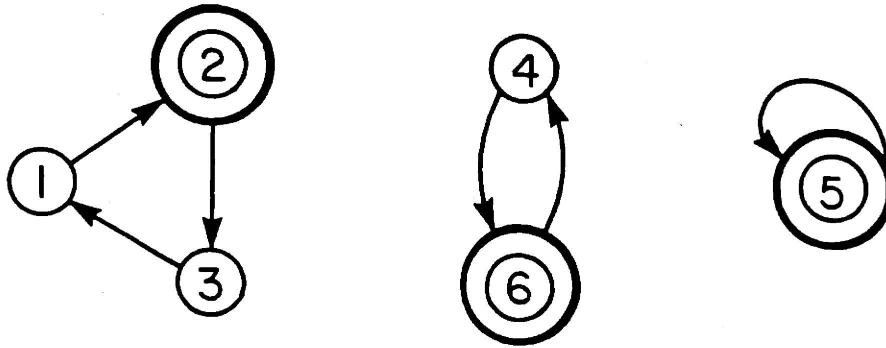
```
SIZEUNION(x,y)

        IF VALIDSIZE(x)=FALSE   'Check that the parameters are
                                'valid names...
             OR
           VALIDSIZE(y)=FALSE
           THEN STOP   'one of the parameters is not a name
        VALIDSIZE(y) <-- FALSE   'The second name is no longer
                                a canonical name, and its size
                                is no longer valid
        SIZEOF(x) <-- SIZEOF(x)+SIZEOF(y)   'Add the sizes of
                                'the two classes, to get the
                                'size of the new one
        RETURN   'That's it
```

It is interesting to note that each the above functions runs in constant time, as there are no loops in any of the code. There are constraints on the parameters, but that will pose no difficulty in using these algorithms in later examples of a FIND-UNION pairs.

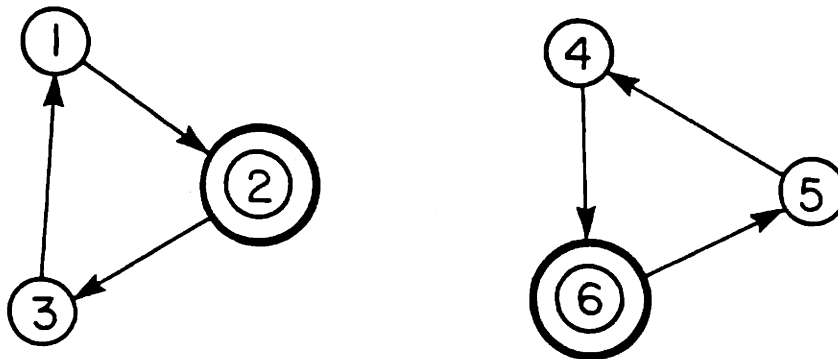
O(n) FIND, O(1) UNION

Although this particular FIND-UNION pair did not end up to be directly useful in this thesis, the concepts are used in conjunction with other FIND-UNION algorithms in appendix A. Its function, in the context in appendix B, will be discussed later in this appendix. One special constraint that is present in the use of this specific algorithm, is that the two elements that are given as parameters to the UNION operator, must be canonical names. The idea here is to use a data structure that forms a loops of pointers for each equivalence class. By manipulating the pointers, two loops may be quickly combined into one. The FIND function traverses this loop each time it performs it task, and takes considerably more than constant time.

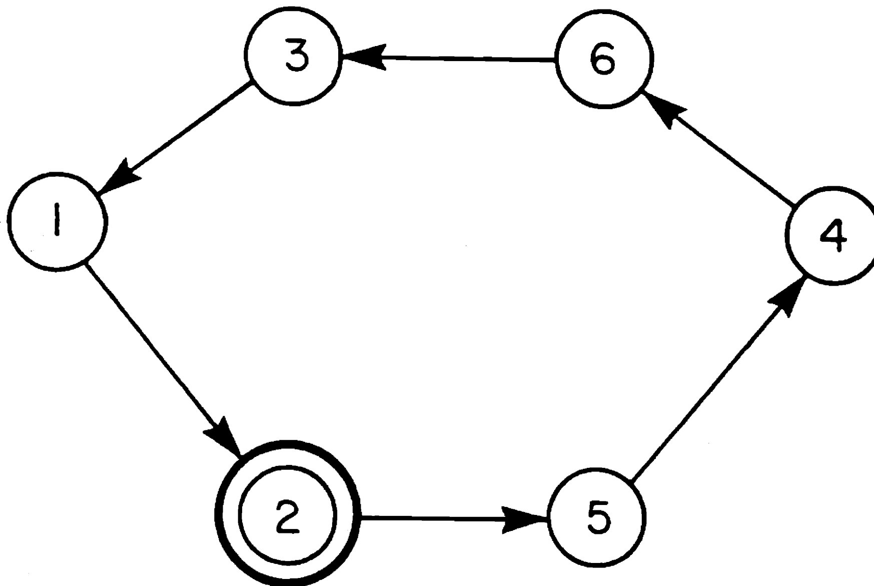
It is easiest to give a visual example of this algorithm, before the explicit form is presented. Assume that we have the partition {{1,2,3},{4,6},{5} of the set of integer from 1,...6. We might depict this partition as:



The canonical names of the partitions are 2, 6 and 5 respectively. The result of a UNION(6,5) would then result in the following:



Notice that only the two pointers that came out of the canonical elements 5 and 6 changed during this operation. The other transformation was the removal of the second circle around element 5, which keeps the number of canonical elements per equivalence class down to one. The result of the operation UNION(2,6) would then be:



As a standard, we always make the first parameter of the UNION operator into the canonical name of the new set in this algorithm. Again notice that only two pointers had to be moved to perform the UNION.

The FIND operation is performed by chasing the pointers around the loop, until an element is identified that is the canonical node.

The formal definition of this algorithm is:

INITIALIZATION2

```
    DIMENSION POINTERS(n)   'The vector of arrows
    DIMENSION CANONICAL(n)   'The logical variable that
```

```
                                'tells if element n is a
                                'canonical name
    FOR i=1 to n  'Initialize the above arrays
          CANONICAL(i) <-- TRUE  'All elements start out
                                       'as canonical names
          POINTERS(i) <-- i  'Each element starts out
                                       'pointing at itself
    NEXT i


FIND2(x)

    LOOP: IF CANONICAL(x)=TRUE  'see if we've got a canonical name
                THEN RETURNS(x)  'if so, return that name
    ELSE x <-- POINTERS(x) 'otherwise, chase around the
                                loop one step...
    GOTO LOOP  '... and try again


UNION2(x,y)

    IF CANONICAL(y)=FALSE  'we better not get a parameter
          OR CANONICAL(x)=FALSE  'that isn't a
                                      'canonical name
          THEN STOP  'Invalid parameter
    CANONICAL(y) <-- FALSE  'element y is no longer the
                                'canonical name for the
```

```
                                  'equivalence class
TEMP <-- POINTERS(y)   'remember what y used to point to
POINTERS(y) <-- POINTERS(x)   ' point y at what x used
                                        to point at...
POINTERS(x) <-- TEMP   'and point x at what y used to
                                              'point at.
RETURN  'That's it
```

O(1) FIND, O(log n) UNION

To be precise, the algorithms we will give can perform a total of n UNION operations in time O(n log n) worst case, and hence the average time per UNION is a worst case average O(log n). This FIND-UNION pair has as its basis, the pair of algorithms FIND1 and UNION1. A very fast FIND algorithm is produced by keeping a table of the canonical values as part of the data base. As with FIND1, only a lookup is then required to perform the FIND operation. The reason why the UNION1 function took so long (O(n)) to execute, was that a search of the entire table was required to find all the elements in a given class. The type of data base (a series of pointers) maintained for FIND2 and UNION2 will be used in this algorithm to swiftly scan through ONLY the elements in a SPECIFIC equivalence class when a UNION is performed.

The data base that we will maintain has three parts:

1) A quick look table for FIND

2) A set of lists. One list for each equivalence class. Each lists contain all the elements of its class.

3) A table that tells the size of each equivalence class.

We start with the initialization:

```
INITIALIZATION3

     DIMENSION M(n)   'reserve space for the array

     DIMENSION SIZEOF(n)   'there are n elements that we
                               'have to remember sizes for

     DIMENSION POINTERS(n)   'The vector of arrows

     FOR i=1 TO n   'Initialize the entire table so that...
          M(n) <-- n   '..the canonical name for each
                   'singleton equivalence class is the
                   'name of the singleton

          POINTERS(i) <-- i   'Each element starts out
                              'pointing at itself

          SIZEOF(i) <-- 1   'All classes start out
                                  'as size 1
     NEXT i
```

Next we define the FIND function, which is actually identical to FIND1.

```
FIND3(x)

        RETURNS (M(x))  'return the x'th entry in the table
```

Finally we come to the UNION operator. Basically what we will do in this function is identical to what UNION1 did, in that all entries in the table M(*) that need to be updated will be changed. Instead of searching the table to find out which entries should change, the circular lists will provide the names of all the elements that are in a given equivalence class. As the final speedup, we will select which of the two equivalence classes (of the parameters to UNION) has the fewest entries in M(*) to change. By making this selection of which equivalence class should have its name changed, we are able to achieve the O(log n) time bound for UNION3(x,y).

```
UNION3(x,y)

        X <-- FIND3(x)   'First get the true canonical
        Y <-- FIND3(y)   'names of the parameters
```

```
'Now make sure that X refers to the smaller class
IF SIZEOF(X) > SIZEOF(Y)    'and if its not...
        THEN  [          'then...
        TEMP <-- X     'Switch them
        X <-- Y
        Y <-- X  ]

'Now run through all the elements of this smaller class

CHANGE <-- X  'The first element to change is X
FOR i=1 to SIZEOF(X)  'We even know how many changes
                                          'to make
        M(CHANGE) <-- Y  'Give this element the same
                         'name as the other class
        CHANGE <-- POINTERS(CHANGE)  'Use the pointers
                         'to get to the next element in
                         'the list
NEXT i  'repeat till done

'Now we update the size data

SIZEOF(X) <-- SIZEOF(X)+SIZEOF(Y)  'Add the sizes of the two
               classes, to get the size of the new one

'Lastly we must update the list of what's in the X class
TEMP <-- POINTERS(Y)  'remember what Y used to point to...
POINTERS(Y) <-- POINTERS(X)  ' point Y at what X used
```

```
                                        to point at...
POINTERS(X) <-- TEMP  'and point X at what Y used to point at.

RETURN  'That's it
```

To obtain the computational complexity bound, we consider how many times the entry in M for a given node can change. The key point to notice is that every time an entry of M (say M(x)) changes, we know that the size of the equivalence class that x becomes a part of is at least twice as large as its old equivalence class. Hence no entry in M may change more than log(n) times. Since there are n elements, in the course of n-1 union operations no more than O(n(log n)) changes may be entries of M. In the above procedure, the only statements that might be executed more than n-1 times (once per union) are the statements including the place where M() is changed. With the above bound on the number of times that these statements can execute, we have the desired total bound of O(log n) work per union (averaged over n-1 unions)

For further reading on more advanced algorithms, the reader is referred to the paper by Tarjan [22] in which the analysis of a nearly linear (almost O(1) UNION and FIND) set union algorithm algorithm is presented.

REFERENCES

1 Alon Atai, Richard Lipton, Christos Papadimitriou, M. Rodeh, Covering Graphs by Simple Circuits, MIT Laboratory for Computer Science TM-155, February 1980.

2 Paul Baran, On Distributed Communications Networks, IEEE Trans. on Communications (CS-12).

3 Dimitri Bertsekas, Dynamic Behavior of Shortest Path Algorithms for Communication Networks, MIT Laboratory for Information and Decision Systems Report, LIDS-TH-1005, June 1980

4 J Clausen, L. A. Hansen, "Finding k edge-disjoint spanning trees of minimum total weight in a network: An application of matroid theory," Math. Prog. Study 13 (1980), pp.88-101.

5 S. M. Ch'ase, An Implemented Graph Algorithm for winning Shannon Switching Games, Communications of the ACM 15 (1972), pp. 253-256.

6 Jack Edmonds, Edge Disjoint Branchings, Combinatorial Algorithms, Edited by R, Rustin.

7 Jack Edmonds, "Minimum partition of a matroid into

independent subsets," Journal of Research of the Nation Bureau of Standards 69B (1965)

8 Jack Edmonds, Maximum Matching and a Polyhedron With 0, 1-vertices, Journal of Research of the National Bureau of Standards-B. Mathematics and Mathematical Physics, Vol. 69B, Nos. 1 and 2, January-June 1965.

9 Shimon Even, R. Tarjan, A Combinatorial Problem Which Is complete on Polynomial Space, Journal of the A.C.M., Vol. 23, NO. 4, October 1976, pp.710-719.

10 Steven G. Finn, Resynch Procedures and a Fail-Safe Network Protocol, IEEE Transactions on Communications, Vol. COM-27, No.6, June 1979

11 Daniel Friedman, Communications Complexity of Distributed Shortest Path Algorithms, MIT thesis, December 1978, Department of Electrical Engineering and Computer Science.

12 R. G Gallager, P. A. Humblet, P. M. Spira, A Distributed Algorithm for Minimum-Weight Spanning Trees, ACM Trans. on Programming Languages and Systems, Vol. 5, No. 1, January 1983, pp 66-77

13 T.Kameda, On Maximally Distant Spanning Trees of a Graph, Computing 17, pp. 115-119, (1976).

14 T. Kameda, S. Toida, Efficient Algorithms For Determining An Extremal of a Graph, 14th Annual Symposium on Switch and Automata THeory, pp. 12-15, (1973).

15 Genya Kishi, Yoji Kajantani, Maximally Distant Trees and Principal Partition of a Linear Graph, IEEE Transactions on Circuit Theory, Vol. CT-16, NO. 3, August 1969, pp. 323-330.

16 E. L. Lawler, Combinatorial Optimization: Networks and Matroids, Holt, Rinehart, and Winston, New York, 1976

17 Laszlo Lovasz, On Two Minimax Theorems on Graph, Journal of Combinatorial Theory (B) 21, 96-103, (1976).

18 K. Maruyama, G. Markowsky, On the Generation of Explicit Routing Tables, IBM.

19 James Roskind, Robert E. Tarjan, A Note On Finding Minimum-Cost Edge-Disjoint Spanning Trees, To be published.

20 Mischa Schwartz, Thomas Stern, ROuting Techniques In Computer Communication Networks, IEEE Transactions on Communications, Vol. Com-28, No. 4, pp 539-552, April 1980.

21 Yossi Shiloach, Edge-Disjoint Branchings in Directed

Multigraphs, Information Processing Letters, Vol. 8, No. 1, pp. 24-27, January 1979.

22 Robert Tarjan, A Good Algorithm for Edge-Disjoint Branching, Information Processing Letters, Vol. 3, No. 2, pp. 51-53, November 1974.

23 Robert Tarjan, Finding Edge Disjoint Spanning Trees, 8th Hawaii International conference on Systems Sciences, pp. 251-252, January 1975.

24 Christos H. Papadimitriou, Kenneth Striglitz, Combinatorial Optimization: Algorithms and Complexity, Prentice Hall, (1982)